常用的 loader 有哪些，该如何配置？

之前说了一些关于 entry 和 output 的相关配置。今天来说一说 loader。

一、为什么需要使用 Loader

Webpack 的一大特点是： 代码转换 TypeScript 编译成 JavaScript、SCSS,LESS 编译成 CSS 等。这里面说的是进行代码转换。为什么需要代码转换呢。对于目前编程来说，我们需要更高级的语法，和更高级的预处理器，可以更方便我们的编程，以及多人开发。这里面有我们熟知的 ES6，SASS LESS，Stylus，typeScript 等等。这些高级语言和预处理器，在我们的项目中使用的话是非常便于我们开发的，但是每种高级处理器都不能直接在浏览器上运行，只能挨个编译然后组合，最终适应到浏览器可以执行的内容，进行执行。但是这样的过程，在没有外力帮助的情况下是很难完成的，之后就有类似 gulp 和 Webpack 这种类似的工具。以上说的代码转换就是 Webpack 中 loader 做的事情。

二、常用 Loader 有哪些，该如何配置

Webpack 允许我们使用 loader 来处理文件，loader 是一个导出为 function 的 node 模块。可以将匹配到的文件进行一次转换，同时 loader 可以链式传递。

loader 的使用方式

一般 loader 的使用方式分为三种：
1：在配置文件 Webpack.config.js 中配置

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

2：通过命令行参数方式

Webpack --module-bind 'txt=raw-loader'

3：通过内联使用

import txt from 'raw-loader!./file.txt';

但是第一种是最常用的，我们配置更多是采用第一种配置方法。

Webpack 常用的 loader

- 样式：style-loader、css-loader、less-loader、sass-loader 等
- 文件：raw-loader、file-loader 、url-loader 等
- 编译：babel-loader、coffee-loader 、ts-loader 等
- 校验测试：mocha-loader、jshint-loader 、eslint-loader 等

- vue-loader、coffee-loader、babel-loader 等可以将特定文件格式转成 js 模块、将其他语言转化为 js 语言和编译下一代 js 语言
- file-loader、url-loader 等可以处理资源，file-loader 可以复制和放置资源位置，并可以指定文件名模板，用 hash 命名更好利用缓存。
- url-loader 可以将小于配置 limit 大小的文件转换成内敛 Data Url 的方式，减少请求。
- raw-loader 可以将文件已字符串的形式返回
- imports-loader、exports-loader 等可以向模块注入变量或者提供导出模块功能，常见场景是：
  1:jquery 插件注入$,imports-loader?$=jquery
  2:禁用 AMD，imports-loader?define=false
  等同于：var $ = require("jquery") 和 var define = false;
- expose-loader:暴露对象为全局变量

常用 loader 配置

babel-loader

babel-loader 用 babel 转换代码。

安装

npm install babel-loader@8.0.0-beta.0 @babel/core @babel/preset-env Webpack

用法

```
module: {
  rules: [
    {
      test: /\.js$/,
      exclude: /(node_modules|bower_components)/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env']
        }
      }
    }
  ]
}
```

- test: 一个正则表达式，匹配文件名
- use: 一个数组，里面放需要执行的 loader，倒序执行，从右至左。
- exclude: 取消匹配 node_modules 里面的文件

babel 的配置建议在根目录下新建一个.babelrc 文件

```
{
    "presets": [
        "env",
        "stage-0",
        "react"
    ],
    "plugins": [
        "transform-runtime",
        "transform-decorators-legacy",
        "add-module-exports"
    ]
}
```

- presets: 预设，一个预设包含多个插件 起到方便作用 不用引用多个插件
- env: 只转换新的句法，例如 const let => ..等 不转换 Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise、Object.assign。
- stage-0: es7 提案转码规则 有 0 1 2 3 阶段 0 包含 1 2 3 里面的所有
- react: 转换 react jsx 语法
- plugins: 插件 可以自己开发插件 转换代码(依赖于 ast 抽象语法数)

- transform-runtime: 转换新语法，自动引入 polyfill 插件，另外可以避免污染全局变量
- transform-decorators-legacy: 支持装饰器
- add-module-exports: 转译 export default {}; 添加上 module.exports = exports.default 支持 commonjs

CSS-loader，Style-loader

- css-loader: 支持 css 中的 import
- style-loader: 把 css 写入 style 内嵌标签

安装

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

使用

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


file-loader url-loader

默认情况下，生成的文件的文件名就是文件内容的 MD5 哈希值并会保留所引用资源的原始扩展名。

安装

```
npm install --save-dev file-loader
```

使用

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(png|jpg|gif)$/,
        use: [
          {
```

```
            loader: 'file-loader',
            options: {}
          }
        ]
      }
    ]
  }
}
```

url-loader 功能类似于 file-loader，但是在文件大小（单位 byte）低于指定的限制时，可以返回一个 DataURL。

安装

