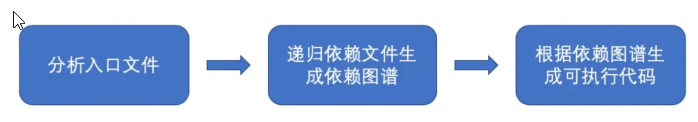
## webpack原理解析

webpack的本质是一个 bundler。作用是将相互依赖、关联的多个文件，打包成一个浏览器可以执行的代码块。想要了解 bundler 的原理，最好的方法就是实现他。下面就实现一个简单的 bundler。

**思路**

https://mmbiz.qpic.cn/mmbiz_png/xydpfFxSShvWObS6W9XJMruew3X6CPtFIMpcbUbL0NuxDNevMWVMo0GbR3kzpbdwbTEqic0ATJicIicicYZYYiaL35w/640?wx_fmt=png&tp=webp&wxfrom=5&wx_lazy=1&wx_co=1



实现 bundler 思路上分这四步：

1. 分析入口文件：入口文件是整个项目的开始，通过生成该文件的抽象语法树进行语法分析。根据抽象语法树可以获取该文件所有的依赖文件。同时要将每一个依赖文件中的内容 由es6 转换成浏览器可以识别的 es5 代码。
2. 生成依赖图谱：以入口文件为“根”，通过依赖文件，递归遍历所有的文件，并对每个文件进行分析，生成依赖图谱。
3. 生成可执行代码：上述操作已经获取到各个文件的依赖关系，同时也获取到每个文件的可执行代码。从入口 文件开始，根据依赖关系依次执行每个文件的代码。

**代码实现**

https://mmbiz.qpic.cn/mmbiz_png/xydpfFxSShvWObS6W9XJMruew3X6CPtFIMpcbUbL0NuxDNevMWVMo0GbR3kzpbdwbTEqic0ATJicIicicYZYYiaL35w/640?wx_fmt=png&tp=webp&wxfrom=5&wx_lazy=1&wx_co=1

需要被打包的三个js文件：index.js、hello.js、world.js

其中 index.js 依赖 hello.js , hello.js 依赖 world.js

*// index.js*

import Hello from "./hello.js";

console.log(Hello);

*// hello.js*

import { world } from "./world.js";

const Hello = `hello ${world}`;

export default Hello;

// world.js

export const world = "world";

下面打包的 js 的代码：

1. 分析入口文件：

const fs = require("fs");const path = require("path");const parser = require("@babel/parser"); *//babel中的解析工具；将字符串转换为ast*const traverse = require("@babel/traverse").default;const babel = require("@babel/core");*/\*\* \* 对某一路径文件·进行·分析 \* @param {} filename 文件名 \*/*const moduleAnalyser = (filename) => { const content = fs.readFileSync(filename, "utf-8"); const ast = parser.parse(content, { *//2.同构babel.parse 生成抽象语法树（AST）* sourceType: "module", }); const dependencies = {}; traverse(ast, {    *//3.找出依赖文件* ImportDeclaration({ node }) { const dirname = path.dirname(filename); *//文件夹路径* const newFile = "./" + path.join(dirname, node.source.value); *//获取相对于跟目录的路径* console.log(newFile); dependencies[node.source.value] = newFile; }, }); *//4.将babel的抽象语法树转换成浏览器可以执行的代码* const { code } = babel.transformFromAst(ast, null, { *//对语法进行编译，从es6语法转换成浏览器可以执行的语法* presets: ["@babel/preset-env"],  });  return { filename, *//入口文件* dependencies, *//依赖文件* code, };};

代码说明如下：

* 先通过 node 中的 fs 模块读取文件内容
* 通过 @babel/parser 包将文件内容生成 AST 抽象语法树，结果如下：
* 通过 @babel/traverse 包解析生成的抽象语法树，获取到依赖文件
* 将文件的代码从 es6 转换成浏览器可以识别的 es5 语法
* 最终将 文件名称、依赖文件、转换后的可执行代码返回 ，格式如下：

