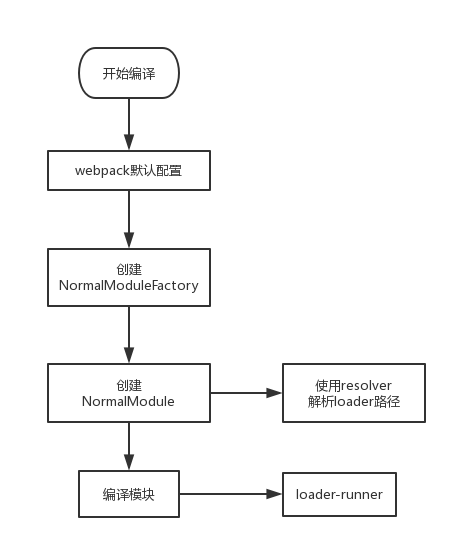
## **1.loader运行的总体流程**

1. Compiler.js中会将用户配置与默认配置合并，其中就包括了loader部分
2. webpack就会根据配置创建两个关键的对象——NormalModuleFactory和ContextModuleFactory。它们相当于是两个类工厂，通过其可以创建相应的NormalModule和ContextModule
3. 在工厂创建NormalModule实例之前还要通过loader的resolver来解析loader路径
4. 在NormalModule实例创建之后，则会通过其.build()方法来进行模块的构建。构建模块的第一步就是使用loader来加载并处理模块内容。而loader-runner这个库就是webpack中loader的运行器
5. 最后，将loader处理完的模块内容输出，进入后续的编译流程



## **2. loader配置**

loader是导出为一个函数的node模块。该函数在loader转换资源的时候调用。给定的函数将调用loader API，并通过this上下文访问。

### **2.1 匹配(test)单个 loader**

匹配(test)单个 loader，你可以简单通过在 rule 对象设置 path.resolve 指向这个本地文件

{

test: /\.js$/

use: [

{

loader: path.resolve('path/to/loader.js'),

options: {/\* ... \*/}

}

]

}

### **2.2 匹配(test)多个 loaders**

，你可以使用 resolveLoader.modules 配置，webpack 将会从这些目录中搜索这些 loaders。

resolveLoader: {

modules: [path.resolve('node\_modules'), path.resolve(\_\_dirname, 'src', 'loaders')]

},

### **2.3 npm link**

* 第1步: 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
* 第2步: 在本地npm模块根目录下执行npm link,把本地模块注册到全局
* 第3步: 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\_modules目录 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

npm link

### **2.4 alias**

resolveLoader: {

alias: {

"babel-loader": resolve('./loaders/babel-loader.js'),

"css-loader": resolve('./loaders/css-loader.js'),

"style-loader": resolve('./loaders/style-loader.js'),

"file-loader": resolve('./loaders/file-loader.js'),

"url-loader": resolve('./loaders/url-loader.js')

}

},

## **3. loader用法**

### **3.1 单个loader用法**

* loader只能传入一个包含包含资源文件内容的字符串
* 同步 loader 可以简单的返回一个代表模块转化后的值
* loader 也可以通过使用 this.callback(err, values...) 函数，返回任意数量的值
* loader 会返回一个或者两个值。第一个值的类型是 JavaScript 代码的字符串或者 buffer。第二个参数值是 SourceMap,它是个 JavaScript 对象

### **3.2 多个loader**

* 当链式调用多个 loader 的时候，请记住它们会以相反的顺序执行。取决于数组写法格式，从右向左或者从下向上执行。
* 最后的 loader 最早调用，将会传入原始资源内容。
* 第一个 loader 最后调用，期望值是传出 JavaScript 和 source map(可选)
* 中间的 loader 执行时，会传入前一个 loader 传出的结果。

