webpack

WebPack可以看做是模块打包机：它做的事情是，分析你的项目结构，找到JavaScript模块以及其它的一些浏览器不能直接运行的拓展语言（Scss，TypeScript等），并将其打包为合适的格式以供浏览器使用。

构建就是把源代码转换成发布到线上的可执行 JavaScrip、CSS、HTML 代码，包括如下内容。

代码转换：TypeScript 编译成 JavaScript、SCSS 编译成 CSS 等。
文件优化：压缩 JavaScript、CSS、HTML 代码，压缩合并图片等。
代码分割：提取多个页面的公共代码、提取首屏不需要执行部分的代码让其异步加载。
模块合并：在采用模块化的项目里会有很多个模块和文件，需要构建功能把模块分类合并成一个文件。
自动刷新：监听本地源代码的变化，自动重新构建、刷新浏览器。
代码校验：在代码被提交到仓库前需要校验代码是否符合规范，以及单元测试是否通过。
自动发布：更新完代码后，自动构建出线上发布代码并传输给发布系统。
构建其实是工程化、自动化思想在前端开发中的体现，把一系列流程用代码去实现，让代码自动化地执行这一系列复杂的流程。 构建给前端开发注入了更大的活力，解放了我们的生产力。

2. 初始化项目
   mkdir vue-webpack
   cd vue-webpack
   npm init -y
3. 快速上手
   3.1 webpack核心概念
   Entry：入口，Webpack 执行构建的第一步将从 Entry 开始，可抽象成输入。
   Module：模块，在 Webpack 里一切皆模块，一个模块对应着一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。
   Chunk：代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。
   Loader：模块转换器，用于把模块原内容按照需求转换成新内容。
   Plugin：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的事情。
   Output：输出结果，在 Webpack 经过一系列处理并得出最终想要的代码后输出结果。
   Webpack 启动后会从Entry里配置的Module开始递归解析 Entry 依赖的所有 Module。每找到一个 Module， 就会根据配置的Loader去找出对应的转换规则，对 Module 进行转换后，再解析出当前 Module 依赖的 Module。 这些模块会以 Entry 为单位进行分组，一个 Entry 和其所有依赖的 Module 被分到一个组也就是一个 Chunk。最后 Webpack 会把所有 Chunk 转换成文件输出。 在整个流程中 Webpack 会在恰当的时机执行 Plugin 里定义的逻辑。

3.2 配置webpack
npm install webpack webpack-cli -D

3.2.1 创建src目录
mkdir src
3.2.2 创建dist目录
mkdir dist
3.2.3 基本配置文件
const path=require('path');
module.exports={
entry: './src/index.js',
output: {
path: path.resolve(__dirname,'dist'),
filename:'bundle.js'
},
module: {},
plugins: [],
devServer: {}
}
创建dist
创建index.html
配置文件webpack.config.js
entry：配置入口文件的地址
output：配置出口文件的地址
module：配置模块,主要用来配置不同文件的加载器
plugins：配置插件
devServer：配置开发服务器
const path=require('path');
module.exports={
entry: './src/index.js',
output: {
path: path.resolve(__dirname,'dist'),
filename:'bundle.js'
},
module: {},
plugins: [],
devServer: {}
}
3.2.4 创建index.html文件
在dist目录下创建index.html文件

4. 配置开发服务器
   npm i webpack-dev-server –D
   contentBase 配置开发服务运行时的文件根目录
   host：开发服务器监听的主机地址
   compress 开发服务器是否启动gzip等压缩
   port：开发服务器监听的端口
   - devServer:{
   - contentBase:path.resolve(__dirname,'dist'),
   - host:'localhost',
   - compress:true,
   - port:8080
   - }
   - "scripts": {
   - "build": "webpack –mode development",
   - "dev": "webpack-dev-server –open –mode development "
   - }
5. 支持加载css文件
   5.1 什么是Loader
   通过使用不同的Loader，Webpack可以要把不同的文件都转成JS文件,比如CSS、ES6/7、JSX等

