

06 基础概念和常见配置项介绍（二）

更新时间：2019-06-21 09:49:42



“

构成我们学习最大障碍的是已知的东西，而不是未知的东西。

—— 贝尔纳

”

除了上篇介绍的跟 **entry** 和 **output** 相关的配置之外，本文将继续介绍 Webpack 其他重要配置。

resolve

Webpack 进行构建的时候会从入口文件开始（**entry**）遍历寻找各个模块的依赖，**resolve** 配置是帮助 Webpack 查找依赖模块的，通过 **resolve** 的配置，可以帮助 Webpack 快速查找依赖，也可以替换对应的依赖（比如开发环境用 **dev** 版本的 **lib** 等）。**resolve** 的基本配置语法如下：

```
module.exports = {  
  resolve: {  
    // resolve的配置  
  }  
};
```

下面来介绍下常用的 **resolve** 配置。

resolve.extensions

resolve.extensions 是帮助 Webpack 解析扩展名的配置，默认值：`['.wasm', '.mjs', '.js', '.json']`，所以我们引入 **js** 和 **json** 文件，可以不写它们的扩展名，通常我们可以加上 **.css**、**.less** 等，但是要确保同一个目录下没有重名的 **css** 或者 **js** 文件，如果存在的话，还是写全路径吧。

```
module.exports = {
  resolve: {
    extensions: ['.js', '.json', '.css']
  }
};
```

resolve.alias

`resolve.alias` 是最常用的配置，通过设置 `alias` 可以帮助 `webpack` 更快查找模块依赖，而且也能使我们编写代码更加方便。例如，我们在实际开发中经常会把源码都放到 `src` 文件夹，目录结构如下：

```
src
├─ lib
│   └─ utils.js
├─ pages
│   └─ demo
│       └─ index.js
```

在 `src/pages/demo/index.js` 中如果要引用 `src/lib/utils.js` 那么可以通过：`import utils from '../../lib/ut`
`ils'`；，如果目录更深一些，会越来越难看，这是可以通过设置 `alias` 来缩短这种写法，例如：

```
module.exports = {
  resolve: {
    alias: {
      src: path.resolve(__dirname, 'src'),
      '@lib': path.resolve(__dirname, 'src/lib')
    }
  }
};
```

经过设置了 `alias`，我们可以在任意文件中，不用理会目录结构，直接使用 `require('@lib/utls')` 或者 `require('src/lib/utls')` 来帮助 `Webpack` 定位模块。

Tips:

1. `alias` 的名字可以使用 `@` `!` `~` 等这些特殊字符，实际使用中 `alias` 都使用一种，或者不同类型使用一种，这样可以跟正常的模块引入区分开，增加辨识度；
2. 使用 `@` 注意不要跟 `npm` 包的 `scope` 冲突！
3. 这时在 `vscode` 中会导致我们检测不到 `utls` 中的内容，不能帮我们快速编写代码，可以通过在项目根目录创建 `jsconfig.json` 来帮助我们定位：

```
//jsconfig.json
{
  "compilerOptions": {
    "baseUrl": "./src",
    "paths": {
      "@lib/": ["src/lib"]
    }
  }
}
```

`alias` 还常被用于给生产环境和开发环境配置不同的 `lib` 库，例如下面写法，在线下开发环境使用具有 `debug` 功能的 `dev` 版本 `San`：

```
module.exports = {
  resolve: {
    alias: {
      san: process.env.NODE_ENV === 'production' ? 'san/dist/san.min.js' : 'san/dist/san.dev.js'
    }
  }
};
```

alias 还支持在名称末尾添加 **\$** 符号来缩小范围只命中以关键字结尾的导入语句，这样可以做精准匹配：

```
module.exports = {
  resolve: {
    alias: {
      react$: '/path/to/react.min.js'
    }
  }
};
```

```
import react from 'react'; // 精确匹配，所以 react.min.js 被解析和导入
import file from 'react/file.js'; // 非精确匹配，触发普通解析
```

resolve.mainFields

有一些我们用到的模块会针对不同宿主环境提供几份代码，例如提供 **ES5** 和 **ES6** 的两份代码，或者提供浏览器环境和 **nodejs** 环境两份代码，这时候在 **package.json** 文件里会做如下配置：

```
{
  "jsnext:main": "es/index.js", //采用ES6语法的代码入口文件
  "main": "lib/index.js", //采用ES5语法的代码入口文件，node
  "browser": "lib/web.js" //这个是专门给浏览器用的版本
}
```

