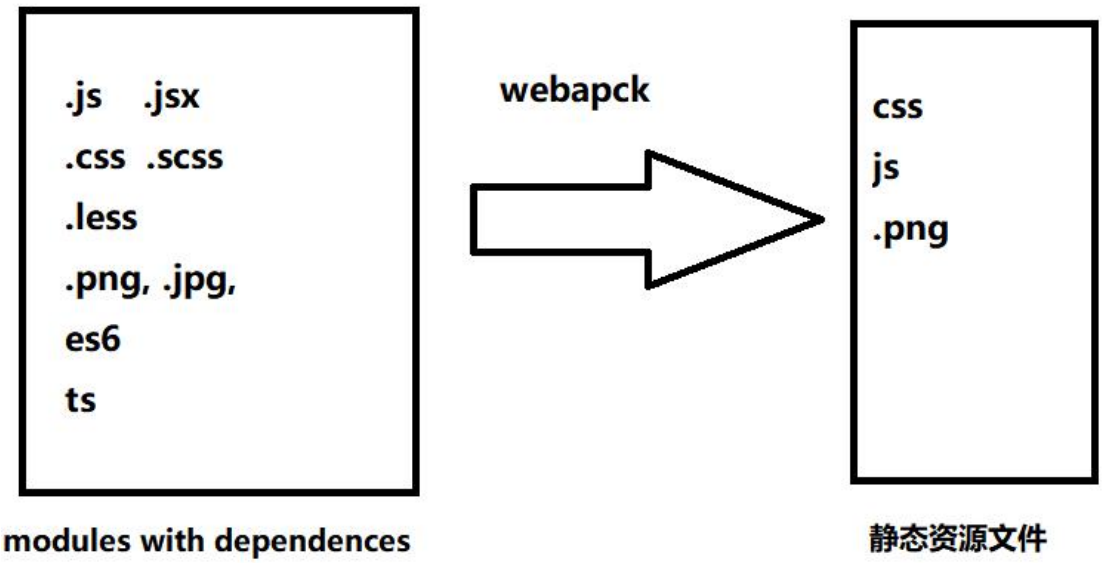
目录

什么是 WebPack?

简单地说，Webpack 其最核心的功能就是 解决模板之间的依赖，把各个模块按照特定的规则和顺序组织在一起，最终合并成一个 JS 文件（比如 bundle.js）。这个整个过程也常常被称为是模块打包。

WebPack 可以看做是模块打包机：它做的事情是，分析你的项目结构，找到 JavaScript 模块以及其它的一些浏览器不能直接运行的拓展语言（Sass，TypeScript 等），并将其转换和打包为合适的格式供浏览器使用。



为什么要用 Webpack

一直以来，在开发 Web 页面或 Web 应用程序的时候，都习惯性的将不同资源放置在不同的文件目录之中，比如图片放置在 images（或 img）下，样式文件放置在 styles(或 css)中，脚本文件放在 js 和模板文件放置在 pages 中。一直以来，发布的时候都会一次性的将所有资源打包发布，不管这些资源用到了还是没用到（事实上很多时候自己都分不清楚哪资源被使用）。用一句话来描述就是：依赖太复杂，太混乱，无法维护和有效跟踪。比如哪个样式文件引用了 a.img，哪个样式文件引用了 b.img；另外页面到底是引用了 a.css 呢还是 b.css 呢?

而 Webpack 这样的工具却能很多好的解决它们之间的依赖关系，使其打包后的结果能运行在浏览器上。其目前的工作方式主要被分为两种：

将存在依赖关系的模块按照特定规则合并成为单个.js 文件，一次性全部加载进页面

在页面初始时加载一个入口模块，其他模块异步加载

Webpack 的主要优势：

- webpack-dev-server 搭建本地环境，进行热更新
- 预处理（Less，Sass，ES6，TypeScript,……）
- 图片添加 hash,方便线上 CDN 缓存
- 自动处理 CSS3 属性前缀
- 单文件，多文件打包 。。。。
- 有完备的代码分割解决方案
- 模块化开发（import，require） ，支持多种模块标准
- 压缩 js 代码
- 复制

相比于 Parcel、Rollup 具有同等功能的工具而言，Webpack 还具有其他的优势：

- Webpack 支持多种模块标准：这对于一些同时使用多种模块标准的工程非常有用，Webpack 会帮我们处理好不同类型模块之间的依赖关系
- Webpack 有完备的代码分割解决方案：它可以分割打包后的资源，首屏只加载必要的部分，不太重要的功能放到后面动态地加载
- Webpack 可以处理各种类型的资源：除了 JavaScript 之外，Webpack 还可以处理样式、模板、图片等资源。开发者要做的只是将之些资源导入，而无需关注其他. 另外，Webpack 还拥有一个强大的社区。这也是其受开发者青睐的原因之一。接下来，我们还是实际一点，动手来撸码。

Webpack4 与 Webpack3 版本的区别：

1. 增加了 mode 属性，可以是 development 或 production

```
"scripts": {
   "dev": "webpack --mode development",
   "build": "webpack --mode production"
}
```

通过 mode，你可以轻松设置打包环境。

如果你将 mode 设置成 development，你将获得最好的开发阶段体验。这得益于 webpack 针对开发模式提供的特性：

- 浏览器调试工具
- 注释、开发阶段的详细错误日志和提示
- 快速和优化的增量构建机制

如果你将 mode 设置成了 production, webpack 将会专注项目的部署，包括以下特性：

- 开启所有的优化代码
- 更小的 bundle 大小
- 去除掉只在开发阶段运行的代码
- Scope hoisting 和 Tree-shaking

2. webpack4 在 loader,optimize 上进行了很多改动。

module.loaders 改为 module.rules

module.loaders.loader 改成 module.loaders.use

loader 的”-loader”均不可省略

3. 插件和优化

webpack4 删除了合并相同的文件 CommonsChunkPlugin 插件，它使用内置 API optimization.splitChunks 和 optimization.runtimeChunk，这意味着 webpack 会默认为你生成共享的代码块。

其它插件变化如下:
NoEmitOnErrorsPlugin 废弃，使用 optimization.noEmitOnErrors 替代，在生产环境中默认开启该插件。
ModuleConcatenationPlugin 废弃，使用 optimization.concatenateModules 替代，在生产环境默认开启该插件。
NamedModulesPlugin 废弃，使用 optimization.namedModules 替代，在生产环境默认开启。
uglifyjs-webpack-plugin 升级到了 v1.0 版本，默认开启缓存和并行功能。

之前是这样:

```
plugins : [
new webpack.optimize.CommonsChunkPlugin({
        name : 'main',
        children : true,
        minChunks : 2,
})]
```

此段代码已经在 webpack4 的高版本里面被移除
换成了如下的配置选项:

```
optimization: {
         splitChunks: {
             cacheGroups: {
                 commons: {
                     name: "commons",
                     chunks: "initial",
                     minChunks: 2
                 }
             }
         }
    }
```

4. 升级 extract-text-webpack-plugin 插件　　yarn add extract-text-webpack-plugin@next -D

Webpack5 展望

已经有不少关于 webpack5 的计划正在进行中了，包括以下:

对 WebAssembly 的支持更加稳定

支持开发者自定义模块类型

去除 ExtractTextWebpackPlugig 插件，支持开箱即用的 CSS 模块类型

支持 Html 模块类型

持久化缓存

从零开始构建你自己的开发环境

为了更好的理解 Webpack 能帮我们做什么，我打算从零开始构建一个属于自己的开发环境。可能在后面的内容中会涉及到很多关键词，比如 Webpack、loaders、Babel、sourcemaps、Vue、React、TypeScript，CSS Modules 等等。