test：匹配处理文件的扩展名的正则表达式
use：loader名称，就是你要使用模块的名称
include/exclude:手动指定必须处理的文件夹或屏蔽不需要处理的文件夹
query：为loaders提供额外的设置选项
5.2 loader三种写法
css-loader
style-loader
5.2.1 loader
加载CSS文件，CSS文件有可能在node_modules里，比如bootstrap和antd

module: {
rules: [
{
test: /.css/,

- loader:['style-loader','css-loader']
  }
  ]
  }

5.2.2 use
module: {
rules: [
{
test: /.css/,
- use:[‘style-loader’,’css-loader’]
}
]
},
5.2.3 use+loader
module: {
rules: [
{
test: /.css/,
include: path.resolve(__dirname,’src’),
exclude: /node_modules/,
use: [{
loader: ‘style-loader’,
options: {
insertAt:’top’
}
},’css-loader’]
}
]
}

1. 插件
   在 webpack 的构建流程中，plugin 用于处理更多其他的一些构建任务
   模块代码转换的工作由 loader 来处理
   除此之外的其他任何工作都可以交由 plugin 来完成
   6.1 自动产出html
   我们希望自动能产出HTML文件，并在里面引入产出后的资源

npm i html-webpack-plugin -D
minify 是对html文件进行压缩，removeAttrubuteQuotes是去掉属性的双引号
hash 引入产出资源的时候加上查询参数，值为哈希避免缓存
template 模版路径
plugins: [

- new HtmlWebpackPlugin({
- minify: {
- removeAttributeQuotes:true
- },
- hash: true,
- template: ‘./src/index.html’,
- filename:’index.html’

})]

1. 支持图片
   7.1 手动添加图片
   npm i file-loader url-loader -D
   file-loader 解决CSS等文件中的引入图片路径问题
   url-loader 当图片小于limit的时候会把图片BASE64编码，大于limit参数的时候还是使用file-loader 进行拷贝
   7.2 JS中引入图片
   7.2.1 JS
   let logo=require('./images/logo.png');
   let img=new Image();
   img.src=logo;
   document.body.appendChild(img);
   7.2.2 webpack.config.js
   {
   test:/.(jpg|png|bmp|gif|svg|ttf|woff|woff2|eot)/,
   use:[
   {
   loader:'url-loader',
   options:{limit:4096}
   }
   ]
   }
   7.3 在CSS中引入图片
   还可以在CSS文件中引入图片

7.3.1 CSS
.logo{
width:355px;
height:133px;
background-image: url(./images/logo.png);
background-size: cover;
}
7.3.2 HTML

8. 分离CSS
   因为CSS的下载和JS可以并行,当一个HTML文件很大的时候，我们可以把CSS单独提取出来加载

mini-css-extract-plugin
filename 打包入口文件
chunkFilename 用来打包import('module')方法中引入的模块
8.1 安装依赖模块
npm install –save-dev mini-css-extract-plugin
8.2 配置webpack.config.js

plugins: [
//参数类似于webpackOptions.output

- new MiniCssExtractPlugin({
- filename: '[name].css',
- chunkFilename:'[id].css'
- }),

{
test: /.css/,
include: path.resolve(__dirname,'src'),
exclude: /node_modules/,
use: [{

- loader: MiniCssExtractPlugin.loader
  },'css-loader']
  }

8.3 压缩JS和CSS
npm i uglifyjs-webpack-plugin optimize-css-assets-webpack-plugin -D
const UglifyJsPlugin = require("uglifyjs-webpack-plugin");
const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");
module.exports = {
mode: 'development',
optimization: {
minimizer: [
new UglifyJsPlugin({
cache: true,//启动缓存
parallel: true,//启动并行压缩
//如果为true的话，可以获得sourcemap
sourceMap: true // set to true if you want JS source maps
}),
//压缩css资源的
new OptimizeCSSAssetsPlugin({})
]
},
8.3 css和image存放单独目录
outputPath 输出路径
publicPath指定的是构建后在html里的路径
output: {
path: path.resolve(__dirname,'dist'),
filename: 'bundle.js',

