实战：手写一个 markdown-loader

更新时间：2019-07-08 14:12:13

> 古之立大事者，不唯有超世之才，亦必有坚韧不拔之志。
>
> ——苏轼

根据上面的工作流程描述，我们知道在 Webpack 中，真正起编译作用的便是我们的 loader，`loader` 实际就是处理单个模块的解析器（加载器不如解析器更好理解），平时我们进行 babel 的 ES6 编译，SCSS、LESS 等编译都是在 loader 里面完成的。loader 可以是异步的，也可以是同步的，同步的则直接返回处理后的模块内容，异步则调用异步回调函数输出处理后的模块内容。

实际上，loader 只是一个普通的 `funciton`，它会传入匹配到的文件内容（String 类型的 source），`loader` 做的是只需要对这些字符串做些处理就好了。本小节将从写一个 `markdown-loader` 入手，介绍 `loader` 编写中常见的问题和工具。

loader 的用法

首先回顾下 Webpack 中的 loader 有两种使用方式，然后是 loader 的链式调用和执行顺序。

webpack config 配置形式

```
// webpack.config.js
module.exports = {
    entry: './babel.js',
    mode: 'development',
    devtool: false,
    module: {
        rules: [
            {
                test: /\.js$/,
                use: [
                    {
                        loader: 'babel-loader',
                        options: {
                            presets: [
                                [
                                    '@babel/preset-env',
                                    {
                                        useBuiltIns: 'usage'
                                    }
                                ]
                            ]
                        }
                    }
                ]
            }
        ]
    }
};
```

内联 **inline** 写法

```
import utils from 'raw-loader!../../utils.js';
```

loader 的链式调用

loader 是支持链式调用的，例如下面两种形式的链式调用方法：

```
// webpack.config.js
module.exports = {
    //   ...
    module: {
        rules: [
            {
                test: /\.less$/,
                use: [
                    'style-loader',
                    {
                        loader: 'css-loader',
                        options: {
                            modules: true
                        }
                    },
                    'less-loader' // 将 Less 编译为 CSS
                ]
            }
        ]
    }
};
```

内联的链式调用写法：

```
import 'style-loader!css-loader!less-loader?a=b!../../common.less';
```

loader 的执行顺序

在之前的章节介绍过 loader 的执行顺序是**从后往前**或者**从右到左**，即从最后的 loader 开始往反方向执行，例如内联的链式写法：`import 'style-loader!css-loader!less-loader?a=b!../../common.less';` 实际解析顺序是 Webpack 会首先读取`common.less`的内容（source），然后交给`less-loader`编译处理，`less-loader`接收到的是`common.less`的 source（内容），同时还能够接收到`a=b`的 options（选项参数），处理结束后将内容返回或者扔个`callback`即到达第二个`css-loader`处理，`css-loader`处理结束后会扔个`style-loader`，以此类推。

loader 开发基础知识

在编写 loader 之前，我们需要先了解一些 loader 知识。

loader 本质是一个函数

loader 是本质上是一个函数，通过接受处理的内容，然后处理后返回结果。loader 的函数一般写法是：

```
module.exports = function(content, sourcemap) {
    // 处理 content 操作...
    return content;
};
```

在上面的示例中，我们看到 loader 实际是一个`funtion`，所以我们使用了 `return` 的方式返回 loader 处理后的数据。但其实这并**不是我们最推荐的写法**，在大多数情况下，我们还是更希望使用 `this.callback` 方法去返回数据。如果改成这种写法，示例代码可以改写为：

```
module.exports = function(content) {
    // 处理 content 操作...
    this.callback(null, content);
};
```

`this.callback` 可以传入四个参数（其中后两个参数可以省略），分别是：

- `error`：当 loader 出错时向外抛出一个 `Error`对象，成功则传入`null`；
- `content`：经过 loader 编译后需要导出的内容，类型可以是为`String`或者`Buffer`；
- `sourceMap`：为方便调试生成的编译后内容的 source map；
- `ast`：本次编译生成的 AST 静态语法树，之后执行的 loader 可以直接使用这个 AST，可以省去重复生成 AST 的过程。

> Tips：一定要注意，编写 loader 的时候，如果要使用`this.callback`或者后面提到的`loader-utils`的`getOptions`等方法，`this`是 webpack 调用 loader 时候传入的自定义的特殊上下文，所以这时候**不应该使用箭头函数**！

loader 异步处理数据

在上面的示例中，不论是使用 `return` 还是 `this.callback`，本质上都是同步的执行过程，假如我们的 loader 里存在异步操作，比如拉取异步的请求等又该怎么办呢？在 loader 中提供了两种异步写法。

第一种是使用`async/await`异步函数写法：

```
module.exports = async function(content) {
    function timeout(delay) {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                // 模拟一些异步操作处理 content
                resolve(content);
            }, delay);
        });
    }
    const data = await timeout(1000);
    return data;
};
```