● 配置版本说明：

node: v10.16.0

Yarn: 1.16.0

Webpack: 4.34.0

Webpack-cli: 3.3.4

演示的系统是 window ， 以及 node 的下载，安装成以后，会自带 npm 命令，如果想体验 yarn 命令 可以通过 npm install yarn -g 或者下载安装

环境支持：

当使用 webpack4 时，确保使用 Node.js 的版本 >= 8.9.4。因为 webpack4 使用了很多 JS 新的语法，它们在新版本的 v8 里经过了优化。

● 安装命令 yarn 或者 npm 的选择：

yarn 的安装和使用参考：https://blog.csdn.net/x550392236/article/details/79205812

● 全局安装　不推荐

yarn global add webpack webpack-cli

或者：　npm install webpack webpack-cli -g

可以帮助我们在命令行行里使用 webpack 等相关指令

webpack -v

● 卸载

npm uninstall webpack webpack-cli -g 或者

yarn global remove webpack webpack-cli

● 局部安装　项目内安装 推荐

yarn add webpack webpack-cli --dev/-D　或者

npm install webpack webpack-cli --save--dev/-D

--dev/-D 指安装到 devDependencies：开发时的依赖包

--save/-S　dependencies：运行程序时的依赖包

webpack -v //command not found 默认在全局环境中查找

npx webpack -v// npx 帮助我们在项目目中的 node_modules里里里查找 webpack

● 指定版本安装：

npm info webpack//查看 webpack 的历史发布信息

npm install webpack@x.xx webpack-cli -D

npm i -D 是 npm install --save-dev 的简写，是指安装模块并保存到

注意，在命令终端使用 npm i 安装依赖关系时，如果带后缀 -D（或--save-dev） 安装的包会记录在 "devDependencies" 下； 如果使用 --save 后缀安装的包会记录在 "dependencies"下。

● webpack 配置文文件

当我们使用用 npx webpack index.js 时，表示的是使用用 webpack 处理理打包，名为 index.js 的入入口口模块。默 认放在当前目目录下的 dist目目录，打包后的模块名称是 main.js，当然我们也可以修改 webpack 有默认的配置文文件，叫 webpack.config.js，我们可以对这个文文件进行行行修改，进行行行个性化配置

- 默认的配置文文件：webpack.config.js

npx webpack //执行行行命令后，webpack 会找到默认的配置文文件，并使用用执行行行

- 不使用用默认的配置文文件： webpackconfig.js

npx webpack --config webpackconfig.js //指定 webpack 使用用 webpack.config.js文文件来作为 配置文文件并执行行行

- 修改 package.json scripts 字段：有过 vue react 开发经验的同学 习惯使用用 npm run 来启动，我们也 可以修改下

"scripts":{ "bundle":"webpack"//这个地方方不不要添加 npx ,因为 npm run 执行行行的命令，会优先使用用项目目工工程里里里 的包，效果和 npx非非常类似 }
npm run bundle

https://www.jianshu.com/p/a4d2d14f4c0e

Step01:初始化项目,生成 package.json

进入到新创建的项目目录下,执行 yarn init -y 或者 npm init 生成 package.json

首先在你的本地创建一个项目，比如我这里创建了一个 webpack-sample 项目：

mkdir webpack-sample && cd webpack-sample

进入到新创建的项目目录下，执行 yarn init 或者 yarn init -y 命令来初始化项目，执行完该命令之后，在你的命令终端会看到类似下图这样的命令询问，你可以根据你自己的需要去输入你想要的内容，或者一路 Enter 键执行下去：

此时你的项目根目录下会增加一些文件和文件夹：

|--webpack-sample/

|----package.json

|----package-lock.json

其中 package.json 文件里将包含一些项目信息：

而 **package-lock.json** 文件是当 node_modules/ 或 package.json 发生变化时自动生成的文件，它的主要功能是 确定当前安装的包的依赖，以便后续重新安装的时候生成相同的依赖，而忽略项目开发过程中有些依赖已经发生的更新。

我们对 package.json 文件只做一个修改，删除"main": "index.js"入口，并添加 "private":true 选项，以便确保安装包是私有的，这样可以防止意外发布你的代码

Step02：安装 Webpack 和初始配置 Webpack

在这一步，先来安装 Webpack。执行下面的命令安装 Webpack 配置所需要的包：

yarn add webpack webpack-cli webpack-dev-server --dev

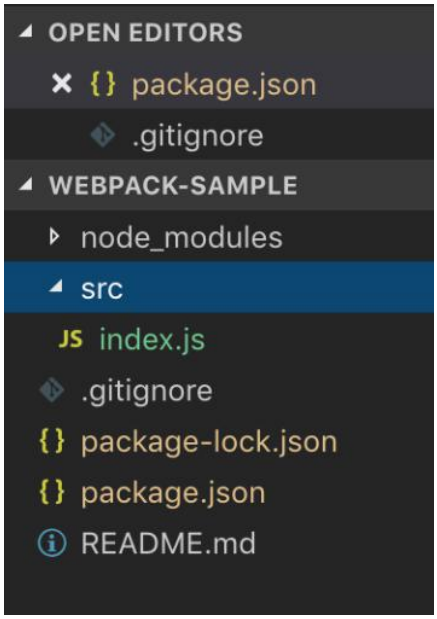
此时打开 package.json 文件，你会发现在文件中有一个新增项 devDependencies：

```
{
 // 其他项信息在这省略，详细请查看该文件
"devDependencies": {
"webpack": "^4.35.0",
"webpack-cli": "^3.3.5",
 "webpack-dev-server": "^3.7.2"
 }
}
```

为了验证 Webpack 是否能正常工作，这个时候我们需要创建一些新的文件。在 webpack-sample 根目录下创建/src 目录，并且在该目录下创建一个 index.js 文件：

mkdir src && cd src && touch index.js

执行完上面的命令，你会发现你的项目目录结构变成下图这样：



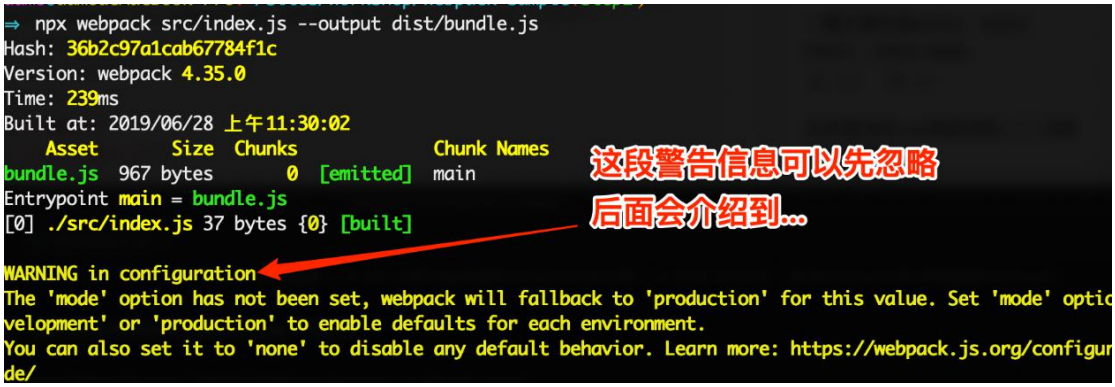
我们在新创建的/src/index.js 文件下添加一行最简单的 JavaScript 代码：

console.log("Hello, Webpack!(^_^)~~")