## **4 用法准则**

### **4.1 简单**

* loaders 应该只做单一任务。这不仅使每个 loader 易维护，也可以在更多场景链式调用。

### **4.2 链式(Chaining)**

* 利用 loader 可以链式调用的优势。写五个简单的 loader 实现五项任务，而不是一个 loader 实现五项任务

### **4.3 模块化(Modular)**

保证输出模块化。loader 生成的模块与普通模块遵循相同的设计原则。

### **4.4 无状态(Stateless)**

确保 loader 在不同模块转换之间不保存状态。每次运行都应该独立于其他编译模块以及相同模块之前的编译结果。

### **4.5 loader工具库**

[loader-utils](https://github.com/webpack/loader-utils) 包。它提供了许多有用的工具，但最常用的一种工具是获取传递给 loader 的选项

[schema-utils](https://github.com/webpack-contrib/schema-utils) 包配合 loader-utils，用于保证 loader 选项，进行与 JSON Schema 结构一致的校验

### **4.6 loader依赖**

如果一个 loader 使用外部资源（例如，从文件系统读取），必须声明它。这些信息用于使缓存 loaders 无效，以及在观察模式(watch mode)下重编译。

### **4.7 模块依赖**

根据模块类型，可能会有不同的模式指定依赖关系。例如在 CSS 中，使用 @import 和 url(...) 语句来声明依赖。这些依赖关系应该由模块系统解析。

### **4.8 绝对路径**

* 不要在模块代码中插入绝对路径，因为当项目根路径变化时，文件绝对路径也会变化。
* loader-utils 中的 stringifyRequest 方法，可以将绝对路径转化为相对路径。

### **4.9 同等依赖**

* 如果你的 loader 简单包裹另外一个包，你应该把这个包作为一个 peerDependency 引入。
* 这种方式允许应用程序开发者在必要情况下，在package.json 中指定所需的确定版本。

## **5. API**

### **5.1 缓存结果**

* 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。

module.exports = **function**(source) {

// 关闭该 Loader 的缓存功能

**this**.cacheable(false);

**return** source;

};

### **5.2 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

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);

});

};

### **5.3 返回其它结果**

Loader有些场景下还需要返回除了内容之外的东西。

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**;

};

完整格式

**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

);

### **5.4 raw loader**

* 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;

### **5.5 获得options**

* 可以获得给Loader配置的options

**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;

};

### **5.6 其它 Loader API**

* [完整API](https://webpack.js.org/api/loaders/)

| **方法名** | **含义** |
| --- | --- |
| 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: {...}) |
| loader-utils.stringifyRequest | Turns a request into a string that can be used inside require() or import while avoiding absolute paths. Use it instead of JSON.stringify(...) if you're generating code inside a loader 把一个请求字符串转成一个字符串，以便能在require或者import中使用以避免绝对路径。如果你在一个loader中生成代码的话请使用这个而不要用JSON.stringify() |
| loader-utils.interpolateName | Interpolates a filename template using multiple placeholders and/or a regular expression. The template and regular expression are set as query params called name and regExp on the current loader's context. 使用多个占位符或一个正则表达式转换一个文件名的模块。这个模板和正则表达式被设置为查询参数，在当前loader的上下文中被称为name或者regExp |

## **6.loader实战**

### **6.1 项目准备**

cnpm i webpack webpack-cli webpack-dev-server html-webpack-plugin @babel/core @babel/preset-env babel-loader css-loader file-loader less less-loader style-loader url-loader -D

#### **6.1.1 webpack.config.js**

**const** path = require('path');**const** HtmlWebpackPlugin = require('html-webpack-plugin');module.exports = {

mode: 'development',

context: process.cwd(),

devtool: 'source-map',

entry: "./src/index.js",

output: {

path: path.resolve(\_\_dirname, "dist"),

filename: "bundle.js"

},

resolveLoader: {

modules: [path.resolve('./loaders'), 'node\_modules']

},

devServer: {

contentBase: path.resolve(\_\_dirname, 'dist')