在 **Webpack** 中，会根据 **resolve.mainFields** 的设置去决定使用哪个版本的模块代码，在不同的 **target** 下对应的 **resolve.mainFields** 默认值不同，默认 **target=web** 对应的默认值为：

```
module.exports = {
  resolve: {
    mainFields: ['browser', 'module', 'main']
  }
};
```

所以在 **target=web** 打包时，会寻找 **browser** 版本的模块代码。

下面是不常用的或者比较简单的配置：

- **resolve.mainFiles**：解析目录时候的默认文件名，默认是 **index**，即查找目录下面的 **index + resolve.extensions** 文件；
- **resolve.modules**：查找模块依赖时，默认是 **node_modules**；
- **resolve.symlinks**：是否解析符合链接（软连接，**symlink**）；
- **resolve.plugins**：添加解析插件，数组格式；
- **resolve.cachePredicate**：是否缓存，支持 **boolean** 和 **function**，**function** 传入一个带有 **path** 和 **require** 的对象，必须返回 **boolean** 值。

module

在 **webpack** 解析模块的同时，不同的模块需要使用不同类型的模块处理器来处理，这部分的设置就在 **module** 配置中。**module** 有两个配置：**module.noParse** 和 **module.rules**，

module.noParse

`module.noParse` 配置项可以让 **Webpack** 忽略对部分没采用模块化的文件的递归解析和处理，这样做的好处是能提高构建性能，接收的类型为正则表达式，或者正则表达式数组或者接收模块路径参数的一个函数：

```
module.exports = {
  module: {
    // 使用正则表达式
    noParse: /jquery|lodash/

    // 使用函数，从 Webpack 3.0.0 开始支持
    noParse: (content) => {
      // content 代表一个模块的文件路径
      // 返回 true or false
      return /jquery|lodash/.test(content);
    }
  }
}
```

Tips: 这里一定要确定被排除出去的模块代码中不能包含 `import`、`require`、`define` 等内容，以保证 **webpack** 的打包包含了所有的模块，不然会导致打包出来的 `js` 因为缺少模块而报错。

parser 来控制模块化语法

因为 **webpack** 是以模块化的 **JavaScript** 文件为入口，所以内置了对模块化 **JavaScript** 的解析功能。支持 **AMD**、**Commonjs**、**SystemJs**、**ES6**。 `parse` 属性可以更细粒度的配置哪些模块语法要解析，哪些不解析。简单来说，如果设置 `parser.commonjs=false`，那么代码里面使用 **commonjs** 的 `require` 语法引入模块，对应的模块就不会被解析到依赖中，也不会被处理，支持的选项包括：

```
module: {
  rules: [{
    test: /\.js$/,
    use: ['babel-loader'],
    parser: {
      amd: false, // 禁用 AMD
      commonjs: false, // 禁用 CommonJS
      system: false, // 禁用 SystemJS
      harmony: false, // 禁用 ES6 import/export
      requireInclude: false, // 禁用 require.include
      requireEnsure: false, // 禁用 require.ensure
      requireContext: false, // 禁用 require.context
      browserify: false, // 禁用 browserify
      requireJs: false, // 禁用 requirejs
    }
  }]
}
```

Tips: `parser` 是语法层面的限制，`noParse` 只能控制哪些文件不进行解析。

module.rules

`module.rules` 是在处理模块时，将符合规则条件的模块，提交给对应的处理器来处理，通常用来配置 `loader`，其类型是一个数组，数组里每一项都描述了如何去处理部分文件。每一项 `rule` 大致可以由以下三部分组成：

