第八节 模块化

1.定义

2.模块化的演变过程

- 2.1 Stage 1- 文件划分方式
- 2.2 Stage 2- 命名空间方式
- 2.3 Stage 3- IIFE
- 2.4 Stage 4- CommonJS规范
- 2.5 Stage 5- AMD(Asynchronous Module Definition)
- 2.6 Stage 6 -Sea.js + CMD

3.模块化规范

4.常用的模块化打包工具

5.ES Modules

- 5.1 基本特性
  - 5.1.1 通过script添加 type = moudle 的属性,就可以以ES Module 的标准执行其中的JS代码了
  - 5.1.2 ES module自动采用严格模式,忽略 'use strict'
  - 5.1.3 每个es module都是运行在单独的私有作用域中
  - 5.1.4 es module是通过cors的方式请求外部 JS模块的
  - 5.1.5 es module的script标签会延迟执行脚本
- 5.2 ES Module 导出
  - 5.2.1 命名导出
  - 5.2.2 默认导出
- 5.3 ES Module 导入
- 5.4 ES Modules in Browser
- 5.5 ES Modules in node
  - 5.5.1 方式一:
  - 5.5.2 :ES Modules 与CommonJS交互
  - 5.5.3 node.js 12.0版本之后支持esm
  - 5.5.4 使用babel转换

1.定义

模块化开发当下最重要的前端开发范式之一
随着前端代码的日益复杂,不得不花大量的时间去维护代码
而模块化开发是一种主流的代码组织方式
复杂代码按照功能的不同,划分为不同的模块,单独维护的模式,去提高开发效率,降低维护成本
模块化只是思想

2.模块化的演变过程

2.1 Stage 1– 文件划分方式

按照功能和相关的状态数据,写到同一个文件中
约定每一个文件都是一个独立的模块
使用按个模块就引入到页面中,一个script就是一个模块
然后在代码中直接调用代码中的全局成员

```
// module-a.js
var name = 'module-a'

function method1() {
    console.log(name + '#methods');
}

function method2() {
    console.log(name + '#methods');
}

// module-b.js
var name = 'module-b';

function method1() {
    console.log(name + '#methods');
}

function method2() {
    console.log(name + '#methods');
}
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>基于文件的划分模块</h1>
    <script src="./module-a.js"></script>
    <script src="./module-b.js"></script>
    <script>
        // 命名冲突
        method1()
        // 模块成员可以被修改
        name = 'foo'

    </script>
</body>
</html>
```

缺点:

1.所有模块在全局范围内工作,污染全局作用域,没有独立的私有空间

2.命名冲突问题

3.无法管理模块依赖关系

早期模块化完全依靠约定

2.2 Stage 2– 命名空间方式

1用对象包裹属性和方法,减少命名冲突问题

但是没有私有空间,还是会污染全局变量,模块中的属性和方法都可以随意改变

也无法解决项目依赖问题

```js
var moduleB = {
    name = 'module-b',

    method1:function () {
        console.log(name + '#methods')
    },
```

```
7
8       method2:function () {
9           console.log(name + '#methods')
10      }
11
12  }
```

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale
   =1.0">
6      <title>Document</title>
7  </head>
8  <body>
9      <h1>基于文件的划分模块</h1>
10     <script src="./module-a.js"></script>
11     <script src="./module-b.js"></script>
12     <script>
13         moduleA.method1()
14         moduleB.method1()
15
16         moduleA.name = 'xxx'
17
18     </script>
19 </body>
20 </html>
```

2.3 Stage 3- IIFE

使用立即执行函数创建一个私有空间

```
1  ;(function(){
2      var name = 'module-a';
3
```

```
    function method1() {
        console.log(name + '#methods');
    }

    function method2() {
        console.log(name + '#methods');
    }
    window.moduleA = {
        method1: method1,
        method2: method2
    }
})()
// 如果模块需要依赖其他模块可以传进去参数
(function ($) {
    var name = 'module-b';

    function method1() {
        console.log(name + '#methods');
        $('body').animate({ margin: '200px' });
    }

    function method2() {
        console.log(name + '#methods');
    }
    window.moduleB = {
        method1: method1,
        method2: method2,
    };
})(jQuery);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
=1.0">
    <title>Document</title>
</head>
```

