## 手写webpack核心原理

## 一、核心打包原理

### 1.1 打包的主要流程如下

1. 需要读到入口文件里面的内容。
2. 分析入口文件，递归的去读取模块所依赖的文件内容，生成AST语法树。
3. 根据AST语法树，生成浏览器能够运行的代码

### 1.2 具体细节

1. 获取主模块内容
2. 分析模块
   * 安装@babel/parser包（转AST）
3. 对模块内容进行处理
   * 安装@babel/traverse包（遍历AST收集依赖）
   * 安装@babel/core和@babel/preset-env包 （es6转ES5）
4. 递归所有模块
5. 生成最终代码

## 二、基本准备工作

我们先建一个项目

项目目录暂时如下：

已经把项目放到 **「github」**：https://github.com/Sunny-lucking/howToBuildMyWebpack。可以卑微地要个star吗

我们创建了add.js文件和minus.js文件,然后 在index.js中引入，再将index.js文件引入index.html。

代码如下:

add.js

export default (a,b)=>{  
  return a+b;  
}

minus.js

export const minus = (a,b)=>{  
    return a-b  
}

index.js

import add from "./add"  
import {minus} from "./minus";  
  
const sum = add(1,2);  
const division = minus(2,1);  
  
console.log(sum);  
console.log(division);

index.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <title>Title</title>  
</head>  
<body>  
<script src="./src/index.js"></script>  
</body>  
</html>

现在我们打开index.html。你猜会发生什么？？？显然会报错，因为浏览器还不能识别import语法

不过没关系，因为我们本来就是要来解决这些问题的。

## 三、获取模块内容

好了，现在我们开始根据上面核心打包原理的思路来实践一下，第一步就是 实现获取模块内容。

我们来创建一个bundle.js文件。

*// 获取主入口文件*  
const fs = require('fs')  
const getModuleInfo = (file)=>{  
    const body = fs.readFileSync(file,'utf-8')  
    console.log(body);  
}  
getModuleInfo("./src/index.js")

目前项目目录如下

我们来执行一下bundle.js，看看时候成功获得入口文件内容

哇塞，不出所料的成功。一切尽在掌握之中。好了，已经实现第一步了，且让我看看第二步是要干嘛。哦？是分析模块了

## 四、分析模块

分析模块的主要任务是 将获取到的模块内容 解析成AST语法树，这个需要用到一个依赖包@babel/parser

npm install @babel/parser

ok,安装完成我们将@babel/parser引入bundle.js,

*// 获取主入口文件*  
const fs = require('fs')  
const parser = require('@babel/parser')  
const getModuleInfo = (file)=>{  
    const body = fs.readFileSync(file,'utf-8')  
    const ast = parser.parse(body,{  
        sourceType:'module' *//表示我们要解析的是ES模块*  
    });  
    console.log(ast);  
}  
getModuleInfo("./src/index.js")

我们去看下@babel/parser的文档：

可见提供了三个API，而我们目前用到的是parse这个API。

它的主要作用是 parses the provided code as an entire ECMAScript program，也就是将我们提供的代码解析成完整的ECMAScript代码。

再看看该API提供的参数

我们暂时用到的是sourceType，也就是用来指明我们要解析的代码是什么模块。

好了，现在我们来执行一下 bundle.js，看看AST是否成功生成。

成功。又是不出所料的成功。

不过，我们需要知道的是，当前我们解析出来的不单单是index.js文件里的内容，它也包括了文件的其他信息。而它的内容其实是它的属性program里的body里。如图所示

我们可以改成打印ast.program.body看看

*// 获取主入口文件*  
const fs = require('fs')  
const parser = require('@babel/parser')  
const getModuleInfo = (file)=>{  
    const body = fs.readFileSync(file,'utf-8')  
    const ast = parser.parse(body,{  
        sourceType:'module' *//表示我们要解析的是ES模块*  
    });  
    console.log(ast.program.body);  
}  
getModuleInfo("./src/index.js"

执行看，现在打印出来的就是 index.js文件里的内容（也就是我们再index.js里写的代码啦）.

