第2章 Webpack 初探

2-1 webpack 究竟是什么？

在浏览器中是不识别 ES Moudule 模块引入的。所以需要借助 webpack 工具来进行翻译

```
// ES Moudule 模块引入方式
import Header from './header.js';
import Sidebar from './sidebar.js';
import Content from './content.js';

new Header();
new Sidebar();
new Content();
```

具体的翻译命令为：npx webpack index.js(入口文件)

2-2 什么是模块打包工具？

Webpack 可以识别翻译打包 ES Moudule，CommonJS，CMD，ADM 语法

CommonJS

```
var Header = require('./header.js');
```

```
module.exports = Header;
```

2-3 Webpack 的正确安装方式

1.全局安装 webpack：npm install webpack webpack-cli -g
不推荐全局安装 webpack 因为若是有项目使用的不是 4 这个版本的话会报错
2.若是全局安装的话打包直接使用 webpack index.js
卸载全局安装的 webpack：npm uninstall webpack webpack-cli -g

3.在项目内安装 webpack：npm install webpack webpack-cli -D/--save-dev
查看版本号：npx webpack -v

4.生成 package.json 文件的时候执行：npm init -y 就会按照默认的配置 package.json 文件了
5.查看 webpack 的所有版本：npm info webpack
6.安装特定的版本号的 webpack：npm install webpack@4.16.5 webpack-cli -D
7.修改 package.json："private": true,//项目是私人的不会上传到 npm 上去

2-4 使用 Webpack 的配置文件

1.配置了 webpack.config.js 文件的 entry 和 output 的话在打包的时候就刻意直接使用 npx webpack。不需要带上入口文件名了
2.默认的配置文件必须叫 webpack.config.js
3.若默认的配置文件不叫 webpack.config.js 的话。也可以在打包的时候将配置文件以参数的形式传进去，例如：npx webpack --config webpackconfig.js
4.在 package.json 中配置，就可以使用 npm run bundle 来打包文件

```
"scripts": {
    "bundle":"webpack"
  },
```

该命令会先在该项目下查找 webpack

总结：3 种 webpack 打包方式

```
webpack index.js
npx webpack index.js
npm run bundle -> webpack
```

webpack-cli 此工具用于在命令行中运行 webpack

2-5 浅析 Webpack 打包输出内容

```
E:\练习文档\webpack-test\node_modules\webpack\bin\webpack.js
Hash: 723fb1cb4ea78ee426c0
Version: webpack 4.40.2
Time: 585ms
Built at: 2019-09-19 15:38:28
    Asset       Size  Chunks             Chunk Names
bundle.js  1.29 KiB       0  [emitted]  main
Entrypoint main = bundle.js
[0] ./src/index.js + 3 modules 754 bytes {0} [built]
    | ./src/index.js 183 bytes [built]
    | ./src/header.js 184 bytes [built]
    | ./src/sidebar.js 198 bytes [built]
    | ./src/content.js 189 bytes [built]

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for th
is value. Set 'mode' option to 'development' or 'production' to enable defaults
for each environment.
You can also set it to 'none' to disable any default behavior. Learn more: https
://webpack.js.org/configuration/mode/
```

报错的意思是没有指定打包的环境。但是其实已经默认设置为 mode:'production'
若是显示的声明则不会报错。

mode:'production'打包的文件会被压缩

mode:'development'打包之后的文件不会被压缩

第3章 Webpack 的核心概念

3-1 什么是 loader

file-loader 打包图片的包
npm install file-loader -D

loader 让 webpack 能够去处理那些非 JavaScript 文件（webpack 自身只理解 JavaScript）。loader 可以将所有类型的文件转换为 webpack 能够处理的有效模块，然后你就可以利用 webpack 的打包能力，对它们进行处理。

3-2 使用 Loader 打包静态资源（图片篇）

url-loader 会将图片转换成一个 base64 位的字符串，而不会生成一个图片文件

```
但是可以设置：limit:2048//如果图片大小超过 2048 将不压缩成 base64
```

3-3 使用 Loader 打包静态资源（样式篇 - 上）

1.打包 css 需要两个 loader：css-loader，style-loader
css-loader 会分析出 css 之间的关系，将几个 css 文件合并成一个 css 文件。
style-loader 会将 css-loader 生成的 css 文件挂载到页面的 head 部分

2. 打包 sass 文件需要两个 loader：sass-loader，node-sass
3. Loader 的执行时有顺序的。从下到上，从右到左（一组链式的 loader 将按照相反的顺序执行）

```
use: [
  'style-loader',
  'css-loader',
  'sass-loader'
]
```

4. 给 css 增加厂商前缀需要 Postcss-loader，autoprefixer
新建 postcss.config.js 文件。在里面配置

```
module.exports = {
    plugins:[
        require('autoprefixer')
    ]
}
```

```
use: [
  'style-loader',
  'css-loader',
  'sass-loader',
  'postcss-loader'
]
```

3-4 使用 Loader 打包静态资源（样式篇 - 下）

1.比如打包 index.sass 时遇到引进的另外的 sass 时。有时候不会走 postcss-loader 和 sass-loader。所以要加上 importLoaders:2。让他在打包时又遇到的 sass 文件必须走前面两步

```
test: /\.scss$/,
use: [
    'style-loader',
    {
        loader:'css-loader',
        options:{
            importLoaders:2,
            modules:true
        }
    },
    'sass-loader',
    'postcss-loader'
]
```

2.模块化引进 css。要在 css-loader 的配置项里面增加 modules:true。还要再引进和使用的时候增加模块名

```
import avatar from './avatar.jpg';
import './index.scss';
import createAvatar from './createAvatar';
```

```
import avatar from './avatar.jpg';
import style from './index.scss';
import createAvatar from './createAvatar';

createAvatar();

var img = new Image();
img.src = avatar;
img.classList.add(style.avatar);

var root = document.getElementById('root');
root.append(img);
```

使用 webpack 打包字体文件使用 file-loader

3-5 使用 plugins 让打包更便捷

html-webpack-plugin：会在打包结束后，自动生成一个 html 文件，并把打包生成的 js 自动引入到这个 html 文件中。但是这个 html 文件没有内容。只引进了 js 文件。
所以可以在 src 目录下创建一个 html 文件作为模板。然后再 plugins 下配置

```
plugins:[new HtmlWebpacePlugin({
        template:'src/index.html'
    })],
```

每次打包前 webpack 的 output.path 目录中的所有文件将被删除一次，但是目录本身不会。
clean-webpack-plugin

```
plugins: [new HtmlWebpacePlugin({
        template: 'src/index.html'
    }), new CleanWebpackPlugin(['dist'])],
```

