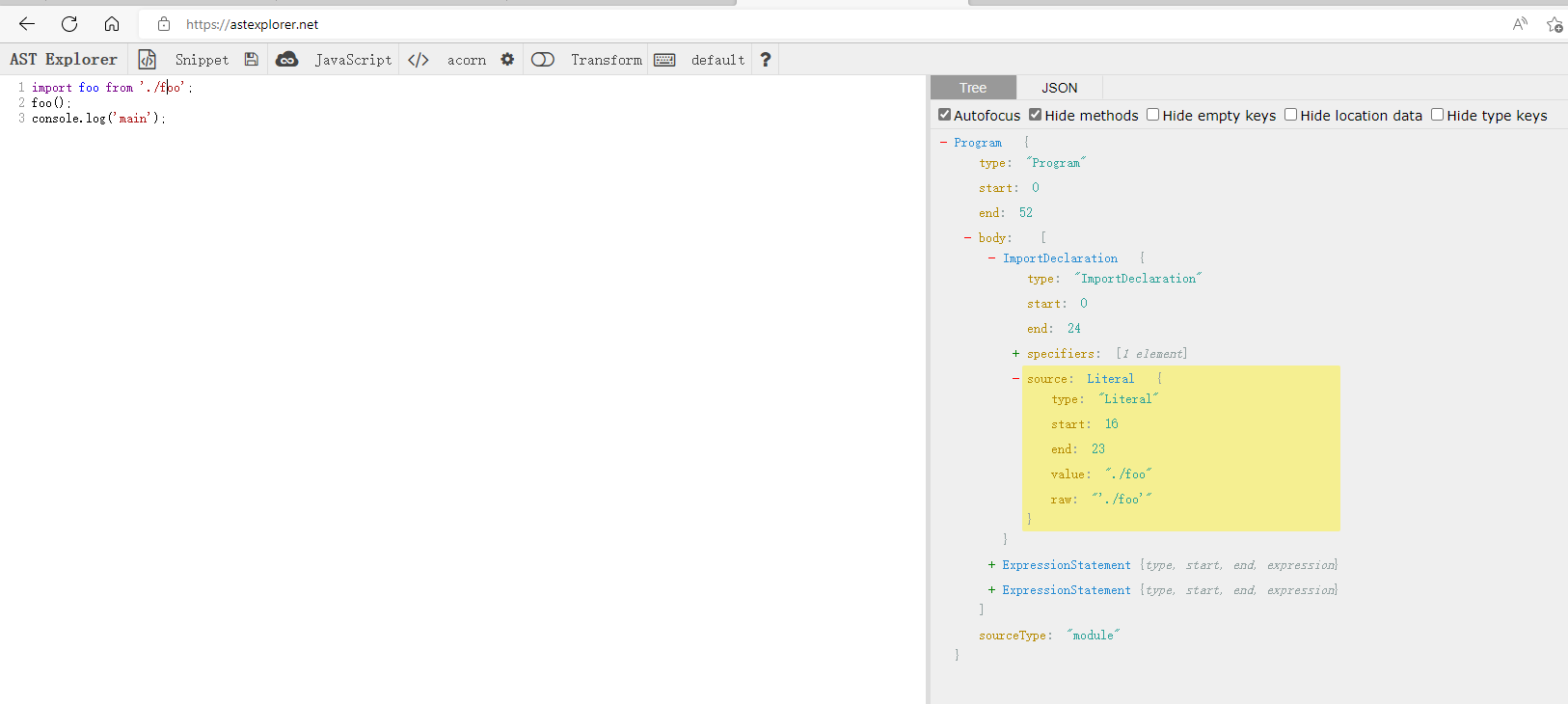
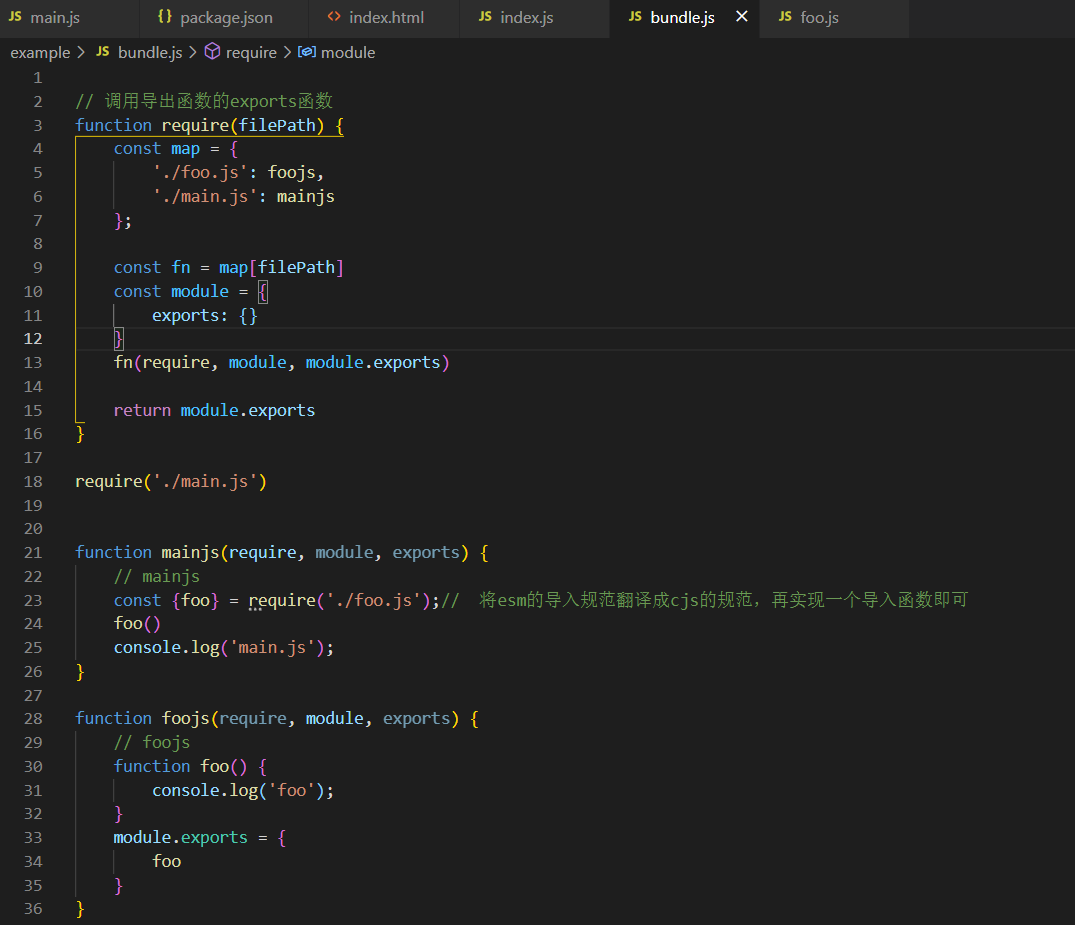
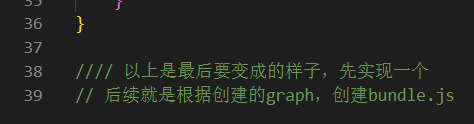
依赖关系通过ast获取：



