

- 个人笔记
  - 1 webpack 基本概念
    - 1.1 webpack 的 5 个核心概念
  - 2 webpack 初体验
  - 3 webpack 开发环境的基本配置
    - 3.1 创建配置文件 webpack.config.js
    - 3.2 打包样式资源 css less
    - 3.3 打包 HTML 资源:plugins
    - 3.4 图片资源的打包
    - 3.5 webpack 打包其他资源
    - 3.6 devServer
    - 3.7 开发环境的配置总结:
  - 4 webpack 构建生产环境介绍
    - 4.1 在开发模式下提取 CSS 为单独文件
    - 4.2 CSS 兼容性处理
    - 4.3 压缩 CSS
    - 4.4 js 语法检查 eslint

# Webpack个人笔记

## 1 webpack 基本概念

webpack 是一种前端资源构建工具，一个静态的模块打包器 (module bundler)

webpack 会将所有的 js/json/less/img/css 等文件均作为模块处理

它会设定一个入口文件，通过分析各个模块的依赖关系形成一个依赖关系图，然后依次将所有模块打包成静态资源 (bundle)

中文文档不是最新的，可以直接查看英文文档 <https://webpack.js.org/>

Asset	Size	Chunks	Chunk Names
css/index.17de3eba30.css	15 bytes	0 [emitted] [immutable]	main
index.html	343 bytes	[emitted]	
js/main.541ec397dd.js	1010 bytes	0 [emitted] [immutable]	main
js/test.819c26e1d5.js	948 bytes	1 [emitted] [immutable]	test

chunk都是根据0,1,2,3作为id编号的，第一个chunk包含两个bundle，一个是js，一个是css，第二个chunk包含一个js文件

## 1.1 webpack 的 5 个核心概念

- Entry
  - 入口(Entry)指示 webpack 以哪个文件为入口起点开始打包，分析构建内部依赖图。
- Output
  - 输出(Output)指示 webpack 打包后的资源 bundles 输出到哪里去，以及如何命名。
- Loader
  - loader 是预处理器, 让 webpack 能够去处理那些非 JavaScript 文件(webpack 自身只理解 JavaScript)
    - 识别出应该被loader转化的文件，使用test属性
    - 转换这些文件，使他们添加到依赖图中，最终添加到bundle中，使用use属性
- Plugins
  - 插件(Plugins)可以用于执行范围更广的任务。插件的范围包括，从打包优化和压缩，一直到重新定义环境中的变量等。plugins 选项用于以各种方式自定义 webpack 构建过程。
  - loader即为文件加载器，操作的是文件，将文件A通过loader转换成文件B，是一个单纯的文件转化过程。
  - plugin即为插件，是一个扩展器，丰富webpack本身，增强功能，针对的是在loader结束之后，webpack打包的整个过程，他并不直接操作文件，而是基于事件机制工作，监听webpack打包过程中的某些节点，执行广泛的任务。
- Mode

开发环境是程序员们专门用于开发的服务器，配置可以比较随意，为了开发调试方便，一般打开全部错误报告。(程序员接到需求后，开始写代码，开发，运行程序，看看程序有没有达到预期的功能；

测试环境：一般是克隆一份生产环境的配置，一个程序在测试环境工作不正常，那么肯定不能把它发布到生产机上。(程序员开发完成后，交给测试部门全面的测试，看看所实现的功能有没有bug，测试人员会模拟各种操作情况；

生产环境是指正式提供对外服务的，一般会关掉错误报告，打开错误日志。(就是线上环境，发布到对外环境上，正式提供给客户使用的环境。

模式(Mode)指示 webpack 使用相应模式的配置。

选项	描述	特点
development	会将 DefinePlugin 中 process.env.NODE_ENV 的值设置为 development。启用 NamedChunksPlugin 和 NamedModulesPlugin。	能让代码本地调试运行的环境
production 插件更为	会将 DefinePlugin 中 process.env.NODE_ENV 的值设置为 production。启用 FlagDependencyUsagePlugin, FlagIncludedChunksPlugin, ModuleConcatenationPlugin, NoEmitOnErrorsPlugin, OccurrenceOrderPlugin, SideEffectsFlagPlugin 和 TerserPlugin。	能让代码优化上线运行的环境

## 2 webpack 初体验

- 1 初始化 package.json(即初始化项目), 在 webpack 文件加下直接初始化, 这里初始化过程中填写的项目内容后期均可以在 package.json 文件中进行修改

```
npm init
```

See `npm help init` for definitive [documentation](#) on these fields and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and save it as a dependency in the package.json file.

Press ^C at any time to quit.

package name: (webpack) my\_webpack 设置项目名

version: (1.0.0) 项目版本

description: my first try 项目描述

entry point: (index.js) 项目入口文件, 所以说这里就已经确定了入口文件

test command: 项目启动的时候要用什么命令来执行脚本文件 (默认为node app.js)

git repository: github的远程库的地址

keywords:

author: 项目作者

license: (ISC) 项目许可证

About to write to C:\Users\17273\OneDrive\文档\front-to-end\mynote\webpack\package.jso

```
{
  "name": "my_webpack",
  "version": "1.0.0",
  "description": "my first try",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

上面填写的内容生成的

- package.json 中的内容：

```
{
  "name": "my_webpack",
  "version": "1.0.0",
  "description": "my first try",
  "main": "index.js",
  ▶ Debug
  "scripts": {
    | "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    | "webpack": "^4.44.2"
  },
}
```

- 2 将 webpack 直接安装到 webpack 文件夹下

```
npm install webpack webpack-cli -g //全局安装
npm install webpack webpack-cli -D //本地安装
```

- 注意：这一步可能会出现以下问题

```
PS C:\Users\17273\OneDrive\文档\front-to-end\mynote\webpack> npm install webpack webpack-cli -g
npm ERR! code ERR_TLS_CERT_ALTNAME_INVALID
npm ERR! errno ERR_TLS_CERT_ALTNAME_INVALID
npm ERR! request to https://registry.cnpmjs.org/webpack failed, reason: Hostname/IP does not match certificate's altnames: Host: registry.cnpmjs.org. is not in the cert's altnames: DNS:r.cnpmjs.org

npm ERR! A complete log of this run can be found in:
npm ERR! C:\Users\17273\AppData\Roaming\npm-cache\_logs\2020-09-19T11_48_05_194Z-debug.log
```

- 解决办法：执行下面的命令

```
npm config set registry http://registry.npmjs.org
// 如果上面的操作执行了还有问题，则可以将镜像改为淘宝
npm config set registry http://registry.npm.taobao.org
```

- 3 编译打包

- 开发环境指令：webpack ./src/js/index.js -o ./build/js/built.js --mode=production
- 生产环境指令：webpack ./src/js/index.js -o ./build/js/built.js --mode=production
- 以开发环境打包得到结果：

```
PS C:\Users\17273\OneDrive\文档\front-to-end\mynote\webpack\02_webpack初体验> webpack ./src/js/index.js -o ./build/js/built.js --mode=production
Hash: 8a7af6826f22a2012d4f
Version: webpack 4.44.2
Time: 389ms
Built at: 2020/09/19 下午8:04:29
    Asset      Size  Chunks             Chunk Names
  built.js  947 bytes          0  [emitted]  main
Entrypoint main = built.js
[0] ./src/js/index.js 65 bytes {0} [built]
```

- 观察开发环境和生产环境下的打包结果，可以发现在开发环境下的打包结果压缩了

- 另外，可以发现 webpack 可以直接编译打包的文件有：js 和 json
- 其他的文件不能直接编译打包，比如 css,img.less 等，这些文件需要借助 loader 工具预先处理，html 等文件需要借助插件 plugin 进行预先处理

## 3 webpack 开发环境的基本配置

- 注意：一旦确定以某个 js 文件作为入口文件，则需要将其他需要编译打包的文件 **import** 到该入口文件中，例如：



### 3.1 创建配置文件 webpack.config.js

- 这是 webpack 的配置文件，直接指示 webpack 如何干活，干哪些活
- 配置文件的基本内容如下：只包含 js 或者 json 的编译打包时 webpack.config.js 的文件的内容：

```
// 引入nodejs的内置path模块，处理路径问题
const {resolve} = require('path');

// 设置编译打包的输出
module.exports = {
  // 入口文件
  entry: './src/js/index.js',
  // 输出配置
  output: {
    filename: './build/js/built.js',
    // 输出路径
    // __dirname是nodejs的一个变量，表示当前文件(webpack.config.js)的目录
    path: resolve(__dirname, 'built.js')
    // 上面两句话的意思：文件输出到webpack.config.js文件的目录/build/built.js
  },
  // loader配置
  module: {
    // 详细地loader配置
    rules: []
  }
};
```

```
// plugin配置
plugins:[
  // 详细配置
],
// 设置开发环境
mode: 'development'
}
```

- 编译打包的结果:

```
PS C:\Users\17273\OneDrive\文档\front-to-end\mynote\webpack\03_打包样式资源> webpack
Hash: 302972a64e1d4e6db995
Version: webpack 4.44.2
Time: 86ms
Built at: 2020/09/19 下午8:30:28
   Asset      Size  Chunks             Chunk Names
built.js  3.86 KiB       0  [emitted]  main
Entrypoint main = built.js
[./src/js/index.js] 65 bytes {0} [built]
```

## 3.2 打包样式资源 css less

- 首先确定入口文件, 将其他需要打包的文件(css,less)引入到入口文件中



```
webpack > 03_打包样式资源 > src > js > JS index.js > ...
1 import '../css/index.css'
2 import '../less/index.less'
3
4 function add(a, b){
5   return a+nb;
6 }
7 console.log(add(1, 2));
```

- 1 下载安装 loader 包:css-loader style-loader less-loader以及less

```
// 第一种: 全局安装:
npm install style-loader css-loader less-loader less -g

// 使用了第一种方法后面webpack有问题, 可以使用第二种:
npm install style-loader --save
npm install css-loader --save
npm install less-loader --save
npm install less --save
```

- 2 在上面的 webpack.config.js 中直接修改 module 的内容