版本问题
以上的用法会报错。报 `CleanWebpackPlugin is not a constructor` 和参数错误
应该改为

```
引用：const {CleanWebpackPlugin} = require('clean-webpack-plugin');
使用：new CleanWebpackPlugin()
```

3-6 Entry 与 Output 的基础配置

打包两个文件

```
entry: {
        main:'./src/index.js',
        sub:'./src/index.js'
    }
```

```
output: {
         publicPath:'http://cdn.com.cn',//若将打包之后的文件放在 cdn 上的话，可以在这里加入 cdn 的地址。
        filename: '[name].js', //输出的名字为 main.js 和 sub.js
        path: path.resolve(__dirname, 'dist')
    }
```

3-7 SourceMap 的配置

若是没有 source-map 只能知道打包之后的文件哪里出错了。而不能知道源文件中出错的位置
Source-map 是一个映射关系，他知道 dist 目录下 js 文件中的错误代码对应的是 src 目录下相应文件的错误代码的位置

```
devtool:'source-map',
```

会在打包目录 dist 下生成 main.js.map

Inline-source-map 会将映射内容放在打包之后的 main.js 文件里。所以不会生成 main.js.map 文件。会将出错的行列都标出来
inline-cheap-source-map 只会标出出错的行。不会标出列

推荐在开发过程中使用

```
mode: 'development',
devtool:'cheap-module-eval-source-map',
```

线上代码使用

```
mode: 'production',
devtool:'cheap-module-source-map',
```

3-8 使用 WebpackDevServer 提升开发效率

1. A.只要改变 src 目录下的代码，就会自动打包
2. B.自动在浏览器上刷新 index.html
3. 在 package.json 中添加 watch。（套餐 A）

```
"scripts": {
    "bundle": "webpack",
    "watch":"webpack --watch"
},
```

4. 在 webpack.config.js 中添加并且安装 webpack-dev-server（A+B）

```
devServer:{
        contentBase:'./dist'
},
```

在 package.json 的 scripts 中添加 `start`

```
"start":"webpack-dev-server",
```

5. 在 2 的基础上在 webpack.config.js 的 devServer 增加。当执行 npm run start 命令的时候会直接打开 localhost:8080

```
open:true
```

注：这种方法生成的打包文件不会放在 dist 目录下。而是放在电脑的内存中

可以配置服务的端口号

```
port:8080
```

扩展：
如果你有单独的后端开发服务器 API，并且希望在同域名下发送 API 请求 ，那么代理某些 URL 会很有用。

```
proxy: {"/api": "http://localhost:3000"}
```

自己搭建一个服务器

1 在 package.json 中添加 middleware

```
"server":"node server.js"
```

2 安装 webpack-dev-middleware express

3 在 output 中添加

```
publicPath:'/',
```

4 在根目录下增加 server.js

```
const express = require('express')
const webpack = require('webpack')
const webpackDevMiddleware = require('webpack-dev-middleware')
const config = require('./webpack.config.js')
const complier = webpack(config)//webpack 编译

const app = express()
app.use(webpackDevMiddleware(complier,{
    publicPath:config.output.publicPath
}))
app.listen(3000, () => {
    console.log('server in 3000')
})
```

3-9 Hot Module Replacement 热模块更新（1）

参考：https://juejin.im/post/5d8b755fe51d45781332e919

只要代码一改变就会重新渲染页面。需要如果只改变 css 文件的话就刷新 css 文件。其他的不改变

1.在 webpack.config.js 的 devServer 增加

```
hot:true//开启 Hot Module Replacement 的功能
hotOnly:true//即便 Hot Module Replacement 的功能没有生效。也不让浏览器自动刷新
```

2.引进 webpack

```
const webpack = require('webpack')
```

3.在 webpack.config.js 的 plugins 增加

```
new webpack.HotModuleReplacementPlugin()
```

若是改变 js 文件的话还需要在入口文件中增加（css-loader 底层已经实现了这段代码，所以 css 文件的改变不用做此配置。使用 vue 脚手架来编写项目的时候也不用自己编写这段代码，因为 vue-loader 做了此项的配置）

```
if(module.hot){//若是没有这段话的也可以不用设置以上的 hotOnly:true 属性
    module.hot.accept('./number',()=>{
        number();
    })
}
```

3-11 使用 Babel 处理 ES6 语法（1）

1.安装 babel-loader @babel/core
2.在 webpack.config.js 的 module.rules 中配置规则

```
{
      test: /\.js$/,
      exclude: /node_modules/,//如果你的 js 文件是在 node_modules 目录下的。那我就不用使用 babel-loader 来处理 js 了
      loader: "babel-loader"
}
```

6．安装@babel/preset-env，使用 babel-loader 只是作为和 webpack 通信的桥梁。实际上 babel-loader 并不会把 es6 代码翻译成 es5.所以还需要借助@babel/preset-env
7．配置 options

```
options:{
    presets:['@babel/preset-env']
}
```

8．以上操作只是对一些语法做了转变。但是像 map，Promise 这样的语法还是没有进行 es5 转换。所以还需要借助@babel/polyfill
9. 在 index.js 文件中引进

```
import "@babel/polyfill";//配置了 useBuiltIns:'usage'的话就不需要这一句了，会自动引入
```

10．以上操作打包出来的 main.js 会变得很大。因为他把全部的 es6 的 polyfill 都写出来了。根据业务代码增加 polyfill

```
presets:[['@babel/preset-env',{
     useBuiltIns:'usage',
      "corejs": "3"//需要安装 core-js@3 --save，然后在这里引进这一句。不然会报错
}]]
```

若是开发类库或者是第三方组件的话这种方法不适用，可以使用 transform-runtime 方法
https://www.babeljs.cn/docs/babel-plugin-transform-runtime

1.安装@babel/plugin-transform-runtime 和@babel/runtime
2.配置

```
"plugins": [
    [
        "@babel/plugin-transform-runtime",
         {
             "absoluteRuntime": false,
              "corejs": 2,
              "helpers": true,
              "regenerator": true,
              "useESModules": false
         }
    ]
]
```

扩展：

1.配置目标浏览器

```
presets:[['@babel/preset-env',{
    targets:{
        chrome:'67'
    },
    useBuiltIns:'usage'
}]]
```

2.可以将 babel 的配置项 options 中的内容放在.babelrc 文件中

第4章 Webpack 的高级概念

4-1 Tree Shaking 概念详解

Es6 语法转换只转换业务上有用到的。没用到的不进行转换。

Tree Shaking 只支持 es module 的语法。因为 es module 的底层是一种静态的引入。而 require 是一种动态的引入

1.当 mode: 'development'时，在 webpack.config.js 配置

```
optimization:{
    usedExports:true
},
```