- publicPath:'/'
  },

{
test:/.(jpg|jpeg|png|bmp|gif|svg|ttf|woff|woff2|eot)/,
use:[
{
loader:'url-loader',
options:{
limit: 4096,

- outputPath: 'images',
- publicPath:'/images'

}
}
]
}

plugins: [
new MiniCssExtractPlugin({

- filename: 'css/[name].css',
- chunkFilename:'css/[id].css'

}),
8.3 在HTML中使用图片
npm i html-withimg-loader -D
index.html

webpack.config.js

{

- test: /.(html|htm)$/,
- use: 'html-withimg-loader'

}

1. 编译less 和 sass
   9.1 安装less
   npm i less less-loader -D
   npm i node-sass sass-loader -D
   9.2 编写样式
   less

color:orange;
.less-container{
color:@color;
}
sass

$color:green;

.sass-container{
color:green;
}
webpack.config.js

{
test: /.less/,
include: path.resolve(__dirname,'src'),
exclude: /node_modules/,
use: [{
loader: MiniCssExtractPlugin.loader,
},'css-loader','less-loader']
},
{
test: /.scss/,
include: path.resolve(__dirname,'src'),
exclude: /node_modules/,
use: [{
loader: MiniCssExtractPlugin.loader,
},'css-loader','sass-loader']
},

10. 处理CSS3属性前缀
    为了浏览器的兼容性，有时候我们必须加入-webkit,-ms,-o,-moz这些前缀

Trident内核：主要代表为IE浏览器, 前缀为-ms
Gecko内核：主要代表为Firefox, 前缀为-moz
Presto内核：主要代表为Opera, 前缀为-o
Webkit内核：产要代表为Chrome和Safari, 前缀为-webkit
npm i postcss-loader autoprefixer -D
postcss-loader
index.css

::placeholder {
color: red;
}
postcss.config.js

module.exports={
plugins:[require('autoprefixer')]
}
webpack.config.js

{
test:/.css$/,
use:[MiniCssExtractPlugin.loader,'css-loader','postcss-loader'],

include:path.join(__dirname,'./src'),
exclude:/node_modules/
}

11. 转义ES6/ES7/JSX
    Babel其实是一个编译JavaScript的平台,可以把ES6/ES7,React的JSX转义为ES5
    babel-plugin-proposal-decorators
    11.1 安装依赖包
    npm i babel-loader @babel/core @babel/preset-env @babel/preset-react -D
    npm i @babel/plugin-proposal-decorators @babel/plugin-proposal-class-properties -D
    11.2 decorator
    //Option+Shift+A
    function readonly(target,key,discriptor) {
    discriptor.writable=false;
    }

class Person{
@readonly PI=3.14;
}
let p1=new Person();
p1.PI=3.15;
console.log(p1)
jsconfig.json

{
“compilerOptions”: {
“experimentalDecorators”: true
}
}
11.3 webpack.config.js
{
test: /.jsx?$/,
use: {
loader: ‘babel-loader’,
options:{
“plugins”: [
[“@babel/plugin-proposal-decorators”, { “legacy”: true }],
[“@babel/plugin-proposal-class-properties”, { “loose” : true }]
]
}
},
include: path.join(__dirname,’src’),
exclude:/node_modules/
}

11.4 babel runtime
babel 在每个文件都插入了辅助代码，使代码体积过大
babel 对一些公共方法使用了非常小的辅助代码，比如 _extend
默认情况下会被添加到每一个需要它的文件中。你可以引入 @babel/runtime 作为一个独立模块，来避免重复引入
babel-plugin-transform-runtime
npm install –save-dev @babel/plugin-transform-runtime
npm install –save @babel/runtime
.babelrc

{
“presets”: [“@babel/preset-env”],
“plugins”: [
[
“@babel/plugin-transform-runtime”,
{
“corejs”: false,
“helpers”: true,
“regenerator”: true,
“useESModules”: true
}
]
]
}
webpack打包的时候，会自动优化重复引入公共方法的问题