```
module: {
  rules:[
    // 详细地loader配置
    // 对于不同的文件需要配置不同的loader处理
```

```

// 1 css文件
{
  // 设置匹配哪些文件，使用正则表达式，下面表示匹配以css结尾的文件
  test: /\.css$/,
  // 进行哪些loader处理
  use: [
    // user中loader执行顺序：从右至左 依次执行
    // 首先css-loader将css文件变成commonjs模块加载到目标js文件中，以
字符串样式呈现
    // 然后style-loader将目标js文件中的样式资源提取出来插入style标签
中，添加到head中
    'style-loader', 'css-loader'
  ]
},

// 2 less文件
{
  // 设置匹配哪些文件，使用正则表达式，下面表示匹配以css结尾的文件
  test: /\.less$/,
  // 进行哪些loader处理
  use: [
    // user中loader执行顺序：从右至左 依次执行
    // 首先less-loader将less文件编译成css文件
    // 然后css-loader将css文件变成commonjs模块加载到目标js文件中，以字
符串样式呈现
    // 最后style-loader将目标js文件中的样式资源插入style标签中，添加到
head中
    'style-loader', 'css-loader', 'less-loader'
  ]
}
]
},

```

- 3 然后使用 webpack 编译打包，效果如下：

```

PS C:\Users\17273\OneDrive\文档\front-to-end\mynote\webpack\03_打包样式资源> webpack
Hash: 130e93b21b06f44eae8a
Version: webpack 4.44.2
Time: 668ms
Built at: 2020/09/19 下午9:30:40
    Asset      Size  Chunks             Chunk Names
  built.js  19.9 KiB          0  [emitted]  main
Entrypoint main = built.js
[./node_modules/css-loader/dist/cjs.js!./node_modules/less-loader/dist/cjs.js!./src/less/index.less] 314 bytes {main} [built]
[./node_modules/css-loader/dist/cjs.js!./src/css/index.css] 390 bytes {main} [built]
[./src/css/index.css] 531 bytes {main} [built]
[./src/js/index.js] 121 bytes {main} [built]
[./src/less/index.less] 578 bytes {main} [built]
+ 2 hidden modules

```

### 3.3 打包 HTML 资源:plugins

- html 文件不用 import 进入口文件，在 html 中也不需要引入 js 文件或者 less 文件
- 1 下载 html-webpack-plugin 插件处理包

```
npm install html-webpack-plugin --save
```

- 2 修改 webpack.config.js 中 plugins 部分的内容

```
plugins:[  
  // html-webpack-plugin插件配置  
  new HtmlWebpackPlugin()  
],
```

- 3 执行 webpack 命令，得到结果：

```
PS C:\Users\17273\OneDrive\文档\front-to-end\mynote\webpack\04_打包html资源> webpack  
Hash: aeb2a4a74da5e80ee1bb  
Version: webpack 4.44.2  
Time: 200ms  
Built at: 2020/09/19 下午10:37:55
```

Asset	Size	Chunks	Chunk Names
built.js	3.77 KiB	main [emitted]	main
index.html	226 bytes	[emitted]	

```
Entrypoint main = built.js  
[./src/index.js] 0 bytes {main} [built]  
Child HtmlWebpackCompiler:  
  1 asset  
  Entrypoint HtmlWebpackPlugin_0 = __child-HtmlWebpackPlugin_0  
  1 module
```

打包的结果产生了两个文件

### 3.4 图片资源的打包

- 1 安装包：
  - 首先，需要使用html-loader引入 html 中的图片到目标 js 文件中，所以先安装 html-loader

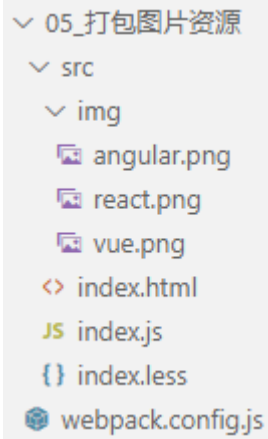
```
npm install html-loader ---save
```

- 默认情况下，webpack 处理不了 img 图片，处理样式中的图片资源需要借助url-loader,而它又是依赖于file-loader产生作用的，所以首先就需要下载这两个 loader 包；另外，处理 html 中的图片需要使用'html-loader'先将图片引入到目标 js 文件中，然后再将使用html-loader解析图片的地址，所以也需要下载 html-loader这个包

```
npm install url-loader file-loader html-loader ---save
```



- 2 搭配项目：



- ./src/index.html 文件的内容：不需要引入 less 文件

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Document</title>
</head>
<body>
  <div id="div1"></div>
  <div id="div2"></div>
  <div id="div3"></div>
  
</body>
</html>
```

- 3 配置 webpack.config.js 文件：

- 执行顺序：
  - (1) 通过入口文件开始打包，html-loader 解析 HTML 文件中的图片文件
  - (2) less-loader,css-loader,style-loader 处理样式文件
  - (3) url-loader 解析样式中的图片路径问题，file-loader 处理其他文件格式
  - (4) Plugins 中的 html-webpack-plugin 则负责打包 HTML 文件；

```
const {resolve} = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',
  output:{
    filename:'built.js',
    path:resolve(__dirname, 'build')
  },
  module:{
    rules:[
```

```

        // less文件
        {
            test: /\.less$/,
            use: ['style-loader', 'css-loader', 'less-loader']
        },
        {
            // 1 匹配html文件, 将html文件中的img引入目标js文件中, 以
commonjs模块化的形式
            test: /\.html$/,
            loader: 'html-loader'
        },
        // png,jpg,gif
        {
            // 2 匹配图片, 解析目标js文件中的样式的图片路径问题
            test: /\. (png|jpg|gif) $/,
            loader: 'url-loader',
            options: {
                // 限制如果图片大小小于12kb, 则使用base64处理, 这样可以减轻服
务器的压力, 但是如果图片太大使用base64处理就会导致请求速度变慢
                limit: 12 * 1024,
                // 修改图片的命名, 取图片的hash的前10位, ext表示取文件原来的
扩展名
                name: '[hash:10].[ext]'
            }
        }
    ],
    },
    plugins: [
        new HtmlWebpackPlugin({
            template: './src/index.html'
        })
    ],
    mode: 'development'
}

```

- 4 执行 webpack 指令, 得到效果:

- 打包得到了 4 个文件, **built.js,index.html**, 还有两张图片
- 本来有 3 张图片, 打包时却只有 2 张图片, 这是因为还有一张图片的大小小于 12kb, 所以使用 base64 编码处理了

```
PS C:\Users\17273\OneDrive\文档\front-to-end\mynote\webpack\05_打包图片资源> webpack
Hash: 19e7b47a6727a78aa3fd
Version: webpack 4.44.2
Time: 1720ms
Built at: 2020/09/20 上午10:00:06

   Asset      Size  Chunks             Chunk Names
3745252f95.png 27.8 KiB          0 [emitted] [immutable]
   built.js    33.6 KiB          0 [emitted]
c1e8d38c83.png 15.6 KiB          0 [emitted] [immutable]
   index.html   380 bytes          0 [emitted]

Entrypoint main = built.js
[./node_modules/css-loader/dist/cjs.js!./node_modules/less-loader/dist/cjs.js!./src/index.less] 1.26 KiB {main} [built]
[./src/img/angular.png] 58 bytes {main} [built]
[./src/img/react.png] 12.1 KiB {main} [built]
[./src/img/vue.png] 58 bytes {main} [built]
[./src/index.js] 23 bytes {main} [built]
[./src/index.less] 569 bytes {main} [built]
+ 3 hidden modules
Child HtmlWebpackCompiler:
  1 asset
    Entrypoint HtmlWebpackPlugin_0 = __child-HtmlWebpackPlugin_0
    [./node_modules/html-webpack-plugin/lib/loader.js!./src/index.html] 450 bytes {HtmlWebpackPlugin_0} [built]

v build
  3745252f95.png
  JS built.js
  c1e8d38c83.png
  <> index.html
```

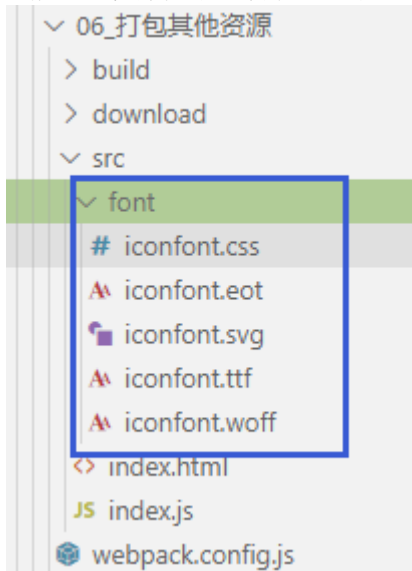
总共打包得到4个文件

### 3.5 webpack 打包其他资源

- 其他文件的打包方式：file-loader,排除 js,css,html 文件

```
module:{
  rules:[
    {
      test:/\.css$/,
      use:['style-loader','css-loader']
    },
    //打包其他资源（除了html,css,js之外的资源）
    {
      // 排除js,css,html文件
      exclude:/\. (js|css|html) $/,
      loader:'file-loader',
      options:{
        name: '[hash:10].[ext]'
      }
    }
  ]
}
```