},

module: {

rules: [

{

test: /\.jsx?$/,

use: {

loader: "babel-loader"

},

include: path.join(\_\_dirname, "src"),

exclude: /node\_modules/

}

]

},

plugins: [

**new** HtmlWebpackPlugin({

template: './src/index.html'

})

]

};

#### **6.1.2 src\index.js**

src\index.js

**const** sum = (a, b) => {

**return** a + b;

}console.log(sum(1, 2));

#### **6.1.3 src\index.html**

src\index.html

<!DOCTYPE html><html lang="en"><head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title></head><body>

<div id="root"></div></body></html>

### **6.2 babel-loader**

* [babel-loader](https://github.com/babel/babel-loader/blob/master/src/index.js)
* [@babel/core](https://babeljs.io/docs/en/next/babel-core.html)

| **属性** | **值** |
| --- | --- |
| this.request | /loaders/babel-loader.js!/src/index.js' |
| this.userRequest | /src/index.js |
| this.rawRequest | ./src/index.js |
| this.resourcePath | /src/index.js |

**const** babel = require("@babel/core");**function** **loader**(source) {

//C:\vipdata\prepare6\zhufeng\_loader\loaders\babel-loader.js!C:\vipdata\prepare6\zhufeng\_loader\src\index.js

console.log('this.request', **this**.request);

**const** options = {

presets: ['@babel/preset-env'],

//sourceMaps: true 如果没有此参数，则不会生成sourcemap文件

//注意此处的sourceMap是从转译前的代码到转译后的代码的map映射,如果此处不生成无法调试真正的源码了

sourceMaps: true,

//如果没有此参数则生成的map文件里文件中unknown

filename: **this**.request.split('!')[1].split('/').pop()

}

//在webpack.config.js中 增加devtool: 'source-map',这个source-map是从转译后的源码到bundle.js之间的map映射

**let** { code, map, ast } = babel.transform(source, options);

**return** **this**.callback(null, code, map, ast);

}

module.exports = loader;

本案例是是学习如何编写loader以及如何通过this.callback返回多个值

### **6.3 banner-loader**

* 本案例学习如何获取参数，验证参数，实现异步loader以及使用缓存

#### **6.3.1 banner-loader.js**

loaders\banner-loader.js

**const** loaderUtils = require('loader-utils');**const** validateOptions = require('schema-utils');**const** fs = require('fs');**function** **loader**(source) {

//把loader改为异步,任务完成后需要手工执行callback

**let** cb = **this**.async();

//启用loader缓存

**this**.cacheable && **this**.cacheable();

//用来验证options的合法性

**let** schema = {

type: 'object',

properties: {

filename: {

type: 'string'

},

text: {

type: 'string'

}

}

}

//通过工具方法获取options

**let** options = loaderUtils.getOptions(**this**);

//用来验证options的合法性

validateOptions(schema, options);

**let** {filename } = options;

fs.readFile(filename, 'utf8', (err, text) => {

cb(err, text + source);

});

}

module.exports = loader;

#### **6.3.2 banner.js**

loaders\banner.js

/\*\*copyright: zhufengjiagu\*/

#### **6.3.3 webpack.config.js**

webpack.config.js

module: {

rules: [

{

test: /\.jsx?$/,+ use: [{+ loader: 'banner-loader',+ options: { filename: path.resolve(\_\_dirname, 'loaders/banner.js') }+ }, {

loader: "babel-loader"

}],

include: path.join(\_\_dirname, "src"),

exclude: /node\_modules/

}

]

},

### **6.4 file**

* file-loader 并不会对文件内容进行任何转换，只是复制一份文件内容，并根据配置为他生成一个唯一的文件名。
* 主案例主要是学习loader-utils用法

#### **6.4.1 file-loader**

* [loader-utils](https://github.com/webpack/loader-utils)
* [file-loader](https://github.com/webpack-contrib/file-loader/blob/master/src/index.js)
* [public-path](https://webpack.js.org/guides/public-path/" \l "on-the-fly)