保存之后回到命令终端，第一次执行有关于 Webpack 相关的命令：

webpack src/index.js --output dist/bundle.js

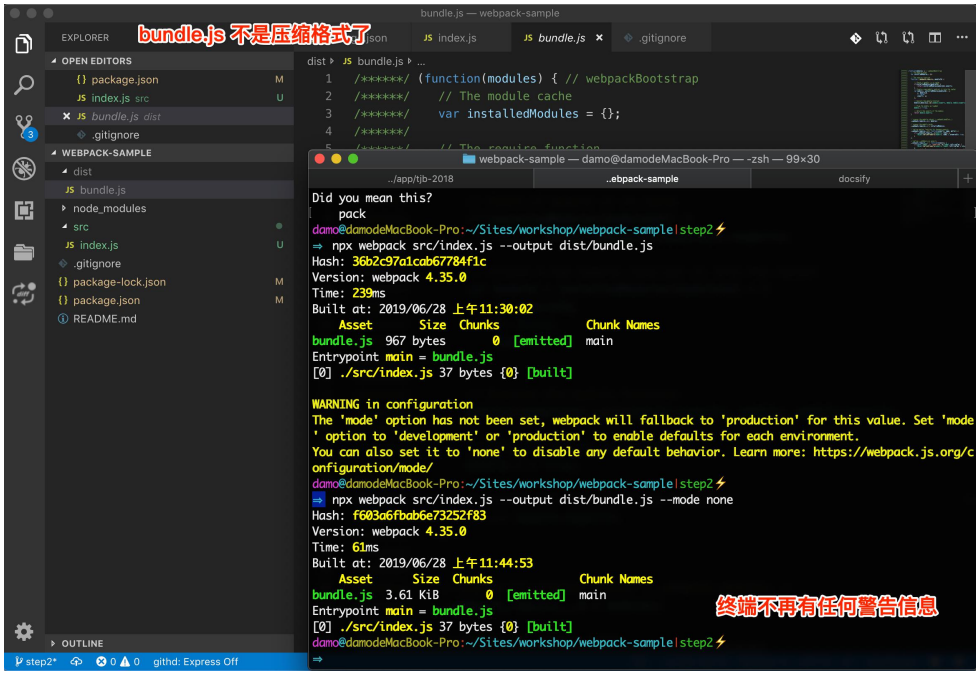
执行完上面的命令后，如果看到下图这样的结果，那么要恭喜你，Webpack 的安装已经成功，你可以在你的命令终端执行有关于 Webpack 相关的命令：



回到项目中，会发现项目根目下自动创建了一个/dist 目录，而且该目录下包含了一个 bundle.js 文件：

执行完上面的命令之后，可以看到有相关的警告信息。那是因为 Webpack4 增加了 mode 属性，用来表示不同的环境。mode 模式具有 development，production 和 none 三个值，其默认值是 production 。也就是说，在执行上面的命令的时候，我们可以带上相应的 mode 属性的值，比如说，设置 none 来禁用任何默认行为

npx webpack src/index.js --output dist/bundle.js --mode none



执行到这里，只知道我们可以运行 Webpack 相关命令。并不知道/src/index.js 的代码是否打包到/dist/bundle.js 中。为此，我们可以在/dist 目录下创建一个 index.html

cd dist && touch index.html

并且将生成出来的 bundle.js 引入到新创建的 index.html：

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">
        <title>Hello Webpack (^_^) ~~~</title>
    </head>
    <body>
        <script src="./bundle.js"></script>
    </body>
</html>
```

在浏览器中打开/dist/index.html，或者在命令行中执行：

⇒ yarn global add http-server　或者　npm i -g http-server

⇒ http-server dist

http-server 是一个启动服务器的 npm 包，执行上面的命令之后，就可以在浏览器中访问 http://127.0.0.1:8080/（访问的/dist/index.html），在浏览器的 console.log 控制台中，可以看到 src/index.js 的脚本输出的值：

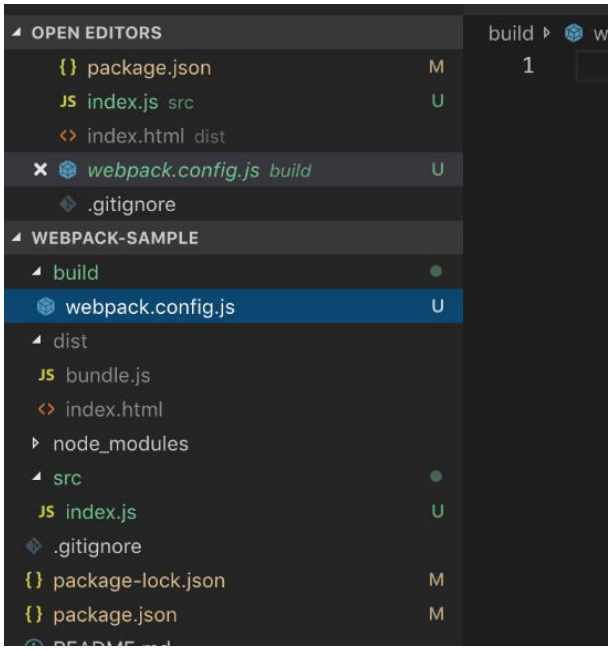
或者通过 vscode 的插件，安装一个 live Server 也可以直接在 vscode 里直接起服务，https://jingyan.baidu.com/article/9989c746e671e6f648ecfea7.html

不过，当你需要构建的东西越复杂，需要的标志就会越多。在某种程度上说，就会变得难以控制。这个时候我们就需要一个文件来管理这些配置。接下来我们需要创建一个 webpack.config.js 这样的一个文件，用来配置 Webpack 要做的事情。注意，这个文件是一个 node.js 文件，所以你可以在任何节点文件中执行任何你能够执行的操作。你也可以写成 json 文件，但是 node 文件更强大一些。

首先创建 Webpack 的配置文件，在 webpack-sample 根目录下创建一个/build 目录，然后在该目录下添加一个名为 webpack.config.js 文件：

mkdir build && cd build && touch webpack.config.js

执行完上面的命令之后，你会发现你的项目文件目录结构变成下面这样了：



这个时候，新创建的 webpack.config.js 文件里面是一片空白，它就是 Webpack 的配置文件，将会导出一个对象的 JavaScript 文件。我们需要在这个文件中添加一些配置：

```
var webpack = require('webpack');

var path = require('path');

var DIST_PATH = path.resolve(__dirname, '../dist');

// 声明/dist 的路径 为成绝对路径

module.exports = {

// 入口 JS 路径

// 指示 Webpack 应该使用哪个模块，来作为构建其内部依赖图的开始

entry: path.resolve(__dirname,'../src/index.js'),

支持单文件，多文件打包

 // entry: './src/index.js',  //方式一

 // entry: ['./src/index.js','./src/main.js'], //方法二
```

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

// 编译输出的 JS 入路径
// 告诉 Webpack 在哪里输出它所创建的 bundle，以及如何命名这些文件
output: {
      path: DIST_PATH, // 创建的 bundle 生成到哪里
      filename: 'bundle.js', // 创建的 bundle 的名称
 },
// 模块解析
module: { },
// 插件
plugins: [ ],
// 开发服务器
devServer: { }
}
```

Webpack 配置是标准的 Node.js CommonJS 模块，它通过 require 来引入其他模块，通过 module.exports 导出模块，由 Webpack 根据对象定义属性进行解析。