- 例如：字体文件的打包：在网上下载一个字体包，然后找到下面的文件：



- html 中内容如下：创建四个 span,引入四个图标字体

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Document</title>
</head>
<body>
  <span class="iconfont icon-icon-test"></span>
  <span class="iconfont icon-icon-test2"></span>
  <span class="iconfont icon-icon-test3"></span>
  <span class="iconfont icon-icon-test1"></span>
</html>
```

- 在 index.js 中需要引入 iconfont.css
- 向上面说的添加 module 的 rules,配置 webpack.config.js 文件
- 执行 webpack 命令后效果：

PS C:\Users\17273\OneDrive\文档\front-to-end\mynote\webpack\3.webpack 开发环境配置\06\_打包其他资源> webpack

Hash: 48c6e4801c0252d39edd

Version: webpack 4.44.2

Time: 655ms

Built at: 2020/09/20 上午11:04:17

Asset	Size	Chunks	Chunk Names
02cf02b9fa.eot	3.87 KiB	[emitted] [immutable]	
7e9e8d6597.woff	2.63 KiB	[emitted] [immutable]	
9e85820ca4.ttf	3.71 KiB	[emitted] [immutable]	
built.js	27.4 KiB	main [emitted]	main
cd3663039e.svg	7.96 KiB	[emitted] [immutable]	
index.html	483 bytes	[emitted]	

Entrypoint main = built.js

[./../node\_modules/css-loader/dist/cjs.js!./src/font/iconfont.css] C:/Users/17273/OneDrive/文档/front-to-end/myr/dist/cjs.js!./src/font/iconfont.css 4.99 KiB {main} [built]

[./src/font/iconfont.css] 540 bytes {main} [built]

[./src/font/iconfont.eot?t=1600569627541] 58 bytes {main} [built]

[./src/font/iconfont.svg?t=1600569627541] 58 bytes {main} [built]

[./src/font/iconfont.ttf?t=1600569627541] 58 bytes {main} [built]

[./src/font/iconfont.woff?t=1600569627541] 59 bytes {main} [built]

[./src/index.js] 66 bytes {main} [built]

+ 3 hidden modules

## 3.6 devServer

- 之前使用 **webpack** 打包需要每次添加一个新的内容就使用 **webpack** 命令编译打包一次，有些麻烦，所以出现了 **devServer**,实现自动编译，自动打开浏览器，自动刷新
- 开发服务器有一个特点：只会在内存中编译打包，不会输出打包结果，帮助我们在浏览器中实时观测页面的变化
- 1 首先需要下载webpack-dev-server这个包

```
npm install webpack-dev-server --save
```

- 2 在 **webpack.config.js** 文件中配置 **devserver**:

```
const { resolve } = require("path");
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',
  output:{
    filename:'built.js',
    path:resolve(__dirname, 'build')
  },
  module:{
    rules:[
      {
        test:/\.css$/,
        use:['style-loader','css-loader']
      },
      //打包其他资源（除了html,css,js之外的资源）
      {
        // 排出js,css,html文件
        exclude:/\. (js|css|html) $/,
        loader:'file-loader',
        options:{
          name: '[hash:10].[ext]'
        }
      }
    ]
  },
  plugins:[
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ],
  mode:'development',

  // 开发服务器 devServer
  devServer: {
    // 项目构建后的路径
    contentBase: resolve(__dirname, 'build'),
```

```

    // 启动gzip压缩
    compress: true,
    // 指定端口号
    port: 3000,
    // 在启动devServer指令后, 自动打开在浏览器中打开`localhost:3000`页面
    open: true
  }
}

```

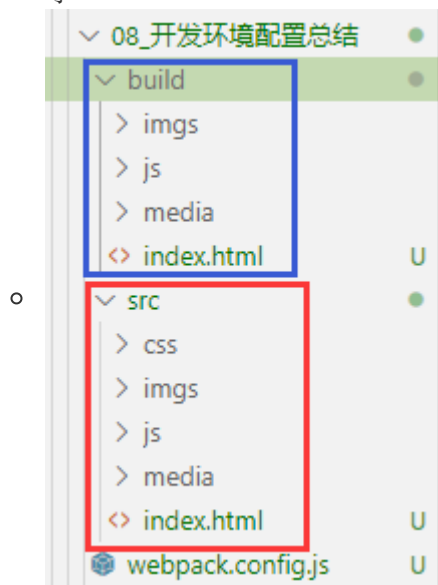
- 3 启动 devServer 的指令:

```
npx webpack-dev-server
```

- 也就是说每次只要修改./src中的内容, 即可直接在浏览器中看到变化, 并且可以发现并没有生成新的打包文件, 可以帮助我们得到满意的效果后再打包

### 3.7 开发环境的配置总结:

- webpack.config.js 文件配置如下, 负责匹配各个类型的文件, 从而实现打包处理
  - 这里指定了各个文件的打包规则
  - 并且对于图片及其他文件的输出位置做了规定, 让打包得到的文件关系更加清晰明了



```

// nodejs的路径模块
const {resolve} = require('path');
// html文件打包的插件
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  // 入口文件
  entry: './src/js/index.js',
  // 只决定入口文件的输出位置和名字
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build')
  }
}

```

```
},
// 配置各个资源的匹配规则和打包方式
module:{
  rules:[
    // less    less文件和css文件会打包到js文件中保存
    {
      test:/\.less$/,
      use:['style-loader','css-loader','less-loader']
    },
    // css
    {
      test:/\.css$/,
      use:['style-loader','css-loader']
    },
    // 样式中的图片资源
    {
      test:/\. (png|jpg|gif) $/,
      loader:'url-loader',
      options:{
        limit:10*1024,
        name:'[hash:10].[ext]',
        // 设定图片输出到哪个位置
        outputPath:'imgs'
      }
    },
    // html中的图片资源
    {
      test:/\.html$/,
      loader:'html-loader'
    },
    // 其他的资源
    {
      exclude:/\. (html|css|less|js|jpg|png|gif) $/,
      loader:'file-loader',
      options:{
        name:'[hash:10].[ext]',
        // 设置其他文件打包到media目录下
        outputPath:'media'
      }
    }
  ]
},
plugins:[
  // 打包html文件
  new HtmlWebpackPlugin({
    template:'./src/index.html'
  })
],
```

```
// 设置开发模式
mode: 'development',
// 开启自启动浏览器，自更新服务
devServer: {
  contentBase: resolve(__dirname, 'build'),
  compress: true,
  port: 3000,
  open: true
}
}
```

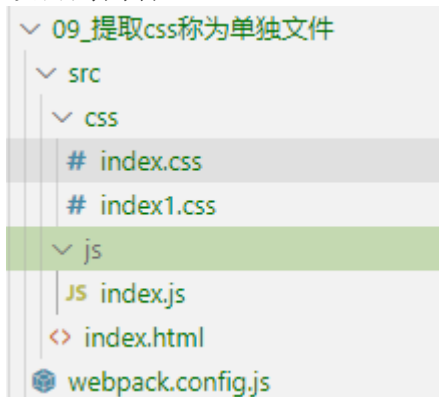
## 4 webpack 构建生产环境

### 4.1 提取 css 为单独文件

- 在开发模式下，如果想要提取 css 文件为一个单独的文件，不和js文件混杂，需要下载一个插件：
  - 这个插件在下载后，存在一个 loader，可以将目标 js 文件中的样式提取出来成为一个单独的 main.css 文件，然后在打包得到的 index.html 中会自动引入该 css 文件

```
npm install mini-css-extract-plugin --save
```

- 项目的内容：



- 修改 webpack.config.js 的内容：

```
const {resolve} = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
```



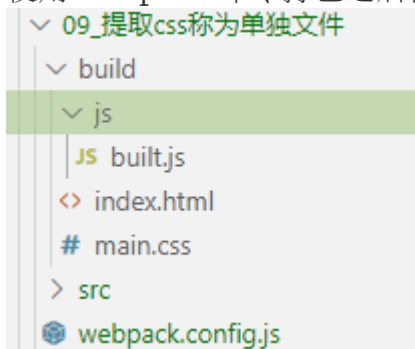
签中

```
// css文件
{
  test: /\.css$/,
  use: [
    // 这个插件.loader取代了style-loader
    // style-loader将样式内容从js中文件中读出，然后放在style标

    // 这个loader直接提取js中的样式内容成为单独文件
    MiniCssExtractPlugin.loader,
    'css-loader'
  ],
  // options:{
  //   outputPath: 'css'
  // }
}

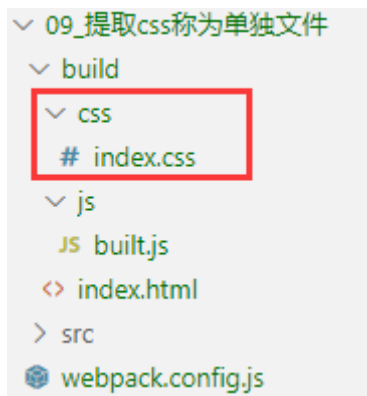
],
},
plugins: [
  new HtmlWebpackPlugin({
    template: './src/index.html'
  }),
  new MiniCssExtractPlugin()
],
mode: 'development'
}
```

- 使用 webpack 命令打包之后得到的结果为：



- 当然我们可以自定义 CSS 文件的名字和位置：

```
plugins: [
  new HtmlWebpackPlugin({
    template: './src/index.html'
  }),
  new MiniCssExtractPlugin({
    // 对输出的文件进行重命名
    filename: 'css/index.css'
  })
],
```



## 4.2 css 兼容性处理

- 如果 css 样式中存在一些浏览器不兼容的样式，就会出现兼容性问题，此时就需要使用 postcss 处理
- 首先需要下载两个包：postcss-loader postcss-preset-env

```
npm install postcss-loader postcss-preset-env --save
```

- 在 css 文件中添加两个不兼容的样式：

```
display: flex;
backface-visibility: hidden;
```

- 写配置 webpack.config.js

```
const { resolve } = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

// 设置nodejs环境变量，不设置时默认是生产环境
process.env.NODE_ENV = 'development';

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          MiniCssExtractPlugin.loader,
          'css-loader',
          // 修改loader的配置
          {
            loader: "postcss-loader",
```