**const** { getOptions, interpolateName } = require('loader-utils');**function** **loader**(content) {

**let** options=getOptions(**this**)||{};

**let** url = interpolateName(**this**, options.filename || "[hash].[ext]", {content});

**this**.emitFile(url, content);

**return** `module.exports = ${JSON.stringify(url)}`;

}

loader.raw = true;module.exports = loader;

* 通过 loaderUtils.interpolateName 方法可以根据 options.name 以及文件内容生成一个唯一的文件名 url（一般配置都会带上hash，否则很可能由于文件重名而冲突）
* 通过 this.emitFile(url, content) 告诉 webpack 我需要创建一个文件，webpack会根据参数创建对应的文件，放在 public path 目录下
* 返回 module.exports = ${JSON.stringify(url)},这样就会把原来的文件路径替换为编译后的路径

#### **6.4.2 url-loader**

**let** { getOptions } = require('loader-utils');**var** mime = require('mime');**function** **loader**(source) {

**let** options=getOptions(**this**)||{};

**let** { limit, fallback='file-loader' } = options;

**if** (limit) {

limit = parseInt(limit, 10);

}

**const** mimeType=mime.getType(**this**.resourcePath);

**if** (!limit || source.length < limit) {

**let** base64 = `data:${mimeType};base64,${source.toString('base64')}`;

**return** `module.exports = ${JSON.stringify(base64)}`;

} **else** {

**let** fileLoader = require(fallback || 'file-loader');

**return** fileLoader.call(**this**, source);

}

}

loader.raw = true;module.exports = loader;

### **6.5 pitch**

* 以a!b!c!module为例,正常调用顺序应该是c=>b=>a,但是真正调用顺序是 a(pitch)=>b(pitch)=>c(pitch)=>c=>b=>a,如果其中任何一个pitching loader返回了值就相当于在它以及它右边的loader已经执行完毕
* 比如如果b返回了字符串result\_b, 接下来只有a会被系统执行，且a的loader收到的参数是result\_b
* loader根据返回值分为两种，一种是返回js代码(一个module的代码，含有类似module.export语句)的loader,另一种是不能作为最左边loader的其他loader
* 有时候我们想把两个第一种loader 连接起来，比如style-loader!css-loader! 问题是css-loader的返回值是一串js代码，如果按正常方式写style-loader的参数就是一串代码字符串
* 为了解决这种问题，我们需要在style-loader里执行require(css-loader!resources)

pitch与loader本身方法的执行顺序图

|- a-loader `pitch`

|- b-loader `pitch`

|- c-loader `pitch`

|- requested module is picked up **as** a dependency

|- c-loader normal execution

|- b-loader normal execution

|- a-loader normal execution

#### **6.5.1 loaders\loader1.js**

loaders\loader1.js

**function** **loader**(source) {

console.log('loader1',**this**.data);

**return** source+"//loader1";

}

loader.pitch = **function** (remainingRequest,previousRequest,data) {

data.name = 'pitch1';

console.log('pitch1');

}module.exports = loader;

#### **6.5.2 loaders\loader2.js**

loaders\loader2.js

**function** **loader**(source) {

console.log('loader2');

**return** source+"//loader2";

}

loader.pitch = **function** (remainingRequest,previousRequest,data) {

console.log('remainingRequest=',remainingRequest);

console.log('previousRequest=',previousRequest);

console.log('pitch2');

//return 'console.log("pitch2")';

}module.exports = loader;

#### **6.5.3 loaders\loader3.js**

loaders\loader3.js

**function** **loader**(source) {

console.log('loader3');

**return** source+"//loader3";

}

loader.pitch = **function** () {

console.log('pitch3');

}module.exports = loader;

#### **6.5.4 webpack.config.js**

{

test: /\.js$/,

use: ['loader1', 'loader2', 'loader3']