上面很简单，到目前为止只通过 entry 设置了入口起点，然后通过 output 配置了打包文件

输出的目的地和方式。你可能也发现了，在配置文件中还有 module、plugins 和 devServer 没有添加任何东西。不需要太急，后面会一步一步带着大家把这里的内容补全的，而且随着配置的东西越来越多，整个 webpack.config.js 也会更变越复杂。

完成 webpack.config.js 的基础配置之后，回到 package.json 文件，并在"scripts"下添加 "build": "webpack --config ./build/webpack.config.js"：

```
// package.json {
// ...
"scripts": {
    "build": "webpack --config ./build/webpack.config.js",
   "test": "echo \"Error: no test specified\" && exit 1"
},
 }
```

这样做的，为让我们直接在命令终端执行相关的命令就可以实现相应的功能。比如上面配置的 build，在命令终端执行：

```
⇒  npm run build
```

面的命令执行的效果前面提到的 webpack src/index.js --output dist/bundle.js --mode none 等同。同样有警告信息，主要是 mode 的配置没有添加。在上面的配置中添加:

```
{
"scripts": {
```

"build": "webpack --config ./build/webpack.config.js --mode production",

"test": "echo \"Error: no test specified\" && exit 1"

}, }

再次执行 npm run build，不会再有警告信息。你可以试着修改/src/index.js 的代码：

alert(`Hello, Webpack! Let's Go`);

重新编译之后，打开/dist/index.html 你会发现浏览器会弹出 alert()框：

为了开发方便，不可能通过 http-server 来启用服务。我们可以把这部分事件放到开发服务器中来做，对应的就是 devServer，所以我们接着在 webpack.config.js 中添加 devServer 相关的配置：

webpack-dev-server

webpack-dev-server 是一个用来快速搭建本地运行环境的工具。webpack-dev-server 是一个小型的 Node.js Express 服务器,它使用 webpack-dev-middleware 来服务于 webpack 的包,除此自外，它还有一个通过 Sock.js 来连接到服务器的微型运行时.

命令简单 webpack-dev-server 或配置命令脚本快捷运行,模拟服务器运行情况，进行上线前调试。

安装： yarn add webpack-dev-server --dev

webpack-dev-server 和 webpack-dev-middleware 的区别

webpack-dev-server

webpack-dev-server 实际上相当于启用了一个 express 的 Http 服务器+调用 webpack-dev-middleware。它的作用主要是用来伺服资源文件。这个 Http 服务器和 client 使用了 websocket 通讯协议，原始文件作出改动后，webpack-dev-server 会用 webpack 实时的编译，再用 webpack-dev-middleware 将 webpack 编译后文件会输出到内存中。适合纯前端项目，很难编写后端服务，进行整合。

webpack-dev-middleware

webpack-dev-middleware 输出的文件存在于内存中。你定义了 webpack.config，webpack 就能据此梳理出 entry 和 output 模块的关系脉络，而 webpack-dev-middleware 就在此基础上形成一个文件映射系统，每当应用程序请求一个文件，它匹配到了就把内存中缓存的对应结果以文件的格式返回给你，反之则进入到下一个中间件。

因为是内存型文件系统，所以重建速度非常快，很适合于开发阶段用作静态资源服务器；因为 webpack 可以把任何一种资源都当作是模块来处理，因此能向客户端反馈各种格式的资源，所以可以替代 HTTP 服务器。事实上，大多数 webpack 用户用过的 webpack-dev-server 就是一个 express + webpack-dev-middleware 的实现。二者的区别仅在于 webpack-dev-server 是封装好的，除了 webpack.config 和命令行参数之外，很难去做定制型开发。而 webpack-dev-middleware 是中间件，可以编写自己的后端服务然后把它整合进来，相对而言比较灵活自由。

webpack-hot-middleware：

是一个结合 webpack-dev-middleware 使用的 middleware，它可以实现浏览器的无刷新更新（hot reload），这也是 webpack 文档里常说的 HMR（Hot Module Replacement）。HMR 和热加载的区别是：热加载是刷新整个页面。

```
// webpack.config.js
//  开发服务器
devServer: {
   hot: true, //  热更新，无需手动刷新
   contentBase: DIST_PATH,  //热启动文件所指向的文件路径
   // host: '0.0.0.0', // host 地址
   port: 8080, //  服务器端口
   historyApiFallback: true, //  该选项的作用所用 404 都连接到 index.html
   proxy: {
      "/api": "http://localhost:3000"
      //  代理到后端的服务地址，会拦截所有以 api 开头的请求地址
 }
}
```

有关于 devServer 更详细的配置参数描述

```
devServer:{
```

port:7788,//控制端口

open:true //是否自动打开默认浏览器

headers :在所有响应中添加首部内容

historyApiFallback: 使用 HTML5 History API 时，任意的 404 响应都可能需要被替代为 index.html

host:’0.0.0.0’ //指定使用一个 host。默认是 localhost。如果你希望服务器外部可访问，指定如下

hot:true //启用 webpack 的模块热替换特性

https：true //默认情况下，dev-server 通过 HTTP 提供服务。也可以选择带有 HTTPS 的 HTTP/2 提供服务

Info: true //是否要输出一些打包信息

noInfo: true //启用 noInfo 后，诸如「那些显示的 webpack 包(bundle)信息」的消息将被隐藏

Proxy：//这样启用代理 ，https://www.webpackjs.com/configuration/dev-server/#devserver-proxy

progress ： 将运行进度输出到控制台

publicPath ： //此路径下的打包文件可在浏览器中访问，默认 publicPath 是 "/"

//https://www.webpackjs.com/configuration/dev-server/#devserver-publicpath-

useLocalIp: true //是否在打包的时候用自己的 IP

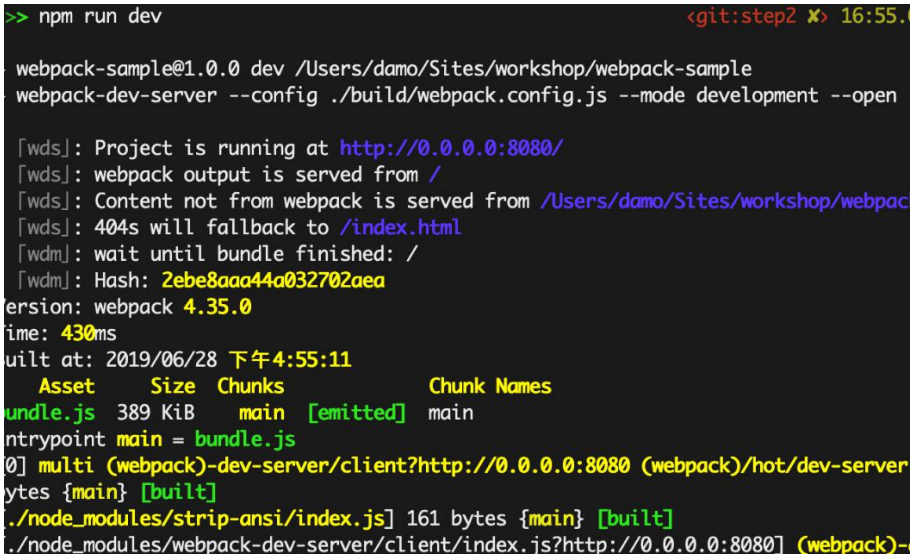
watchContentBase：true //告诉服务器监视 devserver.contentbase 选项提供的文件。文件更改将触发整页重新加载

