学习 Webpack5 之路（实践篇）

上传日期：2021.08.03 626

本篇将从实践出发，在第一章节《基础配置》中使用 webpack 搭建一个基础的支持模块化开发的项目，在第二章节《进阶配置》中使用 webpack 搭建一个 SASS + TS + React 的项目。

前言

在上篇 《学习 Webpack5 之路（基础篇）》 中介绍了 Webpack 是什么，为什么选择 Webpack，Webpack 的基本概念介绍 3 个问题。

本篇将从实践出发，在第一章节《基础配置》中使用 webpack 搭建一个基础的支持模块化开发的项目，在第二章节《进阶配置》中使用 webpack 搭建一个 SASS + TS + React 的项目。

本文依赖的 webpack 版本信息如下：

- webpack-cli@4.7.2
- webpack@5.46.0

一、基础配置

接下来一起配置一个基础的 Webpack。

将支持以下功能：

- 分离开发环境、生产环境配置；
- 模块化开发；
- sourceMap 定位警告和错误；
- 动态生成引入 bundle.js 的 HTML5 文件；
- 实时编译；
- 封装编译、打包命令。

想直接看配置的同学 -> 本文源码地址： webpack Demo0

1. 新建项目

新建一个空项目：

```
// 新建 webpack-demo 文件夹
mkdir webpack-demo

// 进入 webpack-demo 目录
cd ./webpack-demo

// 初始化项目
npm init -y
```

新建 2 个 js 文件，并进行模块化开发：

```
// 进入项目目录
cd ./webpack-demo

// 创建 src 文件夹
mkdir src
```

```
// 创建 js文件
touch index.js
touch hello.js
```

index.js：

```
// index.js

import './hello.js'

console.log('index')
```

hello.js：

```
// hello.js
console.log('hello webpack')
```

项目结构如下：

```
- src
    - index.js
    - hello.js
- package.json
- node_modules
```

2. 安装

1. 安装 Node

Node 需要是**最新版本**，推荐使用 nvm 来管理 Node 版本。

> 将 ***Node.js*** 更新到最新版本，也有助于提高性能。除此之外，将你的 *package* 管理工具（例如 `npm` 或者 `yarn`）更新到最新版本，也有助于提高性能。较新的版本能够建立更高效的模块树以及提高解析速度。

- Node： 安装地址
- nvm： 安装地址

我安装的版本信息如下：

- node v14.17.3
- npm v6.14.13)

2. 安装 webpack

```
npm install webpack webpack-cli --save-dev
```

3. 新建配置文件

development(开发环境) 和 **production(生产环境)** 这两个环境下的构建目标存在着巨大差异。为代码清晰简明，为每个环境编写**彼此独立的 webpack 配置。**

```
// 进入项目目录
cd ./webpack-demo

// 创建 config 目录
mkdir config
```

```
// 进入 config 目录
cd ./config

// 创建通用环境配置文件
touch webpack.common.js

// 创建开发环境配置文件
touch webpack.dev.js

// 创建生产环境配置文件
touch webpack.prod.js
```

webpack-merge

使用 webpack-marge 合并通用配置和特定环境配置。

安装 webpack-merge：

```
npm i webpack-merge -D
```

通用环境配置：

```
// webpack.common.js

module.exports = {} // 暂不添加配置
```

开发环境配置：

```
// webpack.dev.js

const { merge } = require('webpack-merge')
const common = require('./webpack.common')

module.exports = merge(common, {}) // 暂不添加配置
```

生产环境配置：

```
// webpack.prod.js

const { merge } = require('webpack-merge')
const common = require('./webpack.common')

module.exports = merge(common, {}) // 暂不添加配置
```

项目结构如下：

```
- config
    - webpack.common.js
    - webpack.dev.js
    - webpack.prod.js
- src
    - index.js
    - hello.js
- package.json
- node_modules
```

4. 入口（entry）

入口起点(entry point) 指示 webpack 应该使用哪个模块，来作为构建其内部 依赖图(dependency graph) 的开始。进入入口起点后，webpack 会找出有哪些模块和库是入口起点（直接和间接）依赖的。