还有一种方式是使用`this.async`方法获取一个异步的`callback`，然后返回它，上面的示例代码使用`this.async`修改如下：

```
module.exports = function(content) {
    function timeout(delay) {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                // 模拟一些异步操作处理 content
                resolve(content);
            }, delay);
        });
    }
    const callback = this.async();
    timeout(1000).then(data => {
        callback(null, data);
    });
};
```

> Tips：`this.async`获取的 callback，参数也是跟`this.callback`的参数一致，即 error，content，sourcemap 和 ast。

处理二进制数据

像 `file-loader` 这样的 Loader，实际处理的内容是二进制数据，那么就需要通过设置`moudle.exports.raw = true;`来告诉 Webpack 给 loader 传入二进制格式的数据，代码可以如下：

```
module.exports = function(source) {
    if (source instanceof Buffer) {
        // 一系列操作
        return source; // 本身也可以返回二进制数据提供给下一个loader
    }
};
moudle.exports.raw = true; // 不设置，就会拿到字符串
```

loader 的 pitch

前面一直在提醒大家 loader 的执行顺序是**从右到左**的链式调用。这种说法实际说的是 loader 中 `module.exports` 出来的执行方法顺序。在一些场景下，loader 并不依赖上一个 loader 的结果，而只关心原输入内容。这时候，要拿到一开始的文件原内容，就需要使用 `module.exports.pitch = function();`， `pitch` 方法在 loader 中便是从左到右执行的，并且可以通过 data 这个变量来进行 `pitch` 和 `normal` 之间的传递。例如下面的代码：

```
module.exports.pitch = function(remaining, preceding, data) {
    if (somothingFlag()) {
        return 'module.exports = require(' + JSON.stringify('-!' + remaining) + ');';
    }
    data.value = 1;
};
```

`pitch`它可以接受三个参数，最重要的就是第三个参数 `data`，你可以为其挂在一些所需的值，一个 rule 里的所有的 loader 在执行时都能拿到这个值：

```
module.exports = function(content) {
    // this.data test
    console.log('this data', this.data.value);
    return content;
};

module.exports.pitch = (remaining, preceding, data) => {
    data.value = 'test';
};
```

loader 的结果缓存

Webpack 增量编译机制会观察每次编译时的变更文件，在默认情况下，Webpack 会对 loader 的执行结果进行缓存，这样能够大幅度提升构建速度，不过我们也可以手动关闭它，示例代码如下：

```
module.exports = function(content) {
    // 关闭loader缓存
    this.cacheable(false);
    return content;
};
```

使用 Webpack 的 loader 工具库

我们在 `webpack.config.js` 书写 loader 配置时，经常会见到 `options` 这样一个配置项，或者在写内联调用 loader 的时候会通过`querystring`的形式传入 options，这就是 Webpack 为 loader 用户提供的自定义配置。在我们的 loader 里，可以拿到这些自定义配置。为了方便编写 loader，Webpack 官方将编写 loader 中常用的工具函数打包成了`loader-utils`和`schema-utils`模块，这里面包括了常用的获取 loader 选项（options）和参数验证等方法。

`loader-utils`工具库

`loader-utils`提供了各种跟 loader 选项（options）相关的工具函数，例如下面的代码：

```
const { getOptions, stringifyRequest, parseQuery  } = require('loader-utils');

module.exports = function(content) {
    // getOptions 用于在loader里获取传入的options，返回的是对象值。
    const options = getOptions(this);

    // stringifyRequest转换路径，避免require()或impot时使用的绝对路径
    stringifyRequest(this, './test.js'); // Result =>  "\"./test.js\""

    // parseQuery获取query参数的，这个很简单就不说啦
    parseQuery('?name=kev&age=14') // Result => {name: 'kev', age: '14'}
};
```

`schema-utils`工具库

`schema-utils`是 loader 和 plugin 的参数认证器，检测传入的参数是否符合预期，基本用法是：

```
const validateOptions = require('schema-utils');
// 下面是一个schema描述
const schema = {
    type: 'object',
    properties: {
        name: {
            type: 'string'
        },
        test: {
            anyOf: [{type: 'array'}, {type: 'string'}, {instanceof: 'RegExp'}]
        },
        transform: {
            instanceof: 'Function'
        },
        sourceMap: {
            type: 'boolean'
        }
    },
    additionalProperties: false
};
module.exports = function(source) {
    // 验证参数的类型是否正确。
    validateOptions(schema, options, 'loader name');
};
```