```
<body>
    <h1>基于文件的划分模块</h1>
    <script src="./module-a.js"></script>
    <script src="./module-b.js"></script>
    <script>
        moduleA.method1()
        moduleB.method1()
        // 无法习惯
        // moduleA.name = 'xxx'
    </script>
</body>
</html>
```

问题还是有一个,都是通过script标签手动的引入用到的模块,不受我们的控制
比如我们的代码依赖某一个模块,在script标签中忘记引用
比如我们的代码不依赖某个一模块,但是script标签忘记删除了

这个时候我们需要一个
    模块化标准 + 模块加载器

2.4 Stage 4– CommonJS规范

- 一个文件就是一个模块
- 每个模块都有独立的作用域
- 通过module.exports导出成员
- 通过require函数载入模块

无法在浏览器端使用,因为CommonJS是以同步模式加载模块
node是在启动的时候去加载模块,执行过程中是不需要加载模块,只会使用模块
如果在浏览器中加载了多余的模块,那么就会造成大量无用的代码,浪费请求资源

2.5 Stage 5– AMD(Asynchronous Module Definition)

目前绝大多数第三方库都支持AMD规范
但是AMD使用起来相对复杂
如果js划分模块比较细的话,模块文件请求频繁
Require.js实现了该规范

```
define(
    'module-a', // 模块名称
    ['jquery','./module2'], // 模块的依赖项
    function($,module2){ // 函数的参数是前面的依赖项导出的成员
```

```
        return { // 导出的成员
            start: function() {
                $('body').animate({margin: '200px'})
                module2()
            }
        }
    }
)
```

```
// 手动载入一个模块
require(['./module1'],function(module1) {
    module1.start()
})
```

2.6 Stage 6 –Sea.js + CMD

```
// CMD 规范（类似 CommonJS 规范）
define(function (require, exports, module) {
    // 通过 require 引入依赖
  var $ = require('jquery')
  // 通过 exports 或者 module.exports 对外暴露成员
  module.exports = function () {
    console.log('module 2~')
    $('body').append('<p>module2</p>')
  }
})
```

3.模块化规范

ECMAScript 2015(ES6)中模块化是比较新的功能,各种浏览器兼容问题随之而来
可以通过各种打包工具,转换编译成适合浏览器的代码

4.常用的模块化打包工具

1.webpack
2.parcel
3.rollup

5.ES Modules

5.1 基本特性

5.1.1 *通过script添加 type = moudle 的属性,就可以以ES Module 的标准执行其中的JS代码了*

5.1.2 ES module自动采用严格模式,忽略 'use strict'

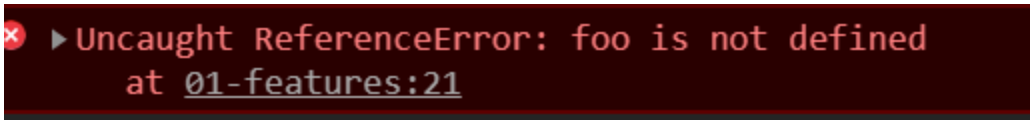
打印的this是undefined