```
npm install --save-dev url-loader
```

用法

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(png|jpg|gif)$/,
        use: [
          {
            loader: 'url-loader',
            options: {
              limit: 8192
            }
          }
        ]
      }
    ]
  }
}
```


三、Loader 的扩展

一个 Loader 的职责是单一的，只需要完成一种转换。 如果一个源文件需要经历多步转换才能正常使用，就通过多个 Loader 去转换。 在调用多个 Loader 去转换一个文件时，每个 Loader 会链式的顺序执行， 第一个 Loader 将会拿到需处理的原内容，上一

个 Loader 处理后的结果会传给下一个接着处理，最后的 Loader 将处理后的最终结果返回给 Webpack。

所以，在你开发一个 Loader 时，请保持其职责的单一性，你只需关心输入和输出。

Loader 基础

由于 Webpack 是运行在 Node.js 之上的，一个 Loader 其实就是一个 Node.js 模块，这个模块需要导出一个函数。 这个导出的函数的工作就是获得处理前的原内容，对原内容执行处理后，返回处理后的内容。

一个最简单的 Loader 的源码如下：

```
module.exports = function(source) {
  // source 为 compiler 传递给 Loader 的一个文件的原内容
  // 该函数需要返回处理后的内容，这里简单起见，直接把原内容返回了，相当于该 Loader 没有做任何转换
  return source;
};
```

由于 Loader 运行在 Node.js 中，你可以调用任何 Node.js 自带的 API，或者安装第三方模块进行调用：

```
const sass = require('node-sass');
module.exports = function(source) {
  return sass(source);
};
```

Loader 进阶

以上只是个最简单的 Loader，Webpack 还提供一些 API 供 Loader 调用，下面来一一介绍。

获得 Loader 的 options

如何在自己编写的 Loader 中获取到用户传入的 options 呢？需要这样做：

```
const loaderUtils = require('loader-utils');
module.exports = function(source) {
```

```
  // 获取到用户给当前 Loader 传入的 options
  const options = loaderUtils.getOptions(this);
  return source;
};
```

返回其它结果

上面的 Loader 都只是返回了原内容转换后的内容，但有些场景下还需要返回除了内容之外的东西。

例如以用 babel-loader 转换 ES6 代码为例，它还需要输出转换后的 ES5 代码对应的 Source Map，以方便调试源码。 为了把 Source Map 也一起随着 ES5 代码返回给 Webpack，可以这样写：

```
module.exports = function(source) {
  // 通过 this.callback 告诉 Webpack 返回的结果
  this.callback(null, source, sourceMaps);
  // 当你使用 this.callback 返回内容时，该 Loader 必须返回 undefined，
  // 以让 Webpack 知道该 Loader 返回的结果在 this.callback 中，而不是 return 中
  return;
};
```


其中的 this.callback 是 Webpack 给 Loader 注入的 API，以方便 Loader 和 Webpack 之间通信。 this.callback 的详细使用方法如下：

```
this.callback(
    // 当无法转换原内容时，给 Webpack 返回一个 Error
    err: Error | null,
    // 原内容转换后的内容
    content: string | Buffer,
    // 用于把转换后的内容得出原内容的 Source Map，方便调试
    sourceMap?: SourceMap,
    // 如果本次转换为原内容生成了 AST 语法树，可以把这个 AST 返回，
    // 以方便之后需要 AST 的 Loader 复用该 AST，以避免重复生成 AST，提升性能
    abstractSyntaxTree?: AST
);
```

Source Map 的生成很耗时，通常在开发环境下才会生成 Source Map，其它环境下不用生成，以加速构建。 为此 Webpack 为 Loader 提供了 this.sourceMap API 去告诉 Loader 当前构建环境下用户是否需要 Source Map。 如果你编写的 Loader 会生成 Source Map，请考虑到这点。

同步与异步

Loader 有同步和异步之分，上面介绍的 Loader 都是同步的 Loader，因为它们的转换流程都是同步的，转换完成后再返回结果。 但在有些场景下转换的步骤只能是异步完成的，例如你需要通过网络请求才能得出结果，如果采用同步的方式网络请求就会阻塞整个构建，导致构建非常缓慢。

在转换步骤是异步时，你可以这样：

```
module.exports = function(source) {
    // 告诉 Webpack 本次转换是异步的，Loader 会在 callback 中回调结果
    var callback = this.async();
    someAsyncOperation(source, function(err, result, sourceMaps, ast) {
        // 通过 callback 返回异步执行后的结果
        callback(err, result, sourceMaps, ast);
    });
};
```


处理二进制数据

在默认的情况下，Webpack 传给 Loader 的原内容都是 UTF-8 格式编码的字符串。 但有些场景下 Loader 不是处理文本文件，而是处理二进制文件，例如 file-loader，就需要 Webpack 给 Loader 传入二进制格式的数据。 为此，你需要这样编写 Loader：

```
module.exports = function(source) {
    // 在 exports.raw === true 时，Webpack 传给 Loader 的 source 是 Buffer 类型的
    source instanceof Buffer === true;
    // Loader 返回的类型也可以是 Buffer 类型的
    // 在 exports.raw !== true 时，Loader 也可以返回 Buffer 类型的结果
    return source;
};
// 通过 exports.raw 属性告诉 Webpack 该 Loader 是否需要二进制数据
module.exports.raw = true;
```