}

### **6.6 run-loader**

* [LoaderRunner](https://github.com/webpack/loader-runner/blob/v2.4.0/lib/LoaderRunner.js)
* [NormalModuleFactory-noPreAutoLoaders](https://github.com/webpack/webpack/blob/v4.39.3/lib/NormalModuleFactory.js" \l "L180)
* [NormalModule-runLoaders](https://github.com/webpack/webpack/blob/v4.39.3/lib/NormalModule.js" \l "L292)

run-loader.js

//@ts-check**const** path = require('path');**const** fs = require('fs');**const** readFile = fs.readFileSync;//入口文件**let** entry = './src/title.js';**let** options = {

resource: path.join(\_\_dirname, entry),//要加载的资源,它是个绝对路径

loaders: [

path.join(\_\_dirname, 'loaders/a-loader.js'),//C:\vipdata\lesson\201911projects\7.loader\loaders\a-loader.js

path.join(\_\_dirname, 'loaders/b-loader.js'),//C:\vipdata\lesson\201911projects\7.loader\loaders\b-loader.js

path.join(\_\_dirname, 'loaders/c-loader.js')//C:\vipdata\lesson\201911projects\7.loader\loaders\c-loader.js

]

}**function** **createLoaderObject**(loaderPath) {

**let** loaderObject = { data: {} };

loaderObject.path = loaderPath;// 存放此loader的绝地路径

loaderObject.normal = require(loaderPath);

loaderObject.pitch = loaderObject.normal.pitch;

**return** loaderObject;

}

**function** **runLoaders**(options, finalCallback) {

**let** loaderContext = {};//这个对象将会成为webpack loader中的this

**let** resource = options.resource;//获取要加载的资源 C:\vipdata\lesson\201911projects\7.loader\src\title.js

**let** loaders = options.loaders;

loaders = loaders.map(createLoaderObject);

loaderContext.loaderIndex = 0;//当前正在执行的loader的索引

loaderContext.readResource = readFile;//指定一个读取文件的方法 readFile

loaderContext.resource = resource;//resource放置着要读取的资源

loaderContext.loaders = loaders;//缓存了loaderObject的数组

Object.defineProperty(loaderContext, 'request', {

**get**() {

**return** loaderContext.loaders.map(loaderObject => loaderObject.path)

.concat(loaderContext.resource).join('!')

}

});

Object.defineProperty(loaderContext, 'previousRequest', {

**get**() {

**return** loaderContext.loaders.slice(0, loaderContext.loaderIndex).map(loaderObject => loaderObject.path)

.join('!')

}

});

Object.defineProperty(loaderContext, 'remainingRequest', {

**get**() {

**return** loaderContext.loaders.slice(loaderContext.loaderIndex + 1).map(loaderObject => loaderObject.path)

.concat(loaderContext.resource).join('!')

}

});

Object.defineProperty(loaderContext, 'data', {

**get**() {

**return** loaderContext.loaders[loaderContext.loaderIndex].data;

}

});

iteratePitchingLoaders(loaderContext, finalCallback);

//在这里要读取要加载的模块的内容

**function** **processResource**(loaderContext, finalCallback) {

**let** buffer = loaderContext.readResource(loaderContext.resource);// 读取文件的内容

iterateNormalLoaders(loaderContext, buffer, finalCallback);//进行正掌中的loader执行

}

**function** **convertArgs**(args, raw) {

**if** (!raw && Buffer.isBuffer(args))

args = args.toString("utf8");