```
1     </script>
```

```
        <script type="module">
        console.log(this);
    </script>
```

5.1.3 每个es module都是运行在单独的私有作用域中

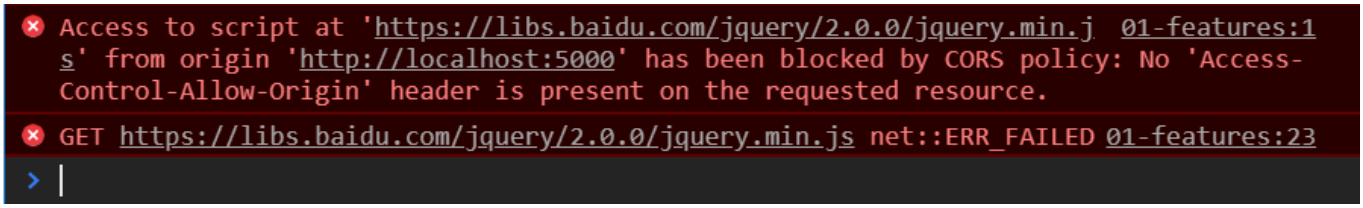
```
    <script type="module">
        let foo = '12312'
    </script>
    <script>
        console.log(foo)
    </script>
```



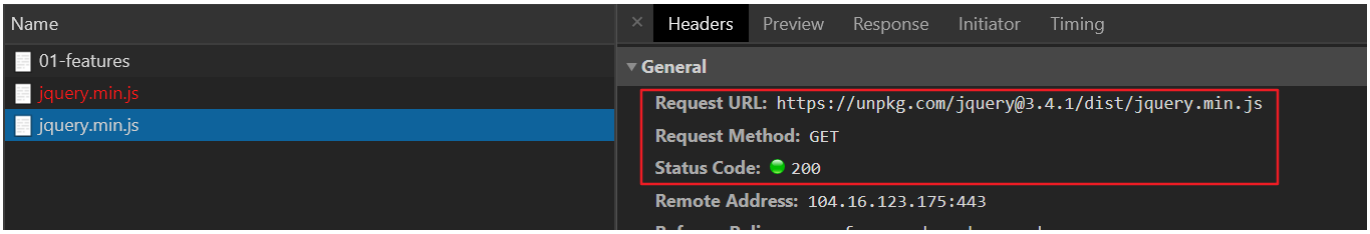
5.1.4 es module是通过cors的方式请求外部 JS模块的

如果我们的js不在同源地址上的话,我们在请求其他模块的时候,响应头中添加cors的标头,允许跨域

```
//这个地址不支持cors
<script type="module" src="https://libs.baidu.com/jquery/2.0.0/jquery.min.js"></script>
```



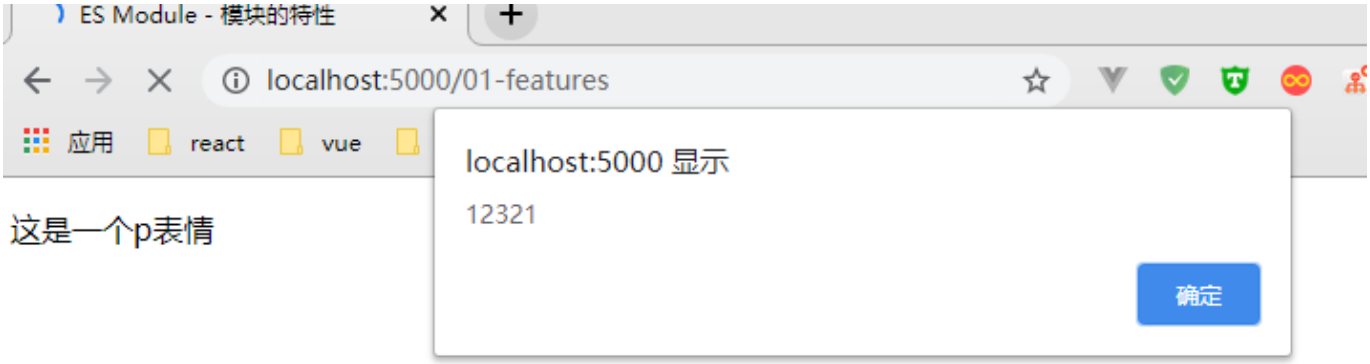

```
    <script type="module" src="https://unpkg.com/jquery@3.4.1/dist/jquery.min.js"></script>
```



5.1.5 es module的script标签会延迟执行脚本

相当于添加一个defer属性,不会阻塞html往下执行

```
    <script type='module'>
        alert('12321')
    </script>
    <p>这是一个p表情</p>
```



5.2 ES Module 导出

5.2.1 命名导出

- 一个模块可以有多个命名导出

```
// 写法1
export const name = 'calculator';
export const add = function(a, b) { return a + b; };

// 写法2
const name = 'calculator';
```

```
const add = function(a, b) { return a + b; };
export { name, add };
```

- 在使用命名导出时，可以通过as关键字对变量重命名