## 五、收集依赖

现在我们需要 遍历AST，将用到的依赖收集起来。什么意思呢？其实就是将用import语句引入的文件路径收集起来。我们将收集起来的路径放到deps里。

前面我们提到过，遍历AST要用到@babel/traverse依赖包

npm install @babel/traverse

现在，我们引入。

const fs = require('fs')  
const path = require('path')  
const parser = require('@babel/parser')  
const traverse = require('@babel/traverse').default  
const getModuleInfo = (file)=>{  
    const body = fs.readFileSync(file,'utf-8')  
    const ast = parser.parse(body,{  
        sourceType:'module' *//表示我们要解析的是ES模块*  
    });  
      
    *// 新增代码*  
    const deps = {}  
    traverse(ast,{  
        ImportDeclaration({node}){  
            const dirname = path.dirname(file)  
            const abspath = './' + path.join(dirname,node.source.value)  
            deps[node.source.value] = abspath  
        }  
    })  
    console.log(deps);  
  
  
}  
getModuleInfo("./src/index.js")

我们来看下官方文档对@babel/traverse的描述

好吧，如此简略

不过我们不难看出，第一个参数就是AST。第二个参数就是配置对象

我们看看我们写的代码

traverse(ast,{  
    ImportDeclaration({node}){  
        const dirname = path.dirname(file)  
        const abspath = './' + path.join(dirname,node.source.value)  
        deps[node.source.value] = abspath  
    }  
})

配置对象里，我们配置了ImportDeclaration方法，这是什么意思呢？我们看看之前打印出来的AST。

ImportDeclaration 方法代表的是对type类型为ImportDeclaration的节点的处理。

这里我们获得了该节点中source的value，也就是node.source.value，

这里的value指的是什么意思呢？其实就是import的值，可以看我们的index.js的代码。

import add from "./add"  
import {minus} from "./minus";  
  
const sum = add(1,2);  
const division = minus(2,1);  
  
console.log(sum);  
console.log(division);

可见，value指的就是import后面的 './add' 和 './minus'

然后我们将file目录路径跟获得的value值拼接起来保存到deps里，美其名曰：收集依赖。

ok，这个操作就结束了，执行看看收集成功了没？

oh my god。又成功了。

## 六、ES6转成ES5（AST）

现在我们需要把获得的ES6的AST转化成ES5的AST，前面讲到过，执行这一步需要两个依赖包

npm install @babel/core @babel/preset-env

我们现在将依赖引入并使用

const fs = require('fs')  
const path = require('path')  
const parser = require('@babel/parser')  
const traverse = require('@babel/traverse').default  
const babel = require('@babel/core')  
const getModuleInfo = (file)=>{  
    const body = fs.readFileSync(file,'utf-8')  
    const ast = parser.parse(body,{  
        sourceType:'module' *//表示我们要解析的是ES模块*  
    });  
    const deps = {}  
    traverse(ast,{  
        ImportDeclaration({node}){  
            const dirname = path.dirname(file)  
            const abspath = "./" + path.join(dirname,node.source.value)  
            deps[node.source.value] = abspath  
        }  
    })  
      
    新增代码  
    const {code} = babel.transformFromAst(ast,null,{  
        presets:["@babel/preset-env"]  
    })  
    console.log(code);  
  
}  
getModuleInfo("./src/index.js")

我们看看官网文档对@babel/core 的transformFromAst的介绍

害，又是一如既往的简略。。。

简单说一下，其实就是将我们传入的AST转化成我们在第三个参数里配置的模块类型。

好了，现在我们来执行一下，看看结果

我的天，一如既往的成功。可见 它将我们写const 转化成var了。

好了，这一步到此结束，咦，你可能会有疑问，上一步的收集依赖在这里怎么没啥关系啊，确实如此。收集依赖是为了下面进行的递归操作。

## 七、递归获取所有依赖

经过上面的过程，现在我们知道getModuleInfo是用来获取一个模块的内容，不过我们还没把获取的内容return出来，因此，更改下getModuleInfo方法

const getModuleInfo = (file)=>{  
    const body = fs.readFileSync(file,'utf-8')  
    const ast = parser.parse(body,{  
        sourceType:'module' *//表示我们要解析的是ES模块*  
    });  
    const deps = {}  
    traverse(ast,{  
        ImportDeclaration({node}){  
            const dirname = path.dirname(file)  
            const abspath = "./" + path.join(dirname,node.source.value)  
            deps[node.source.value] = abspath  
        }  
    })  
    const {code} = babel.transformFromAst(ast,null,{  
        presets:["@babel/preset-env"]  
    })  
    *// 新增代码*  
    const moduleInfo = {file,deps,code}  
    return moduleInfo  
}