**else** **if** (raw && **typeof** args === "string")

args = **new** Buffer(args, "utf8"); // eslint-disable-line

}

//执行正常的

**function** **iterateNormalLoaders**(loaderContext, args, finalCallback) {

**if** (loaderContext.loaderIndex < 0) {

**return** finalCallback(null, args);

}

**let** currentLoaderObject = loaderContext.loaders[loaderContext.loaderIndex];

**let** normalFn = currentLoaderObject.normal;//拿到当前的normal函数

**let** isSync = true;//默认当前的执行模式是同步模式

**const** innerCallback = loaderContext.callback = (err, args) => {

loaderContext.loaderIndex--;

iterateNormalLoaders(loaderContext, args, finalCallback)

}

loaderContext.async = () => {//如果调了它，会把同步变成异步

isSync = false;

**return** innerCallback;

}

args = convertArgs(args, normalFn.raw)

args = normalFn.call(loaderContext, args);

**if** (isSync) {//如果是同步的话。

loaderContext.loaderIndex--;

iterateNormalLoaders(loaderContext, args, finalCallback);

} **else** {

//如果是异步，就不会再递归迭代了

}

}

**function** **iteratePitchingLoaders**(loaderContext, finalCallback) {

**if** (loaderContext.loaderIndex >= loaderContext.loaders.length) {

loaderContext.loaderIndex--;//因为已经越界了,所以让索引减1

**return** processResource(loaderContext, finalCallback);

}

**let** currentLoaderObject = loaderContext.loaders[loaderContext.loaderIndex];//0 获取当前的loaderObject

**let** pitchFn = currentLoaderObject.pitch;

**if** (!pitchFn) {//如果没有pitch函数，则直接 跳过当前loader,执行下一个loader

loaderContext.loaderIndex++;

**return** iteratePitchingLoaders(loaderContext, finalCallback);

}

**let** args = pitchFn.call(loaderContext, loaderContext.remainingRequest, loaderContext.previousRequest, loaderContext.data);

**if** (args) {//如果有返回值，则跳过当前和后面的normal函数以及读取模块文件的逻辑 ，直接把结果返回给前一个loader的normal参数

loaderContext.loaderIndex--;

**return** iterateNormalLoaders(loaderContext, args, finalCallback);

} **else** {

loaderContext.loaderIndex++;

**return** iteratePitchingLoaders(loaderContext, finalCallback);

}

}

}console.time('cost');

runLoaders(options, (err, result) => {

// title//c-loader//b-loader//a-loader

console.log('经过loader编译后的结果', result);

console.timeEnd('cost');

});

### **6.7 样式处理**

* [css-loader](https://github.com/webpack-contrib/css-loader/blob/master/lib/loader.js) 的作用是处理css中的 @import 和 url 这样的外部资源
* [style-loader](https://github.com/webpack-contrib/style-loader/blob/master/index.js) 的作用是把样式插入到 DOM中，方法是在head中插入一个style标签，并把样式写入到这个标签的 innerHTML里
* [less-loader](https://github.com/webpack-contrib/less-loader) 把less编译成css
* [pitching-loader](https://webpack.js.org/api/loaders/" \l "pitching-loader)
* [loader-utils](https://github.com/webpack/loader-utils)
* [!!](https://webpack.js.org/concepts/loaders/" \l "configuration)

#### **6.7.1 loader类型**

* [loader的叠加顺序](https://github.com/webpack/webpack/blob/v4.39.3/lib/NormalModuleFactory.js" \l "L339) = post(后置)+inline(内联)+normal(正常)+pre(前置)

**let** result = [useLoadersPost,useLoaders,useLoadersPre];

loaders = results[0].concat(loaders, results[1], results[2]);

useLoadersPost+inlineLoader+useLoaders(normal loader)+useLoadersPre

[Configuration](https://webpack.js.org/concepts/loaders/" \l "configuration)

| **符号** | **变量** | **含义** |  |
| --- | --- | --- | --- |
| -! | noPreAutoLoaders | 不要前置和普通loader | Prefixing with -! will disable all configured preLoaders and loaders but not postLoaders |
| ! | noAutoLoaders | 不要普通loader | Prefixing with ! will disable all configured normal loaders |
| !! | noPrePostAutoLoaders | 不要前后置和普通loader,只要内联loader | Prefixing with !! will disable all configured loaders (preLoaders, loaders, postLoaders) |