```
const name = 'calculator';
const add = function(a, b) { return a + b; };
export { name, add as getSum }; // 在导入时即为 name 和 getSum
```

5.2.2 默认导出

模块默认导出只能有一个

```
export default {
    name: 'calculator',
    add: function(a, b) {
        return a + b;
    }
};
// 导出字符串
export default 'This is calculator.js';
// 导出 class
export default class {...}
// 导出匿名函数
export default function() {...}
```

5.3 ES Module 导入

- 导入带有命名导出的模块
  - import后面要跟一对大括号来将导入的变量名包裹起来
  - 变量名需要与导出的变量名完全一致
  - 导入变量的效果相当于在当前作用域下声明了这些变量（name和add），并且不可对其进行更改，也就是所有导入的变量都是只读的

```
// calculator.js
const name = 'calculator';
const add = function(a, b) { return a + b; };
export { name, add };
```

```
5
6 // index.js
7 import { name, add } from './calculator.js';
8 add(2, 3);
```

可以对到导入的变量重命名

```
1 import { name, add as calculateSum } from './calculator.js';
2 calculateSum(2, 3);
```

导入多个变量时,可以采用整体导入的方式

```
1 import * as calculator from './calculator.js';
2 console.log(calculator.add(2, 3));
3 console.log(calculator.name);
```

- 导入默认导出的模块
  - React就是默认导出
  -

```
1 // index.js
2 import React, { Component } from 'react';
```

5.4 ES Modules in Browser

Polyfill兼容方案

1.使用browser-es-module-loader

可以通过http://unpkg.com获取该js文件 babel转换成es模块

```
1 https://unpkg.com/browser-es-module-loader@0.4.1/dist/babel-browser-b
  uild.js
```

或者broswer-es-moudle-loader

```
1 https://unpkg.com/browser-es-module-loader@0.4.1/dist/browser-es-modu
```

```
le-loader.js
```

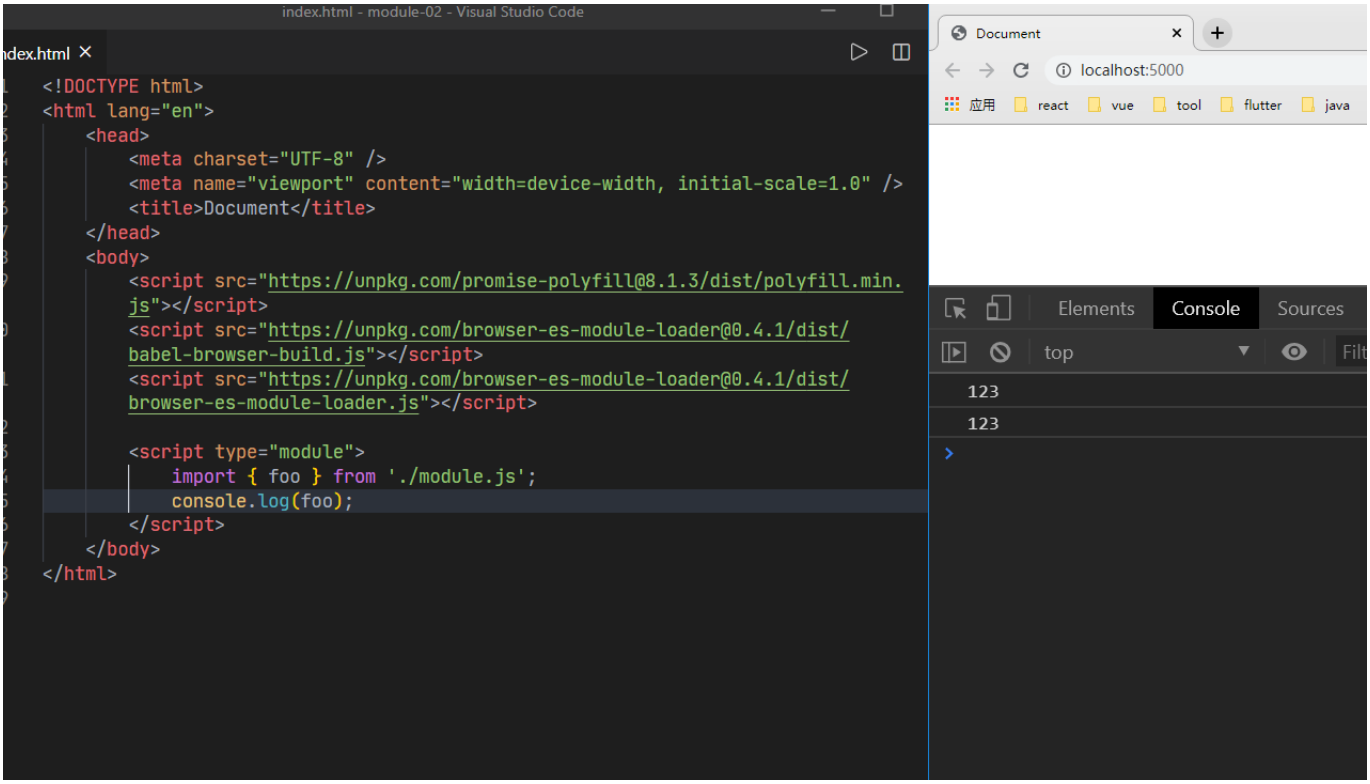
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script src="https://unpkg.com/browser-es-module-loader@0.4.1/dist/babel-browser-build.js"></script>
    <script src="https://unpkg.com/browser-es-module-loader@0.4.1/dist/browser-es-module-loader.js"></script>
    <script type="module">
        import {foo} from './module.js'
        console.log(foo);
    </script>
</body>
</html>
```