在本例中，使用 src/index.js 作为项目入口，webpack 以 src/index.js 为起点，查找所有依赖的模块。

修改 webpack.commom.js：

```
module.exports = merge(common, {
  // 入口
  entry: {
    index: './src/index.js',
  },
})
```

5. 输出（output）

output 属性告诉 webpack 在哪里输出它所创建的 *bundle*，以及如何命名这些文件。

生产环境的 output 需要通过 contenthash 值来区分版本和变动，可达到清缓存的效果，而**本地环境为了构建效率，则不引人 contenthash。**

新增 paths.js，封装路径方法：

```
const fs = require('fs')
const path = require('path')

const appDirectory = fs.realpathSync(process.cwd());
const resolveApp = relativePath => path.resolve(appDirectory, relativePath);

module.exports = {
  resolveApp
}
```

修改开发环境配置文件 webpack.dev.js：

```
module.exports =  merge(common, {
  // 输出
  output: {
    // bundle 文件名称
    filename: '[name].bundle.js',

    // bundle 文件路径
    path: resolveApp('dist'),

    // 编译前清除目录
    clean: true
  },
})
```

修改生产环境配置文件 webpack.prod.js：

```
module.exports =  merge(common, {
  // 输出
  output: {
    // bundle 文件名称 【只有这里和开发环境不一样】
    filename: '[name].[contenthash].bundle.js',

    // bundle 文件路径
    path: resolveApp('dist'),

    // 编译前清除目录
    clean: true
  },
})
```

上述 filename 的占位符解释如下：

- `[name]` - chunk name（例如 `[name].js` -> `app.js`）。如果 chunk 没有名称，则会使用其 id 作为名称
- `[contenthash]` - 输出文件内容的 md4-hash（例如 `[contenthash].js` -> `4ea6ff1de66c537eb9b2.js`）

6. 模式（mode）

通过 `mode` 配置选项，告知 webpack 使用相应模式的内置优化。

选	

项	描述
项	描述
development	会将 DefinePlugin 中 process.env.NODE_ENV 的值设置为 development . 为模块和 chunk 启用有效的名。
production	会将 DefinePlugin 中 process.env.NODE_ENV 的值设置为 production 。为模块和 chunk 启用确定性的混淆名称， FlagDependencyUsagePlugin ， FlagIncludedChunksPlugin ， ModuleConcatenationPlugin ， NoEmitOnErrorsPlugin 和 TerserPlugin 。

修改开发环境配置文件 webpack.dev.js：

```
module.exports =  merge(common, {
  // 开发模式
  mode: 'development',
})
```

修改生产环境配置文件 webpack.prod.js：

```
module.exports =  merge(common, {
  // 生产模式
  mode: 'production',
})
```

7. Source Map

当 webpack 打包源代码时，可能会很难追踪到 error 和 warning 在源代码中的原始位置。

为了更容易地追踪 error 和 warning，JavaScript 提供了 source maps 功能，可以将编译后的代码映射回原始源代码。

修改开发环境配置文件 webpack.dev.js：

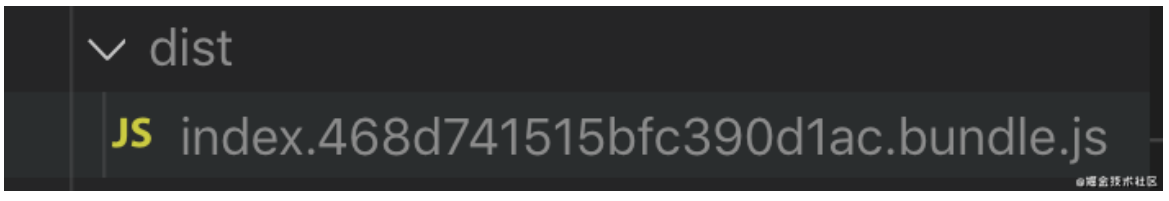
```
module.exports =  merge(common, {
  // 开发工具，开启 source map，编译调试
  devtool: 'eval-cheap-module-source-map',
})
```

source map 有许多 可用选项 。本例选择的是 `eval-cheap-module-source-map`

> 注：为加快生产环境打包速度，不为生产环境配置 *devtool*。

完成上述配置后，可以通过 `npx webpack --config config/webpack.prod.js` 打包编译。

编译后，会生成这样的目录结构：



8. HtmlWebpackPlugin

`npx webpack --config config/webpack.prod.js` 后仅生成了 bundle.js，我们还需要一个 HTML5 文件，用来动态引入打包生成的 bundle 文件。