11.5 ESLint校验代码格式规范
eslint
eslint-loader
configuring
babel-eslint
Rules
ESlint 语法检测配置说明
npm install eslint eslint-loader babel-eslint –D
.eslintrc.js

module.exports = {
root: true,
//指定解析器选项
parserOptions: {
sourceType: ‘module’
},
//指定脚本的运行环境
env: {
browser: true,

},
// 启用的规则及其各自的错误级别
rules: {
“indent”: [“error”, 4],//缩进风格
“quotes”: [“error”, “double”],//引号类型
“semi”: [“error”, “always”],//关闭语句强制分号结尾
“no-console”: “error”,//禁止使用console
“arrow-parens”: 0 //箭头函数用小括号括起来
}
}
module: {
//配置加载规则
rules: [
{
test: /.js$/,
loader: ‘eslint-loader’,
enforce: “pre”,
include: [path.resolve(__dirname, ‘src’)], // 指定检查的目录
options: { fix: true } // 这里的配置项参数将会被传递到 eslint 的 CLIEngine
},

12. 如何调试打包后的代码
    webpack通过配置可以自动给我们source maps文件，map文件是一种对应编译文件和源文件的方法

source-map 把映射文件生成到单独的文件，最完整最慢
cheap-module-source-map 在一个单独的文件中产生一个不带列映射的Map
eval-source-map 使用eval打包源文件模块,在同一个文件中生成完整sourcemap
cheap-module-eval-source-map sourcemap和打包后的JS同行显示，没有映射列
devtool:’eval-source-map’
13.打包第三方类库
13.1 直接引入
import _ from ‘lodash’;
alert(_.join([‘a’,’b’,’c’],’@’));
13.2 插件引入
_ 函数会自动添加到当前模块的上下文，无需显示声明

- new webpack.ProvidePlugin({
- _:’lodash’
- })
  没有全局的$函数，所以导入依赖全局变量的插件依旧会失败

13.3 expose-loader
不需要任何其他的插件配合，只要将下面的代码添加到所有的loader之前

require(“expose-loader?libraryName!./file.js”);
{
test: require.resolve(“jquery”),
loader: “expose-loader?jQuery”
}
require(“expose-loader?$!jquery”);
13.4 externals
如果我们想引用一个库，但是又不想让webpack打包，并且又不影响我们在程序中以CMD、AMD或者window/global全局等方式进行使用，那就可以通过配置externals

const jQuery = require(“jquery”);
import jQuery from ‘jquery’;

- externals: {
- jquery: ‘jQuery’//如果要在浏览器中运行，那么不用添加什么前缀，默认设置就是global
- },
  module: {
  1. watch
     当代码发生修改后可以自动重新编译

watch: true,
watchOptions: {
ignored: /node_modules/, //忽略不用监听变更的目录
poll:1000, //每秒询问的文件变更的次数
aggregateTimeout: 500, //防止重复保存频繁重新编译,500毫秒内重复保存不打包
}
webpack定时获取文件的更新时间，并跟上次保存的时间进行比对，不一致就表示发生了变化,poll就用来配置每秒问多少次
当检测文件不再发生变化，会先缓存起来，等待一段时间后之后再通知监听者，这个等待时间通过aggregateTimeout配置
webpack只会监听entry依赖的文件
我们需要尽可能减少需要监听的文件数量和检查频率，当然频率的降低会导致灵敏度下降

15. 添加商标
    - new webpack.BannerPlugin(‘珠峰培训’),
16. 拷贝静态文件
    有时项目中没有引用的文件也需要打包到目标目录

npm i copy-webpack-plugin -D
new CopyWebpackPlugin([{
from: path.resolve(__dirname,’src/assets’),//静态资源目录源地址
to:path.resolve(__dirname,’dist/assets’) //目标地址，相对于output的path目录
}])

17. 打包前先清空输出目录

npm i clean-webpack-plugin -D
new CleanWebpackPlugin([path.resolve(__dirname,'dist')])