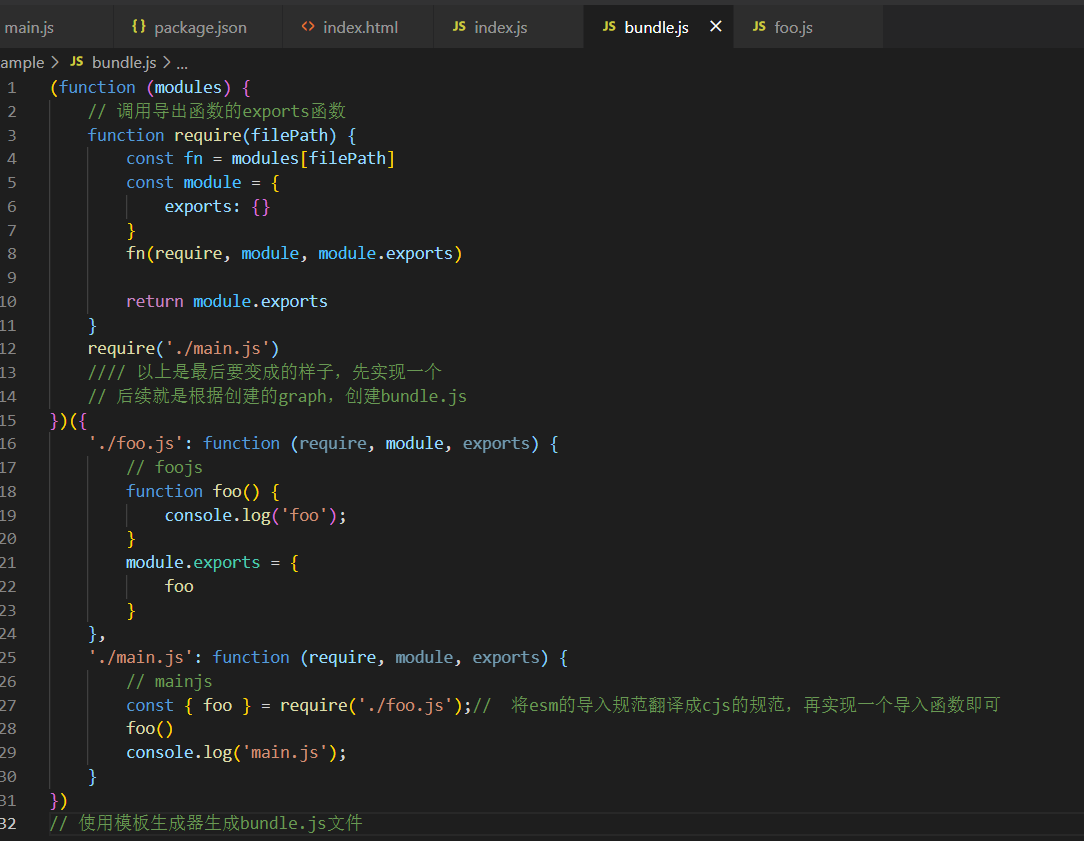
见source.value

bundle.js重构前：





优化：

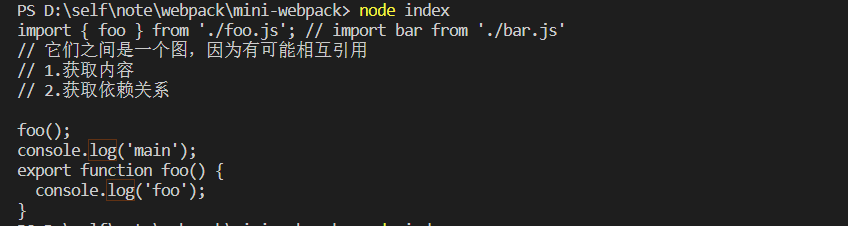


但是这里需要的是cjs的规范，所以就需要把esm的规范转换为cjs规范

这里的转换就用到了babel-core

配置前后：

前：

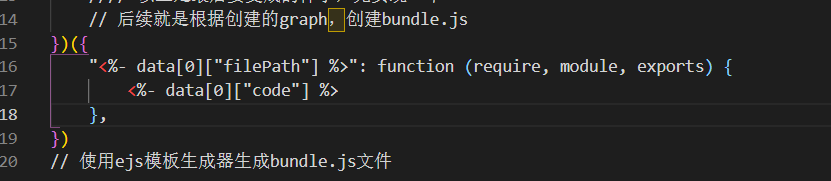


后：