2.在 package.json 中配置（因为像@babel/polly-fill 这样的库并没有导出任何模块。只是在 window 里面绑定了一些方法。这样子用 Tree Shaking 打包的时候就会忽略掉这个库）

```
"sideEffects":["@babel/polly-fill"],
```

若是没有不使用 Tree Shaking 打包的文件时可以写

```
"sideEffects":false,
```

一般遇到 css 文件也不使用 Tree Shaking

```
"sideEffects":"["*.css"]"
```

注：在'development'模式下，即使使用了 Tree Shaking 来打包也不会将你的代码直接从 main.js 中去除掉，只会提示一下

```
/*! exports provided: add, minus */
/*! exports used: add */
```

因为开发模式下的代码会进行调试，若是去除掉的话如果有 source map 报错信息的话对应的行数就会出错了

若是在 mode: 'production'模式下的时候。Tree Shaking 会自动配置好 optimization，但是 sideEffects 还是需要自己配置的

4-2 Develoment 和 Production 模式的区分打包

1.可以将线上和开发环境的配置文件分为两个 webpack.dev.js 和 webpack.prod.js
在 package.json 中配置 scripts

```
"scripts": {
    "build": "webpack --config webpack.prod.js",
    "dev": "webpack-dev-server --config webpack.dev.js"
}
```

改动 webpack 配置文件的话需要自己手动重启下服务。
2.因为 webpack.dev.js 和 webpack.prod.js 中有太多相同的代码。所以可以新建一个 webpack.common.js 文件，将相同的代码写在里面
3.需要将 webpack.common.js 和 webpack.dev.js 或者 webpack.prod.js 合并起来。所以需要依靠 webpack-merge 插件
4.在 webpack.dev.js 和 webpack.prod.js 引进

```
const merge = require('webpack-merge')
const commonConfig = require('./webpack.common.js')
```

5.进行合并导出

```
module.exports =merge(commonConfig,devConfig)
```

6.可以将所有 webpack 配置放在一个 build 文件里。然后将 package.json 中的 scripts 改成

```
"scripts": {
    "build": "webpack --config ./webpack/webpack.prod.js",
    "dev": "webpack-dev-server --config ./webpack/webpack.dev.js"
}
```

4-3 Webpack 和 Code Splitting（1）

Code Splitting：代码分割
在开发环境下想要在每次改动代码之后重新打包代码。所以可以在 package.json 中配置

```
"dev-build":"webpack --config ./build/webpack.dev.js",
```

手动分割代码
例如：安装 loadsh --save，是一个 JavaScript 工具库（http://lodash.think2011.net/）

```
import _ from 'lodash';
console.log(_.join(['a','d','c'],'***'));
```

这样子写的话打包会将 lodash 工具库也打包进 index.js 文件里面。这样会导致如果工具库比较大的话如果更改了业务代码下次用户加载 index.js 文件的话相对来说会比较慢。
所以可以新建一个 lodash.js 文件，然后将 lodash 挂载在 window._上

```
import _ from 'lodash';
window._ = _;
```

然后再 webpack.common.js 的配置文件里面增加入口文件 lodash

```
entry: {
        lodash:'./src/lodash.js',//需要写在 main 上面。不然页面会报错。
        main:'./src/index.js'
},
```

打包之后的文件为 index.html，main.js，lodash.js

4-4 Webpack 和 Code Splitting（2）

Code Splitting 不是 webpack 所特有的。但是 webpack 有一些插件可以很好的进行 Code Splitting 的代码分割。

使用 webpack 插件进行代码分割

1.在使用的地方直接可以引进工具库

然后在 webpack.common.js 里面配置。打包之后它就会自动进行代码分割了

```
optimization:{
        splitChunks:{
            chunks:'all'
        }
},
```

同步的加载代码

```
1    import _ from 'lodash';
2    console.log(_.join(['a','d','c'],'***'));
```

打包之后的文件为 index.html，main.js，vendors~main.js

异步的方式的引进工具类

不需要配置 `optimization`，直接异步引进就可以了

```
// 异步引进代码
function getComponent(){
    return import('lodash').then(({default:_})=>{
        var element = document.createElement('div');
        element.innerHTML = _.join(['xie','ting'],'-');
        return element;
    })
}

getComponent().then(element=>{
    document.body.appendChild(element);
})
```

打包之后的文件为 index.html，main.js，0.js

4-5 SplitChunksPlugin 配置参数详解（1）

Webpack 中的代码分割底层使用了 SplitChunksPlugin 插件

1.异步加载工具库的时候更改工具库的 0.js 名字

```
return import( /* webpackChunkName:'lodash' */ 'lodash').then(({default:_})=>{
        var element = document.createElement('div');
        element.innerHTML = _.join(['xie','ting'],'-');
        return element;
})
```

这样子打包之后的文件名字是 vendors~lodash.js

2.在 webpack.common.js 里面配置，这样打包出来的工具库的文件名就是 lodash.js

```
splitChunks:{
            chunks:'all',//Code Splitting
            cacheGroups:{
                vendors:false,
                default:false
            }
}
```

注：如果没有 `splitChunks` 的配置项为空，也是可以的。因为有默认配置内容

```
splitChunks: {

    chunks: "async",//在做代码分割的时候只对异步代码生效.若想对同步异步的代码都进行分割可以改为‘all’

    minSize: 30000,//当引入的库大小超过 30000 的时候，就会进行分隔代码

    minChunks: 1,//当引入的模块代码至少被用了 1 次的时候才进行代码分割

    maxAsyncRequests: 5,//同时加载的模块数。如果页面中加载了 10 个模块。当打包前 5 个的时候会进行代码分割。后面 5 个就不会进行处理了

    maxInitialRequests: 3,//入口文件进行加载的时候，js 最多只能被分割成 3 个

    automaticNameDelimiter: '~',//组和文件之间名字的连接符

    name: true,

    cacheGroups: {//缓存组。同步代码会在打包的时候会走这里。符合要求的代码会放在同一个组里

        vendors: {
```

```
            test: /[\\/]node_modules[\\/]/, //同步代码会在打包的时候检测该工具库是否在node_modules 目录下。打包之后的文件名为 vendors~main.js。Vendors：符合 vendors 配置的文件。Main：入口文件为 main.js。

            //filename:'vendors.js',//更改打包之后的文件的命名为 vendors.js

            priority: -10//组之间的优先级

        },

    default:{//若是文件不符合 vendors 的配置，则会走这里，打包之后的文件叫 dafault~main.js

            minChunks: 2,

            priority: -20,

            reuseExistingChunk: true，//如果一个模块之前已经被打包过，之后再遇到的时候就不会再次打包了

            //filename：'common.js'

        }

    }}
```