18. 服务器代理
如果你有单独的后端开发服务器 API，并且希望在同域名下发送 API 请求，那么代理某些 URL 会很有用。

18.1 不修改路径
//请求到 /api/users 现在会被代理到请求 http://localhost:3000/api/users。

proxy: {
"/api": 'http://localhost:3000'
}
18.2 修改路径
proxy: {
"/api": {
target: 'http://localhost:3000',
pathRewrite:{"^/api":""}
}
}
18.3 before after
before 在 webpack-dev-server 静态资源中间件处理之前，可以用于拦截部分请求返回特定内容，或者实现简单的数据 mock。

before(app){
app.get('/api/users', function(req, res) {
res.json([{id:1,name:'zfpx1'}])
})
}
18.4 webpack-dev-middleware
webpack-dev-middleware就是在 Express 中提供 webpack-dev-server 静态服务能力的一个中间件

npm install webpack-dev-middleware –save-dev
const express = require('express');
const app = express();
const webpack = require('webpack');
const webpackDevMiddleware = require('webpack-dev-middleware');
const webpackOptions = require('./webpack.config');
webpackOptions.mode = 'development';
const compiler = webpack(webpackOptions);
app.use(webpackDevMiddleware(compiler, {}));
app.listen(3000);
webpack-dev-server 的好处是相对简单，直接安装依赖后执行命令即可
而使用webpack-dev-middleware的好处是可以在既有的 Express 代码基础上快速添加 webpack-dev-server 的功能，同时利用 Express 来根据需要添加更多的功能，如 mock 服

务、代理 API 请求等

19. resolve解析
    19.1 extensions
    指定extension之后可以不用在require或是import的时候加文件扩展名,会依次尝试添加扩展名进行匹配

resolve: {
extensions: [“.js”,”.jsx”,”.json”,”.css”]
},
19.2 alias
配置别名可以加快webpack查找模块的速度

每当引入bootstrap模块的时候，它会直接引入bootstrap,而不需要从node_modules文件夹中按模块的查找规则查找
const bootstrap =
path.resolve(__dirname,’node_modules/_bootstrap@3.3.7@bootstrap/dist/css/bootstrap.css’);
resolve: {

- alias:{
- “bootstrap”:bootstrap
- }
  },
  19.3 modules
  对于直接声明依赖名的模块（如 react ），webpack 会类似 Node.js 一样进行路径搜索，搜索node_modules目录
  这个目录就是使用resolve.modules字段进行配置的 默认配置
  resolve: {
  modules: [‘node_modules’],
  }
  如果可以确定项目内所有的第三方依赖模块都是在项目根目录下的 node_modules 中的话
  resolve: {
  modules: [path.resolve(__dirname, ‘node_modules’)],
  }
  19.4 mainFields
  默认情况下package.json 文件则按照文件中 main 字段的文件名来查找文件

resolve: {
// 配置 target = “web” 或者 target = “webworker” 时 mainFields 默认值是：
mainFields: [‘browser’, ‘module’, ‘main’],
// target 的值为其他时，mainFields 默认值为：
mainFields: [“module”, “main”],
}
19.5 mainFiles

当目录下没有 package.json 文件时，我们说会默认使用目录下的 index.js 这个文件，其实这个也是可以配置的

```
resolve: {
mainFiles: ['index'], // 你可以添加其他默认使用的文件名
},
```

19.6 resolveLoader

resolve.resolveLoader用于配置解析 loader 时的 resolve 配置,默认的配置：

```
module.exports = {
resolveLoader: {
modules: [ 'node_modules' ],
extensions: [ '.js', '.json' ],
mainFields: [ 'loader', 'main' ]
}
};
```

20. noParse
    module.noParse 字段，可以用于配置哪些模块文件的内容不需要进行解析
    不需要解析依赖（即无依赖） 的第三方大型类库等，可以通过这个字段来配置，以提高整体的构建速度

```
module.exports = {
// …
module: {
noParse: /jquery|lodash/, // 正则表达式
// 或者使用函数
noParse(content) {
return /jquery|lodash/.test(content)
},
}
}…
```