引入 `HtmlWebpackPlugin` 插件，生成一个 HTML5 文件， 其中包括使用 `script` 标签的 body 中的所有 webpack 包。

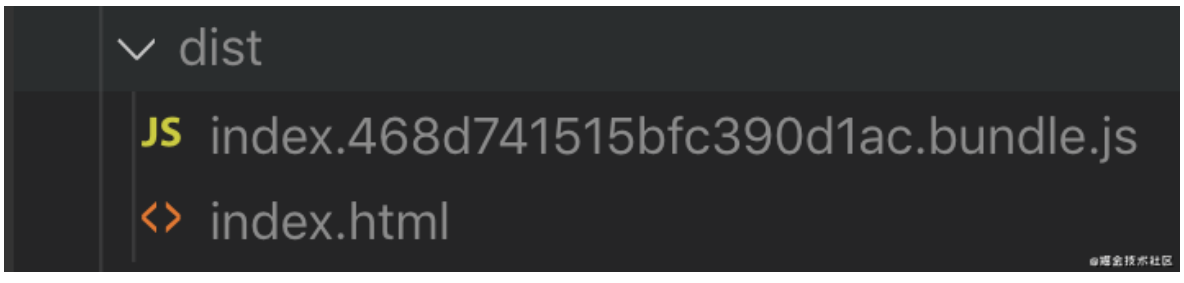
安装：

```
npm install --save-dev html-webpack-plugin
```

修改通用环境配置文件 webpack.commom.js：

```
module.exports = {
  plugins: [
    // 生成html，自动引入所有bundle
    new HtmlWebpackPlugin({
      title: 'release_v0',
    }),
  ],
}
```

重新 webpack 编译 `npx webpack --config config/webpack.prod.js` ，生成的目录结构如下：



新生成了 index.html，动态引入了 bundle.js 文件：

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="utf-8" />
  <title>release_v0</title>
  <meta name="viewport" content="width=device-width,initial-scale=1" />
  <script defer="defer" src="index.468d741515bfc390d1ac.bundle.js"></script>
 </head>
 <body></body>
</html>
```

9. DevServer

在每次编译代码时，手动运行 `npx webpack --config config/webpack.prod.js` 会显得很麻烦， webpack-dev-server 帮助我们在代码发生变化后自动编译代码。

`webpack-dev-server` 提供了一个基本的 web server，并且具有实时重新加载功能。

`webpack-dev-server` 默认配置 `conpress: true` ，为每个静态文件开启 gzip compression 。

安装：

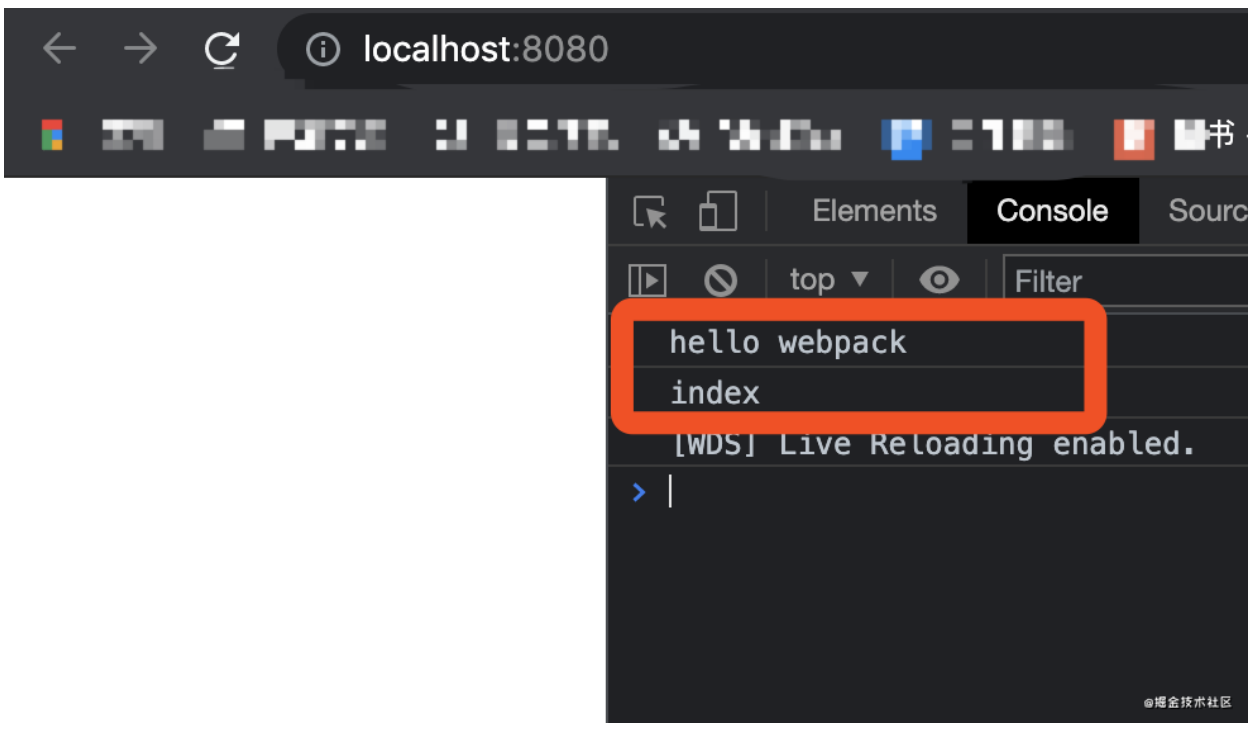
```
npm install --save-dev webpack-dev-server
```

修改开发环境配置文件 webpack.dev.js：