我们返回了一个对象 ，这个对象包括**「该模块的路径（file）」**，**「该模块的依赖（deps）」**，**「该模块转化成es5的代码」**

该方法只能获取一个模块的的信息，但是我们要怎么获取一个模块里面的依赖模块的信息呢？

没错，看标题，，你应该想到了就算递归。

现在我们来写一个递归方法，递归获取依赖

const parseModules = (file) =>{  
    const entry =  getModuleInfo(file)  
    const temp = [entry]  
    for (let i = 0;i<temp.length;i++){  
        const deps = temp[i].deps  
        if (deps){  
            for (const key in deps){  
                if (deps.hasOwnProperty(key)){  
                    temp.push(getModuleInfo(deps[key]))  
                }  
            }  
        }  
    }  
    console.log(temp)  
}

讲解下parseModules方法：

1. 我们首先传入主模块路径
2. 将获得的模块信息放到temp数组里。
3. 外面的循坏遍历temp数组，此时的temp数组只有主模块
4. 里面再获得主模块的依赖deps
5. 遍历deps，通过调用getModuleInfo将获得的依赖模块信息push到temp数组里。

目前bundle.js文件：

const fs = require('fs')  
const path = require('path')  
const parser = require('@babel/parser')  
const traverse = require('@babel/traverse').default  
const babel = require('@babel/core')  
const getModuleInfo = (file)=>{  
    const body = fs.readFileSync(file,'utf-8')  
    const ast = parser.parse(body,{  
        sourceType:'module' *//表示我们要解析的是ES模块*  
    });  
    const deps = {}  
    traverse(ast,{  
        ImportDeclaration({node}){  
            const dirname = path.dirname(file)  
            const abspath = "./" + path.join(dirname,node.source.value)  
            deps[node.source.value] = abspath  
        }  
    })  
    const {code} = babel.transformFromAst(ast,null,{  
        presets:["@babel/preset-env"]  
    })  
    const moduleInfo = {file,deps,code}  
    return moduleInfo  
}  
  
*// 新增代码*  
const parseModules = (file) =>{  
    const entry =  getModuleInfo(file)  
    const temp = [entry]  
    for (let i = 0;i<temp.length;i++){  
        const deps = temp[i].deps  
        if (deps){  
            for (const key in deps){  
                if (deps.hasOwnProperty(key)){  
                    temp.push(getModuleInfo(deps[key]))  
                }  
            }  
        }  
    }  
    console.log(temp)  
}  
parseModules("./src/index.js")

按照目前我们的项目来说执行完，应当是temp 应当是存放了index.js,add.js,minus.js三个模块。,执行看看。牛逼！！！确实如此。

不过现在的temp数组里的对象格式不利于后面的操作，我们希望是以文件的路径为key，{code，deps}为值的形式存储。因此，我们创建一个新的对象depsGraph。

const parseModules = (file) =>{  
    const entry =  getModuleInfo(file)  
    const temp = [entry]   
    const depsGraph = {} *//新增代码*  
    for (let i = 0;i<temp.length;i++){  
        const deps = temp[i].deps  
        if (deps){  
            for (const key in deps){  
                if (deps.hasOwnProperty(key)){  
                    temp.push(getModuleInfo(deps[key]))  
                }  
            }  
        }  
    }  
    *// 新增代码*  
    temp.forEach(moduleInfo=>{  
        depsGraph[moduleInfo.file] = {  
            deps:moduleInfo.deps,  
            code:moduleInfo.code  
        }  
    })  
    console.log(depsGraph)  
    return depsGraph  
}

ok，现在存储的就是这种格式啦

## 八、处理两个关键字

我们现在的目的就是要生成一个bundle.js文件，也就是打包后的一个文件。其实思路很简单，就是把index.js的内容和它的依赖模块整合起来。然后把代码写到一个新建的js文件。