2. 生成依赖图谱

*/\*\* \* 生成依赖图谱 \* @param {\*} entry \*/*const makeDependienceGraph = (entry) => { const entryModule = moduleAnalyser(entry); const graphArr = [entryModule]; for (let i = 0; i < graphArr.length; i++) { const item = graphArr[i]; const { dependencies } = item; if (dependencies) { for (let j in dependencies) { graphArr.push(moduleAnalyser(dependencies[j])); } } } const graph = {}; graphArr.forEach((item) => { graph[item.filename] = { dependencies: item.dependencies, code: item.code, }; }); return graph; *//console.log(graph);*};

代码说明如下：

* 通过上一步的 moduleAnalyser 解析入口文件
* 将返回的解析后的对象存放在依赖数组中
* 遍历数组的值，当遍历的对象存在依赖时，解析依赖对象，并存放到依赖数组中，实现对所以文件的遍历，形成依赖图谱
* 将依赖数组中的值转换成对象形式，如下：

3. 生成可执行代码

*/\*\* \* 生成可执行代码 \* @param {\*} entry \*/*const generateCode = (entry) => {const graph = JSON.stringify(makeDependienceGraph(entry));return ` (function(graph){ function load(module){ function localRequire(relativePath){ return load(graph[module].dependencies[relativePath]) } var exports = {}; (function(require,exports,code){ eval(code) })(localRequire,exports,graph[module].code) return exports } load('${entry}') })(${graph}) `;};

下面这一步的目的是，根据上一步生成的依赖图谱，生成一段可以在浏览器中执行的代码，并执行，代码说明如下：

* 新建一个匿名自执行函数，传入上面获取到的依赖图谱。
* 根据入口文件名称，执行依赖图谱中对应的代码，在通过 eval 执行代码的过程中，代码中存在 require 函数和 exports 方法。
* 重写 require 方法名为 localRequire , 取代代码中的 require 方法。改 require 方法的作用是递归执行依赖的 js 文件，直到依赖图谱中的文件都执行完成。

上面部分形成的模版字符串，如下：

(function(graph){ function load(module){ function localRequire(relativePath){ return load(graph[module].dependencies[relativePath]) } var exports = {}; (function(require,exports,code){ eval(code) })(localRequire,exports,graph[module].code) return exports } load('./src/index.js') })({"./src/index.js":{"dependencies":{"./hello.js":"./src/hello.js"},"code":"\"use strict\";\n\nvar \_hello = \_interopRequireDefault(require(\"./hello.js\"));\n\nfunction \_interopRequireDefault(obj) { return obj && obj.\_\_esModule ? obj : { \"default\": obj }; }\n\nconsole.log(\_hello[\"default\"]);"},"./src/hello.js":{"dependencies":{"./world.js":"./src/world.js"},"code":"\"use strict\";\n\nObject.defineProperty(exports, \"\_\_esModule\", {\n value: true\n});\nexports[\"default\"] = void 0;\n\nvar \_world = require(\"./world.js\");\n\nvar Hello = \"hello \".concat(\_world.world);\nvar \_default = Hello;\nexports[\"default\"] = \_default;"},"./src/world.js":{"dependencies":{},"code":"\"use strict\";\n\nObject.defineProperty(exports, \"\_\_esModule\", {\n value: true\n});\nexports.world = void 0;\nvar world = \"world\";\nexports.world = world;"}})

在浏览器中执行，结果如下：

执行成功～

总结

https://mmbiz.qpic.cn/mmbiz_png/xydpfFxSShvWObS6W9XJMruew3X6CPtFIMpcbUbL0NuxDNevMWVMo0GbR3kzpbdwbTEqic0ATJicIicicYZYYiaL35w/640?wx_fmt=png&tp=webp&wxfrom=5&wx_lazy=1&wx_co=1

通过上述几个步骤，编写了个简单的 bundler ，实现了打包的功能。通过这个例子主要想表达 bundler 的实现过程，同时也能够体现出 webpack 的实现原理。

结合之前的两篇文章，依次说明了 ：webpack核心概念、webpack高阶概念、webpack原理。当然知道这些之后，距离成为一名优秀的 webpack 配置工程师还差好远。webapck中的 loader、 plugin 千千万，如何才能更近一步呢？学习像 Create-React-app这些脚手架中优秀的配置也许是个不错的选择，加油！