```
module.exports = merge(common, {
  devServer: {
    // 告诉服务器从哪里提供内容，只有在你想要提供静态文件时才需要。
    contentBase: './dist',
  },
})
```

完成上述配置后，可以通过 `npx webpack serve --open --config config/webpack.dev.js` 实时编译。

效果如图：



10.执行命令

上述配置文件完成后，优化 webpack 的实时编译、打包编译指令。

通过 cross-env 配置环境变量，区分开发环境和生产环境。

安装：

```
npm install --save-dev cross-env
```

修改 package.json：

```
{
    "scripts": {
        "dev": "cross-env NODE_ENV=development webpack serve --open --config
config/webpack.dev.js",
        "build": "cross-env NODE_ENV=production webpack --config config/webpack.prod.js"
      },
}
```

现在可以运行 webpack 指令：

- npm run dev：本地构建；
- npm run build：生产打包。

以上我们完成了一个基于 webpack 编译的支持模块化开发的简单项目。

> 源码地址：*webpack Demo0*

> 源码地址：[webpack Demo0]

二、进阶配置

本章节将继续完善配置，在上述配置基础上，用 Webpack 搭建一个 SASS + TS + React 的项目。

将支持以下功能：

- 加载图片；
- 加载字体；
- 加载 CSS；
- 使用 SASS；
- 使用 PostCSS，并自动为 CSS 规则添加前缀，解析最新的 CSS 语法，引入 css-modules 解决全局命名冲突问题；
- 使用 React；
- 使用 TypeScript。

想直接看配置的同学 -> 本文源码地址： [webpack Demo1]

1. 加载图片（Image）

在 webpack 5 中，可以使用内置的 [Asset Modules]，将 images 图像混入我们的系统中。

修改通用环境配置文件 webpack.commom.js：

```
const { resolveApp } = require('./paths');
module.exports = {
    module: {
        rules: [
          {
            test: /\.(png|svg|jpg|jpeg|gif)$/i,
            include: [
              resolveApp('src'),
            ],
            type: 'asset/resource',
          },
        ],
      },
}
```

> 在实际开发过程中，推荐将大图片上传至 *CDN*，提高加载速度。

2. 加载字体（Font）

使用 [Asset Modules] 接收字体文件。

修改通用环境配置文件 webpack.commom.js：