我们把这段代码格式化一下

*// index.js*  
"use strict"  
var \_add = \_interopRequireDefault(require("./add.js"));  
var \_minus = require("./minus.js");  
function \_interopRequireDefault(obj) { return obj && obj.\_\_esModule ? obj : { "default": obj }; }  
var sum = (0, \_add["default"])(1, 2);  
var division = (0, \_minus.minus)(2, 1);  
console.log(sum); console.log(division);  
*// add.js*  
"use strict";  
Object.defineProperty(exports, "\_\_esModule", {  value: true});  
exports["default"] = void 0;  
var \_default = function \_default(a, b) {  return a + b;};  
exports["default"] = \_default;

但是我们现在是不能执行index.js这段代码的，因为浏览器不会识别执行require和exports。

不能识别是为什么？不就是因为没有定义这require函数，和exports对象。那我们可以自己定义。我们创建一个函数

const bundle = (file) =>{  
    const depsGraph = JSON.stringify(parseModules(file))  
      
}

我们将上一步获得的depsGraph保存起来。现在返回一个整合完整的字符串代码。

怎么返回呢？更改下bundle函数

const bundle = (file) =>{  
    const depsGraph = JSON.stringify(parseModules(file))  
    return `(function (graph) {  
                function require(file) {  
                    (function (code) {  
                        eval(code)  
                    })(graph[file].code)  
                }  
                require(file)  
            })(depsGraph)`  
      
}

我们看下返回的这段代码

 (function (graph) {  
        function require(file) {  
            (function (code) {  
                eval(code)  
            })(graph[file].code)  
        }  
        require(file)  
    })(depsGraph)

其实就是

1. 把保存下来的depsGraph，传入一个立即执行函数。
2. 将主文件路径传入require函数执行
3. 执行reuire函数的时候，又立即执行一个立即执行函数，这里是把code的值传进去了
4. 执行eval（code）。也就是执行code这段代码

我们再来看下code的值

*// index.js*  
"use strict"  
var \_add = \_interopRequireDefault(require("./add.js"));  
var \_minus = require("./minus.js");  
function \_interopRequireDefault(obj) { return obj && obj.\_\_esModule ? obj : { "default": obj }; }  
var sum = (0, \_add["default"])(1, 2);  
var division = (0, \_minus.minus)(2, 1);  
console.log(sum); console.log(division);

没错执行这段代码的时候，又会用到require函数。此时require的参数为add.js的路径，哎，不是绝对路径，需要转化成绝对路径。因此写一个函数absRequire来转化。怎么实现呢？我们来看下代码

(function (graph) {  
    function require(file) {  
        function absRequire(relPath) {  
            return require(graph[file].deps[relPath])  
        }  
        (function (require,code) {  
            eval(code)  
        })(absRequire,graph[file].code)  
    }  
    require(file)  
})(depsGraph)

实际上是实现了一层拦截。

1. 执行require（'./src/index.js'）函数
2. 执行了

(function (require,code) {  
    eval(code)  
})(absRequire,graph[file].code)

1. 执行eval，也就是执行了index.js的代码。
2. 执行过程会执行到require函数。
3. 这时会调用这个require，也就是我们传入的absRequire
4. 而执行absRequire就执行了return require(graph[file].deps[relPath])这段代码，也就是执行了外面这个require

在这里return require(graph[file].deps[relPath])，我们已经对路径转化成绝对路径了。因此执行外面的require的时候就是传入绝对路径。

1. 而执行require（"./src/add.js"）之后，又会执行eval，也就是执行add.js文件的代码。

是不是有点绕？其实是个递归。

这样就将代码整合起来了，但是有个问题，就是在执行add.js的code时候，会遇到exports这个还没定义的问题。如下所示

*// add.js*  
"use strict";  
Object.defineProperty(exports, "\_\_esModule", {  value: true});  
exports["default"] = void 0;  
var \_default = function \_default(a, b) {  return a + b;};  
exports["default"] = \_default;