#### **6.7.2 使用less-loader**

##### **6.7.2.1 index.js**

src\index.js

**import** './index.less';

##### **6.7.2.2 src\index.less**

src\index.less

@color:red;#root{

color:@color;

}

##### **6.7.2.3 src\index.html**

src\index.html

<div id="root">hello</div>

##### **6.7.2.4 webpack.config.js**

webpack.config.js

{

test: /\.less$/,

use: [

'style-loader',

'less-loader'

]

}

#### **6.7.3 less-loader.js**

**let** less = require('less');**function** **loader**(source) {

**let** callback = **this**.async();

less.render(source, { filename: **this**.resource }, (err, output) => {

callback(err, output.css);

});

}module.exports = loader;

#### **6.7.4 style-loader**

**function** **loader**(source) {

**let** script=(`

let style = document.createElement("style");

style.innerHTML = ${JSON.stringify(source)};

document.head.appendChild(style);

module.exports = "";

`);

**return** script;

} module.exports = loader;

#### **6.7.5 两个左侧模块连用**

##### **6.7.5.1 less-loader.js**

**let** less = require('less');**function** **loader**(source) {

**let** callback = **this**.async();

less.render(source, { filename: **this**.resource }, (err, output) => {

callback(err, `module.exports = ${JSON.stringify(output.css)}`);

});

}module.exports = loader;

##### **6.7.5.2 style-loader.js**

**let** loaderUtils = require("loader-utils");**function** **loader**(source) {

}//https://github.com/webpack/webpack/blob/v4.39.3/lib/NormalModuleFactory.js#L339

loader.pitch = **function** (remainingRequest, previousRequest, data) {

//C:\webpack-analysis2\loaders\less-loader.js!C:\webpack-analysis2\src\index.less

console.log('previousRequest', previousRequest);//之前的路径

//console.log('currentRequest', currentRequest);//当前的路径

console.log('remainingRequest', remainingRequest);//剩下的路径

console.log('data', data);

// !! noPrePostAutoLoaders 不要前后置和普通loader

//\_\_webpack\_require\_\_(/\*! !../loaders/less-loader.js!./index.less \*/ "./loaders/less-loader.js!./src/index.less");

**let** style = `

var style = document.createElement("style");

style.innerHTML = require(${loaderUtils.stringifyRequest(**this**, "!!" + remainingRequest)});

document.head.appendChild(style);

`;

**return** style;

}module.exports = loader;

#### **6.7.6 css-loader.js**

* css-loader 的作用是处理css中的 @import 和 url 这样的外部资源
* [postcss](https://github.com/postcss/postcss" \l "usage)
* Avoid CSS @import CSS @importallows stylesheets to import other stylesheets. When CSS @import isused from an external stylesheet, the browser is unable to downloadthe stylesheets in parallel, which adds additional round-trip timesto the overall page load.

##### **6.7.6.1 src\index.js**

src\index.js

require('./style.css');

##### **6.7.6.2 src\style.css**

**@import** './global.css';.avatar {

width: 100px;

height: 100px;

background-image: url('./baidu.png');

background-size: cover;

}div{

color:red;

}

##### **6.7.6.3 src\global.css**

body {

background-color: green;

}

##### **6.7.6.4 webpack.config.js**

+ {+ test: /\.css$/,+ use: [+ 'style-loader',+ 'css-loader'+ ]+ },+ {+ test: /\.png$/,+ use: [+ 'file-loader'+ ]+ }

##### **6.7.6.5 css-loader.js**

loaders\css-loader.js

**var** postcss = require("postcss");**var** loaderUtils = require("loader-utils");**var** Tokenizer = require("css-selector-tokenizer");