```
module.exports = {
    module: {
        rules: [
            {
               test: /.(woff|woff2|eot|ttf|otf)$/i,
               include: [
                  resolveApp('src'),
                ],
               type: 'asset/resource',
             },
          ]
     }
 }
```

> 在这里我发现，我不新增对 *ttf* 文件的配置，也能够引入字体不报错，不知道是什么问题，先记录一下，

> 有知道原因的大佬移步评论区。

> 在实际开发过程中，推荐将字体文件压缩上传至 *CDN*，提高加载速度。如配置字体的文字是固定的，还可以针对固定的文字生成字体文件，可以大幅缩小字体文件体积。

3. 加载 CSS

为了在 JavaScript 模块中 `import` 一个 CSS 文件，需要安装并配置 style-loader 和 css-loader 。

3.1 style-loader

style-loader 用于将 CSS 插入到 DOM 中，通过使用多个 `<style></style>` 自动把 styles 插入到 DOM 中.

3.2 css-loader

css-loader 对 `@import` 和 `url()` 进行处理，就像 js 解析 `import/require()` 一样，让 CSS 也能模块化开发。

3.3 安装配置

安装 CSS 相关依赖：

```
npm install --save-dev style-loader css-loader
```

修改通用环境配置文件 webpack.commom.js：

```
module.exports = {
    module: {
        rules: [
            {
                test: /\.css$/,
                include: paths.appSrc,
                use: [
                  // 将 JS 字符串生成为 style 节点
                  'style-loader',
                  // 将 CSS 转化成 CommonJS 模块
                  'css-loader',
                ],
              },
          ]
        }
    }
```

4. 使用 SASS

4.1 Sass

Sass 是一款强化 CSS 的辅助工具，它在 CSS 语法的基础上增加了变量、嵌套、混合、导入等高级功能。

4.2 sass-loader

sass-loader 加载 Sass/SCSS 文件并将他们编译为 CSS。

4.3 安装配置

安装 SASS 相关依赖：

```
npm install --save-dev sass-loader sass
```

修改通用环境配置文件 webpack.commom.js：

```
module.exports = {
    module: {
        rules: [
        {
        test: /.(scss|sass)$/,
        include: paths.appSrc,
        use: [
          // 将 JS 字符串生成为 style 节点
          'style-loader',
          // 将 CSS 转化成 CommonJS 模块
          'css-loader',
          // 将 Sass 编译成 CSS
          'sass-loader',
        ],
      },
```

5. 使用 PostCSS

5.1 PostCSS

PostCSS 是一个用 JavaScript 工具和插件转换 CSS 代码的工具。

- 可以自动为 CSS 规则添加前缀；
- 将最新的 CSS 语法转换成大多数浏览器都能理解的语法；
- css-modules 解决全局命名冲突问题。

5.2 postcss-loader

postcss-loader 使用 PostCSS 处理 CSS 的 loader。

5.3 安装配置

安装 PostCSS 相关依赖：

```
npm install --save-dev postcss-loader postcss postcss-preset-env
```

修改通用环境配置文件 webpack.commom.js：

```
const { resolveApp } = require('./paths');
module.exports = {
    module: {
        rules: [
{
  test: /\.module\.(scss|sass)$/,
  include: paths.appSrc,
  use: [
    // 将 JS 字符串生成为 style 节点
    'style-loader',
    // 将 CSS 转化成 CommonJS 模块
    {
      loader: 'css-loader',
      options: {
        // Enable CSS Modules features
        modules: true,
        importLoaders: 2,
        // 0 => no loaders (default);
        // 1 => postcss-loader;
        // 2 => postcss-loader, sass-loader
      },
    },
    // 将 PostCSS 编译成 CSS
    {
      loader: 'postcss-loader',
      options: {
        postcssOptions: {
          plugins: [
            [
              // postcss-preset-env 包含 autoprefixer
              'postcss-preset-env',

            ],
```

```
                ],
              },
            },
          },
          // 将 Sass 编译成 CSS
          'sass-loader',
        ],
      },
    ],
  },
}
```

> 为提升构建效率，为 *loader* 指定 *include*，通过使用 `include` 字段，仅将 *loader* 应用在实际需要将其转换的模块。

5. 使用 React + TypeScript

为了让项目的配置灵活性更高，不使用 create-reate-app 一键搭建项目，而是手动搭建 React 对应的配置项。

安装 React 相关：

```
npm i react react-dom @types/react @types/react-dom -D
```

安装 TypeScript 相关：