```

    options: {
      postcssOptions: {
        ident: "postcss",
        plugins: () => [
          // 帮postcss找到package.json中browserslist里面的配置，通过
配置加载指定的css兼容性样式
          require("postcss-preset-env")(),
        ],
      },
    },
  ],
}
]
},
plugins: [
  new HtmlWebpackPlugin({
    template: './src/index.html'
  }),
  new MiniCssExtractPlugin({
    filename: 'css/built.css'
  })
],
mode: 'development'
};

```

- 在 **package.json** 中添加以下内容，指定在不同的模式下兼容哪些版本的浏览器，默认情况下，使用生产模式的兼容性

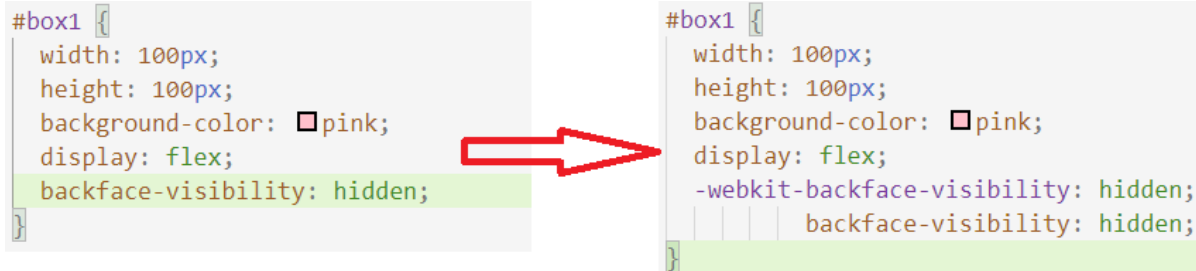
```

"browserslist":{
  // 开发环境
  "development":[
    "last 1 chrome version",
    "last 1 firefox version",
    "last 1 safari version"
  ],
  // 生产环境
  "production":[
    ">0.2%",
    "not dead",
    "not op_mini all"
  ]
}

```

- 要想将兼容性修改为开发环境下的配置，需要设置 **node** 环境变量：  
**process.env.NODE\_ENV = 'development';**
  - 不设置**node**环境变量时，默认是匹配下面的生产环境的兼容性
  - 可以添加别的配置项

- 上面的内容就是帮 postcss 找到 package.json 中 browserlist 里面的配置，通过配置加载指定的 css 兼容性样式
- 然后执行 webpack 命令，可以得到此时打包得到的 css 文件的内容为：



```
#box1 {
  width: 100px;
  height: 100px;
  background-color: pink;
  display: flex;
  backface-visibility: hidden;
}
```

```
#box1 {
  width: 100px;
  height: 100px;
  background-color: pink;
  display: flex;
  -webkit-backface-visibility: hidden;
  backface-visibility: hidden;
}
```

## 4.3 压缩 css

- 需要引入一个插件：

```
npm install optimize-css-assets-webpack-plugin --save
```

- 在 webpack.config.js 中添加配置：

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

plugins:[
  new OptimizeCssAssetsWebpackPlugin()
]
```

- 运行 webpack 指令后得到结果：
  - 可以发现此时 css 文件中的内容就被压缩啦

```
webpack > 4.webpack生产环境 > 11_压缩css > build > css > # built.css > #box1
1 #box1{width:100px;height:100px;background-color: pink;displ
```

## 4.4 js 语法检查 eslint

- 语法检查只检查用户自己写的代码，不检查第三方库引入的代码
- 首先需要下载 eslint 和 eslint-loader

```
npm install eslint eslint-loader --save
```

在 package.json 设置检查规则：

```
"eslintConfig": {
  "extends": "airbnb-base",
  "env": {
    "browser": true // 设置匹配浏览器环境，从而可以识别浏览器的全局变量
  }
}
```

安装包：下载`eslint-plugin-import`和`eslint-config-airbnb-base`

- 添加配置

```
rules:[
  {
    test:/\.js$/,
    // 不检查第三方库引入的代码
    exclude:/node_modules/,
    loader:'eslint-loader',
    options:{
      fix:true; // 自动修复格式中存在的问题
    }
  }
]
```

但是注意：这里对于js文件的语法检查，不包括将es6语句转换为js语句，所以如果js文件中存在es6新语法，则打包后的文件在不兼容该新语法的浏览器中就会报错

例如：

```
const add = (x, y) => x + y;
// 下一行eslint所有规则均失效（下一行不进行eslint检查）
// eslint-disable-next-line
console.log(add(1, 2));
```

打包后：

```
eval("var add = (x, y) => {\n  return x + y;\n};\n// 下一行eslint所有规则均失效（下一行不进行eslint检查）\n// eslint-disable-next-line\nconsole.log(add(1, 2));\n\n\n//\nsourceURL=webpack:///./src/js/index.js?");
```

箭头函数还是箭头函数，ie11是不支持箭头函数的，需要使用babel转换

## 4.5 js兼容性处理，es6->es5

### 4.5.1 方法1

打包时将es6新语法转换为js语句,注意各个包之间的版本兼容性,参考[webpack](#),[es6转es5](#).

```
webpack 4.x | babel-loader 7.x | babel 6.x 版本
npm install -D babel-loader@7 babel-core@6 babel-preset-es2015 webpack@4
可以是babel-preset-env或者babel-preset-es2015

#webpack 4.x | babel-loader 8.x | babel 7.x 最新版本
yarn add babel-loader @babel/core @babel/preset-env -D
```

本文选择后面的配置：

```
rules:[
  {
    loader:'babel-loader',
    options:{
      "presets": ["@babel/preset-env"]
    }
  }
]
```

使用webpack命令打包后，可以发现箭头函数和const等es6新语法已经变成了es5语法：

```
"use strict";
eval("\n\nvar add = function add(x, y) {\n  return x + y;\n}; // 下一行
eslint所有规则均失效（下一行不进行eslint检查）\n// eslint-disable-next-
line\nconsole.log(add(1, 2));\nvar a = 1;\n// eslint-disable-next-
line\nconsole.log(a);\n\n// # sourceMappingURL=webpack:///./src/js/index.js?");
```

这种方法的缺点：它只能转换一些es6的基本语法，类似于promise等不能做转换

## 方法2：es6全部转es5

这种方法的缺点：引入文件的体积过大，为了解决部分兼容性问题，引入了所有的兼容性代码

第一步：yarn add babel-polyfill -D  
或者 yarn add @babel/polyfill -D

第二步：在入口文件中引入：

```
import 'babel-polyfill'
// 或者 import '@babel/polyfill'
const promise = new Promise((resolve) => {
  console.log("lalalal");
});
```

第三步：

使用webpack命令打开即可，可以看到在未引入之前，打包后的promise是不能被识别的；在引入该包后，可以执行Promise语句

```
Built at: 2021/05/29 下午9:30:10
Asset      Size  Chunks  Chunk Names
index.html 272 bytes  [emitted]
js/built.js 4.17 KiB  main [emitted] main
Entrypoint main = js/built.js
[./src/js/index.js] 352 bytes {main} [built]
Child HtmlWebpackCompiler:
```

❌ SCRIPT5009: "Promise"未定义  
index.js (14,1)

```
index.html 272 bytes  [emitted]
js/built.js 595 KiB  main [emitted] main
Entrypoint main = js/built.js
```

lalalal

## 方法3：按需加载

第一步：yarn add core-js -D

第二步：配置

首先：去除方法2中引入的兼容包

其次：

```
presets: [  
  [  
    "@babel/preset-env",  
    // 按需加载  
    {  
      useBuiltIns: "usage",  
      corejs: {  
        //core-js的版本  
        version: 3,  
      },  
      //需要兼容的浏览器  
      targets: {  
        chrome: "60",  
        firefox: "60",  
        ie: "9",  
        safari: "10",  
        edge: "17",  
      },  
    },  
  ],  
],  
],
```

第三步：webpack打包

Asset	Size	Chunks	Chunk Names
index.html	272 bytes	[emitted]	
js/built.js	119 KiB	main	[emitted] main

lalalal

## 4.6 html, js代码压缩

js压缩：设置mode:"production"，去除空格和注释

html压缩：去除空格和注释

```
new HtmlWebpackPlugin({
  template: "./src/index.html",
  minify: {
    collapseWhitespace: true,
    removeComments: true,
  },
}),
```

## 4.7 生产环境的配置总结

```
const { resolve } = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const optimizeCssAssetsWebpackPlugin = require("optimize-css-assets-
webpack-plugin");
```

```
// 设置nodejs的环境变量，指定使用package.json中的browserlist的哪个环境
process.env.NODE_ENV = "production";
```

```
// 复用
```

```
const commonCssLoader = [
  MiniCssExtractPlugin.loader,
  "css-loader",
  {
    // css样式的兼容
    // 还需要再package.json中指定开发和生产环境兼容的版本
    loader: "postcss-loader",
    options: {
      ident: "postcss",
      plugins: () => {
        require("postcss-preset-env")();
      },
    },
  },
];
```

```
module.exports = {
  entry: "./src/js/index.js",
  output: {
    filename: "js/built.js",
    path: resolve(__dirname, "build"),
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [...commonCssLoader],
      },
    ],
  },
};
```



```

    {
      test: /\.less$/,
      use: [...commonCssLoader, "less-loader"],
    },
    // 正常来讲一个文件只能同时被一个loader处理, 先执行babel-loader, 再执行
eslint-loader
    // eslint-loader    js语法检查
    {
      test: /\.js$/,
      exclude: /node_modules/, // 排除这个
      enforce: "pre", // 指定eslint-loader先执行
      loader: "eslint-loader",
      options: {
        // 自动修复代码中存在的格式问题
        fix: true,
      },
    },
    // es6转js
    {
      test: /\.js$/,
      loader: "babel-loader",
      exclude: /node_modules/, // 排除第三方库的检查
      options: {
        // "presets": "@babel/preset-env"
        presets: [
          [
            "@babel/preset-env",
            // 按需加载
            {
              useBuiltIns: "usage",
              corejs: {
                // core-js的版本
                version: 3,
              },
              // 需要兼容的浏览器
              targets: {
                chrome: "60",
                firefox: "60",
                ie: "9",
                safari: "10",
                edge: "17",
              },
            },
          ],
        ],
      },
    },
    // png, jpg, gif
    {