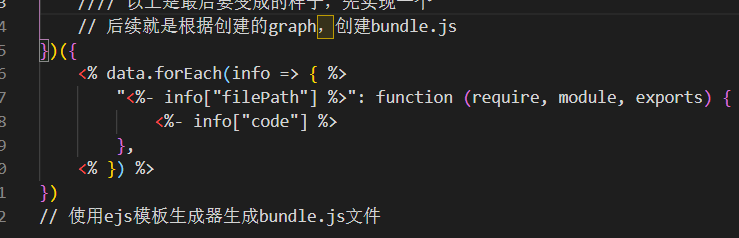
然后就是把数据给到模板，去生成内容。用到的就是graph的数据。



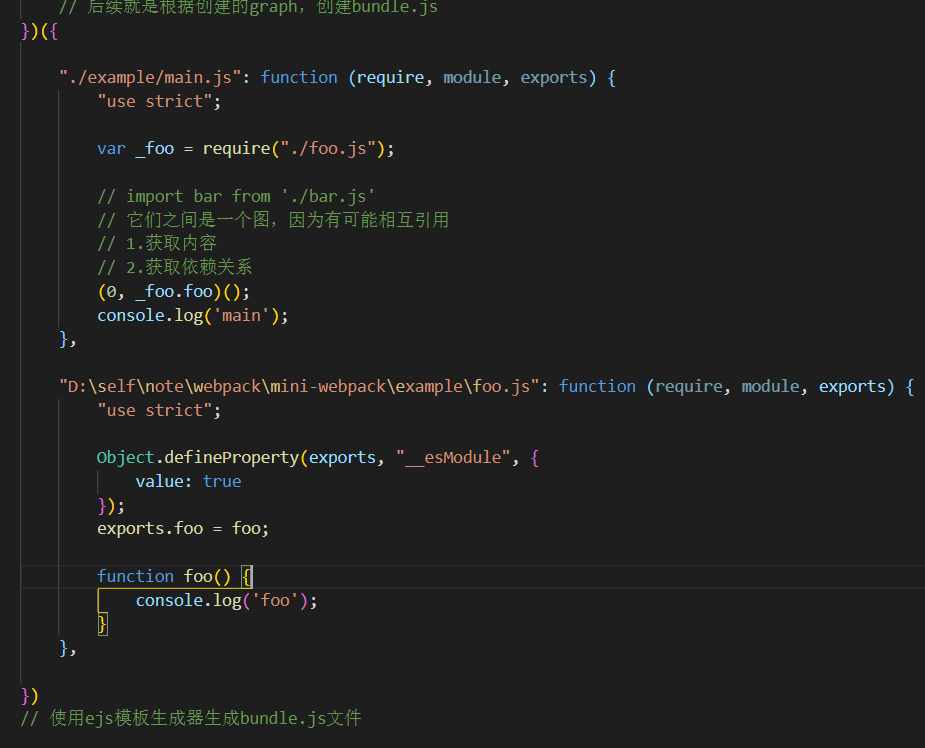


循环生成：

ejs的forEach：



输出的bundle.js:



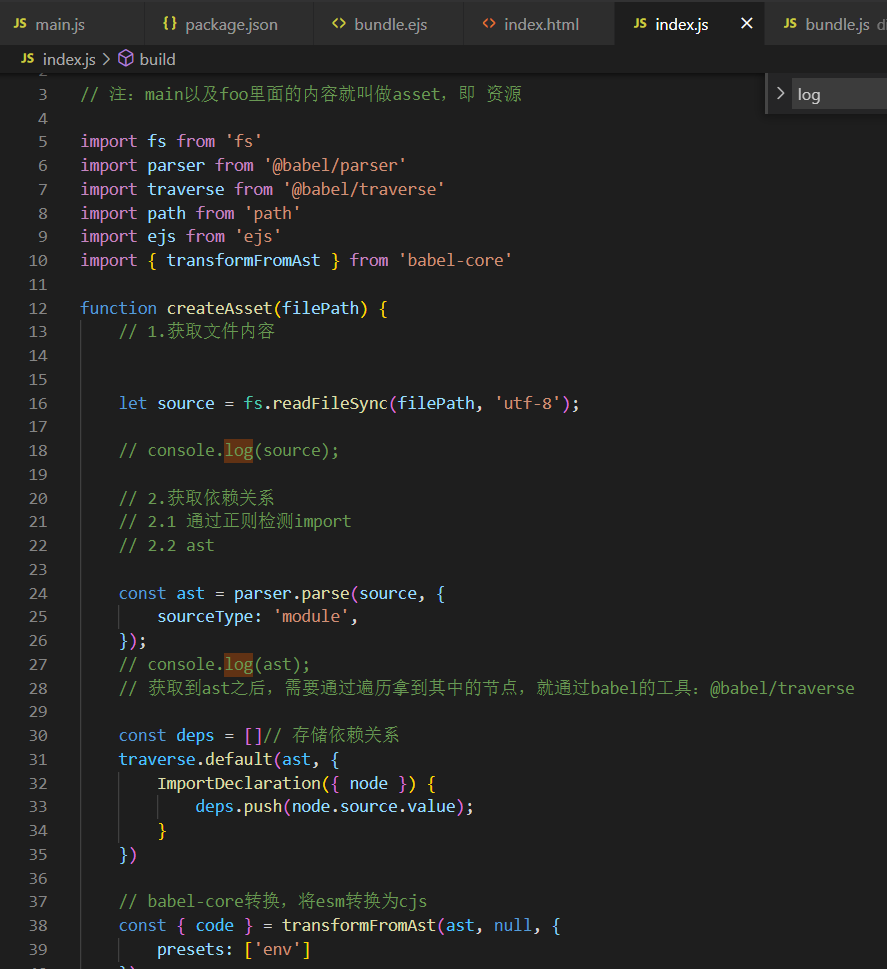
但是问题是在进行main.js的执行的时候，require的是’./foo.js’，但是现在的module中的key是一个大长串，是不会找得到的。