}

和 build 类似，需要在 package.json 的 scripts 中添加相关的命令：

```
// package.json
 "scripts": {
   "build": "webpack --config ./build/webpack.config.js --mode production",
   "dev": "webpack-dev-server --config ./build/webpack.config.js ",
   "test": "echo \"Error: no test specified\" && exit 1"
 },
```

保存所有文件，在命令行中执行 npm run dev 就可以启动服务器：



你可以验证一下，修改/src/index.js：

```
document.addEventListener('DOMContentLoaded', () => {
        const h1Ele = document.createElement('h1');
        document.body.append(h1Ele);
        h1Ele.innerText = 'Hello Webpack (^_^)'
        h1Ele.style.color = '#f46';
 })
```

保存该文件之后，浏览器会立刻刷新，你将看到修改之后的变化：

多文件打包，glob 配置动态入口

安装：yarn add glob --dev

const glob = require("glob");

const htmls = glob.sync('src/components/**/*.html');

//扫描出入口页面模板的路径，如 src/components/index/index.html，存放在 htmls 对象里

//入口文件

```
var SRC_PATH = path.resolve(__dirname,'../src');
var newEntries = {};
// var files = glob.sync(path.join(SRC_PATH,'js/*.js')); // 方式一
var files = glob.sync(SRC_PATH+'/js/*.js');  //方式二
files.forEach(function(file,index){
//    var substr =  file.split('/').pop().split('.')[0];
  var substr = file.match(/src\/js\/(\S*)\.js/)[1];
  newEntries[substr] = file;
})
```

多文件的时候，需要修改 output 输出文件里的 动态文件配置为[name]

// [\s]---表示，只要出现空白就匹配；

// [\S]---表示，非空白就匹配；

hash、chunkhash、contenthash 区别

hash 一般是结合 CDN 缓存来使用，通过 webpack 构建之后，生成对应文件名自动带上对应的 MD5 值。如果文件内容改变的话，那么对应文件哈希值也会改变，对应的 HTML 引用的 URL 地址也会改变，触发 CDN 服务器从源服务器上拉取对应数据，进而更新本地缓存。但是在实际使用的时候，这几种 hash 计算还是有一定区别。

[hash:8] 可以用来控制输出多少位的 hash

Hash

Hash： hash 是跟整个项目的构建相关，只要项目里有文件更改，整个项目构建的 hash 值都会更改，并且全部文件都共用相同的 hash 值

```
filename: 'bundle.[name].[hash].js',
```

chunkhash

chunkhash： 采用 hash 计算的话，每一次构建后生成的哈希值都不一样，即使文件内容压根没有改变。这样子是没办法实现缓存效果，我们需要换另一种哈希值计算方式，即 chunkhash。

chunkhash 和 hash 不一样，它根据不同的入口文件(Entry)进行依赖文件解析、构建对应的 chunk，生成对应的哈希值。我们在生产环境里把一些公共库和程序入口文件区分开，

单独打包构建，接着我们采用 chunkhash 的方式生成哈希值，那么只要我们不改动公共库的代码，就可以保证其哈希值不会受影响。

```
filename: 'bundle.[name].[chunkhash].js',
```

contenthash

contenthash： 在 chunkhash 的例子，我们可以看到由于 index.css 被 index.js 引用了，所以共用相同的 chunkhash 值。但是这样子有个问题，如果 index.js 更改了代码，css 文件就算内容没有任何改变，由于是该模块发生了改变，导致 css 文件会重复构建。

这个时候，我们可以使用 extra-text-webpack-plugin 里的 contenthash 值，保证即使 css 文件所处的模块里就算其他文件内容改变，只要 css 文件内容不变，那么不会重复构建。

```
plugins:[
 new extractTextPlugin('../css/bundle.[name].[contenthash].css')
]
```

Step03：打包之前删除某个文件夹

rimraf 插件

安装：yarn add rimraf --dev

在 package.json 的 script 直接使用，比如：

```
"build": "rimraf dist && webpack --config ./build/webpack.config.js --mode production",
```

或者：

```
 "build": "rm -rf dist && webpack --config ./build/webpack.config.js --mode production"
```

node 常用命令

touch 新建文件
rm -f 删除文件
mkdir 新建文件夹
rm -rf 删除文件夹

Step04：优化 Webpack 配置

在 Step02 中，开发和生产环境相关的配置都集成在 webpack.config.js 一个文件中。为了更好的维护代码，在 Step03 中做一些优化。把 webpack.config.js 拆分成三个部分：

公共配置：把开发和生产环境需要的配置都集中到公共配置文件中，即 webpack.common.js

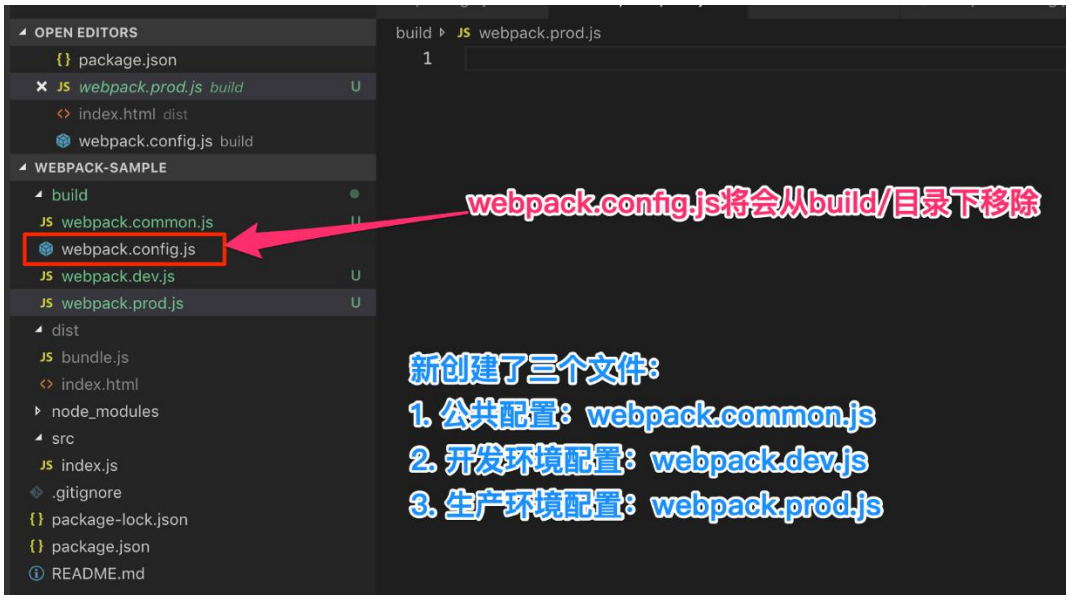
开发环境配置：把开发环境需要的相关配置放置到 webpack.dev.js

生产环境配置：把生产环境需要的相关配置放置到 webpack.prod.js

先在/build 目录下创建上面提到的三个配置文件。在命令终端执行下面的命令即可：

```
⇒ cd build && touch webpack.common.js webpack.dev.js webpack.prod.js
```

这个时候，整个项目目录结构变成下图这样：



遗留下来的 webpack.config.js 文件将会从/build 目录中移除。

Object.assign

为了更好的管理和维护这三个文件，需要安装一个 **webpack-merge** 插件:

yarn add webpack-merge --dev　或者 npm i webpack-merge -D

执行完上面的命令之后，package.json 文件中的 devDependencies 会增加 webpack-merge 相关的配置:

```
// package.json
{
//... 省略的信息请查看原文件
 "devDependencies": {
"webpack": "^4.35.0",
"webpack-cli": "^3.3.5",
"webpack-dev-server": "^3.7.2",
 "webpack-merge": "^4.2.1"
 }
}
```