**const** cssLoader = **function** (inputSource) {

**const** cssPlugin = (options) => {

**return** (root) => {

root.walkAtRules(/^import$/i, (rule) => {

rule.remove();

options.imports.push(rule.params.slice(1, -1));

});

root.walkDecls((decl) => {

**var** values = Tokenizer.parseValues(decl.value);

values.nodes.forEach(**function** (value) {

value.nodes.forEach(item => {

**if** (item.type === "url") {

item.url = "`+require(" + loaderUtils.stringifyRequest(**this**, "!!" + item.url) + ")+`";

}

});

});

decl.value = Tokenizer.stringifyValues(values);

});

};

}

**let** callback = **this**.async();

**let** options = { imports: [] };

**let** pipeline = postcss([cssPlugin(options)]);

pipeline.process(inputSource).then((result) => {

**let** importCss = options.imports.map(url => "`+require(" + loaderUtils.stringifyRequest(**this**, "!!css-loader!" + url) + ")+`").join('\r\n');

callback(

null,

'module.exports=`' + importCss + '\r\n' + result.css + '`'

);

});

};

module.exports = cssLoader;

## **参考**

### **1. PostCSS**

* [PostCSS](https://www.postcss.com.cn/)是一个用 JavaScript 工具和插件转换 CSS 代码的工具
* 增强代码的可读性
* 将未来的 CSS 特性带到今天！
* 终结全局 CSS
* 避免 CSS 代码中的错误
* 强大的网格系统

### **2. 文档**

* [api](http://api.postcss.org/)
* [astexplorer](https://astexplorer.net/" \l "/2uBU1BLuJ1) postcss会帮我们分析出css的抽象语法树

### **3. 类型**

* CSS AST主要有3种父类型
  + AtRule @xxx的这种类型，如@screen
  + Comment 注释
  + Rule 普通的css规则
* 子类型
  + decl 指的是每条具体的css规则
  + rule 作用于某个选择器上的css规则集合

### **4.AST节点**

* nodes: CSS规则的节点信息集合
  + decl: 每条css规则的节点信息
  + prop: 样式名,如width
  + value: 样式值,如10px
* type: 类型
* source: 包括start和end的位置信息，start和end里都有line和column表示行和列
* selector: type为rule时的选择器
* name: type为atRule时@紧接rule名，譬如@import 'xxx.css'中的import
* params: type为atRule时@紧接rule名后的值，譬如@import 'xxx.css'中的xxx.css
* text: type为comment时的注释内容

### **5.操作方法**

#### **5.1 遍历**

* walk: 遍历所有节点信息，无论是atRule、rule、comment的父类型，还是rule、 decl的子类型
* walkAtRules：遍历所有的AtRules
* walkComments 遍历所有的Comments
* walkDecls 遍历所有的Decls
* walkRules 遍历所有的Rules

root.walkDecls(decl => {

decl.prop = decl.prop.split('').reverse().join('');

});

#### **5.2 处理**

* postCss给出了很多操作css规则的方法
* [api](http://api.postcss.org/AtRule.html)
* 处理css的方式其实有2种：编写postcss plugin，如果你的操作非常简单也可以直接利用postcss.parse方法拿到css ast后分析处理

#### **5.3 postcss plugin**

* postcss插件如同babel插件一样，有固定的格式
* 注册个插件名，并获取插件配置参数opts
* 返回值是个函数，这个函数主体是你的处理逻辑，有2个参数，一个是root,AST的根节点。另一个是result，返回结果对象，譬如result.css，获得处理结果的css字符串

**export** **default** postcss.plugin('postcss-plugin-name', **function** (opts) {

opts = opts || {};

**return** **function** (root, result) {

// 处理逻辑

};

});

### **5.4 直接调用postcss命名空间下的方法**

* 可以用postcss.parse来处理一段css文本，拿到css ast，然后进行处理，再通过调用toResult().css拿到处理后的css输出