```

```
test: /\. (png|jpg|gif)$/ ,
loader: "url-loader",
options: {
  limit: 12 * 1024,
  name: "[hash:10].[ext]",
  outputPath: "imgs", // 设定输出路径
  esModule: false, //
},
},
// 处理html文件中的图片
{
  test: /\.html$/ ,
  loader: "html-loader",
},
//打包其他资源（除了html,css,js之外的资源）
{
  // 排除js,css,html文件
  exclude: /\. (js|css|html)$/ ,
  loader: "file-loader",
  options: {
    name: "[hash:10].[ext]",
    outputPath: "media",
  },
},
],
},
plugins: [
  new HtmlWebpackPlugin({
    template: "./src/index.html",
    // html压缩
    minify: {
      collapseWhitespace: true,
      removeComments: true,
    },
  }),
  // 提取css为单独的文件
  new MiniCssExtractPlugin({
    // 对输出的文件进行重命名
    filename: "css/index.css",
  }),
  // 压缩css
  new optimizeCssAssetsWebpackPlugin(),
],
mode: "production",
};
```

## 5 webpack性能优化

开发环境优化：

- 1 优化打包构建速度
- 2 代码调试快速定位

生产环境优化：

- 1 优化打包速度
- 2 提升用户体验感

### 5.1 开发环境中的性能优化

#### 5.1.1 HMR

使用webpack-dev-server可以实现自动打包，但是目前存在的问题是它每次一个模块发生更新，它就会直接重新打包所有的模块，导致速度较低

HMR:hot module Replacement ，一个模块发生变化，只会重新打包这个模块，而不是重新打包所有的模块，极大提升构建速度，基于dev-server

注意版本号：

```
"webpack": "^5.4.0",  
"webpack-cli": "^3.3.12",  
"webpack-dev-server": "^3.11.0"
```

配置：

其他的开发配置：

```
// 开启自启动浏览器，自更新服务  
devServer: {  
  contentBase: resolve(__dirname, "build"),  
  compress: true,  
  port: 3000,  
  open: true,  
  hot: true,    // 开启HMR功能  
},
```

```
[HMR] Waiting for update signal from WDS...
3
[WDS] Hot Module Replacement enabled.
[WDS] Live Reloading enabled.
```

## 开发环境配置----



width: 400px;

修改了css文件的内容, 会导致修改的部分的样式重新打包并且执行

```
[HMR] Waiting for update signal from WDS...
```

3

```
[WDS] Hot Module Replacement enabled.
```

```
[WDS] Live Reloading enabled.
```

```
2 [WDS] App updated. Recompiling...
[WDS] App hot update...
[HMR] Checking for updates on the server...
[HMR] Updated modules:
▶ [HMR] - ./src/css/index.less
[HMR] App is up to date.
```

## 开发环境配置----



存在的问题:

只能处理样式文件的重新打包处理, 因为style-loader内部实现了该功能

js文件, 默认不能使用HMR功能

html文件, 默认不能使用HMR功能, 不用做HMR功能, 因为主界面就一个html文件, 更新时肯定更新的是那个文件

html文件实现HMR功能:

```
// 入口文件: 将html文件加入入口
entry: ["./src/js/index.js", "./src/index.html"]
```

js文件实现HMR功能:

- 修改js代码, 添加支持HMR功能的代码
- 只能处理非入口js文件

```
./index.js
import "../css/iconfont.css";
import "../css/index.less";
import print from "./print";

function add(a, b) {
  return a + b;
}

console.log(add(1, 2));

if (module.hot) {
  module.hot.accept("./print.js", function () {
    // 方法会监听print.js的变化, 一旦发生变化, 其他模块不会重新打包构建, 执行该回调
```

```

    print();
  });
}

./print.js
function print() {
  console.log("print.js执行了");
}

export default print;

```

```
[HMR] Waiting for update signal from WDS...
```

```
3
```

```
[WDS] Hot Module Replacement enabled.
```

```
[WDS] Live Reloading enabled.
```

```
console.log("print.js执行了~~~~");
```

```
[HMR] Waiting for update signal from WDS...
```

```
3
```

```
[WDS] Hot Module Replacement enabled.
```

```
[WDS] Live Reloading enabled.
```

```
2 [WDS] App updated. Recompiling...
```

```
[WDS] App hot update...
```

```
[HMR] Checking for updates on the server...
```

```
print.js执行了~~~~
```

```
[HMR] Updated modules:
```

```
[HMR] - ./src/js/print.js
```

```
[HMR] App is up to date.
```

## 5.1.2 代码调试快速定位

source-map: 一种提供源代码到构建后代码的映射技术, 如果构建后代码出错了, 通过映射关系可以追溯到源代码的错误

inline-source-map: 内联, 只生成一个内联的source-map

hidden-source-map: 外部

eval-source-map: 内联, 每个文件都生成对应的source-map

配置:

```

module.exports = {
  devtools: 'source-map'
}

```

## 5.2 生产环境性能优化

### 5.2.1 oneOf

oneOf: 以下loader只会执行一个, 不能有两个loader执行同一个loader

```

modules: {
  rules: [
    // eslint-loader
    {
    },
    {
      oneOf: [
        // css

```

```

        // less
        // babel-loader
    ]
  },
]
}

```

## 5.2.2 缓存

### 1. babel缓存:

```

{
  test: /\.js$/,
  loader: "babel-loader",
  exclude: /node_modules/, // 排除第三方库的检查
  options: {
    presets: [
      [
        "@babel/preset-env",
        // 按需加载
        {
          useBuiltIns: "usage",
          corejs: {
            //core-js的版本
            version: 3,
          },
          //需要兼容的浏览器
          targets: {
            chrome: "60",
            firefox: "60",
            ie: "9",
            safari: "10",
            edge: "17",
          },
        },
      ],
    ],
    // 开启babel缓存，第二次构建时会读取之前的缓存
    cacheDirectory: true,
  },
},

```

### 2. 文件资源缓存

文件资源缓存：存在一个问题，每次打包都是产生相同的文件名，例如built.js等，如果第一次打包，则会产生缓存；第二次访问发现文件名相同，则会直接从缓存中查找。如果第一次缓存后，修改了某个文件的内容，并且重新打包了，则由于文件名相同，则浏览器会认为该资源不用请求，从而导致新的更新没有立刻被获取到

解决：每次打包都会产生一个hash值，考虑在文件名中添加hash值，这样每次打包后的文件名都不相同，从而可以实时更新缓存

### 1. 修改文件名，每次打包都产生一个唯一的hash值

```
filename: "js/built.[hash:10].js",  
filename: "css/index.[hash:10].css"
```

问题：所有资源添加的hash值相同，则每次重新打包，所有资源的文件名都会修改，从而所有资源都需要重新请求，浪费

### 2. chunkhash: 根据chunk生成的hash值，打包来自于同一个chunk, 则hash值相同

```
filename: "js/built.[chunkhash:10].js",  
filename: "css/index.[chunkhash:10].css"
```

对于属于同一个依赖图中的js和css文件来说，它们同属于一个chunk, 所以生成的hash值是相同的

问题：这样如果一个chunk内的其中一个文件发生变化，会导致最终生成的同属于一个chunk的其他bundle文件名也发生变化，从而导致不必要的更新

### 3. contenthash: 根据文件的内容生成hash值，所以js文件和css文件的hash值不同

```
filename: "js/built.[contenthash:10].js",  
filename: "css/index.[contenthash:10].css"
```

在使用contenthash时，js文件和css文件的hash值不同，所以某个文件发生变化，重新打包后，只有内容变化的文件才会产生新的文件名，从而只会重新请求该文件，其余文件名未发生变化的文件则直接从缓存中获取即可

```
PS E:\workspaces\note\webpack\5.其他\23_babel缓存> webpack  
Hash: 671d964d5767f2a8a34e  
Version: webpack 4.39.2  
Time: 8309ms  
Built at: 2021/05/30 下午5:00:28
```

Asset	Size	Chunks	Chunk Names
css/index.css	13 bytes	0 [emitted]	main
index.html	277 bytes	[emitted]	
js/built.js	1010 bytes	0 [emitted]	main

不使用hash值

```
Hash: 671d964d5767f2a8a34e  
Version: webpack 4.39.2  
Time: 7487ms  
Built at: 2021/05/30 下午5:09:14
```

Asset	Size	Chunks	Chunk Names
css/index.671d964d57.css	13 bytes	0 [emitted]	main
index.html	299 bytes	[emitted]	
js/built.671d964d57.js	1010 bytes	0 [emitted]	main

Entrypoint main = css/index.671d964d57.css js/built.671d964d57.js

使用hash值，每次打包，不管文件内容是否变化，都会产生不同的文件名

```
Hash: a4166d74b929849d8045  
Version: webpack 4.39.2  
Time: 6228ms  
Built at: 2021/05/30 下午5:11:23
```

Asset	Size	Chunks	Chunk Names
css/index.a4166d74b9.css	28 bytes	0 [emitted]	main
index.html	299 bytes	[emitted]	
js/built.77887121b2.js	1010 bytes	0 [emitted]	main

使用contenthash, 只有内容变化的文件才会产生新的文件名

Name	Path	Method	Status	Type	Initiator	Size	Time
localhost	/	GET	200	document	Other	618 B	29 ms
index.17de3eba30.css	/css/index....	GET	200	stylesheet	(index)	330 B	77 ms
built.3181286e4c.js	/js/built.31...	GET	200	script	(index)	1.3 kB	53 ms



修改js文件，css直接从缓存中获取，js发送请求获取

Name	Path	Method	Status	Type	Initiator	Size
localhost	/	GET	200	document	Other	618 B
index.17de3eba30.css	/css/index....	GET	200	stylesheet	(index)	(disk cache)
built.3181286e4c.js	/js/built.31...	GET	200	script	(index)	1.3 kB

## 5.2.2 tree\_shaking

前提：

1. 必须使用ES6模块化 `import { fun1 } from './print';`
2. 开启production环境

作用：只会打包入口文件中使用了的函数

比如在./print.js中声明了两个函数fun1, fun2, 在入口文件中只使用了fun1, 则打包后的结果中只会包含fun1; 如果在入口文件中引入了fun2, 但是没有使用，则打包会出错

package.json中：

如果配置了"sideEffects":false; 表示所有的代码都是没有副作用的代码，都可以进行treeshaking, 从而会导致将入口文件中的css文件treeshaking掉，不会被打包，因为在入口文件中样式并没有执行