loader 中 this 相关的其它方法和属性

- `this.context`: 当前处理转换的文件所在的目录；
- `this.resource`: 当前处理转换的文件完整请求路径，包括 querystring；
- `this.resourcePath`: 当前处理转换的文件的路径；
- `this.resource Query`: 当前处理文件的 querystring；
- `this.target`: Webpack 配置的 target；
- `this.loadMoudle`: 处理文件时，需要依赖其它文件的处理结果时，可以使用 `this.loadMoudle(request: string, callback: function(err, source, sourceMap, module))` 去获取到依赖文件的处理结果；
- `this.resolve`: 获取指定文件的完整路径；
- `this.addDependency`: 为当前处理文件添加依赖文件，以便依赖文件发生变化时重新调用 Loader 转换该文件，`this.addDependency(file: string)`；
- `this.addContextDependency`: 为当前处理文件添加依赖文件目录，以便依赖文件目录里文件发生变化时重新调用 Loader 转换该文件，`this.addContextDependency(dir: string)`；
- `this.clearDependencies`: 清除当前正在处理文件的所有依赖；
- `this.emitFile`: 输出一个文件，使用的方法为 `this.emitFile(name: string, content: Buffer | string, sourceMap: {...})`；
- `this.emitError`：发送一个错误信息。

编写个 markdown loader

现在我们来手动写个 markdown-loader。Markdown-loader 是将 markdown 语法的文件转换成 HTML，这里使用的是showdown来转换 markdown 到 HTML。

首先创建一个`markdown-loader`文件夹，然后添加`package.json`：

```
mkdir markdown-loader && cd $_
npm init -y
```

安装依赖 showdown 和 loader-utils，这里需要添加到`dependencies`，所以用`--save`保存到`package.json`：

```
npm install showdown loader-utils --save
```

下面就是 markdown-loader 的`index.js`内容，几行就可以搞定一个 loader！

```
// index.js
// 引入依赖
const showdown = require('showdown');
const loaderUtils = require('loader-utils');

module.exports = function(content) {
    // 获取 options
    const options = loaderUtils.getOptions(this);
    // 设置 cache
    this.cacheable();
    // 初始化 showdown 转换器
    const converter = new showdown.Converter(options);
    // 处理 content
    content = converter.makeHtml(content);
    // 返回结果
    this.callback(null, content);
};
```

loader 写完了测试需要使用`require.resolve()`方法来使用本地的地址，现在创建一个`test`文件夹，添加`webpack.config.js`内容，loader中添加了`options`用于测试我们的markdown-loader能不能接收 options 参数：

```
module.exports = {
    entry: './index.js',
    module: {
        rules: [
            {
                test: /\.md$/,
                use: [
                    'html-loader',
                    {
                        loader: require.resolve('../index'),
                        options: {
                            simplifiedAutoLink: true,
                            tables: true
                        }
                    }
                ]
            }
        ]
    }
};
```

然后添加`markdown.md`和`index.js`

```
# 测试 markdown

测试自动options 生效，自动检测网址添加 link：www.google.com

## table

| h1    |   h2    |      h3 |
| :---- | :-----: | ------: |
| 100   | [a][1]  | ![b][2] |
| *foo* | **bar** | ~~baz~~ |
```

```
import html from './markdown.md';
console.log(html);
```

配置完成之后，安装`webpack`、`webpack-cli`和`html-loader`，然后执行`npx  webpack`，编译后`markdown.md`的内容被转义成了HTML：

```
<h1 id="markdown">测试 markdown</h1>
<p>测试自动options 生效，自动检测网址添加 link：<a href="http://www.google.com">www.google.com</a></p>
<h2 id="table">table</h2>
<table>
<thead>
<tr>
<th style="text-align:left;">h1</th>
<th style="text-align:center;">h2</th>
<th style="text-align:right;">h3</th>
</tr>
</thead>
<tbody>
<tr>
<td style="text-align:left;">100</td>
<td style="text-align:center;">[a][1]</td>
<td style="text-align:right;">![b][2]</td>
</tr>
<tr>
<td style="text-align:left;"><em>foo</em></td>
<td style="text-align:center;"><strong>bar</strong></td>
<td style="text-align:right;">~~baz~~</td>
</tr>
</tbody>
</table>
```

到此为止，我们就完成了一个简单的 loader：`markdown-loader` 了。

总结

本小节主要让大家了解 Webapck 的 loader 编写知识，loader 本质上是一个函数，这个函数主要接收的参数是需要处理的文件内容，经过异步和同步处理内容之后通过`callback`的方式传递给后面的 loader 或者 Webpack 直接使用。loader 的链式调用顺序是从右到左的，但是也有`pitch`函数可以做到从左到右，这样可以在一些特殊场景中获取原来的文件内容，通过`data`进行传值。最后介绍了 loader-utils和 schema-utils两个在编写 loader 中常用的工具函数，介绍完这些loader 开发的基础知识之后，小节最后使用了`showdown`来编写了一个 markdown-loader，通过实际代码帮助大家理解 loader 知识。

> 本小节 Webpack 相关面试题：
>
> 1. 编写过 Webpack loader吗？

实战：使用 PostCSS 打造移动适配方案

实战：手写一个 prefetch-webpack-plugin 插件