4-7 Lazy Loading 懒加载，Chunk 是什么？

通过异步加载工具库来实现懒加载，这样只有当时点击的时候才加载 lodash.js

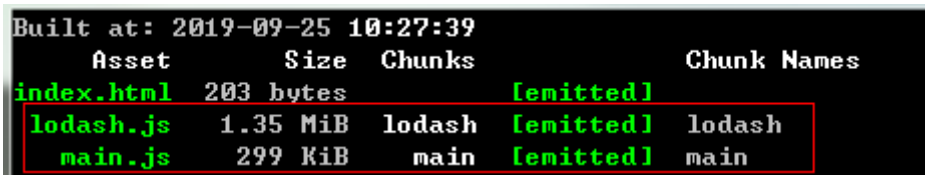
```
// 异步引进代码
function getComponent(){
    return import( /* webpackChunkName:'lodash' */ 'lodash').then(({default:_})=>{
        var element = document.createElement('div');
        element.innerHTML = _.join(['xie','ting'],'-');
        return element;
    })
}
document.addEventListener('click',()=>{
    getComponent().then(element=>{
        document.body.appendChild(element);
    })
})
```

Node.js ES7 异步函数

```
async function getComponent() {
    const {
```

```
        default: _
    } = await import( /* webpackChunkName:'lodash' */ 'lodash');
    const element = document.createElement('div');
    element.innerHTML = _.join(['xie', 'ting'], '-');
    return element;
}
```

打包之后生成了几个 js 文件就有几个 chunk



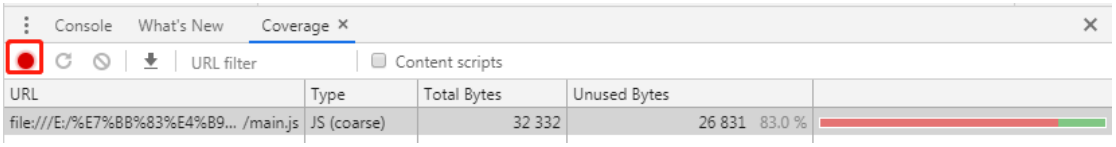
4-8 打包分析，Preloading，Prefetching

打包分析：
Webpack 分析工具的仓库 https://github.com/webpack/analyse
在 package.json 文件里在打包命令上加上

```
"dev-build": "webpack --profile --json > stats.json --config ./build/webpack.dev.js
",
```

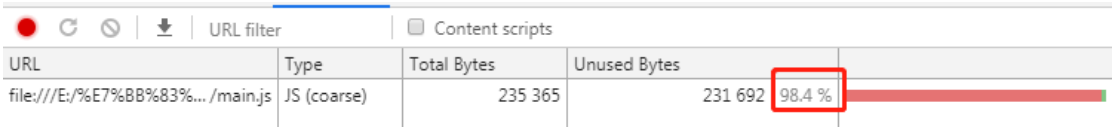
打包之后的将会生成 `stats.json` 文件。将其放进 http://webpack.github.com/analyse 网站里面进行分析

在控制台中按下 Ctrl+shift+p 可以调出输入面板。输入 coverage 调出面板，点击圆圈之后可以看到 main.js 一开始加载的时候的利用率。



将交互的代码放在一个异步加载的模块里面去写。然后使用 import 进行加载

```
document.addEventListener('click', () => {
    import('./click.js').then((func) => {
        func();
    })
})
```



Prefetching：在页面上的内容加载完成之后的空闲时间里偷偷的将一部的代码下载下来。

```
document.addEventListener('click', () => {
    import(/* webpackPrefetch:true */'./click.js').then((func) => {
        func();
    })
})
```

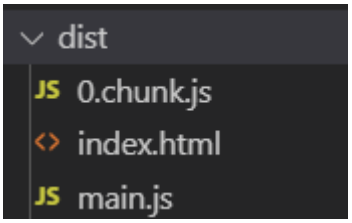
Preloading：是和主文件一起加载下来的

4-9 CSS 文件的代码分割

在 webpack.common.js 的 output 里面增加

```
output: {
        filename: '[name].js', //打包后的文件命名，默认为 main.js
        chunkFilename:'[name].chunk.js',
        path: path.resolve(__dirname, '../dist') //打包后的文件放到该文件夹下。注意要使用绝对路径
}
```

打包之后的文件为



因为 main.js 是入口文件(entry)所以会走 filename 配置。间接加载的文件会走 chunkFileName 配置

若是不做 css 代码的分割的话 webpack 会将 css 代码放在 main.js 文件里面。

1. 安装 npm install --save-dev extract-text-webpack-plugin@next （如果安装 extract-text-webpack-plugin 的话打包会报错）

2.在 webpack.prod.js 中引进

```
const ExtractTextPlugin = require("extract-text-webpack-plugin");
```

3.在 module.rules 中添加

```
{
            test: /\.css$/,
            use: ExtractTextPlugin.extract({
                fallback: "style-loader",
                use: "css-loader"
            })
}
```

4.在 plugins 中声明

```
new ExtractTextPlugin("styles.css")
```

执行 build 之后还是不能将 css 单独打包出来

那是因为之前在 package.json 中配置了 `"sideEffects": false`

所以 tree shinking 没有检测到该模块有被引用所以没有打包出来。（index.html 文件中的背景没有变成红色）

5.将`"sideEffects"`的配置改成`["*.css"]`

然后将 dev 文件中的 optimization 配置放在 common 文件中

```
optimization: {
        usedExports: true,
}
```