接下来分别给 webpack.common.js、webpack.dev.js 和 webpack.prod.js 文件添加相关的配置:

Webpack 公共配置

在公共配置文件 webpack.common.js 文件中添加相应的配置：

```
const webpack = require('webpack');
 const path = require('path');
const DIST_PATH = path.resolve(__dirname, '../dist/');
// 声明/dist 的路径
module.exports = {
 // 入口 JS 路径
// 指示 Webpack 应该使用哪个模块，来作为构建其内部依赖图的开始
entry: path.resolve(__dirname,'../src/index.js'),
// 编译输出的 JS 入路径
// 告诉 Webpack 在哪里输出它所创建的 bundle，以及如何命名这些文件
output: {
        path: DIST_PATH,
         // 创建的 bundle 生成到哪里
        filename: 'bundle.js',
        // 创建的 bundle 的名称
},
// 模块解析
module: { },
// 插件
plugins: [ ]
 }
```

Webpack 开发环境配置

接着给 Webpack 开发环境配置文件 webpack.dev.js 添加下面的相关配置：

```
const webpack = require('webpack');
const path = require('path');
const merge = require('webpack-merge');
const commonConfig = require('./webpack.common.js');
const DIST_PATH = path.resolve(__dirname, '../dist/');

module.exports = merge(commonConfig, {
        mode: 'development',
        // 设置 Webpack 的 mode 模式
```

```
    // 开发环境下需要的相关插件配置
    plugins: [ ],
    // 开发服务器
    devServer: {
        hot: true, // 热更新，无需手动刷新
        contentBase: DIST_PATH,
        // host: '0.0.0.0', // host 地址
        port: 8080, // 服务器端口
        historyApiFallback: true, // 该选项的作用所用 404 都连接到 index.html
        proxy: {
        "/api": "http://localhost:3000" // 代理到后端的服务地址,会拦截所有以 api 开头的请求地址
        }
    }
})
```

Webpack 生产环境配置

继续给 Webpack 生产环境配置文件 webpack.prod.js 添加相关配置：

```
const webpack = require('webpack');
const path = require('path');
const merge = require('webpack-merge');
const commonConfig = require('./webpack.common.js');
module.exports = merge(commonConfig, {
    mode: 'production',
    // 设置 Webpack 的 mode 模式
    // 生产环境下需要的相关插件配置
    plugins: [ ],
})
```

上面的配置信息只是将 Step02 中 webpack.config.js 分成三个文件来配置，随着后续添加相应的配置信息，那么这三个文件中的配置信息会越来越多，

修改完 Webpack 的配置之后，对应的 package.json 中的 scripts 中的信息也要做相应的调整：

```
// package.json
{
// ... 其他配置信息请查看原文件
"scripts": {
```

```
    "build": "webpack --config ./build/webpack.prod.js --mode production",
     "dev": "webpack-dev-server --config ./build/webpack.dev.js --mode development --open",
    "test": "echo \"Error: no test specified\" && exit 1"
 },
}
```

这个时候重新在命令终端执行

```
// 执行 build 命令，重新打包
⇒  npm run build
// 执行 dev 命令
 ⇒  npm run dev
```

这仅仅是最基础部分的优化，因为我们的配置还是最简单的，后续我们添加了别的配置之后，

也会在相应的步骤做相应的优化。

Step05:编译 Webpack 项目中的 html 类型的文件

安装：yarn add html-webpack-plugin --dev

首先在项目根目录新建 index.html,作为模板文件进行打包

在插件里配置：

```
new htmlWebpackPlugin({
        filename: path.resolve(DIST_PATH,'index.html'), //打包后的文件名
        title: '树鱼虚拟充值生态服务平台',   //打包后的页面 title
        template: path.resolve(__dirname,'../index.html'),   //打包的模板文件
        inject: true,
        hash: true,
        favicon: path.resolve(__dirname, '../fav.ico')
    })
```

打包后 title 如何自动的添加到页面中呢?

只需要在模板文件的 title 中添加

<title><%= htmlWebpackPlugin.options.title %></title>

参考链接：https://www.jianshu.com/p/08a60756ffda

```
var htmlWebpackPlugin = require('html-webpack-plugin');
new htmlWebpackPlugin({
        filename: , //就是 html 文件的文件名，默认是 index.html
        title: '标题',
        template: , //指定你生成的文件所依赖哪一个 html 文件模板，模板类型可以是 html、jade、ejs 等
        inject: true,   //true body head false
          true 默认值，script 标签位于 html 文件的 body 底部
          body script 标签位于 html 文件的 body 底部
          head script 标签位于 html 文件的 head 中
          false 不插入生成的 js 文件，这个几乎不会用到的
        hash: true,
        favicon: path.resolve(__dirname, './fav.ico')
          //给你生成的 html 文件生成一个 favicon ,值是一个路径
          //然后再生成的 html 中就有了一个 link 标签
          <link rel="shortcut icon" href="example.ico">
       minify:   //是否对 HTML 进行压缩
      cache：true   //默认是 true 的，表示内容变化的时候生成一个新的文件
      showErrors：当 webpack 报错的时候，会把错误信息包裹再一个 pre 中，默认是 true
      chunks: chunks, //chunks 主要用于多入口文件，当你有多个入口文件，那就回编译后生成多个打包后的文件，那么 chunks 就能选择你要使用那些 js 文件 ，
     excludeChunks：//排除掉一些 js
    })
```

```
var pluginsAll2 = [];
var pages = glob.sync(path.join(htmlPagesPath,'**/*.html'));
pages.forEach(function(page){
   var pagestr = page.match(/pages\/(\S*)\.html/);
   var name = pagestr[1];
   var plug = new htmlWebpackPlugin({
      filename:path.resolve(buildPath,name+'.html'),
      title:'测试',
      template:path.resolve(htmlPagesPath,name+'.html'),
      inject:true,
      chunks:[name],
      favicon: path.resolve(__dirname, './fav.ico')
   })
   pluginsAll2.push(plug);
})
```

Step06:loaders

https://www.webpackjs.com/loaders/
常见的 loader 有：

file-loader,url-loader,style-loader,css-loader,less-loader,sass-loader,babel-loader,raw-loader,vue-loader
webpack >= v2.0.0　开始默认支持 json 文件的导入

webpack 是模块打包工具，而模块不仅是 js，还可以是 css，图片或者其他格式
但是 webpack 默认只知道如何处理 js 模块，那么其他格式的模块处理，和处理方式就需要 loader
如何在 webpack 中配置第三方 loader
配置第三方 loader，需要在 webpack 的配置文件中新增一个 module 节点，节点中是一个一个的规则集合，集合名字是 rules，需要添加 loader 就在 rules 的集合中新增一个规则；每个规则必须的两个配置：

（1）test　：test 搭配的是键值对，值是一个正则表达式，用来匹配要处理的文件类型；

（2）use　：用来指定使用哪个 loader 模块来打包处理该文件；

```
module:{
    rules:[
    {
        test:/\.xxx$/,
        use:{
            loader: 'xxx-load'
        }
      }
    ]
}
```

当 webpack 处理理到不不认识的模块时，需要在 webpack 中的 module 处进行行行配置，当检测到是什什么格式的 模块，使用用什什么 loader 来处理理

● loader：file-loader：处理理静态资源模块

1、作用：file-loader 可以用来帮助 webpack 打包处理一系列的图片文件；比如：**.png** 、 **.jpg** 、**.jepg** 等格式的图片；