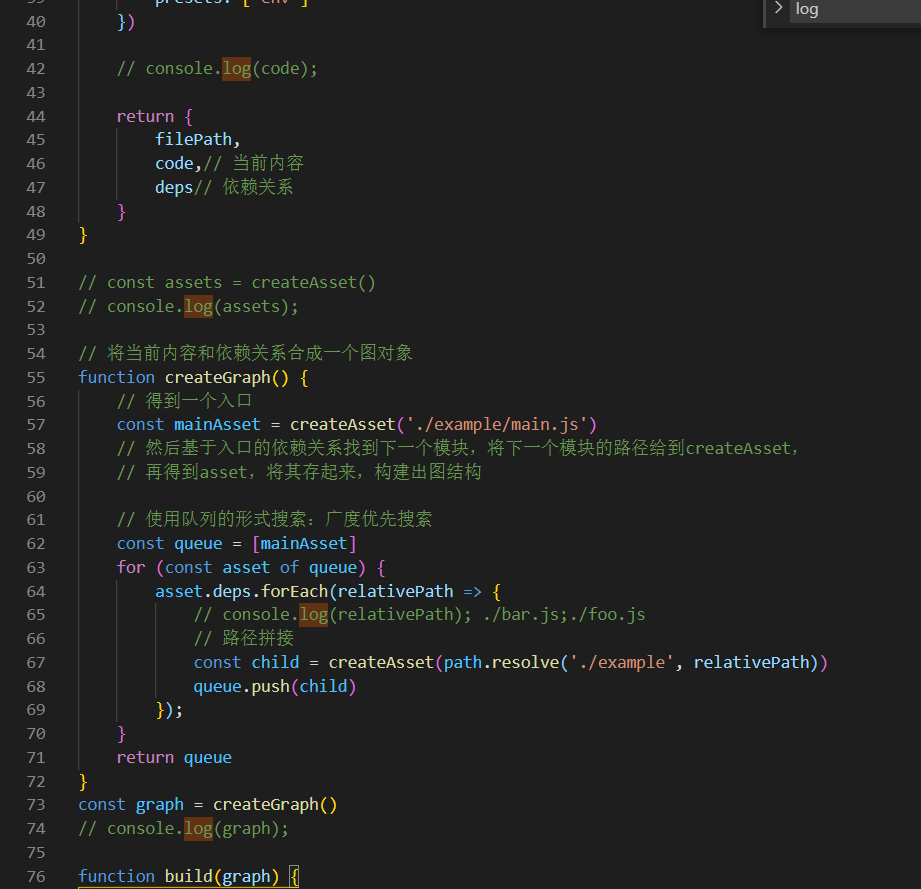
采用的解决方法是：给一个模块一个唯一id，基于id进行查找。并且在解构的时候还可以告知其依赖关系。