如果不希望样式文件被treeshaking, 则"sideEffects":["\*.css"];

## 5.2.3 代码分割

单入口：后两种方式皆可

多入口：三种方式皆可

### 1.配置方式1

```
// 多入口，几个入口就对应几个chunk，一个chunk可以包含多个bundle(js和css文件等)
entry: {
  main: "./src/js/index.js",
  test: "./src/js/print.js",
},
output: {
  // name就是main和test
  filename: "js/[name].[contenthash:10].js",
  path: resolve(__dirname, "build"),
},
```

下面就包含两个chunks，第一个chunk包含两个bundle文件：css和js, 第二个chunk包含一个js文件



Asset	Size	Chunks		Chunk Names
css/index.17de3eba30.css	15 bytes	0	[emitted] [immutable]	main
index.html	343 bytes		[emitted]	
js/main.541ec397dd.js	1010 bytes	0	[emitted] [immutable]	main
js/test.819c26e1d5.js	948 bytes	1	[emitted] [immutable]	test

## 2. 配置方式2

对于单入口：optimization可以将node\_modules中的代码单独打包成一个chunk输出，我们自己设置的入口会作为一个单独的chunk打包

```
module.exports = {
  entry:main: "./src/js/index.js",
  output:{filename: "js/built.[contenthash:10].js"},
  plugins:[],
  // 可以将node_modules中的代码单独打包成一个chunk输出，我们自己设置的入口会作为一个单独的chunk打包
  optimization:{
    splitChunks:{
      chunks:'all',
    }
  }
}
```

对于多页面应用，（多入口）：optimization可以将每个入口都打包成一个chunk,对于第三方库，会生成一个共同的chunk

```
entry: {
  main: "./src/js/index.js",
  test: "./src/js/print.js",
},
output: {
  // name就是main和test
  filename: "js/[name].[contenthash:10].js",
  path: resolve(__dirname, "build"),
},
optimization:{
  splitChunks:{
    chunks:'all',
  }
}
```

优点：如果main和test中均引入了Math模块，在只会打包产生一个Math的chunk,这样方便两个页面都可以使用

## 3. 配置方式3

在单入口配置的情况下，在入口文件中设置将另一个js文件单独打包

```
import('./print')
.then((result)=>{
  console.log('文件加载成功!!!');
});
```

```

        console.log(result);// result是整个module对象, result.fun1
    })
    .catch(()=>{
        console.log('文件加载失败!!!');
    })

```

配置:.eslintrc文件, 解决import动态引入的问题

```

{
  "parserOptions": {
    "ecmaVersion": 2017,
    "sourceType": "module",
    "parser": "babel-eslint"
  },
  "extends": ["eslint:recommended", "prettier"]
}

```

设置打包文件的名字:

```

output: {
  chunkFilename: "js/[name].chunk.js",
}

// 设置名字为print
import(/*webpackChunkName:'[print]'/ " ./print")
  .then((result) => {
    console.log("文件加载成功!!!");
    console.log(result);
  })
  .catch(() => {
    console.log("文件加载失败!!!");
  });

```



```

JS [print].chunk.js
JS built.30e65b4b2d.js
JS vendors~main.chunk.js

```

但是这个方法存在一个问题: 如果在index.js中使用import()加载和打包了print.js,则会在生成的built.js中产生一个记录该文件的hash值,从而当print.js发生变化,则它的contenthash值就变了,从而导致index.js中的内容也会产生更新,从而index.js文件的hashcontent也会发生变化,导致缓存失效。



解决:

```

optimization: {
  splitChunks: {
    chunks: 'all',
  },
  // 将当前模块中记录其他模块的hash单独打包为一个文件 runtime
  runtimeChunk: {
    name: entrypoint => `runtime-${entrypoint.name}`
  }
}

```

runtime将A模块中引用了B模块的部分单独打包成一个runtime模块，从而每次都只会更新B.js和runtime.js



## 5.2.4 懒加载和预加载

### 1. 懒加载LazyLoading

在指定条件满足时，再加载，执行

```

./index.js

document.getElementById("btn").onclick = function () {
  import(/*webpackChunkName: '[print]'*/ './print')
    .then((result) => {
      console.log("文件加载成功!!!");
      console.log(result);
    });
}

```

```
    })
    .catch(() => {
      console.log("文件加载失败!!!");
    });
  });
};
```

第一次加载完之后，第二次就会自动读取缓存

## 2. 预加载

添加：webpackPrefetch:true，在页面第一次显示时就加载好，在指定条件成立后即不需要再加载，立刻执行

```
// 设置名字为print
document.getElementById("btn").onclick = function () {
  import(/*webpackChunkName:'[print]',webpackPrefetch:true*/ "./print")
    .then((result) => {
      console.log("文件加载成功!!!");
      console.log(result);
    })
    .catch(() => {
      console.log("文件加载失败!!!");
    });
};
```

## 5.2.5 PWA

PWA: (Progressive Web App)渐进式网络开发应用程序（离线可访问），通过workbox使用

第一步：安装插件

```
yarn add workbox-webpack-plugin -D
```

第二步：帮助serviceworker快速启动，删除旧的serviceworker，生成一个serviceworker的配置文件

```
var workboxWebpackPlugin = require('workbox-webpack-plugin');
plugins:[
  new workboxWebpackPlugin.GenerateSW({
    clientsClaim:true,
    skipWaiting:true
  })
]
```

eslint不认识浏览器的全局变量，需要修改配置,使其可以识别浏览器端的全部变量，如果需要识别nodejs的全局变量，则设置为node

```
"eslintConfig": {
  "extends": "airbnb-base",
  "env": {
    "browser": true
  }
}
```

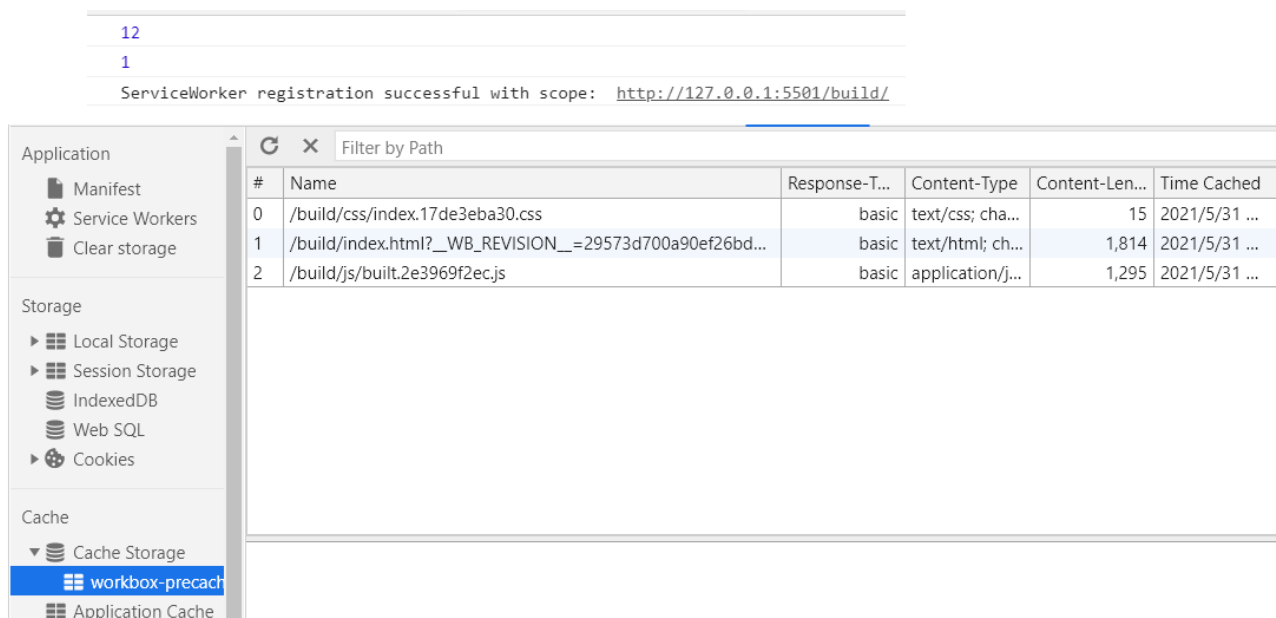
## 注册service-worker

```
index.js
// 注册serviceworker
if (navigator.serviceWorker) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('service-worker.js').then(
      (registration) => {
        // Registration was successful
        console.log(
          'ServiceWorker registration successful with scope: ',
          registration.scope,
        );
      },
      (err) => {
        // registration failed :(
        console.log('ServiceWorker registration failed: ', err);
      },
    );
  });
}
```

注意: **webpack**的版本号要在**4.40.0**及至上, 因为其中用到的一些方法是在这个版本之后加入的

**serviceworker**代码必须运行在服务器上, 安装**serve**,运行,也可以使用**vscode**自带的服务器

```
yarn add serve -g
serve -s build
```



之前在[http交互](#)中是在代码阶段自定义对于静态文件的缓存，这里直接在打包阶段实现文件缓存

注意代码执行的过程中可能会报错误，`TypeError: Failed to register a ServiceWorker for scope ('http://127.0.0.1:5000')`，经过检查发现文件名写错了，注意细心一些

## 5.2.6 多进程打包

开启多进程打包

- 如果使用的好，则可以提升打包速度
- 但是由于进程开启需要时间，大约是600ms，进程之间还需要通信，同样需要花费时间，所以只有工作时间消耗很长的情况下，才会使用多进程打包

下载包：

```
yarn add thread-loader -D
```

修改配置：

```
{
  test: /\.js$/,
  exclude: /node_modules/, // 排除第三方库的检查
  use: [
    // 指定多进程打包
    {
      loader: "thread-loader",
      options: {
        workers: 2, // 设置进程数为2
      },
    },
    {
      loader: "babel-loader",
      options: {
        // "presets": "@babel/preset-env"
      }
    }
  ]
}
```

```
presets: [
  [
    "@babel/preset-env",
    // 按需加载
    {
      useBuiltIns: "usage",
      corejs: {
        //core-js的版本
        version: 3,
      },
      //需要兼容的浏览器
      targets: {
        chrome: "60",
        firefox: "60",
        ie: "9",
        safari: "10",
        edge: "17",
      },
    },
  ],
],
},
],
}
```