2、使用 **file-loader** 打包图片的结果：使用 file-loader 打包的图片会给每张图片都生成一个随机的 hash 值作为图片的名字；

loader: file-loader 原理理是把打包入入口口中识别出的资源模块，移动到输出目目录，并且返回一个地址名称

所以我们什什么时候用用file-loader 呢？ 场景：就是当我们需要模块，仅仅是从源代码挪移到打包目目录，就可以使用用file-loader 来处理理， txt，svg，csv，excel，图片片资源啦等等

```
yarn add file-loader -D   或者  npm install file-loader -D
```

案例：

```
module: {
    rules: [
    {
        test: /\.(png|jpe?g|gif)$/,
        //use 使用用一一个 loader 可以用用对象，字符串串，两个 loader 需要用用数组
        use: {
            loader: "file-loader",
            // options 额外的配置，比比如资源名称
            options: {
                // placeholder  占位符   [name]老老老资源模块的名称
                // [ext]老老资源模块的后缀
                // https://webpack.js.org/loaders/file-loader#placeholders
                name: "[name]_[hash].[ext]",
                //打包后的存放位置
                outputPath: "images/"
            }
        }
    }
    ]
},
```

● url-loader :处理图片 base64

当图片较小的时候会把图片转换成 base64 编码，大于 limit 参数的时候还是使用 file-loader 进行拷贝。这样做是好处是可以直接将图片打包到 bundle.js 里，不用额外请求图片（省去

HTTP 请求）

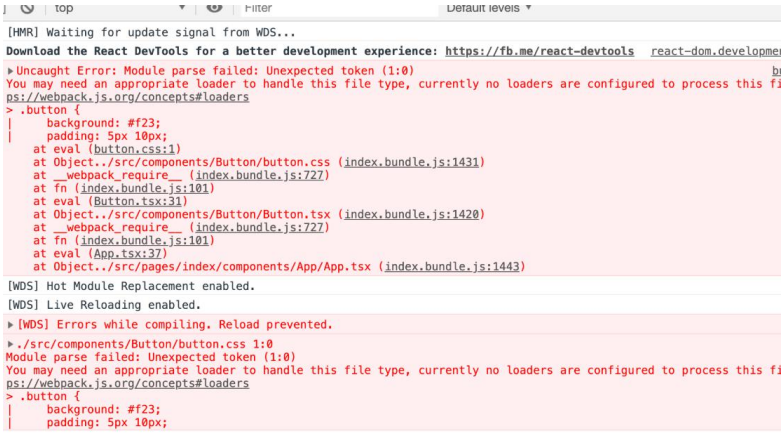
作用： 可以弥补处理file-loader 不能生成 base64 图片格式的缺陷，对小体积的图片比较合适，大图片片不合适。但是，也是需要安装 file-loader
安装 ：
yarn add url-loader -D 或者 npm install url-loader -D
案例；

```
module: {
    rules: [
        {
            test: /\.(png|jpe?g|gif)$/,     //是匹配图片文件后缀名称
            use: {
                loader: "url-loader",       //是指定使用的 loader 和 loader 的配置参数
                options: {
                    name: "[name]_[hash].[ext]",
                    outputPath: "images/",  //图片打包到的文件目录
                    //小于 2048B，才转换成 base64 的文件打成 Base64 的格式，写入 JS
                    limit: 2048 ,
                    publicPath:'/img'  //最终生成的 CSS 代码中，图片 URL 前缀
                    }
            }
        }
    ]
},
```

● CSS 预处理

(1)CSS loader 配置

如果没有安装 style-loader css-loader 直接引入 css 文件就会报错如下：



这是因为在 Webpack 中没有 CSS 相关的 Loader 配置。那么接下来，来解决

CSS Loader 在 Webpack 中的配置：

- css-loader 使你能够使用类似@import 和 url()的方法实现 require()的功能
- style-loader 将所有的计算后的样式加入页面中
- 而 vue-style-loader 是 vue 官方基于 style-loader 开发的适用于 vue 的样式解析，

  sass-loader 用来解析 sass/scss 文件

两者结合在一起能够把样式嵌入 Webpack 打包后的 JavaScript 文件中

安装：yarn add style-loader css-loader --dev

在

```
module:{
   rules:[]
}
```

配置：

```
{
test: /\.css$/,
exclude: /node_modules/,
use: [{ loader: "style-loader"}, { loader: "css-loader" } ]     //方式一
//use: ["style-loader", loader: "css-loader" ]    //方式二
}
```

(2) 配置 less 环境需要安装：

安装：yarn add less less-loader --dev

配置：

```
{
test: /\.less$/,
use: [{ loader: "style-loader" }, { loader: "css-loader" }, { loader: "less-loader" }]   //方式一
use: ['style-loader','css-loader','less-loader']    //方式二
}
```

(3) 配置 scss 环境需要安装：

sass-loader 把 sass 语法转换成 css ，依赖 node-sass 模块

安装：yarn add node-sass sass-loader --dev

//注意：在项目中要 less 预处理和 sass 预处理二者选其一，不要同时混着用

loader 有顺序，从右到左，从下到上

配置：

```
{
test: /\.scss$/,
use: [{ loader: "style-loader" }, { loader: "css-loader" }, { loader: "sass-loader" }]   //方式一
use: ['style-loader','css-loader','sass-loader']    //方式二
}
```

(4) 添加 PostCSS 相关配置

PostCSS 是一个很优秀的东西，他有很多优秀的插件，比如 postcss-preset-env、Autoprefixer 等。另外自己还可以根据自己需要扩展自己想要的 PostCSS 插件。而且在一些移动端的布局中的方案都会需要处理 CSS 单位之间的转换，比如 Flexible（px2rem），vw-layout（px2vw）等。

可以说，现代 Web 开发中 PostCSS 相关的配置是工程体系中必不可少的。接下来，我们就看看如何在该工程体系中添加 PostCSS 相关的配置。要配置 PostCSS 相关的事项，需要：

- 安装 postcss、postcss-loader
- 在 Webpack 配置中添加 PostCSS 相关的配置
- 安装自己需要的 PostCSS 插件
- 在 postcss.config.js 或.postcssrc.js 添加有关于 PostCSS 的插件配置

先来执行第一步，安装 postcss 和 postcss-loader：

安装：yarn add postcss-loader　postcss --dev

配置：

```
{
test: /\.css$/,
exclude: /node_modules/,
use: [{ loader: "style-loader"}, { loader: "css-loader" },{loader: 'postcss-loader'} ]
}
```

如果直接启动会报错：No PostCSS Config found....