就是这样式的：



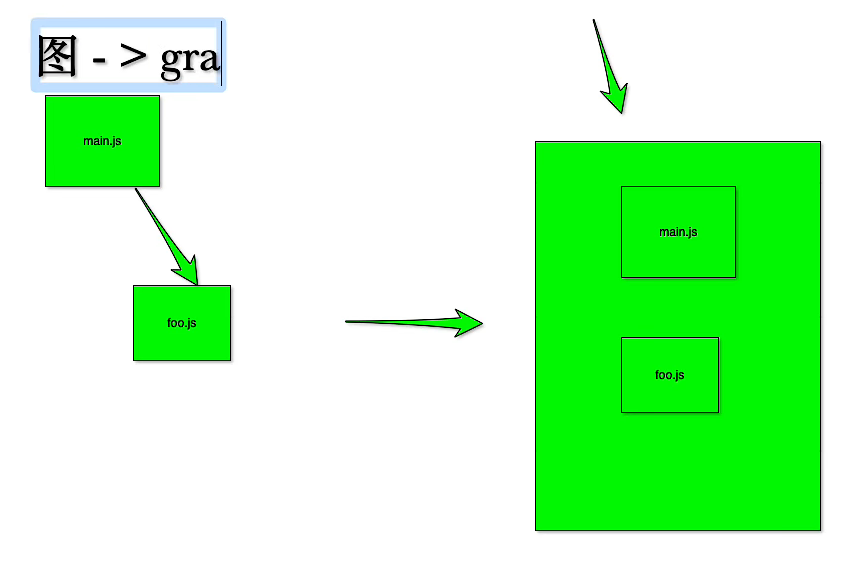
先保存一份index.js:







# 思路总结：



基于文件的内容构建一个图，根据图拿到所有的数据和依赖关系生成一个脚本。

## 1.生成这个图

为什么是图，不是一个树结构，因为有可能会有相互应用，所以不是一个树结构，是一个图结构。

### 1.1 获取内容

关键词：资源：

利用fs获取资源fs.readfilesync

### 1.2 获取依赖关系

可以使用正则去匹配import

**利用ast——抽象语法树**

需要babel，

利用其中名为ImportDeclaration的节点的value

用到的库：

1.@Babel/parser：给他一个code（上面获得的文件内容），一个配置options（{sourceType：“module”}），就返回一个ast

2.为了得到这个节点，需要用到@babel/travsers。（需要调用它的default）设置上一个函数，当遍历到这个节点的时候，就会自动调用设置的回调函数。node.source.value

3.将依赖关系存起来：数组deps

4.将这些合成一个图对象。

从入口文件出发。

基于入口assets就能找到下一个模块

遍历图的方式：队列：广度优先搜索。遍历入口asset的依赖数组

利用path.resolve拼接路径，去读取依赖的内容和依赖（根据上面的createAsset方法）。加入到队列中。

## 2.根据这个图生成脚本。

bundle.js中为了防止不同模块中的变量名称冲突，将各自的模块放在不同的函数中，但是import只能放在顶层作用域中，所以需要将import的esm的模块规范转换为cjs的规范，即require，然后再自己实现一个require导入函数