## 5.2.7 externals

### 第一步：在配置中忽略入口文件中使用到的某些库的打包

```
module.exports = {
  entry:,
  output:,
  ...,
  externals: {
    // 忽略对于某些库的打包
    jquery: "jQuery",
  },
}
```

第二步：在html中引入该库对应的CDN，即网络资源

```
<script
src="https://cdn.bootcdn.net/ajax/libs/jquery/3.6.0/jquery.min.js">
</script>
```

第三步：打包即可，可以发现jquery并未被打包，但是由于在html文件中引入了，所以打包后的文件依旧可以使用

cdn链接可以在: [bootcdn.cn](http://bootcdn.cn)中查找

## 5.2.8 dll

指定webpack那些库不需要打包，并且会将某些库打包成一个单独的chunk

第一步：新建一个配置文件webpack.dll.js,这个配置文件会产生一个单独的chunk

```
const { resolve } = require("path");
const webpack = require("webpack");
// 对某些库进行单独打包，一般值第三方库
module.exports = {
  entry: {
    // 最终打包生成的name是jquery, ['jquery']指定打包的库
    jquery: ["jquery"],
  },
  output: {
    filename: "[name].js",
    path: resolve(__dirname, "dll"),
    library: "[name]_[hash]", // 指定打包的库向外暴露出去的内容的名字
  },
  plugins: [
    // 打包生成一个manifest.json文件，提供和jquery的映射
    new webpack.DllPlugin({
      name: "[name]_[hash]", //映射库暴露的名称
      path: resolve(__dirname, "dll/manifest.json"),
    }),
  ],
  mode: "production",
};
```

当运行webpack时，默认查找的是webpack.config.js配置文件，而我们现在需要运行webpack.dll.js文件，需要使用webpack --config webpack.dll.js



在webpack.config.js中：

```
const webpack = require("webpack");
const addAssetHtmlWebpackPlugin = require("add-asset-html-webpack-plugin");
plugins: [
  // 告诉webpack哪些库不参与打包，同时使用的名称也发生了变化，打包时需要注意修改名称
  new webpack.DllReferencePlugin({
    manifest: resolve(__dirname, "dll/manifest.json"),
  }),
  // 将某个文件单独打包输出，并在html中自动引入，从而将单独打包的第三方库和打包后的文件联系起来
```



```
new addAssetHtmlWebpackPlugin({
  filepath: resolve(__dirname, "dll/jquery.js"),
}),
],
```

使用webpack命令打包得到：



问题：**jquery**同样被打包了，就有点迷，但是可以看到它是打包成了一个单独的**chunk**，直接利用**dll**中打包得到的结果，内容和**dll**中的打包结果**jquery.js**相同

## 5.2.9 性能优化总结

开发环境的优化：

- HMR
- 代码调试优化 source-map

生产环境优化：

- 优化打包构建速度，提升开发者的体验
  - oneOf
  - babel缓存
  - 多进程打包
- 优化代码运行的性能
  - 文件缓存 (hash, chunkhash, contenthash)
  - tree-shaking: 去除没有使用的代码，从而代码体积小，请求快
  - 代码分割 code split
  - 懒加载，预加载 js模块
  - PWA
  - externals
  - dll

# 6 其他

## 6.1 entry的详细配置

```
module.exports = {
  entry: './src/index.js',
  output: {
    // filename: '[name].js',    默认打包后的名称是main.js
    filename: "built.js",
    path: resolve(__dirname, "build"),
  },
  plugins: [new htmlWebpackPlugin()],
  mode: "development",
};
```

// 数组形式的多入口,只会产生一个chunk,一般只会用在HMR功能中让html的热更新生效

```
module.exports = {
  entry: ['./src/index.js', './src/print.js'],
  output: {
    // filename: '[name].js',    默认打包后的名称是main.js,形成一个chunk
    filename: "built.js", // 形成一个chunk
    path: resolve(__dirname, "build"),
  },
  plugins: [new htmlWebpackPlugin()],
  mode: "development",
};
```

// 对象形式的多入口,有几个入口文件就有几个chunk,产生几个bundle文件

```
module.exports = {
  entry: { main: './src/index.js', print: './src/print.js' },
  output: {
    filename: "[name].js", // 会形成多个chunk
    path: resolve(__dirname, "build"),
  },
  plugins: [new htmlWebpackPlugin()],
  mode: "development",
};
```

//混合形式的多入口,有几个入口文件就有几个chunk,产生几个bundle文件

```
module.exports = {
  entry: {      // 两个chunk
    index: ['./src/index.js', './src/print.js'],
    add: './src/add.js'
  },
  output: {
    filename: "[name].js", // 会形成多个chunk
    path: resolve(__dirname, "build"),
  },
  plugins: [new htmlWebpackPlugin()],
```

```
mode: "development",
};
```

在dll中对于jquery打包时就使用了这种用法

## 6.2 output详细配置

```
module.exports = {
  entry: "./src/index.js",
  output: {
    // filename: 'js/[name].js',    默认打包后的名称是main.js
    // 指定入口文件打包后的输出的文件名称
    filename: "built.js",
    // 指定输出的文件目录，所有资源打包的公共目录
    path: resolve(__dirname, "build"),
    // 指定所有资源引入的公共路径
    publicPath: '/',
    chunkFilename: 'js/[name]_chunk.js', // 指定非入口chunk的名称
    library: '[name]', // 整个库向外暴露的变量名
    // libraryTarget: 'window' // 变量名添加到哪个对象上，window global等
    libraryTarget: 'commonjs' // 通过模块化的语法引入
  },
  plugins: [new htmlWebpackPlugin()],
  mode: "development",
};
```

publicPath:

比如打包后，入口文件在html中引入了，则不加publicPath时，路径是js/main.js  
加入publicPath: '/' 后得到的是: /js/main.js ,是从服务器的根路径开始计算的

chunkFilename:

例如使用import()导入某个模块，会造成该模块的单独打包，默认情况下该模块是使用id命名的，即0,1,2等，设定chunkFilename: 'js/[name]\_chunk.js'后，该文件就会被放到js目录下，并且命名为0\_chunk.js

## 6.3 module的详细配置

```
module:{
  rulers:[
    {
      test:/\.css$/,
      use:['style-loader','css-loader'],
    },
    {
      test:/\.js$/,
      exclude:/node_modules/, // 排除第三方库的模块的检查
      include:resolve(__dirname,'src'), // 只检查src下的js文件
      enforce:'pre', //优先执行
      // enforce:'post', //延后执行
    }
  ]
}
```

```

        loader:eslint-loader,
        options:{

        }
    },
    {
        // 以下配置只会生效其中一个
        oneOf:[
            {},
            {}
        ]
    }
]
}

```

## 6.4 resolve详细配置

```

module.exports = {
  entry: {      // 两个chunk
    index:["./src/index.js", "./src/print.js"],
    add:"./src/add.js"
  },
  output: {
    filename: "[name].js", // 会形成多个chunk
    path: resolve(__dirname, "build"),
  },
  module:{
    rulers:[
      {
        test:/\.css$/,
        use:['style-loader','css-loader'],
      },
    ]
  }
  plugins: [new htmlWebpackPlugin()],
  mode: "development",
  // 解析模块的规则
  resolve: {
    // 配置解析模块路径别名,可以简写路径,缺点路径就不会提示了
    alias:{
      $css:resolve(__dirname,'src/css'),
    },
    // 配置省略文件路径的后缀名
    extentions:['.js','css','json'],
    // 告诉webpack, 解析模块时去哪个目录
    modules:[
      resolve(__dirname,'../..../node_modules'),
      'node_modulse'
    ]
  }
}

```

```
}  
};
```

alias: 这样写在入口文件中引入样式表时就可以直接写import '\$css/index.css', webpack就会自动去对应的位置resolve(\_\_dirname, 'src/css')找到该样式文件

对于extentions, 在入口文件中引入样式时就可以直接不写后缀名, webpack就会依次匹配数组中的后缀, 找到一个存在的文件, 则就直接加载该文件。问题: 如果路径下同时存在print.js和print.css, 则容易发生混乱

## 6.5 devServer详细配置

基于开发环境:

```
devServer: {  
  contentBase: resolve(__dirname, 'build'),  
  watchContentBase: true, // 监视文件目录contentBase下的所有文件, 一旦文件变化, 就reload  
  watchOptions: {  
    ignore: /node_modules/, // 忽略第三方库文件变动的监视  
  }  
  // 启动gzip压缩  
  compress: true, // 启动gzip压缩  
  port: 5000, // 开启端口号  
  host: 'localhost', // 域名  
  open: true, // 自动打开浏览器  
  hot: true, // 开启HMR功能  
  clientLogLevel: 'none', // 不显示启动服务器的日志信息  
  quiet: true, // 除了一些基本的启动信息外, 其他内容都不要打印  
  overlay: false, // 一旦出错, 不要全屏提示~  
  proxy: { // 服务器代理, 开发环境的跨域问题  
    '/api': {  
      // 一旦服务器接收到/api/xxx的请求, 就会将该请求转发给另一个服务器3000  
      target: 'http://localhost:3000'  
      pathRewrite: {  
        '^/api: ''', // 路径重写, 将/api/xxx修改为/xxx  
      }  
    }  
  }  
}
```

