# 四 webpack原理

## 1 模块打包器

Webpack 是一个模块打包器。它将根据模块的依赖关系进行静态分析，然后将这些模块按照指定的规则生成对应的静态资源。

**A代码拆分**

webpack 有两种组织模块依赖的方式，同步和异步。异步依赖作为分割点，形成一个新的块。在优化了依赖树后，每一个异步区块都作为一个文件被打包。

**B Loader**

webpack 本身只能处理原生的 JavaScript 模块，但是 loader 转换器可以将各种类型的资源转换成 JavaScript 模块。这样，任何资源都可以成为 Webpack 可以处理的模块。

**C 智能解析**

webpack 有一个智能解析器，几乎可以处理任何第三方库，无论它们的模块形式是 CommonJS、 AMD 还是普通的 JS 文件。

**D 插件系统**

webpack 还有一个功能丰富的插件系统。大多数内容功能都是基于这个插件系统运行的，还可以开发和使用开源的 Webpack 插件，来满足各式各样的需求。

**E 快速运行**

webpack 使用异步 I/O 和多级缓存提高运行效率，这使得 Webpack 能够以令人难以置信的速度快速增量编译。

## 2 手动编写打包器

首先，打包工具会从一个入口文件开始，分析里面的依赖，并进一步地分析依赖中的依赖。

所需依赖：

"devDependencies": {

"babel-core": "^6.26.3",

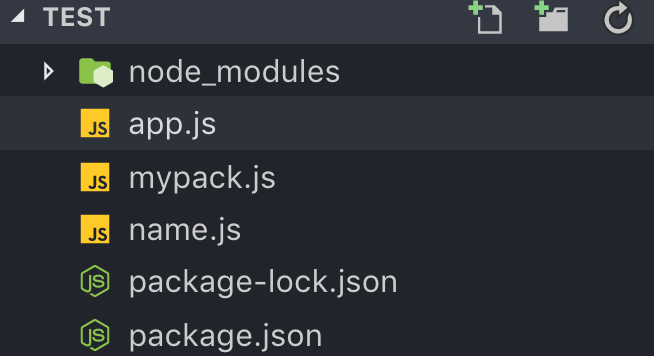
"babylon": "^6.18.0",

"traverse": "^0.6.6",

"babel-preset-env": "^1.7.0"

}

所需文件：



入口文件app.js：

import {name} from './name.js'

let r = `Hello ${name}`;

console.log(r);

export default r;

入口文件引入的依赖文件name.js：

export const name = 'world';

简易打包工具：mypack.js

const fs = require('fs');

const path = require('path');

const babylon = require('babylon');

const traverse = require('babel-traverse').default;

const { transformFromAst } = require('babel-core');

let ID = 0;

*//获取某个文件的依赖函数*

function createAsset(filename) {

const content = fs.readFileSync(filename, 'utf-8');

*//获取文件依赖，手工书写太过笨重，使用babylon可以生成一个模型（ast）*

const ast = babylon.parse(content, {

sourceType: 'module',

});

const dependencies = []; *//保存这个模块依赖的模块的相对路径.*

traverse(ast, { *//遍历ast*

ImportDeclaration: ({node}) => {

dependencies.push(node.source.value);

}

});

const id = ID++; *//为模块分配唯一标识符*

const {code} = transformFromAst(ast, null, { *//兼容低版浏览器*

presets: ['env'],

});

*// 返回有关此模块的所有信息.*

return {id,filename,dependencies,code};

}

*//获取依赖图函数：从入口文件开始*

function createGraph(entry) {

const mainAsset = createAsset(entry); *// 首先解析整个文件*

*//使用`队列{queue}`来解析每个`资产{asset}`的依赖关系.*

*//定义一个只有 入口资产{entry asset} 的数组.*

const queue = [mainAsset];

for (const asset of queue) {

asset.mapping = {};

const dirname = path.dirname(asset.filename);

asset.dependencies.forEach(relativePath => {

*//依赖关系保存的相对路径转变为绝对路径*

const absolutePath = path.join(dirname, relativePath);

*//解析资产,读取其内容并提取其依赖关系.*

const child = createAsset(absolutePath);

*//`asset.mapping`增加属性(child.id)表示一一对应的关系*

asset.mapping[relativePath] = child.id;

*// 将`child`这个资产推入队列,这样它的依赖关系也将被迭代和解析.*

queue.push(child);

});

}

*//队列 就是一个包含目标应用中 每个模块 的数组:*

return queue;

}

*//打包函数*

function bundle(graph) {

let modules = '';

*//返回一个可以在浏览器中运行的包：包含`graph`中每个模块信息的对象*

graph.forEach(mod => {

modules += `${mod.id}: [

function (require, module, exports) { ${mod.code} },

${JSON.stringify(mod.mapping)},

],`;

});

*//`模块的id`作为`key`，数组作为`value` (每个模块中有2个值)*

*//第一个值是用函数包装的每个模块的代码（包装避免污染全局）*

*//第二个值用`stringify`解析模块及其依赖之间的关系(asset.mapping)*

*//最后：实现自调函数的主体.*

*//1 创建一个`require()`⏰函数: 接受 `模块ID` 在构建的`模块`对象查找它.*

*//2 解构`const [fn, mapping] = modules[id]`来获得包装函数、 `mappings`*

const result = `

(function(modules) {

function require(id) { //

const [fn, mapping] = modules[id];

function localRequire(name) { //⏰

return require(mapping[name]); //

}

const module = { exports : {} };

fn(localRequire, module, module.exports);

return module.exports;

}

require(0);

})({${modules}})

`;

return result;

}

*//开始测试*

const graph = createGraph('./app.js');

const result = bundle(graph);

console.log("result====" + result);

输出的结果可以直接在一个JS文件中运行！

## 3 总结

webpack解决了包与包之间潜在的循环依赖难题，同时，按需合并静态文件，以避免浏览器在网络取数阶段的并发瓶颈。除了打包，还可以进一步实现压缩（减少网络传输）和编译（ES6、JSX等语法向下兼容）的功能。

基于对webpack.config.js文件的配置，执行打包时的工作原理，可总结为：把页面逻辑当作一个整体，通过一个给定的入口文件，webpack从这个文件开始，找到所有的依赖文件，进行打包、编译、压缩，最后输出一个浏览器可识别的JS文件。

一个模块打包工具，第一步会从入口文件开始，对其进行依赖分析，第二步对其所有依赖再次递归进行依赖分析，第三步构建出模块的依赖图集，最后一步根据依赖图集构建出最终的代码。