扩：为什么要用这个呢？因为可以自定义去实现。

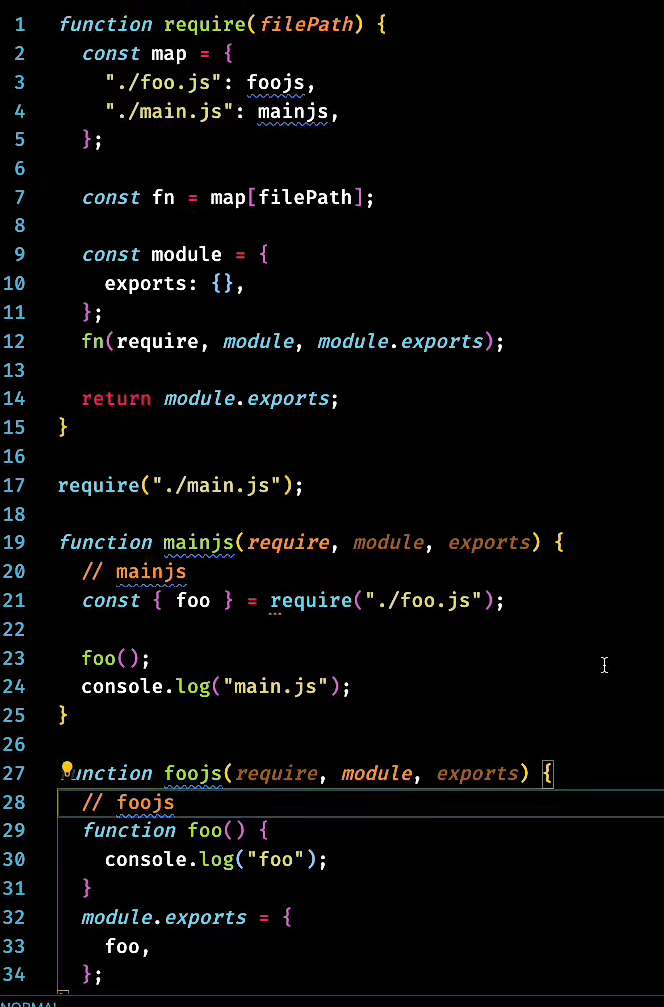
引入文件的require方法是自己实现的，不是cjs原生的。

### 2.1 require函数实现

根据不同的模块找到对应的函数。

require是现成的，module构建一个

require函数的返回结果就是这个模块的module.exports



看上图理解

重构的bundle.js:



### 2.2 基于图结构生成所示的bundle.js

不变的地方可以使用两种方法生成：

1.模板字符串拼接

2.ejs

生成文件方法：build

#### 2.2.1 设置模板

#### 2.2.2 将数据给到模板

模板里用到的是cjs的require，但是实际的模块中使用的是import语法，所以需要转换。用到的是babel/code

transfromFromAst（ast，null，{

presets:[‘env’]

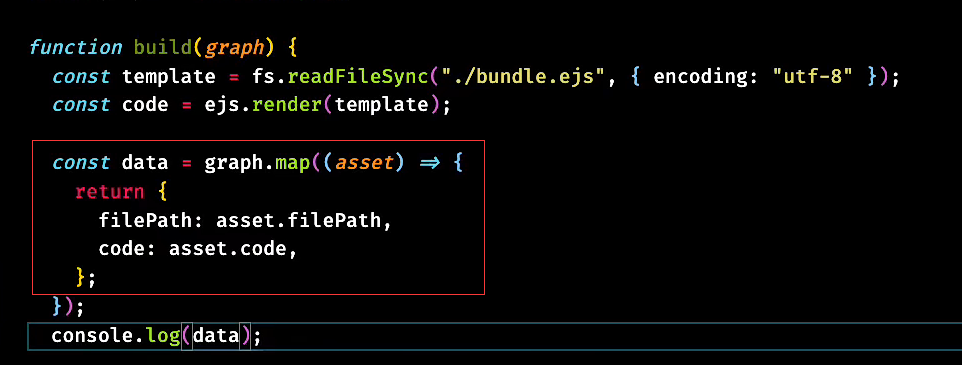
}）

需要用到的预设babel-preset-env，

之后就能将ast从esm规范转换为cjs的规范

**生成模板数据**

data=graph.map()