打包之后的 css 文件并没有进行压缩。

6.安装 optimize-css-assets-webpack-plugin 进行 css 代码的压缩和合并

7.在 prod 文件中引进

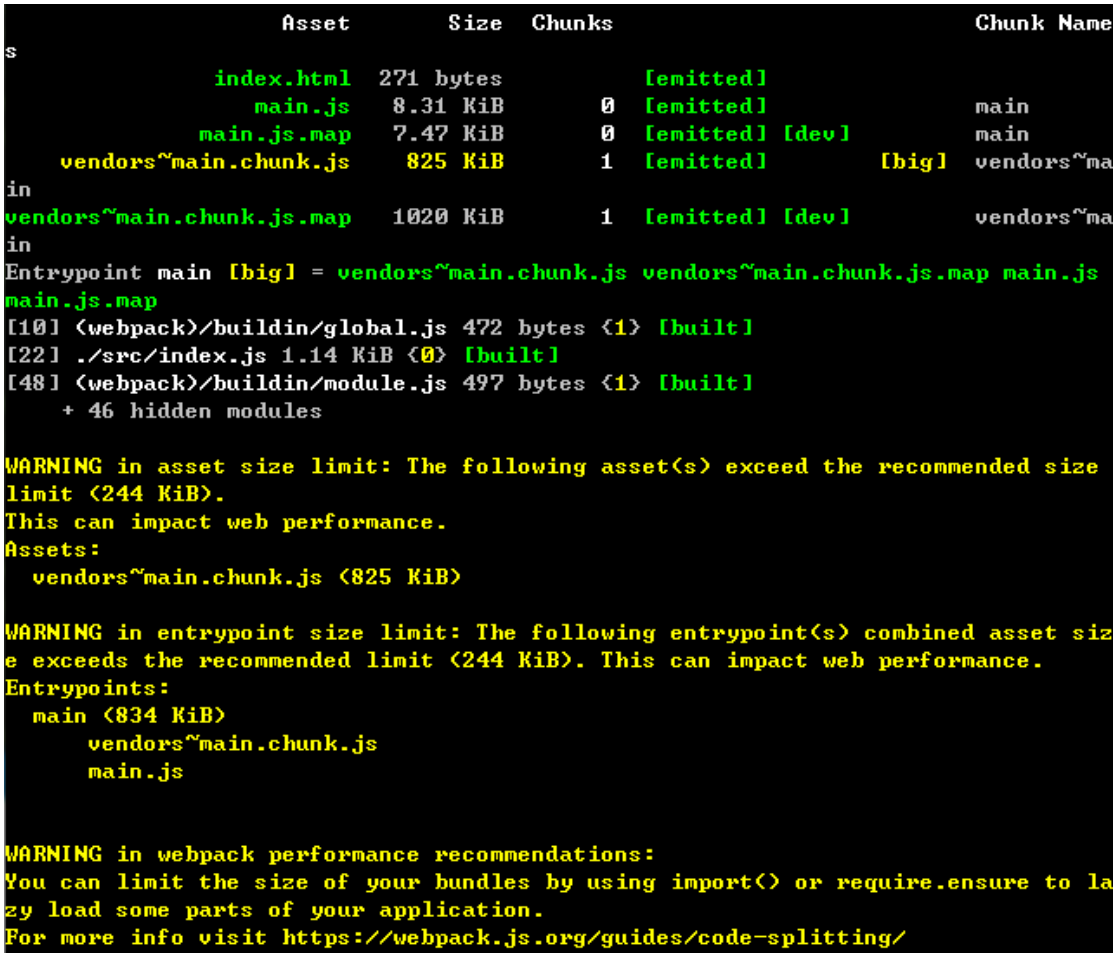
```
const optimizeCss = require('optimize-css-assets-webpack-plugin');
```

8.配置

```
optimization: {
        minimizer:[new optimizeCss({})]
    }
```

4-10 Webpack 与浏览器缓存(Caching)

当打包的项目 js 文件 825kb 大于要求的 144kb，会报这个警告



可以在 common 中加入以下命令，关掉警告

```
performance:false
```

若上线的代码每次更新之后的名字都是相同的，用户有可能会因为浏览器的缓存看不到最新的页面

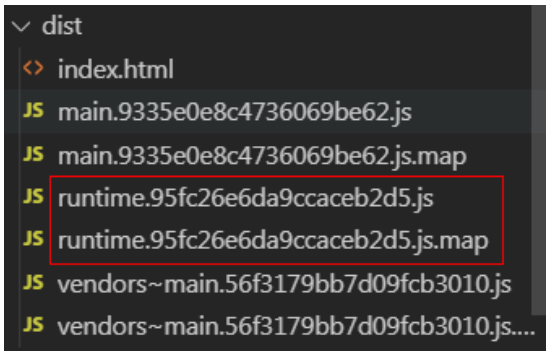
所以可以将 prod 文件的 output 配置改成

```
output: {
        filename: '[name].[contenthash].js', //打包后的文件命名，默认为 main.js
        chunkFilename: '[name].[contenthash].js'
    }
```

Contenthash 要是文件内容没改变 hash 值就不会发生变化

```
runtimeChunk:{
            name:'runtime'//低版本的 webpack 打包的时候没改变文件内容，但是打包的时候会生
成新的 contenthash。可以增加这一句来做兼容
        }
}
```

配置了这一句打包之后会生成一个 runtime.js 文件



Main.js 放置的是业务代码，vendors.js 放置的是工具库打包之后的代码。两者也是有关联的。在 webpack 中将关联的代码归结为 manifest。Manifest 存在于 main.js 和 vendors.js.Manifest 在旧版的 webpack 打包的时候可能会有差异，所以 `Contenthash` 会发生变化。当配置了 `runtimeChunk` 之后就会将相关联的 manifest 放在 runtime 文件中了

4-11 Shimming 的作用

模块内定义的变量只能在模块之内引用。在模块之外是用不了的。例如：父模块中定义并引入了 jquery，但是在子模块中若是也使用 jquery 的话需要重新定义并引入。这样保证了模块和模块之间没有耦合

若是引用的第三方库中使用了 jquery。Jquery 需要我们自己手动引入的话可以借助 shimming 在 common 中引入 webpack 并且在 plugins 中配置

```
new webpack.ProvidePlugin({
            $:'jquery'
})
```

扩展：
若想要将 lodash 中的_.join()改成_join()
可以配置

```
new webpack.ProvidePlugin({
            $:'jquery',
            _join:['lodash','join']
})
```

默认模块中的 this 指向的是自身，而不是 window。想要改变可以安装 imports-loader
在 module 中的 rules 中配置

```
{
            test: /\.js$/,
            exclude: /node_modules/, //如果你的 js 文件是在 node_modules 目录下的。那我就不用使用 babel-loader 来处理 js 了
            use:[{
                loader:"babel-loader"
            },{
```

```
                loader:"imports-loader?this=>window"
            }]
}
```

4-12 环境变量的使用方法

改变【4-2 Develoment 和 Production 模式的区分打包】中的文件区分开发模式和线上模式的合并方法
在 dev 和 prod 中只需要把配置导出来
在 common 中引入

```
const merge = require('webpack-merge')
const devConfig = require('./webpack.dev.js')
const prodConfig = require('./webpack.prod.js')
```

```
module.exports = (env,argv) => {
    if(argv.mode === 'production') {
        return merge(commonConfig, prodConfig);
    }else {
        return merge(commonConfig, devConfig);
    }
}
```

将 package.json 中的配置改为

```
"scripts": {
    "build": "webpack --config ./build/webpack.common.js --mode=production",
    "dev": "webpack-dev-server --config ./build/webpack.common.js",
    "dev-build": "webpack --profile --json > stats.json --config ./build/webpack.co
mmon.js"
}
```

第5章 Webpack 实战配置案例讲解

5-1 Library 的打包

开发一个组件库或者函数库
先在一个空的文件夹中初始化输入 npm init -y
安装 webpack webpack-cli
别人引用我们的库可能有非常多的办法，例如

```
import library from 'library'

const library = require('library');

require(['library'], function(){

})
```