以上代码中最关键的代码是最后一行 module.exports.raw = true;，没有该行 Loader 只能拿到字符串。

缓存加速

在有些情况下，有些转换操作需要大量计算非常耗时，如果每次构建都重新执行重复的转换操作，构建将会变得非常缓慢。 为此，Webpack 会默认缓存所有 Loader 的处理结果，也就是说在需要被处理的文件或者其依赖的文件没有发生变化时， 是不会重新调用对应的 Loader 去执行转换操作的。

如果你想让 Webpack 不缓存该 Loader 的处理结果，可以这样：

```
module.exports = function(source) {
  // 关闭该 Loader 的缓存功能
  this.cacheable(false);
  return source;
};
```

其它 Loader API

除了以上提到的在 Loader 中能调用的 Webpack API 外，还存在以下常用 API：

- this.context：当前处理文件的所在目录，假如当前 Loader 处理的文件是 /src/main.js，则 this.context 就等于 /src。
- this.resource：当前处理文件的完整请求路径，包括 querystring，例如 /src/main.js?name=1。
- this.resourcePath：当前处理文件的路径，例如 /src/main.js。
- this.resourceQuery：当前处理文件的 querystring。
- this.target：等于 Webpack 配置中的 Target
- this.loadModule：但 Loader 在处理一个文件时，如果依赖其它文件的处理结果才能得出当前文件的结果时， 就可以通过 this.loadModule(request: string, callback: function(err, source, sourceMap, module)) 去获得 request 对应文件的处理结果。
- this.resolve：像 require 语句一样获得指定文件的完整路径，使用方法为 resolve(context: string, request: string, callback: function(err, result: string))。

- this.addDependency：给当前处理文件添加其依赖的文件，以便再其依赖的文件发生变化时，会重新调用 Loader 处理该文件。使用方法为 addDependency(file: string)。
- this.addContextDependency：和 addDependency 类似，但 addContextDependency 是把整个目录加入到当前正在处理文件的依赖中。使用方法为 addContextDependency(directory: string)。
- this.clearDependencies：清除当前正在处理文件的所有依赖，使用方法为 clearDependencies()。
- this.emitFile：输出一个文件，使用方法为 emitFile(name: string, content: Buffer|string, sourceMap: {...})。

其它没有提到的 API 可以去 Webpack 官网 查看。

加载本地 Loader

在开发 Loader 的过程中，为了测试编写的 Loader 是否能正常工作，需要把它配置到 Webpack 中后，才可能会调用该 Loader。 在前面的章节中，使用的 Loader 都是通过 Npm 安装的，要使用 Loader 时会直接使用 Loader 的名称，代码如下：

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


如果还采取以上的方法去使用本地开发的 Loader 将会很麻烦，因为你需要确保编写的 Loader 的源码是在 node_modules 目录下。 为此你需要先把编写的 Loader 发布到 Npm 仓库后再安装到本地项目使用。

解决以上问题的便捷方法有两种，分别如下：

Npm link

Npm link 专门用于开发和调试本地 Npm 模块，能做到在不发布模块的情况下，把本地的一个正在开发的模块的源码链接到项目的 node_modules 目录下，让项目可以直接使用本

地的 Npm 模块。 由于是通过软链接的方式实现的，编辑了本地的 Npm 模块代码，在项目中也能使用到编辑后的代码。

完成 Npm link 的步骤如下：

1. 确保正在开发的本地 Npm 模块（也就是正在开发的 Loader）的 package.json 已经正确配置好；
2. 在本地 Npm 模块根目录下执行 npm link，把本地模块注册到全局；
3. 在项目根目录下执行 npm link loader-name，把第 2 步注册到全局的本地 Npm 模块链接到项目的 node_moduels 下，其中的 loader-name 是指在第 1 步中的 package.json 文件中配置的模块名称。

链接好 Loader 到项目后你就可以像使用一个真正的 Npm 模块一样使用本地的 Loader 了。

ResolveLoader

在其它配置项 中曾介绍过 ResolveLoader 用于配置 Webpack 如何寻找 Loader。 默认情况下只会去 node_modules 目录下寻找，为了让 Webpack 加载放在本地项目中的 Loader 需要修改 resolveLoader.modules。

假如本地的 Loader 在项目目录中的 ./loaders/loader-name 中，则需要如下配置：

```
module.exports = {
  resolveLoader:{
    // 去哪些目录下寻找 Loader，有先后顺序之分
    modules: ['node_modules','./loaders/'],
  }
}
```

加上以上配置后， Webpack 会先去 node_modules 项目下寻找 Loader，如果找不到，会再去 ./loaders/ 目录下寻找。