```
npm i -D typescript esbuild-loader
```

> 为提高性能，摒弃了传统的 *ts-loader*，选择最新的 *esbuild-loader*。

修改通用环境配置文件 webpack.commom.js：

```
module.exports = {
   resolve: {
       extensions: ['.tsx', '.ts', '.js'],
   },
   module: {
       rules: [
           {
               test: /\.(js|ts|jsx|tsx)$/,
               include: paths.appSrc,
               use: [
                 {
                   loader: 'esbuild-loader',
                   options: {
                     loader: 'tsx',
                     target: 'es2015',
                   },
                 }
               ]
             },
         ]
     }
 }
```

TypeScript 是 JavaScript 的超集，为其增加了类型系统，可以编译为普通 JavaScript 代码。

为兼容 TypeScript 文件，新增 typescript 配置文件 tsconfig.json：

```
{
    "compilerOptions": {
        "outDir": "./dist/",
        "noImplicitAny": true,

        "module": "es6",
```

```
        "target": "es5",
        "jsx": "react",
        "allowJs": true,
        "moduleResolution": "node",
        "allowSyntheticDefaultImports": true,
        "esModuleInterop": true,
      }
}
```

> 如果想在 *TypeScript* 中保留如 `import _ from 'lodash';` 的语法被让它作为一种默认的导入方式，需要在文件 ***tsconfig.json*** 中设置 `"allowSyntheticDefaultImports" : true` 和 `"esModuleInterop" : true` 。

注意：这儿有坑

1. `"allowSyntheticDefaultImports": true` 配置

TypeScript 配置文件 tsconfig.json 需要加 `"allowSyntheticDefaultImports": true` 配置，否则会提示 `can only be default-imported using the 'allowSyntheticDefaultImports' flag` 。

```
{
    "compilerOptions": {
        "allowSyntheticDefaultImports": true
    },
}
```

不加 `"allowSyntheticDefaultImports": true` 的加报错信息如下：

```
ERROR in /Users/jiaozi/Desktop/code/webpack-demo/src/index.tsx
./src/index.tsx 2:7-12
[tsl] ERROR in /Users/jiaozi/Desktop/code/webpack-demo/src/index.tsx(2,8)
      TS1259: Module '"/Users/jiaozi/Desktop/code/webpack-demo/node_modules/@types/react/index"'
 can only be default-imported using the 'allowSyntheticDefaultImports' flag

ERROR in /Users/jiaozi/Desktop/code/webpack-demo/src/index.tsx
./src/index.tsx 2:7-12
[tsl] ERROR in /Users/jiaozi/Desktop/code/webpack-demo/src/index.tsx(2,8)
      TS1259: Module '"/Users/jiaozi/Desktop/code/webpack-demo/node_modules/@types/react/index"'
 can only be default-imported using the 'allowSyntheticDefaultImports' flag
```


2. tsx 和 jsx 不能混合使用

在 tsx 中引入 jsx 文件报错如下：

```
ERROR in /Users/jiaozi/Desktop/code/webpack-demo/src/App.tsx
./src/App.tsx 2:18-31
[tsl] ERROR in /Users/jiaozi/Desktop/code/webpack-demo/src/App.tsx(2,19)
      TS7016: Could not find a declaration file for module './hello.jsx'. '/Users/jiaozi/Desktop
/code/webpack-demo/src/hello.jsx' implicitly has an 'any' type.
ts-loader-default_074dd68a7db44fad
 @ ./src/index.tsx 8:28-44

webpack 5.47.1 compiled with 1 error in 3268 ms
i [wdm]: Failed to compile.
```


以上我们完成了一个基于 webpack 编译的 SASS + TS + React 项目。

> 源码地址： *webpack Demo1*

三、总结

本文从 Webpack 基础配置、Webpack 进阶配置 2 个角度进行讲述，从 Webpack 实践着手，和你一起了解 Webpack。

下一篇《学习 Webpack5 之路（优化篇）》将从继续优化项目配置，尝试搭建一个最优的 Webpack 配置，敬请期待。

本文源码：

- webpack Demo0
- webpack Demo1

希望能对你有所帮助，感谢阅读～

别忘了点个赞鼓励一下我哦，笔芯♡

往期精彩

- 学习 Webpack5 之路（基础篇）

参考资料

- https://webpack.docschina.org/