在 webpack.config.js 的 output 中加上

```
    libraryTarget:'umd'//不管通过任何形式来引用我的库都是支持的。
    若是'this'的话，library 挂载在全局 this 下面。this.library
    还可以填'window','global'//node 环境下
```

如果使用 script 标签来引入 library 库。希望可以使用 library.math 这样全局变量的形式来使用库。可以在 webpack.config.js 的 output 中加上

```
    library:'library',
```

如果在我们自己编写的库里面引进了 lodash。然后用户引用我们的库之后自己也引用了 lodash。这样代码里面就会有两份 lodash 了。
这时候可以在 webpack.config.js 的 modules 中配置

```
    externals:["lodash"],//打包的时候忽略 lodash 库。不要把他打包进我们的项目里面
```

如果要打包，在 package.json 中将 main 改成

```
    "main": "./dist/library.js",
```

在 npm 官方网站中注册并申请一个账号。然后在本地的命令行中输入 npm adduser,然后输入用户名和密码
然后可以输入 npm publish 上传到 npm 中
别人下载我们这个库的话直接输入 npm install library
库的名字不能和别人的相同。如果要更改可以在 package.json 中更改`"name"`

5-2 PWA 的打包配置

PWA=progressive web application
模拟后端的服务器。安装 http-server
然后在 package.json 的 scripts 中配置

```
"start":"http-server dist"//在 dist 目录下开启一个 http-server
```

这样子的服务当服务器挂掉的时候页面就访问不到了。所以可以使用 PWA 技术来缓存页面

安装 wordbox-webpack-plugin。上线的代码才需要 PWA 处理。本地的代码是不需要的。所以只需要修改 webpack.prod.js 的代码就可以了

1. 引进

```
const workboxPlugin = require('workbox-webpack-plugin')
```

2. 在 plugins 中加入

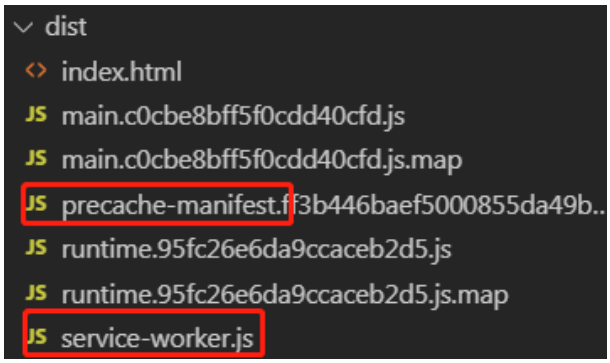
```
new workboxPlugin.GenerateSW({
            clientsClaim:true,
```

```
            skipWaiting:true
})
```

3.在 src 的 index.js 加入

```
if ('serviceWorker' in navigator) {//如果浏览器支持'serviceWorker' 功能的话
    window.addEventListener('load', () => {
        navigator.serviceWorker.register('/service-worker.js')
            .then(registration => {
                console.log('service-worker registed')
            }).catch(error => {
                console.log('service-worker register error')
            })
    })
}
```

4.打包之后 dist 目录下多了这两个目录



5. 运行 npm run start.打开网页 localhost:8080/index.html(后面一定要加 index.html。不然会报无法找到该网页)

5-3 TypeScript 的打包配置

安装 ts-loader 和 typescript

使用 `ts-loader` 来处理 TypeScript 文件

```
const path = require('path');

module.exports = {
    mode: 'production',
    entry: './src/index.tsx',
    module: {
        rules: [{
            test: /\.tsx?$/,
            use: 'ts-loader',
            exclude: /node_modules/
        }]
    },
    output: {
```

```
        filename: 'bundle.js',
        path: path.resolve(__dirname, 'dist')
    }
}
```

使用 ts-loader 来打包 typescript 文件的时候。需要在根目录下建立 tsconfig.json 文件

```
{
    "compilerOpitons": {//编译时的配置
        "outDir": "./dist",//使用ts-loader打包之后的文件存放的位置。可以不写,因为webpack配置中写了
        "module": "es6",//使用的是 es6 模块引入的方式
        "target": "es5",//打包之后的转换成 es5 的代码
        "allowJs": true,//允许在 typescript 文件中使用 js 语法
    }
}
```

在 typescript 中使用 lodash 库中的方法时，编译器并不会报错和语法提示。

可以安装 npm install @types/lodash --save-dev

在该网站里搜索是否有支持的类型文件 https://microsoft.github.io/TypeSearch/

5-4 使用 WebpackDevServer 实现请求转发

在 dev 文件的 devServer 里配置

```
devServer: {
        proxy:{
            '/react/api':'http://www.dell-lee.com'
        }
    }
```

当使用`/react/api`开头的接口请求数据的时候，就会代理到 http://www.dell-lee.com，network 里面显示的还是 localhost：8080

```
proxy:{
    '/react/api':{
        target:'http://www.dell-lee.com',
        secure:false,//实现对 https 网址的转发
        pathRewrite:{
            'header.json':'demo.json'//如果拿 header.json 中的数据，实际上是去 demo.json 中拿
        },
        bypass:function(req,res,proxyOptions){
            if(req.headers.accept.indexOf('html') !== -1){//如果请求的链接是 html。就不会进行代理。该返回什么就返回什么
                    return false;
             }
```

```
        },
        changeOrigin:true,//有些网站对 origin 进行了限制，防止爬虫
        historyApiFallback:true,//若发现请求的路径后端服务器中没有。会转化成对根路径的请求


    }
}
```

如果要代理根路径，默认是不支持的，需要将 index 设置为”

dev-server 实际是底层上使用了 http-proxy-middleware
dev-server 只有在开发环境中可使用

5-5 WebpackDevServer 解决单页面应用路由问题

```
historyApiFallback:true,//若发现请求的路径后端服务器中没有。会转化成对根路径的请求
```

5-6 EsLint 在 Webpack 中的配置（1）

安装 eslint 来规范我们项目中的代码
执行 npx eslint --init.
执行 npx eslint src//检查 src 目录下的代码是否符合 eslint 规范
安装 babel-eslint
在.eslintrc.js 中配置

5-7 EsLint 在 Webpack 中的配置（2）

5-8 webpack 性能优化(1)

1. 跟上技术的迭代（Node, Npm, Yarn）

2. 在尽可能少的模块上应用 Loader
3. Plugin 尽可能精简并确保可靠
4. resolve 参数合理配置

在 common 文件里配置

```
resolve:{
        extensions:['.js','.jsx']//当引入一个目录下的模块的时候，会先到这个目录下找对应的以 js 为后缀的文件。若是找不到就找以 jsx 为后缀的文件。
        mainFiles:['index','child']//当引入一个目录的时候(‘./child/’)。先尝试去找 index 文件。找不到再找 child 文件
        alias:{
            xieting:path.resolve(__dirname,'../src/child')//当看到xieting这个模块的时候，会按照这个路径去找。是这个路径的别名
          }
    },
```

