# 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



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

CommonJS1568873591(1)

1568873643(1)

## 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](mailto: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

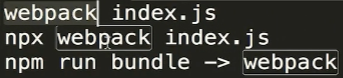
"scripts": {

    "bundle":"webpack"

  },

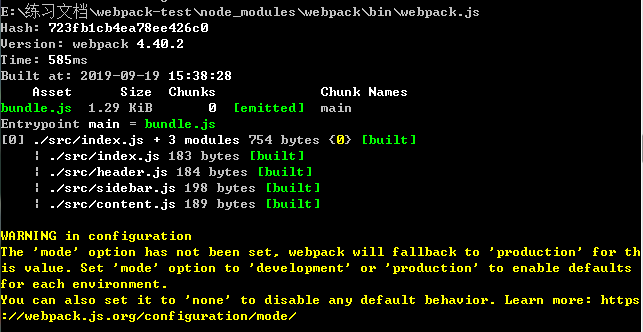
该命令会先在该项目下查找webpack

总结：3种webpack打包方式



webpack-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



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

# 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](https://www.webpackjs.com/concepts/modules)，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

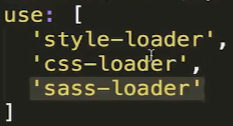
## 3-3 使用 Loader 打包静态资源（样式篇 - 上）

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

1. 打包sass文件需要两个loader：sass-loader，node-sass
2. 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）



1. 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

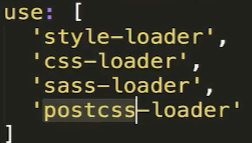
module.exports = {

    plugins:[

        require('autoprefixer')

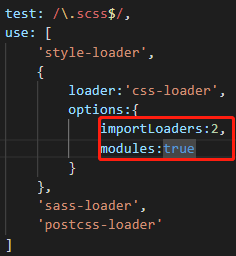
    ]

}

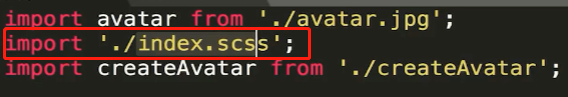


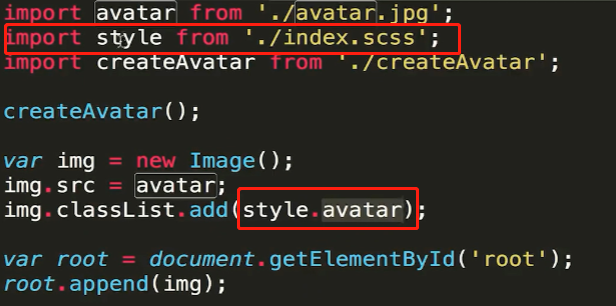
## 3-4 使用 Loader 打包静态资源（样式篇 - 下）

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



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





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"

},

1. 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",

1. 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"

}

1. 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
2. 配置options

options:{

    presets:['@babel/preset-env']

}

1. 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
2. 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

1. 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

# 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

1569294415(1)

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/>）

1569310247(1)

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

1569312837(1)

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');

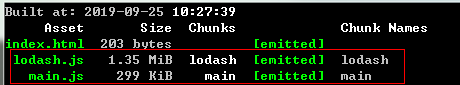
    const element = document.createElement('div');

    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：

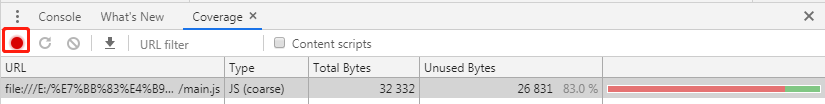
Webpack分析工具的仓库<https://github.com/webpack/analyse>

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

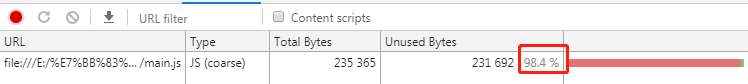
document.addEventListener('click', () => {