2.但是ie不支持promise
还要安装promise-polyfill

```
https://unpkg.com/promise-polyfill@8.1.3/dist/polyfill.min.js
```

其实原理很简单,就是将浏览器中不支持es-moudle,通过babel转换
然后我们的es模块文件,通过ajax请求,请求回来的代码然后通过babel转换,从而支持我们的es-module

但是问题就来了
es模块文件的代码会执行2次



- google支持es-moudle,当中代码会2次,
- 浏览器会执行一次我们的es-module代码
- 然后我们的broser-es-moudle也会执行一次代码

解决方式: 在script标签中加入nomodule

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
    </head>
    <body>
        <script nomodule src="https://unpkg.com/promise-polyfill@8.1.3/dist/polyfill.min.js"></script>
        <script nomodule src="https://unpkg.com/browser-es-module-loader@0.4.1/dist/babel-browser-build.js"></script>
        <script nomodule src="https://unpkg.com/browser-es-module-loader@0.4.1/dist/browser-es-module-loader.js"></script>

```

```
        <script type="module">
            import { foo } from './module.js';
            console.log(foo);
        </script>
    </body>
</html>
```

但是最后不要这么做,
因为在运行阶段动态 的去解析脚本效率 会低下
在生成环境下应该预先把代码编译出来,直接可以在浏览器中工作

5.5 ES Modules in node

5.5.1 方式一:

- 修该js文件扩展名为.mjs
- 运行node命令为

```
node --expermental-modules index.mjs
```

- 可以通过es-module的方式载入原生模块,比如fs

5.5.2 :ES Modules 与CommonJS交互

- ES Module 中可以导入默认导出的CommonJS模块
- 不能直接提取成员,注意import 不能解构出对象

```

import mod from './common.js'
console.log(mod);

// import {foo} from './common.js
// console.log(mod)
```

```
module.exports = {
    foo: 'commonjs exports value'
}
// exports.foo = 'commons.js
```

- 不能直接在commonjs模块中导入require载入ES Module

```
// es-moudle.js
export const foo = 'es module export value'

// common.js
const mod  = require('./es-module.mjs')
```

- ESM 中没有CommonJSS中那些模块全局成员

```
// common.js

// 加载模块函数
console.log(require);

// 模块对象
console.log(module);

// 导出对象别名
console.log(exports);

// 当前文件绝对路径
console.log(__filename);


// 当前文件所在目录
console.log(__dirname);

// esm.mjs
import { fileURLToPath } from 'url'
import { dirname } from 'path'

const __filename = fileURLToPath(import.meta.url)

const __dirname = dirname(__filename)
```

- common.js实现的简单原理

commonjs代码最终传入了这个函数在外侧包裹了一个函数,从而实现私有模块作用域