5. 使用 DllPlugin 提高打包速度

每次打包 webpack 都会将我们引入的模块打包进 vendor.js 里面。但时这些模块的内容我们是不会去改变的。

目标：第三方模块只打包一次

新建 webpack.dll.js 文件。

```
const path = require('path')
module.exports = {
    mode:'production',
    entry:{
        vendors:['lodash','jquery']
    },
    output:{
        filename:'[name].dll.js',
        path:path.resolve(__dirname,'../dll'),
        library:'[name]'
    }
}
```

在 package.json 的 script 加入

```
"build:dll":"webpack --config ./build/webpack.dll.js"
```

打包之后会生成 vendors.dll.js 文件

安装 add-asset-html-webpack-plugin 插件（在 html 上增加一些静态资源）

在 common 文件上配置

```
new AddAssetHtmlWebpackPlugin({
            filepath:path.resolve(__dirname,'../dll/vendors.dill.js')
})
```

这样在页面上就可以使用 vendors 这个全局变量来引用模块的内容了

我们在项目中引入第三方模块的时候，要去使用 dll 文件引入

在 vendors.dll.js 文件中配置

```
plugins:[
        new webpack.DllPlugin({//分析映射关系
            name:'[name]',
            path:path.resolve(__dirname,'../dll/[name].manifest.json')
        })
]
```

在 webpack.common.js 文件中配置

```
New webpack.DllReferencePlugin({
            manifest:path.resolve(__dirname,'../dll/vendors.manifest.json')
})
```

当你在项目中引入第三方模块的时候，打包前会先在 `vendors.manifest.dll.json` 中查找有没有该库的映射关系。如果有该模块的映射关系。打包的时候就不会将该模块进行打包了。使用的时候会直接从全局变量中去拿

优化：将库打包成多个文件

1.将 webpack.dll.js 文件中的 entry 改成

```
entry:{
        vendors:['lodash'],
        jquery:['jquery']
},
```

1. 在 webpack.common.js 文件中需要配置多个

```
new AddAssetHtmlWebpackPlugin({
    filepath:path.resolve(__dirname,'../dll/vendors.dll.js')
}),
new webpack.DllReferencePlugin({
    manifest:path.resolve(__dirname,'../dll/vendors.manifest.json')
})
```

2. 可以优化为

引进 fs 库

```
const fs = require('fs')
```

使用 fs 来读取 dll 目录下的文件

```
const plugins = [
    new HtmlWebpackPlugin({
        template: 'src/index.html'
    }),
    new CleanWebpackPlugin(), //new CleanWebpackPlugin(['dist'])
];
const files = fs.readdirSync(path.resolve(__dirname,'../dll'))
files.forEach(file=>{
    if(/.*\.dll.js/.test(file)){
        plugins.push(new AddAssetHtmlWebpackPlugin({
            filepath:path.resolve(__dirname,'../dll',file)
        }))
```

```
    }
    if(/.*\.manifest.json/.test(file)){
        plugins.push(new webpack.DllReferencePlugin({
            manifest:path.resolve(__dirname,'../dll',file)
        }))
    }
})
```

6. 控制包文件大小
7. thread-loader, parallel-webpack
   , happypack 多进程打包
8. 合理使用 sourceMap
9. 结合 stats 分析打包结果
10. 开发环境内存编译
10. 开发环境无用插件剔除

5-13 多页面打包配置

1.common 中 entry 配置

```
entry: {
        // lodash:'./src/lodash.js',
        main: './src/index.js',
        list:'./src/index.js'
},
```

2.common 中 plugin 配置

```
new HtmlWebpackPlugin({
        template: 'src/index.html',
        filename:'index.html',
        chunks:['runtime','vendors','main']
}),
new HtmlWebpackPlugin({
        template: 'src/index.html',
        filename:'list.html',
        chunks:['runtime','vendors','list']
}),
```

优化，增加页面的时候动态添加 plugin 中的 `HtmlWebpackPlugin`

```
commonConfig.plugins = makePlugins(commonConfig);
```

```
const makePlugins = (configs) => {
    。。。
    Object.keys(configs.entry).forEach(item => {
        plugins.push(
            new HtmlWebpackPlugin({
```

```
                template: 'src/index.html',
                filename: `${item}.html`,
                chunks: ['runtime', 'vendors', item]
            })
        )
    });
    return plugins;
}
```

第 6 章 Webpack 底层原理及脚手架工具分析

6-1 如何编写一个 Loader（1）

Loader 就是一个函数

```
module.exports = function(source) {
  return source.replace('dell', 'dellLee');
}
```

使用：

```
test: /\.js/,
use:[path.resolve(__dirname,'./loaders/replaceLoader.js')]
```

配置项在 loader 里面是使用 this.query 来接收的

```
module.exports = function(source) {
  return source.replace('dell', this.query.name);
}
```

```
{
    use:[path.resolve(__dirname,'./loaders/replaceLoader.js')]
    options: {
        name: 'lee'
    }
},
```

或者配置项可以使用 loader-utils 插件来接收

```
const loaderUtils = require('loader-utils');

module.exports = function(source) {
  const options = loaderUtils.getOptions(this);
  return source.replace('dell', options.name);
}
```

Loader 中有异步函数要使用 this.async()和 callback()

```
const loaderUtils = require('loader-utils');

module.exports = function(source) {
  const options = loaderUtils.getOptions(this);
  const callback = this.async();

  setTimeout(() => {
    const result = source.replace('dell', options.name);
    callback(null, result);
  }, 1000);

}
```

可以配置 resolveLoaders，这样使用的 loader 的时候就不用配路径了

```
resolveLoader: {
    modules: ['node_modules', './loaders']
},
```

```
{
    loader: 'replaceLoader',
},
```

实际上的用处：代码的异常捕获，网站的国际化

```
if(Node全局变量 === '中文') {
  source.repalce('{{title}}','中文标题')
}else{
  source.repalce('{{title}}','english title')
}
```

6-3 如何编写一个 Plugin

Plugin 就是一个类

```
class CopyrightWebpackPlugin {
  constructor() {

  }

  apply(compiler) {

  }
}

module.exports = CopyrightWebpackPlugin;
```

引进

```
const CopyRightWebpackPlugin = require('./plugins/copyright-webpack-plugin');
```