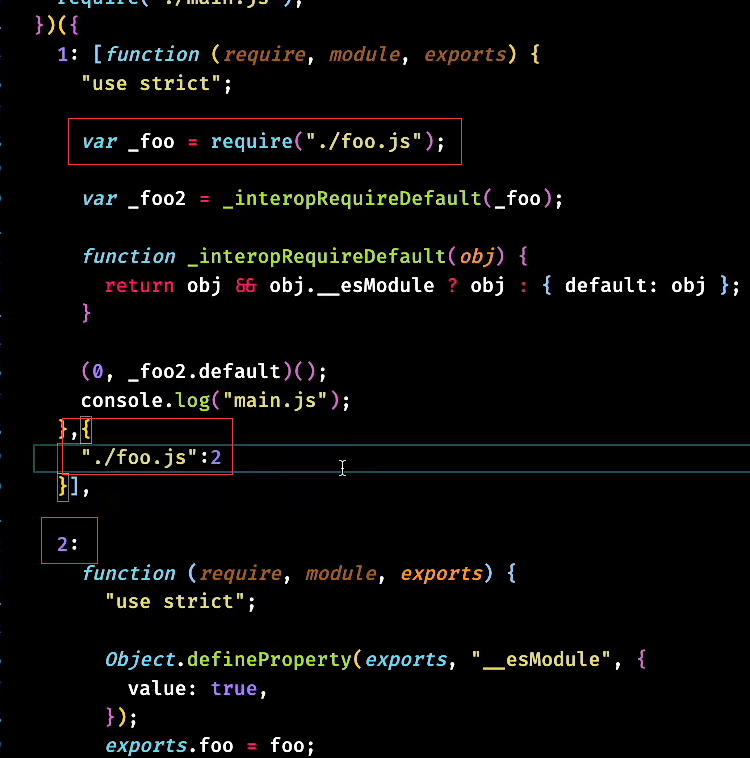
#### 2.2.3 路径匹配



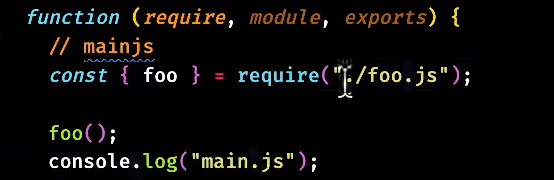
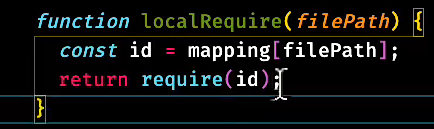
如图所示，module里面key是一个完整的路径，但是方法里面的请求的路径只是一个相对路径，二者是不匹配的。

解决方法：给一个模块一个唯一的id，并且再解构的时候告诉他他的依赖关系是什么。

如图：

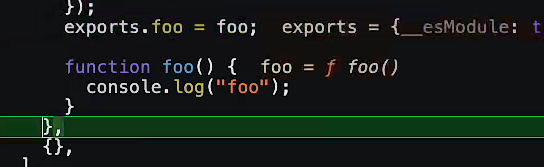


filePath将路径转换为id方法：



这样require的时候就无所畏惧了

后面就是设置模板，设置输入模板的数据的结构：id，mapping等等

：这一步就是将模块要导出的内容挂载到export上，然后将module.exports导出

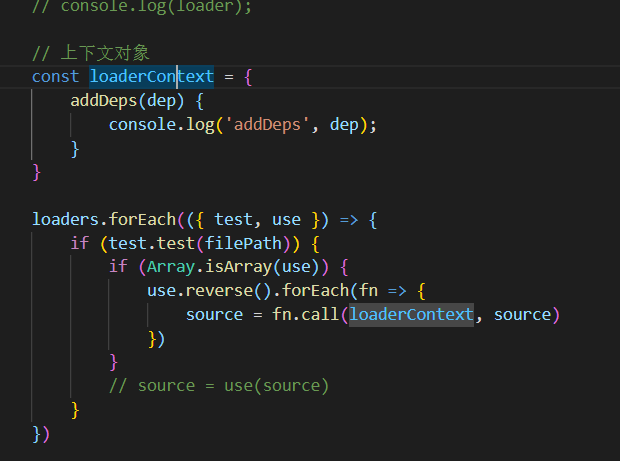
## 3.loader

可以让webpack处理非js文件

### 3.1 webpack.config.js设置modules

就是一个函数，将非js处理为js。

在获取ast之前就需要使用loader对内容进行处理



如果匹配上了就是用loader处理。

这里出现了上下文的接口。详情见共享文档中的链接。

## 4.plugin

最核心的就是事件机制：webpack再底层实现了一个库：tapable

暴漏出了各种各样的hook，就可以侦听各种各样的hook实现功能，就像给按钮注册点击事件，当点击按钮的时候，就能触发相应的事件。

事件中就是两点：

1.注册：

tap注册函数，

2.触发：

与注册对应。

在初始化的时候需要将所有的插件调用一下。因为有可能添加注册事件了。

如何影响webpack，就像loader通过调用this输出一样，插件也可以通过上下文对webpack产生影响。