    使用 noParse 进行忽略的模块文件中不能使用 import、require、define 等导入机制
21. DefinePlugin
    DefinePlugin创建一些在编译时可以配置的全局常量

```
new webpack.DefinePlugin({
PRODUCTION: JSON.stringify(true),
VERSION: "1",
EXPRESSION: "1+2",
COPYRIGHT: {
AUTHOR: JSON.stringify("珠峰培训")
}
})
console.log(PRODUCTION);
console.log(VERSION);
```

console.log(EXPRESSION);
console.log(COPYRIGHT);
如果配置的值是字符串，那么整个字符串会被当成代码片段来执行，其结果作为最终变量的值
如果配置的值不是字符串，也不是一个对象字面量，那么该值会被转为一个字符串，如true，最后的结果是 ‘true’
如果配置的是一个对象字面量，那么该对象的所有 key 会以同样的方式去定义
JSON.stringify(true) 的结果是 ‘true’

22. IgnorePlugin
    IgnorePlugin用于忽略某些特定的模块，让 webpack 不把这些指定的模块打包进去

import moment from ‘moment’;
console.log(moment);
new webpack.IgnorePlugin(/^.\/locale/,/moment$/)
第一个是匹配引入模块路径的正则表达式
第二个是匹配模块的对应上下文，即所在目录名

20. 区分环境变量
    日常的前端开发工作中，一般都会有两套构建环境
    一套开发时使用，构建结果用于本地开发调试，不进行代码压缩，打印 debug 信息，包含 sourcemap 文件
    一套构建后的结果是直接应用于线上的，即代码都是压缩后，运行时不打印 debug 信息，静态文件不包括 sourcemap
    webpack 4.x 版本引入了 mode 的概念
    当你指定使用 production mode 时，默认会启用各种性能优化的功能，包括构建结果优化以及 webpack 运行性能优化
    而如果是 development mode 的话，则会开启 debug 工具，运行时打印详细的错误信息，以及更加快速的增量编译构建
    20.1 环境差异
    生产环境
    可能需要分离 CSS 成单独的文件，以便多个页面共享同一个 CSS 文件
    需要压缩 HTML/CSS/JS 代码
    需要压缩图片
    开发环境
    需要生成 sourcemap 文件
    需要打印 debug 信息
    需要 live reload 或者 hot reload 的功能…
    20.2 获取mode参数
    npm install –save-dev optimize-css-assets-webpack-plugin
    const UglifyJSPlugin = require(‘webpack/lib/optimize/UglifyJsPlugin’);
    const OptimizeCssAssetsPlugin = require(‘optimize-css-assets-webpack-plugin’);
    module.exports=(env,argv) => ({
    optimization: {
    minimizer: argv.mode == ‘production’?[

new UglifyJSplugin({
cache: true,//启用缓存
parallel: true,// 使用多进程运行改进编译速度
sourceMap:true//生成sourceMap映射文件
}),
new OptimizeCssAssetsWebpackPlugin({})
]:[]
}
})
20.3 封装log方法
webpack 时传递的 mode 参数，是可以在我们的应用代码运行时，通过 process.env.NODE_ENV 这个变量获取
export default function log(…args) {
if (process.env.NODE_ENV == 'development') {
console.log.apply(console,args);
}
}
20.4 拆分配置
可以把 webpack 的配置按照不同的环境拆分成多个文件，运行时直接根据环境变量加载对应的配置即可

webpack.base.js：基础部分，即多个文件中共享的配置
webpack.development.js：开发环境使用的配置
webpack.production.js：生产环境使用的配置
webpack.test.js：测试环境使用的配置…
webpack-merge
const { smart } = require('webpack-merge')
const webpack = require('webpack')
const base = require('./webpack.base.js')
module.exports = smart(base, {
module: {
rules: [],
}
})

21. 多入口
有时候我们的页面可以不止一个HTML页面，会有多个页面，所以就需要多入口

const path=require('path');
const HtmlWebpackPlugin=require('html-webpack-plugin');
module.exports={
entry: {
index: './src/index.js',
login: './src/login.js'
},

```
output: {
path: path.resolve(__dirname, 'dist'),
filename: '[name].[hash].js',
publicPath: '/'
},
plugins: [
new HtmlWebpackPlugin({
minify: {
removeAttributeQuotes: true
},
hash: true,
template: './src/index.html',
chunks: ['index'],
filename: 'index.html'
}),
new HtmlWebpackPlugin({
minify: {
removeAttributeQuotes: true
},
hash: true,
chunks: ['login'],
template: './src/login.html',
filename: 'login.html'
})
],
}
```