使用

```
plugins: [
        new CopyRightWebpackPlugin()
]
```

配置参数

```
    plugins: [
        new CopyRightWebpackPlugin({name:xie})
    ],
```

```
constructor(options) {
  console.log(options)
}
```

```
class CopyrightWebpackPlugin {
    apply(compiler) {//存放配置项的内容。打包的内容
        compiler.hooks.compile.tap('CopyrightWebpackPlugin', (compilation) =>
{//tap()同步
            console.log('compiler');
        })
        compiler.hooks.emit.tapAsync('CopyrightWebpackPlugin', (compilation, cb) =>
{//tapAsync()异步
            debugger;
            compilation.assets['copyright.txt']= {
                source: function() {
                    return 'copyright by dell lee'
                },
                size: function() {
                    return 21;
                }
            };
            cb();//异步必须，同步不需要
        })
    }
}
```

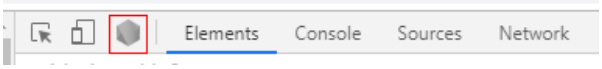
```
module.exports = CopyrightWebpackPlugin;
```

```
"debug": "node --inspect --inspect-brk node_modules/webpack/bin/webpack.js",
```

`--inspect` 开启 node 的调试工具

`--inspect-brk` 执行 webpack 打包的时候在 webpack 代码的第一行打下一个断点

执行 npm run degub 之后打开浏览器的控制台。



点击之后会出现 node 调试工具

6-4 Bundler 源码编写（模块分析 1）

全局安装代码高亮工具 npm i cli-highlight -g

在运行命令行的时候执行 node bundler.js | highlight

安装@babel/parser --save 来分析语法

```
const content = fs.readFileSync(filename, 'utf-8');
console.log(parser.parse(content,{
    sourceType:'module'//模块引入时使用 es6 module 时应该加上的声明
}))
```

`parser.parse` 打印出来的抽象语法树。可以通过 ast.program.body 来分析出 js 文档中的语句类型

```
Node {
  type: 'File',
  start: 0,
  end: 58,
  loc:
   SourceLocation {
     start: Position { line: 1, column: 0 },
     end: Position { line: 3, column: 21 } },
  program:
   Node {
     type: 'Program',
     start: 0,
     end: 58,
     loc: SourceLocation { start: [Object], end: [Object] },
     sourceType: 'module',
     interpreter: null,
     body: [ [Object], [Object] ],
     directives: [] },
  comments: [] }
```

安装@babel/traverse --save 可以帮助我们找出 ast 中是引入依赖的语句

安装@babel/core --save 可以将 es6 的代码转换成 es5 的代码。以便在浏览器中运行

安装@babel/preset-env --save 是 babel/core 转换 es6 的插件。翻译好的代码如下

```
E:\练习文档\bundler>node bundler.js | highlight
"use strict";

var _message = _interopRequireDefault(require("./message.js"));

function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj : { "d
efault": obj }; }

console.log(_message["default"]);
```

入口文件的分析：

```
const fs = require('fs');
const path = require('path');
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;//默认是一个 es module 的导出
const babel = require('@babel/core');

// 对模块进行分析
const moduleAnalyser = (filename) => {
    // 获取入口文件的内容
    const content = fs.readFileSync(filename, 'utf-8');
    // 拿到 js 中的依赖文件
    let ast = parser.parse(content,{
        sourceType:'module'//模块引入时使用 es6 module 时应该加上的声明
    });
    const dependencies = {};
    traverse(ast, {
        ImportDeclaration({ node }) {
            const dirname = path.dirname(filename);//拿到入口文件路径中的目录部分
            const newFile = './' + path.join(dirname, node.source.value);
            dependencies[node.source.value] = newFile;//{'./message.js':'./src/inde
x.js'}
        }
    });
    //将 ast 转化的 es6 语句转化成 es5
    const { code } = babel.transformFromAst(ast, null, {
        presets: ["@babel/preset-env"]
    });
    return {
        filename,
        dependencies,
        code
    }
}

const moduleInfo = moduleAnalyser('./src/index.js');
console.log(moduleInfo)
```

6-6 Bundler 源码编写（ Dependencies Graph ）

```
// 对入口文件的依赖的文件进行分析
const makeDependenciesGraph = (entry) => {
    const entryModule = moduleAnalyser(entry);
    const graphArray = [ entryModule ];
    for(let i = 0; i < graphArray.length; i++) {
        const item = graphArray[i];
        const { dependencies } = item;
        if(dependencies) {
            for(let j in dependencies) {
                graphArray.push(//graphArray 长度增加，就会再执行循环
                    moduleAnalyser(dependencies[j])
                );
            }
        }
    }
    // 将数组转换成对象
    const graph = {};
    graphArray.forEach(item => {
        graph[item.filename] = {
            dependencies: item.dependencies,
            code: item.code
        }
    });
    return graph;
}
```

```
{ './src/index.js':
   { dependencies: { './message.js': './src\\message.js' },
     code: '"use strict";\n\nvar _message = _interopRequireDefault(require("./me
ssage.js"));\n\nfunction _interopRequireDefault(obj) { return obj && obj.__esMod
ule ? obj : { "default": obj }; }\n\nconsole.log(_message["default"]);' },
  './src\\message.js':
   { dependencies: { './word.js': './src\\word.js' },
     code: '"use strict";\n\nObject.defineProperty(exports, "__esModule", {\n  v
alue: true\n});\nexports["default"] = void 0;\n\nvar _word = require("./word.js"
);\n\nvar message = "say ".concat(_word.word);\nvar _default = message;\nexports
["default"] = _default;' },
  './src\\word.js':
   { dependencies: {},
     code: '"use strict";\n\nObject.defineProperty(exports, "__esModule", {\n  v
alue: true\n});\nexports.word = void 0;\nvar word = \'hello\';\nexports.word = w
ord;' } }
```

6-7 Bundler 源码编写（ 生成代码 ）

```
// 最终的生成代码
const generateCode = (entry) => {
    const graph = JSON.stringify(makeDependenciesGraph(entry));//包含 export 和
require，所以需要构建 require 函数
    // 网页上的代码应该放在一个闭包里面，以防污染环境
    return `
        (function(graph){
            function require(module) {
                //将 require（）相对路径转化的函数
                function localRequire(relativePath) {
                    return require(graph[module].dependencies[relativePath]);
                }
                var exports = {};
                (function(require, exports, code){
                    eval(code)
                })(localRequire, exports, graph[module].code);
                return exports;
            };
            require('${entry}')
        })(${graph});
    `;
}

const code = generateCode('./src/index.js');
console.log(code);
```

第 7 章 Create-React-App 和 Vue-Cli 3.0 脚手架工具配置分析

7-1 通过 CreateReactApp 深入学习 Webpack 配置（1）

7-2 通过 CreateReactApp 深入学习 Webpack 配置（2）

7-3 Vue CLI 3 的配置方法及课程总结（1）

7-4 Vue CLI 3 的配置方法及课程总结（2）