我们发现 这里它把exports当作一个对象来使用了，但是这个对象还没定义，因此我们可以自己定义一个exports对象。

(function (graph) {  
    function require(file) {  
        function absRequire(relPath) {  
            return require(graph[file].deps[relPath])  
        }  
        var exports = {}  
        (function (require,exports,code) {  
            eval(code)  
        })(absRequire,exports,graph[file].code)  
        return exports  
    }  
    require(file)  
})(depsGraph)

我们增添了一个空对象 exports，执行add.js代码的时候，会往这个空对象上增加一些属性，

*// add.js*  
"use strict";  
Object.defineProperty(exports, "\_\_esModule", {  value: true});  
exports["default"] = void 0;  
var \_default = function \_default(a, b) {  return a + b;};  
exports["default"] = \_default;

比如，执行完这段代码后

exports = {  
  \_\_esModule：{  value: true}，  
  default：function \_default(a, b) {  return a + b;}  
}

然后我们把exports对象return出去。

var \_add = \_interopRequireDefault(require("./add.js"));

可见，return出去的值，被\_interopRequireDefault接收，\_interopRequireDefault再返回default这个属性给\_add，因此\_add = function \_default(a, b) { return a + b;}

现在明白了，为什么ES6模块 引入的是一个对象引用了吧，因为exports就是一个对象。

至此，处理；两个关键词的功能就完整了。

const fs = require('fs')  
const path = require('path')  
const parser = require('@babel/parser')  
const traverse = require('@babel/traverse').default  
const babel = require('@babel/core')  
const getModuleInfo = (file)=>{  
    const body = fs.readFileSync(file,'utf-8')  
    const ast = parser.parse(body,{  
        sourceType:'module' *//表示我们要解析的是ES模块*  
    });  
    const deps = {}  
    traverse(ast,{  
        ImportDeclaration({node}){  
            const dirname = path.dirname(file)  
            const abspath = "./" + path.join(dirname,node.source.value)  
            deps[node.source.value] = abspath  
        }  
    })  
    const {code} = babel.transformFromAst(ast,null,{  
        presets:["@babel/preset-env"]  
    })  
    const moduleInfo = {file,deps,code}  
    return moduleInfo  
}  
const parseModules = (file) =>{  
    const entry =  getModuleInfo(file)  
    const temp = [entry]  
    const depsGraph = {}  
    for (let i = 0;i<temp.length;i++){  
        const deps = temp[i].deps  
        if (deps){  
            for (const key in deps){  
                if (deps.hasOwnProperty(key)){  
                    temp.push(getModuleInfo(deps[key]))  
                }  
            }  
        }  
    }  
    temp.forEach(moduleInfo=>{  
        depsGraph[moduleInfo.file] = {  
            deps:moduleInfo.deps,  
            code:moduleInfo.code  
        }  
    })  
    return depsGraph  
}  
*// 新增代码*  
const bundle = (file) =>{  
    const depsGraph = JSON.stringify(parseModules(file))  
    return `(function (graph) {  
        function require(file) {  
            function absRequire(relPath) {  
                return require(graph[file].deps[relPath])  
            }  
            var exports = {}  
            (function (require,exports,code) {  
                eval(code)  
            })(absRequire,exports,graph[file].code)  
            return exports  
        }  
        require('${file}')  
    })(${depsGraph})`  
  
}  
const content = bundle('./src/index.js')  
  
console.log(content);

来执行下，看看效果，确实执行成功。接下来，把返回的这段代码写入新创建的文件中

*//写入到我们的dist目录下*  
fs.mkdirSync('./dist');  
fs.writeFileSync('./dist/bundle.js',content)

至次，我们的手写webpack核心原理就到此结束了。

我们参观下生成的bundle.js文件

发现其实就是将我们早期收集的所有依赖作为参数传入到立即执行函数当中，然后通过eval来递归地执行每个依赖的code。

现在我们将bundle.js文件引入index.html看看能不能执行