22. 对图片进行压缩和优化
    image-webpack-loader可以帮助我们对图片进行压缩和优化

```
npm install image-webpack-loader –save-dev
{
test: /.(png|svg|jpg|gif|jpeg|ico)$/,
use: [
'file-loader',
```

- {
- loader: 'image-webpack-loader',
- options: {
- mozjpeg: {
- progressive: true,
- quality: 65
- },
- optipng: {
- enabled: false,

- },
- pngquant: {
- quality: ‘65-90’,
- speed: 4
- },
- gifsicle: {
- interlaced: false,
- },
- webp: {
- quality: 75
- }
- }
- },
  ]
  }
  参考
  参考文档
  webpack-start
  resolve
  常用loader列表
  webpack 可以使用 loader 来预处理文件。这允许你打包除 JavaScript 之外的任何静态资源。你可以使用 Node.js 来很简单地编写自己的 loader。 awesome-loaders

文件
raw-loader 加载文件原始内容（utf-8）
val-loader 将代码作为模块执行，并将 exports 转为 JS 代码
url-loader 像 file loader 一样工作，但如果文件小于限制，可以返回 data URL
file-loader 将文件发送到输出文件夹，并返回（相对）URL
JSON
json-loader 加载 JSON 文件（默认包含）
json5-loader 加载和转译 JSON 5 文件
cson-loader 加载和转译 CSON 文件
转换编译(Transpiling)
script-loader 在全局上下文中执行一次 JavaScript 文件（如在 script 标签），不需要解析
babel-loader 加载 ES2015+ 代码，然后使用 Babel 转译为 ES5
buble-loader 使用 Bublé 加载 ES2015+ 代码，并且将代码转译为 ES5
traceur-loader 加载 ES2015+ 代码，然后使用 Traceur 转译为 ES5
ts-loader 或 awesome-typescript-loader 像 JavaScript 一样加载 TypeScript 2.0+
coffee-loader 像 JavaScript 一样加载 CoffeeScript
模板(Templating)
html-loader 导出 HTML 为字符串，需要引用静态资源
pug-loader 加载 Pug 模板并返回一个函数
jade-loader 加载 Jade 模板并返回一个函数
markdown-loader 将 Markdown 转译为 HTML

react-markdown-loader 使用 markdown-parse parser(解析器) 将 Markdown 编译为 React 组件
posthtml-loader 使用 PostHTML 加载并转换 HTML 文件
handlebars-loader 将 Handlebars 转移为 HTML
markup-inline-loader 将内联的 SVG/MathML 文件转换为 HTML。在应用于图标字体，或将 CSS 动画应用于 SVG 时非常有用
样式
style-loader 将模块的导出作为样式添加到 DOM 中
css-loader 解析 CSS 文件后，使用 import 加载，并且返回 CSS 代码
less-loader 加载和转译 LESS 文件
sass-loader 加载和转译 SASS/SCSS 文件
postcss-loader 使用 PostCSS 加载和转译 CSS/SSS 文件
stylus-loader 加载和转译 Stylus 文件
清理和测试(Linting && Testing)
mocha-loader 使用 mocha 测试（浏览器/NodeJS）
eslint-loader PreLoader，使用 ESLint 清理代码
jshint-loader PreLoader，使用 JSHint 清理代码
jscs-loader PreLoader，使用 JSCS 检查代码样式
coverjs-loader PreLoader，使用 CoverJS 确定测试覆盖率
框架(Frameworks)
vue-loader 加载和转译 Vue 组件
polymer-loader 使用选择预处理器(preprocessor)处理，并且 require() 类似一等模块(first-class)的 Web 组件
angular2-template-loader 加载和转译 Angular 组件