```
let wrap = function(script) {
  return Module.wrapper[0] + script + Module.wrapper[1];
};

const wrapper = [
  '(function (exports, require, module, __filename, __dirname) { ',
  '\n});'
];
```

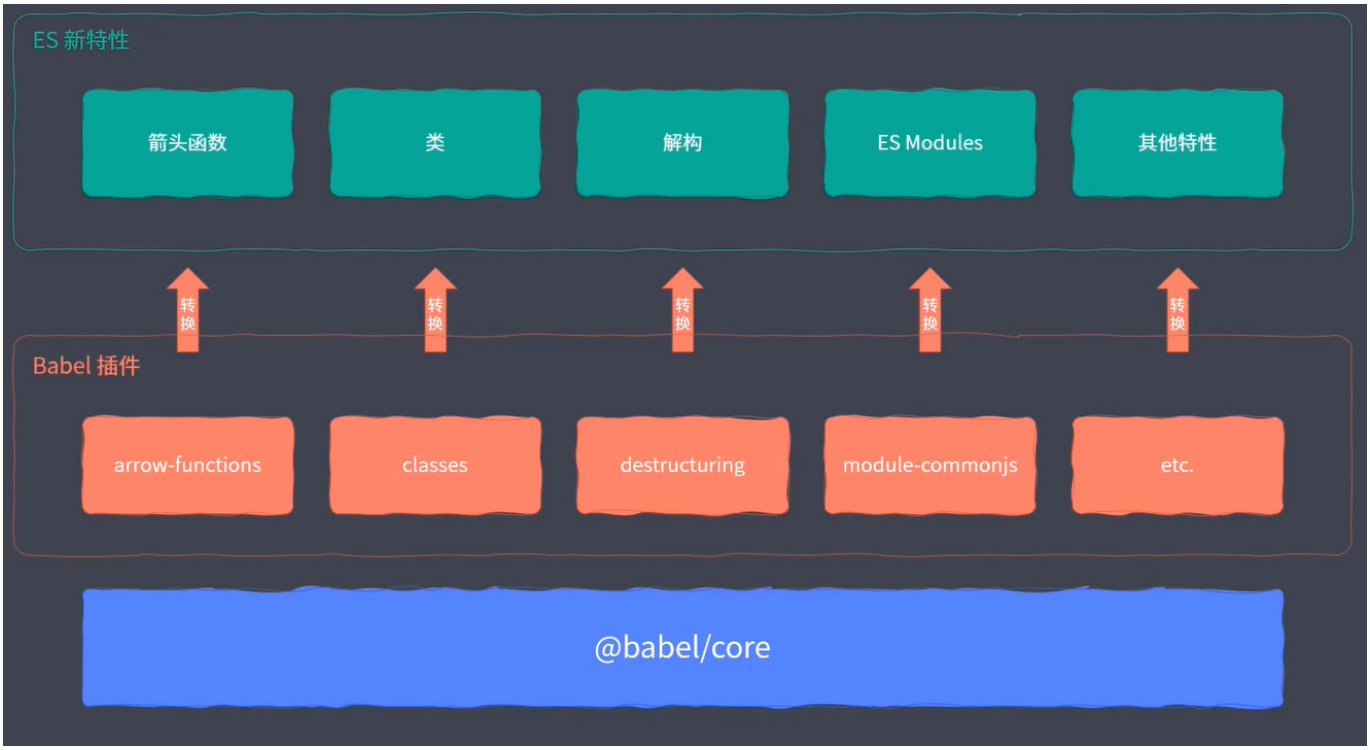
5.5.3 node.js 12.0版本之后支持esm

在package.json中添加字段

```
{
    "type": "module"
}
```

但是common.js总要修改后缀名为common.cjs

5.5.4 使用babel转换



babel中的插件,就封装了某个转换新特性的集合
安装特定的插件

```
yarn add @babel/plugin-transform-modules-commonjs -D
```

修改.babelrc文件

```
{
    "plugins": [
    "@babel/plugin-transform-modules-commonjs"
  ]
}
```