    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: {

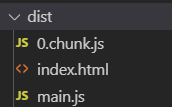
        filename: '[name].js', //打包后的文件命名，默认为main.js

        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

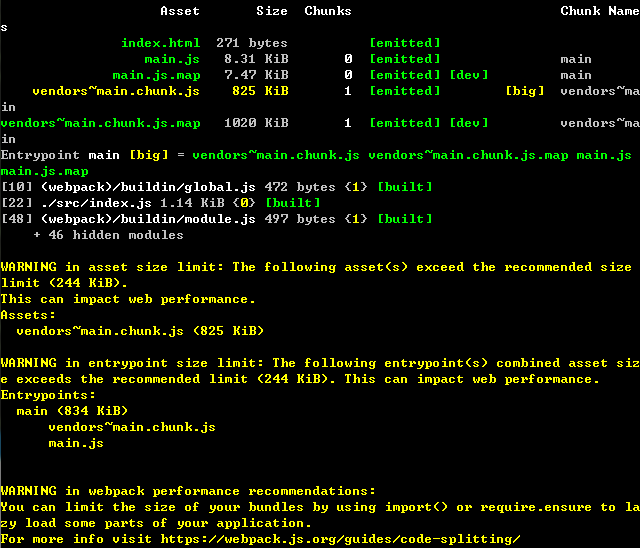
optimization: {

        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

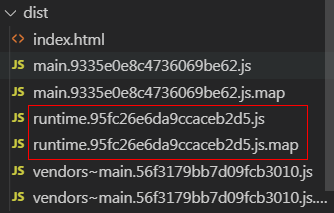
runtimeChunk:{

            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.common.js"

}

# Webpack 实战配置案例讲解

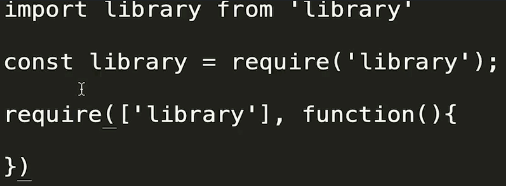
## 5-1 Library 的打包

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



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')

1. 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 => {

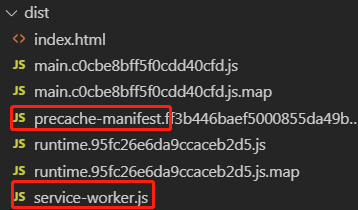
                console.log('service-worker register error')

            })

    })

}

4.打包之后dist目录下多了这两个目录



1. 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)

1569550054(1)

1569549915(1)

1569549937(1)

1569550117(1)

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

         }

    },

1569551538(1)

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')

})

1. 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)

        }))

    }

})

1569564197(1)



1569564324(1)

1569564363(1)

1569564411(1)

1569564435(1)

## 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]

            })

        )

});

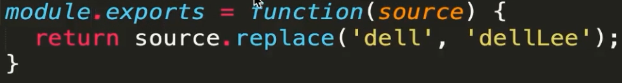
return plugins;

}

# 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）

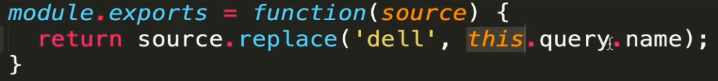
Loader就是一个函数

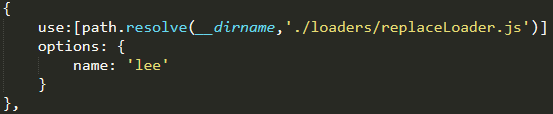


使用：

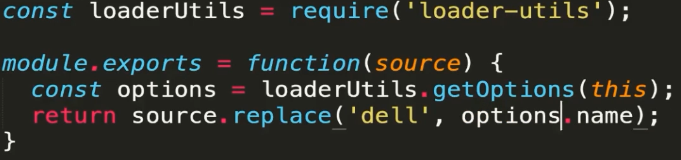
1569722627(1)

