自定义Loader

创建自己的Loader

- Loader是用于对模块的源代码进行转换（处理），之前我们已经使用过很多Loader，比如css-loader、style-loader、babel-loader等。
- 这里我们来学习如何自定义自己的Loader：
  - Loader本质上是一个导出为函数的JavaScript模块；
  - loader runner库会调用这个函数，然后将上一个loader产生的结果或者资源文件传入进去；
- 编写一个hy-loader01.js模块这个函数会接收三个参数：
  - content：资源文件的内容；
  - map：sourcemap相关的数据；
  - meta：一些元数据；

```
module.exports = function(content, map, meta) {
  console.log(content);
  return content;
}
```

在加载某个模块时，引入loader

- 注意：传入的路径和context是有关系的，在前面我们讲入口的相对路径时有讲过。

```
const path = require('path');

module.exports = {
  mode: "development",
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "./build"),
    filename: "bundle.js"
  },
  module: {
    rules: [
      {
        test: /\.js$/i,
        use: [
          "./loaders/hy-loader01.js",
        ]
      }
    ]
  }
}
```

resolveLoader属性

- 但是，如果我们依然希望可以直接去加载自己的loader文件夹，有没有更加简洁的办法呢？
  - 配置resolveLoader属性；

```
resolveLoader: {
  modules: ["./loaders", "node_modules"],
},
```

loader的执行顺序

- 创建多个Loader使用，它的执行顺序是什么呢？
  - 从后向前、从右向左的

```
// hy-loader01.js
module.exports = function(content) {
  console.log("loader01", content);
  return content;
}

// hy-loader02.js
module.exports = function(content) {
  console.log("loader02", content);
  return content;
}

// hy-loader03.js
module.exports = function(content) {
  console.log("loader03", content);
  return content;
}
```

```
rules: [
  {
    test: /\.js$/i,
    use: [
      "hy-loader01",
      "hy-loader02",
      "hy-loader03",
    ]
  }
]
```

pitch-loader和enforce

- 事实上还有另一种Loader，称之为PitchLoader：

```
module.exports.pitch = function() {
  console.log("loader01 pitch");
}
```

执行顺序和enforce

- 其实这也是为什么loader的执行顺序是相反的：
  - run-loader先优先执行PitchLoader，在执行PitchLoader时进行loaderIndex++；
  - run-loader之后会执行NormalLoader，在执行NormalLoader时进行loaderIndex--；
- 那么，能不能改变它们的执行顺序呢？
  - 我们可以拆分成多个Rule对象，通过enforce来改变它们的顺序；
- enforce一共有四种方式：
  - 默认所有的loader都是normal；
  - 在行内设置的loader是inline（在前面将css加载时讲过，import 'loader1!loader2!./test.js'）；
  - 也可以通过enforce设置 pre 和 post；
- 在**Pitching**和**Normal**它们的执行顺序分别是：
  - post, inline, normal, pre；
  - pre, normal, inline, post；

同步的Loader

- 什么是同步的Loader呢？
  - 默认创建的Loader就是同步的Loader；
  - 这个Loader必须通过 return 或者 this.callback 来返回结果，交给下一个loader来处理；
  - 通常在有错误的情况下，我们会使用 this.callback；
- this.callback的用法如下：
  - 第一个参数必须是 Error 或者 null；
  - 第二个参数是一个 string或者Buffer；

```
module.exports = function(content) {
  console.log("loader01", content);
  this.callback(null, content);
}
```

异步的Loader

- 什么是异步的Loader呢？
  - 有时候我们使用Loader时会进行一些异步的操作；
  - 我们希望在异步操作完成后，再返回这个loader处理的结果；
  - 这个时候我们就要使用异步的Loader了；
- loader-runner已经在执行loader时给我们提供了方法，让loader变成一个异步的loader：

```
module.exports = function(content) {
  const callback = this.async();

  setTimeout(() => {
    console.log("loader01", content);
    callback(null, content);
  }, 1000);
}
```

传入和获取参数

- 在使用loader时，传入参数。
- 我们可以通过一个webpack官方提供的一个解析库 loader-utils，安装对应的库。

```
npm install loader-utils -D
```

```
{
  test: /\.js$/i,
  use: {
    loader: "hy-loader01",
    options: {
      name: "why",
      age: 18
    }
  },
}
```

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

module.exports = function(content) {
  // 设置为异步的loader
  const callback = this.async();

  // 获取参数
  const options = getOptions(this);

  setTimeout(() => {
    console.log("loader01", content, options);
    callback(null, content);
  }, 1000);
}
```

校验参数

- 我们可以通过一个webpack官方提供的校验库 schema-utils，安装对应的库：

```
npm install schema-utils -D
```

```json
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "description": "请填入你的名称"
    },
    "age": {
      "type": "number",
      "description": "请填入你的年龄"
    }
  },
  "additionalProperties": true
}
```

```js
const { getOptions } = require('loader-utils');
const { validate } = require("schema-utils")
const loader01Schema = require('../schema/loader01_schema.json');

module.exports = function(content) {
  // 设置为异步的loader
  const callback = this.async();

  // 获取参数
  const options = getOptions(this);

  // 参数校验
  validate(loader01Schema, options);

  setTimeout(() => {
    console.log("loader01", content, options);
    callback(null, content);
  }, 1000);
}
```

babel-loader案例

- 我们知道babel-loader可以帮助我们对JavaScript的代码进行转换，这里我们定义一个自己的babel-loader：

```
const babel = require("@babel/core");
const { getOptions } = require("loader-utils");
const { validate } = require("schema-utils");
const babelSchema = require("../schema/babel_schema.json");

module.exports = function(content) {
  const callback = this.async();

  const options = getOptions(this);
  validate(babelSchema, options);

  babel.transform(content, options, (err, result) => {
    if (err) {
      callback(err);
    } else {
      callback(null, result.code);
    }
  })
}
```

```
{
  "type": "object",
  "properties": {
    "presets": {
      "type": "array"
    }
  },
  "additinalProperties": true
}
```

hymd-loader

```
const marked = require('marked');
const hljs = require('highlight.js');

module.exports = function(content) {
  marked.setOptions({
    highlight: function(code, lang) {
      return hljs.highlight(lang, code).value;
    }
  });

  const htmlContent = marked(content);
  const innerContent = "`" + htmlContent + "`";
  const moduleCode = `var code=${innerContent}; export default code;`

  console.log(moduleCode);

  return moduleCode;
}
```