1. 条件匹配：通过 `test`、`include`、`exclude` 等配置来命中可以应用规则的模块文件；
2. 应用规则：对匹配条件通过后的模块，使用 `use` 配置项来应用 `loader`，可以应用一个 `loader` 或者按照从后往前

的顺序应用一组 loader，当然我们还可以分别给对应 loader 传入不同参数：

3. 重置顺序：一组 loader 的执行顺序默认是**从后到前（或者从右到左）**执行，通过 `enforce` 选项可以让其中一个 loader 的执行顺序放到最前（pre）或者是最后（post）。

条件匹配

如上所述，条件匹配相关的配置有 `test`、`include`、`exclude`、`resource`、`resourceQuery` 和 `issuer`。条件匹配的对象包括三类：`resource`，`resourceQuery` 和 `issuer`。

- `resource`: 请求文件的绝对路径。它已经根据 `resolve` 规则解析；
- `issuer`: 被请求资源（requested the resource）的模块文件的绝对路径，即导入时的位置。

举例来说明：从 `app.js` 导入 `'./style.css?inline'`：

- `resource` 是 `/path/to/style.css`；
- `resourceQuery` 是 `?` 之后的 `inline`；
- `issuer` 是 `/path/to/app.js`。

来看下 `rule` 对应的配置与匹配的对象关系表：

rule 配置项	匹配的对象
<code>test</code>	<code>resource</code> 类型
<code>include</code>	<code>resource</code> 类型
<code>exclude</code>	<code>resource</code> 类型
<code>resource</code>	<code>resource</code> 类型
<code>resourceQuery</code>	<code>resourceQuery</code> 类型
<code>issuer</code>	<code>issuer</code> 类型

举例说明，下面 `rule` 的配置项，匹配的条件为：来自 `src` 和 `test` 文件夹，不包含 `node_modules` 和 `bower_modules` 子目录，模块的文件路径为 `.tsx` 和 `.jsx` 结尾的文件。

```
{
  test: [/\.jsx?$/, /\.tsx?$/],
  include: [
    path.resolve(__dirname, 'src'),
    path.resolve(__dirname, 'test')
  ],
  exclude: [
    path.resolve(__dirname, 'node_modules'),
    path.resolve(__dirname, 'bower_modules')
  ]
}
```

Loader 配置

`loader` 是解析处理器，通过 `loader` 我们可以将 ES6 语法的 `js` 转化成 ES5 的语法，可以将图片转成 base64 的 `dataURL`，在 JavaScript 文件中直接 `import css` 和 `html` 也是通过对应的 `loader` 来实现的。

我们在使用对应的 `loader` 之前，需要先安装它，例如，我们要在 JavaScript 中引入 `less`，则需要安装 `less-loader`：

```
npm i -D less-loader
```

然后在 `module.rules` 中指定 `*.less` 文件都是用 `less-loader`：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.less$/, use: 'less-loader'
      }
    ]
  }
}
```

简单来理解上面的配置，`test` 项使用 `/\.less$/` 正则匹配需要处理的模块文件（即 `less` 后缀的文件），然后交给 `less-loader` 来处理，这里的 `less-loader` 是个 `string`，最终会被作为 `require()` 的参数来直接使用。

这样 `less` 文件都会被 `less-loader` 处理成对应的 `css` 文件。

除了直接在 `webpack.config.js` 是用 `loader` 的方式之外，还可以在对应的 `JavaScript` 文件中是用 `loader`：