## 6.6 optimization的详细配置

生产环境: 分割代码

```
// 安装4.x版本, webpack4.x与terser-webpack-plugin5.x不兼容  
const terserWebpackPlugin = require('terser-webpack-plugin');
```

```

optimization: {
  splitChunks: {
    chunks: 'all',
    minSize: 30*1024, // 分割的chunk最小为30kb
    maxSize: 0, // 最大没有限制
    minChunks: 1, // 要提取的chunk最少被引用一次
    maxAsyncRequest: 5, // 按需加载时并行加载的文件的最大数量为5, chunk数最大为5

    maxInitialRequest: 3, // 入口js文件最大并行请求数量
    automaticNameDelimiter: '~', // 名称连接符
    name: true, // 可以使用命名规则
    cacheGroups: {
      // 分割chunk的组

      node_modules中的文件会被打包到vendors组的chunk中
      vendors: {
        test: /[\\/]node_modules[\\/]$/,
        priority: -10, // 打包优先级
      },
      default: {
        minChunks: 2, // 要提取的chunk最少被引用两次
        priority: -20, // 打包优先级
        reuseExistingChunk: true, // 如果当前要打包的模块和之前已经被
        提取的模块是同一个, 则复用, 不再重新打包
      },
    },
  },
  // 将当前模块中记录其他模块的hash单独打包为一个文件 runtime
  runtimeChunk: {
    name: entrypoint => `runtime-${entrypoint.name}`
  },
  minimizer: {
    // 配置生产环境的压缩方案: js和css
    new terserWebpackPlugin({
      // 开启缓存
      cache: true,
      parallel: true, // 开启多进程打包
      source-map: true
    })
  }
}

```

## 7 Webpack 5

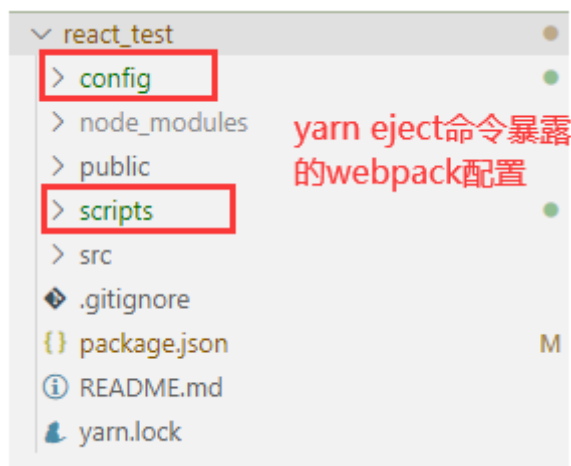
## 7.1 react中的webpack

第一步：使用react脚手架新建react项目

```
npx create-react-app react_test
```

第二步：将webpack配置暴露出来，这个动作是一个不可逆的动作

```
yarn eject
```



可以发现此时package.json中的启动项目的命令变成了原始node命令，不再是使用包启动

```
"scripts": {
  "start": "node scripts/start.js",    // 代表开发环境
  "build": "node scripts/build.js",    // 代表生产环境
  "test": "node scripts/test.js"       // 代表测试环境
},
```

## 7.2 loader

### 7.2.1 loader本身

loader本身就是一个函数

```
console.log("loader", 111);
// 自定义loader
module.exports = function (content, map, meta) {
  // content就是整个loader.js的全部代码,string类型
  console.log(111);
  this.callback(null, content, map, meta);
  // return content;
};

module.exports.pitch = function () {
  console.log("pitch", 111);
};
```

webpack.config.js中设置了多个loader时的执行顺序：

```

var path = require("path");

console.log(__dirname);

module.exports = {
  // 默认入口文件就是src/index.js,输出位置:
  module: {
    rules: [
      {
        test: /\.js$/,
        // loader执行顺序是从由往左执行
        // 但是loader的pitch方法是从左向右执行的
        use: ["loader1", "loader2", "loader3"],
      },
    ],
  },
  // 配置loader的解析规则
  resolveLoader: {
    // 配置loader的解析路径,如果在node_modules中没有找到对应的loader,则自动
    去loaders目录下查找
    modules: ["node_modules", path.resolve(__dirname, "loaders")],
  },
};

```

其中loader2,loader3都是和loader1相同的js文件,得到的执行结果是:

```

PS E:\workspaces\note\webpack\webpack5\loader> webpack
E:\workspaces\note\webpack\webpack5\loader
pitch 111
pitch 222
pitch 333
333
222
111

```

## 7.2.2 同步loader和异步loader

loader的执行分为同步执行和异步执行,

同步loader:



```
// 同步loader
console.log("loader", 111);
// 自定义loader, loader本身就是一个函数
module.exports = function (content, map, meta) {
  // content就是整个loader.js的全部代码, string类型
  console.log(111);
  this.callback(null, content, map, meta);
  return content;
};

module.exports.pitch = function () {
  console.log("pitch", 111);
};
```

异步loader:

```
// 同步loader
console.log("loader", 111);
// 自定义loader, loader本身就是一个函数
module.exports = function (content, map, meta) {
  // content就是整个loader.js的全部代码, string类型
  console.log(111);
  const callback = this.async();
  setTimeout(function () {
    callback();
    console.log("异步loader", 1111);
  }, 1000);
};

module.exports.pitch = function () {
  console.log("pitch", 111);
};
```

```
PS E:\workspaces\note\webpack\webpack5\loader> webpack
E:\workspaces\note\webpack\webpack5\loader
pitch 111
pitch 222
pitch 333
333
222
异步loader 333
111
异步loader 222
异步loader 1111
```

## 7.2.3 loader的options

```
webpack.config.js

use: [
  "loader1",
  "loader2",
```

```

{
  loader: "loader3",
  options: {
    name: "whh",
  },
},
],

```

在loader3.js中获取所设置的options的值:

1. 下载loader-utils库
2. getOptions(this) 获取对应的值

```
const { getOptions } = require("loader-utils");
```

// 自定义loader

```
module.exports = function (content, map, meta) {
  // content就是整个loader.js的全部代码,string类型
  console.log(333);
  const options = getOptions(this);
  console.log(options); // { name: 'whh' }
};
```

定义校验规则: 新建schema.json

```

{
  "type": "object",    // 首先得是一个对象
  "properties": {
    "name": {          // 检验name属性
      "type": "string", // 属性值必须是字符串
      "description": "名称~"
    }
  },
  "additionalProperties": true // 可额外添加属性,即options可以不只有name这个属性
}
```

在loader3.js中获取该规则:

```
const { getOptions } = require("loader-utils");
const { validate } = require("schema-utils");
```

```
const schema = require("../schema.json");
```

// 自定义loader

```
module.exports = function (content, map, meta) {
  // content就是整个loader.js的全部代码,string类型
  console.log(333);
  // 获取options
  const options = getOptions(this);
  console.log(options); // { name: 'whh' }
```

```
// 校验options是否合法
validate(schema, options, {
  name: "loader3",
});
};
```

此时webpack打包，则不会发生错误，因为loader3.js中的option就是一个对象，并且它的name属性是string类型，如果修改为其他类型，则会报错

## 7.2.4 自定义babel-loader

### webpack.config.js

```
var path = require("path");

module.exports = {
  // 默认入口文件就是src/index.js,输出位置:
  module: {
    rules: [
      {
        test: /\.js$/,
        loader: "babelLoader",
        options: {
          // 需要安装这个库
          presets: ["@babel/preset-env"],
        },
      },
    ],
  },
  // 配置loader的解析规则
  resolveLoader: {
    // 配置loader的解析路径，如果在node_modules中没有找到对应的loader，则自动去loaders目录下查找
    modules: ["node_modules", path.resolve(__dirname, "loaders")],
  },
};
```

### babelLoader.js

```
const { getOptions } = require("loader-utils");
const { validate } = require("schema-utils");
const babel = require("@babel/core");
const util = require("util");

const schema = require("../babelSchema.json");
// babel.transform是一个普通的异步方法,用于编译代码
const transform = util.promisify(babel.transform); // 将transform方法转换为promise形式的transform方法
```

```
// 自定义loader
module.exports = function (content, map, meta) {
  // 获取options配置
  const options = getOptions(this) || {}; // 如果没有设置options,则会获得一个空对象
  console.log(options); // { presets: [ '@babel/preset-env' ] }
  // 校验options是否合法
  validate(schema, options, {
    name: "babelLoader",
  });
  // 校验合格后,就可以执行异步loader了
  const callback = this.async();
  // 1. 使用Babel编译代码
  transform(content, options)
    .then(({ code, map }) => {
      callback(null, code, map, meta);
    })
    .catch((e) => {
      callback(e);
    });
};
```

## babelSchema.json

```
{
  "type": "object", // 首先得是一个对象
  "properties": {
    "name": { // 检验name属性
      "type": "array", // 属性值必须是字符串
      "description": "名称~"
    }
  },
  "additionalProperties": true // 可额外添加属性,即options可以不只有name这个属性
}
```

index.js中加入es6语法:

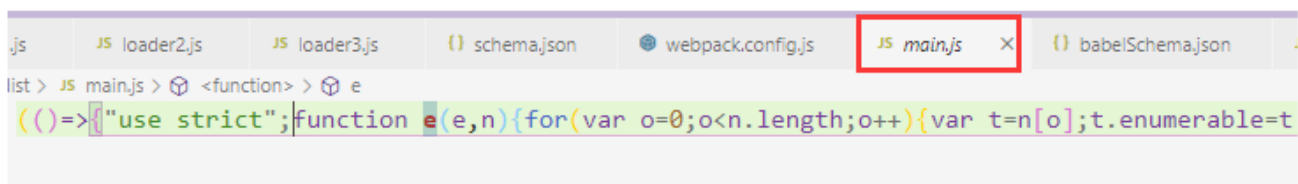
```
console.log("我在测试loader");

class Person {
  constructor(name) {
    this.name = name;
  }
  setName(name) {
    this.name = name;
  }
}

console.log(new Person("whh"));
```

使用webpack打包就可以了，es6的语法就可以实现转换了

```
PS E:\workspaces\note\webpack\webpack5\loader> webpack
{ presets: [ '@babel/preset-env' ] }
asset main.js 489 bytes [emitted] [minimized] (name: main)
./src/index.js 1.02 KiB [built] [code generated]
```



list > JS main.js > <function> > e

```
((=>["use strict";function e(e,n){for(var o=0;o<n.length;o++){var t=n[o];t.enumerable=t
```