配置项在loader里面是使用this.query来接收的

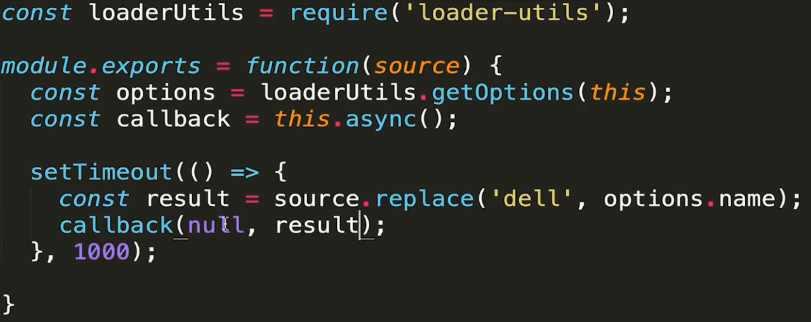




或者配置项可以使用loader-utils插件来接收

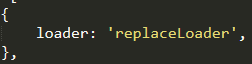


Loader中有异步函数要使用this.async()和callback()

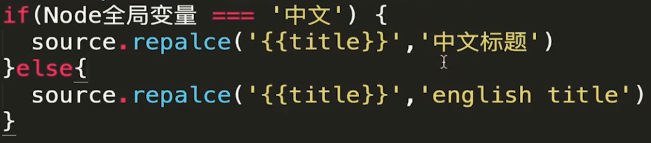


可以配置resolveLoaders，这样使用的loader的时候就不用配路径了

1569725977(1)



实际上的用处：代码的异常捕获，网站的国际化



## 6-3 如何编写一个 Plugin

Plugin就是一个类



引进

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

使用

plugins: [

new CopyRightWebpackPlugin()

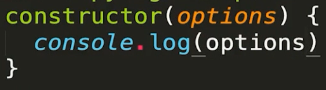
]

配置参数

plugins: [

new CopyRightWebpackPlugin({name:xie})

],



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

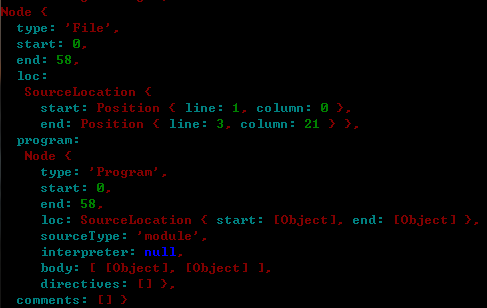
const content = fs.readFileSync(filename, 'utf-8');

console.log(parser.parse(content,{

    sourceType:'module'//模块引入时使用es6 module时应该加上的声明

}))

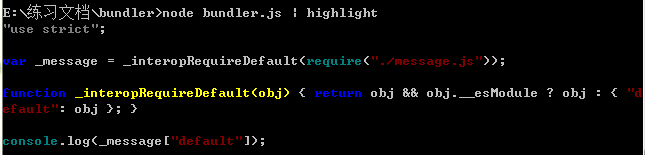
parser.parse打印出来的抽象语法树。可以通过ast.program.body来分析出js文档中的语句类型



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



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/index.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,

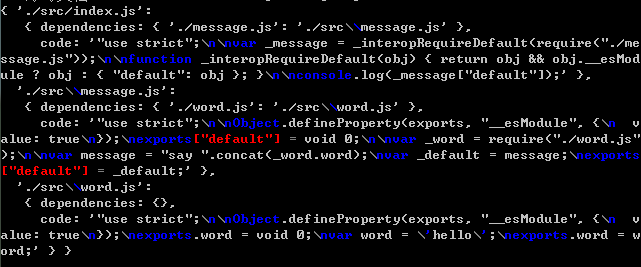
            code: item.code

        }

    });

    return graph;

}



## 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）