```
const html = require('html-loader!./loader.html');
console.log(html);
```

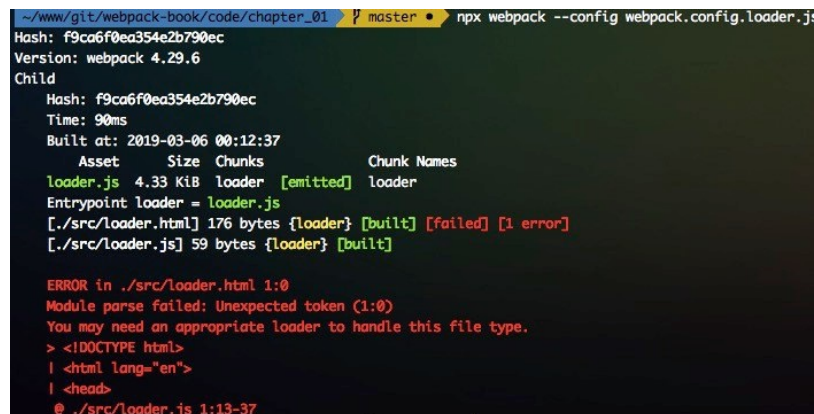
上面的代码，实际是将 `loader.html` 的内容转化成 `string` 变量，直接给输出了，等同于：

```
const html = require('./loader.html');
console.log(html);
```

加上下面配置的效果：

```
module.exports = {
  module: {
    rules: [{test: /\.html$/, use: ['html-loader']}]}
};
```

如果没有 `html-loader`，直接 `require` 一个 `html` 文件，会被当初 `js` 模块来执行，则会报错：



```
~/www/git/webpack-book/code/chapter_01 master • npx webpack --config webpack.config.loader.js
Hash: f9ca6f0ea354e2b790ec
Version: webpack 4.29.6
Child
  Hash: f9ca6f0ea354e2b790ec
  Time: 90ms
  Built at: 2019-03-06 00:12:37
  Asset      Size  Chunks             Chunk Names
  loader.js  4.33 KiB    loader  [emitted]  loader
  Entrypoint loader = loader.js
  [./src/loader.html] 176 bytes {loader} [built] [failed] [1 error]
  [./src/loader.js] 59 bytes {loader} [built]

  ERROR in ./src/loader.html 1:0
  Module parse failed: Unexpected token (1:0)
  You may need an appropriate loader to handle this file type.
  > <!DOCTYPE html>
  | <html lang="en">
  | <head>
  @ ./src/loader.js 1:13-37
```

Tips: `require('html-loader!./loader.html')` 中 `!` 类似 `Unix` 系统中命令行的管道，这里 `!` 隔开的命令是从右到左解析的，即先加载 `loader.html` 然后在将加载的文件内容传给 `html-loader` 处理。

综上，`loader` 有两种配置方式：

1. 使用 `webpack.config.js` 的配置方式：

```
module.exports = {
  module: {
    rules: [{test: /\.html$/, use: ['html-loader']}]}
};
```

2. 在 JavaScript 文件内使用内联配置方式:

```
const html = require('html-loader!./loader.html');
// or
import html from 'html-loader!./loader.html';
```

Tips: use 中传递字符串（如: `use: ['style-loader']`）是 loader 属性的简写方式（如: `use: [{loader: 'style-loader'}]`）

Loader 的参数

给 loader 传参的方式有两种:

1. 通过 `options` 传入
2. 通过 `query` 的方式传入:

```
// inline内联写法, 通过 query 传入
const html = require("html-loader?attrs[]=img:src&attrs[]=img:data-src!./file.html");
// config内写法, 通过 options 传入
module: {
  rules: [{
    test: /\.html$/,
    use: [{
      loader: 'html-loader',
      options: {
        minimize: true,
        removeComments: false,
        collapseWhitespace: false
      }
    }]
  }]
}
// config内写法, 通过 query 传入
module: {
  rules: [{
    test: /\.html$/,
    use: [ {
      loader: 'html-loader?minimize=true&removeComments=false&collapseWhitespace=false',
    }]
  }]
}
```

Loader 的解析顺序

对于一些类型的模块, 简单配置一个 loader 是不能够满足需求的, 例如 `less` 模块类型的文件, 只配置了 `less-loader` 仅仅是将 `Less` 语法转换成了 `CSS` 语法, 但是 `JS` 还是不能直接使用, 所以还需要添加 `css-loader` 来处理, 这时候就需要注意 Loader 的解析顺序了。前面已经提到了, Webpack 的 Loader 解析顺序是从右到左（从后到前）的, 即:

```
// query 写法从右到左，使用!隔开
const styles = require('css-loader!less-loader!./src/index.less');
// 数组写法，从后到前
module.exports = {
  module: {
    rules: [
      {
        test: /\.less$/,
        use: [
          {
            loader: 'style-loader'
          },
          {
            loader: 'css-loader'
          },
          {
            loader: 'less-loader'
          }
        ]
      }
    ]
  }
};
```

如果需要调整 Loader 的执行顺序，可以使用 `enforce`，`enforce` 取值是 `pre|post`，`pre` 表示把放到最前，`post` 是放到最后：

```
use: [
  {
    loader: 'babel-loader',
    options: {
      cacheDirectory: true
    },
    // enforce: 'post' 的含义是把该 loader 的执行顺序放到最后
    // enforce 的值还可以是 pre，代表把 loader 的执行顺序放到最前
    enforce: 'post'
  }
];
```

oneOf: 只应用第一个匹配的规则

`oneOf` 表示对该资源只应用第一个匹配的规则，一般结合 `resourceQuery`，具体代码来解释：

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.css$/,
        oneOf: [
          {
            resourceQuery: /inline/, // foo.css?inline
            use: 'url-loader'
          },
          {
            resourceQuery: /external/, // foo.css?external
            use: 'file-loader'
          }
        ]
      }
    ]
  }
};
```

`plugin` 插件

`plugin` 是 Webpack 的重要组成部分，通过 `plugin` 可以解决 `loader` 解决不了的问题。Webpack 本身就是有很多插件组成的，所以内置了很多插件，我们可以直接通过 `webpack` 对象的属性来直接使用，例如：`webpack.optimize.UglifyJsPlugin`

```
module.exports = {
  //....
  plugins: [
    // 压缩js
    new webpack.optimize.UglifyJsPlugin();
  ]
}
```

除了内置的插件，我们也可以通过 NPM 包的方式来使用插件：

```
// 非默认的插件
const ExtractTextPlugin = require('extract-text-webpack-plugin');
module.exports = {
  //....
  plugins: [
    // 导出css文件到单独的内容
    new ExtractTextPlugin({
      filename: 'style.css'
    })
  ]
};
```

Tips: `loader` 面向的是解决某个或者某类模块的问题，而 `plugin` 面向的是项目整体，解决的是 `loader` 解决不了的问题。

小结

在本小节中，我们讲解了Webpack 相关的除 `entry` 和 `output` 外的基础配置项，这里总结下项目经常配置的并且比较重要的配置项列表，供大家复习本小节内容：

- `resolve`：模块依赖查找相关的配置
 - `resolve.extensions`：可以省略解析扩展名的配置，配置太多反而会导致webpack 解析效率下降；
 - `resolve.alias`：通过设置 `alias` 可以帮助 webpack 更快查找模块依赖，精简代码书写时相对路径的书写；
- `module.rules`：loader 相关的配置，每个 `rule` 重要的内容有：
 - `test`：正则匹配需要处理的模块文件；
 - `use`：loader 数组配置，内部有 `loader` 和 `options`；
 - `include`：包含；
 - `exclude`：排除；
- `plugins`：插件。

本小节 Webpack 相关面试题：

1. 能不能手写一个 Webpack 配置？记住重点配置项：`entry`、`output`、`module.rules`（loader）和 `plugin`。
2. 在 JS 文件中怎么调用 Loader 来处理一个模块？
3. Loader 的解析顺序是怎样的？

专栏代码已经整理好给大家共享出来：

[点此下载代码](#) [代码](#)



05 基础概念和常见配置项介绍
(一)

07 Webpack 中的模块化开发