接下来需要给项目添加 PostCSS 配置相关的信息。先在项目的根目录下创建 postcss.config.js 或.postcssrc.js 文件，并添加相关的配置：

```
// postcss.config.js
module.exports = {
   plugins: { },
}
```

上面的配置还没有添加任何 PostCSS 相关的插件。为了验证我们的配置是否成功，来安装两个 PostCSS 的插件，

- 添加 css3 前缀

postcss-preset-env 将现代 CSS 转换成浏览器能理解的东西

安装 postcss-preset-env，无需再安装 autoprefixer，由于 postcss-preset-env 已经内置了相关

功能。

安装：yarn add postcss-preset-env　--dev

并把这两插件的相关配置中添加到 postcss.config.js 中：

```
module.exports = {
    plugins: {
      'postcss-preset-env': {
        autoprefixer: { flexbox: 'no-2009'},  //可去掉
        stage: 3
      },
    }
  }
```

如果安装了自动补全插件 autoprefixer， yarn add autoprefixer --dev 也可以通过这个配置来生效

```
module.exports = {
    plugins: {
      'autoprefixer': { }
    }
  }
```

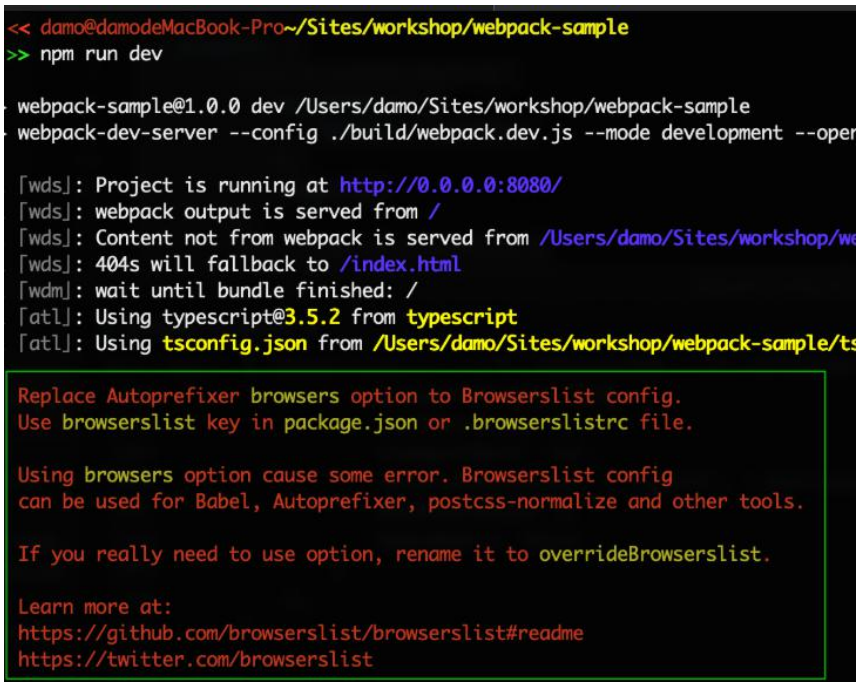
在样式文件中添加一些样式代码来验证是否生效了：

```
.box {
    width: 50px;
    height: 30px;
    background: url(../images/ABC.jpg);
    transform:rotate(30deg);
  }
```

如果生效，会是这样的

```
.box
{
transform:rotate(7deg);
-ms-transform:rotate(7deg);         /* IE 9 */
-moz-transform:rotate(7deg);        /* Firefox */
-webkit-transform:rotate(7deg); /* Safari 和 Chrome */
-o-transform:rotate(7deg);    /* Opera */
}
```

如果就这样执行 npm run dev，会有 autoprefixer 相关的警告信息：



主要是因为 postcss-preset-env 插件涵盖了 autoprefixer 插件的能力，要解决这个警告信息，需要在 package.json 中添加 browserslist 相关的配置信息：

```
  // package.json
  {
  // ... 省略的信息可以查看原文件
"browserslist": [
  "last 5 version",
  "> 2%",
  "IE 8"
 ],
  }
```

- 单位转换 px - rem

把项目中 px 单位转化为 rem,实现响应式，

yarn add postcss-pxtorem -D　或者 npm install postcss-pxtorem -D

```
module.exports = {
        plugins:{
          'postcss-preset-env': {
            autoprefixer: { flexbox: 'no-2009'},
            stage: 3
```

```
        },
        'postcss-pxtorem':{
          rootValue:10,
          minPixelValue:2,
          propWhiteList: []
        }
      }
    }
```

(5) css 抽离 后面插件讲

配置：

```
{     test:/\.css$/,
      use: ExtractTextPlugin.extract({
      fallback: 'style-loader',
      use: [{
            loader: 'css-loader',
            options: { importLoaders: 1 }
            },
        {loader:'postcss-loader',
              options:{   //方式一
                  plugins:[
                      require("autoprefixer")
                    ]
                }
        },
        {loader:'less-loader'}
      ]}),
 }
```

方式二：

在项目根目录 添加一个 postcss.config.js

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

在 package.json 添加

```
"browserslist": [
     "last 1 version",
     "> 1%",
```

```
    "IE 10"
  ],
```

Step07:增加 babel支持

babel7 的一些变化

preset 的变更:

淘汰 es201x，删除 stage-x，推荐 env

如果你还在使用 es201x，官方建议使用 env 进行替换。淘汰并不是删除，只是不推荐使用。

但 stage-x 是直接被删了，也就是说在 babel7 中使用 es201X 是会报错的。

包名称变化

babel 7 的一个重大变化，把所有 babel-* 重命名为 @babel/*，

例如:

babel-cli —> @babel/cli。

babel-preset-env —> @babel/preset-env

安装：yarn add `@babel/core babel-loader @babel/preset-env` -D

其中 preset 是 babel 的配置，babel-loader 在 webpack 中解析 babel 中会用到，polyfill 也是 babel 的一个依赖在 webpack 中解析 babel 中会用到

//babel-loader 是 webpack 与 babel 的通信桥梁，在 webpack 中解析 babel 中会用到，不会做把 es6 转成 es5 的工作，这部分工作需要用到 @babel/preset-env 来做

//@babel/preset-env 里包含了 es6 转 es5 的转换规则

~~babel-preset-es2015~~ 已经被 babel-preset-env 替代，现在已经是`@babel/preset-env`

@babel/preset-env 这个比较强大， 是一个面向未来得库， 即使是 es7,8,9 出来以后，都可以帮忙处理

在开发依赖包安装如下插件:

babel-loader

@babel/core

@babel/preset-env

@babel/plugin-transform-runtime

@babel/plugin-transform-modules-commonjs

在线上依赖包安装

@babel/runtime

https://babeljs.io/setup#installation

```
//babel 配置
{
test:/\.(jsx|js)$/,
use:{
loader:'babel-loader',
options:{
      presets:[ "@babel/preset-env" ]   //方法一
   }
},
exclude:/node_modules/
}
```

.babelrc 配置： 方式二

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

然后可以把配置文件里的配置 babel 的地方去掉 presets 配置

测试代码：

```
//index.js
const arr = [new Promise(() => {}), new Promise(() => {})];
arr.map(item => {  console.log(item); });

var isCol = ['11','22'].includes('11');
console.log(isCol);
```

在 ie10 会报错

通过上面的几步 还不够，includes Promise 等一些还没有转换过来，这时候需要借助@babel/polyfill，把 es 的新特性都装进来，来弥补低版本浏览器中缺失的特性

babel-polyfill

以全局变量的方式注入进来的。windows.Promise，它会造成全局对象的污染

安装： yarn add @babel/polyfill --dev

Webpack.config.js 配置：

```
{
test: /\.js$/,
exclude: /node_modules/,
loader: "babel-loader",
options: {
   presets: ["@babel/preset-env"]
 } }
```

//index.js 顶部

```
 import "@babel/polyfill";
```

会发现打包的体积大了很多，这是因为 polyfill 默认会把所有特性注入进来，假如我想我用到的 es6+，才会注入，没用到的不注入，从而减少打包的体积，可不可以呢
当然可以
修改 Webpack.config.js

```
options: {
presets: [
 [
```

```
"@babel/preset-env",
 {
      targets: {
            edge: "17",
            firefox: "60",
            chrome: "67",
             safari: "11.1"
       },
 useBuiltIns: "usage"//按需注入 ,
 "corejs": "2", // 声明 corejs 版本 ,注意这是是版本 2

 } ]
] }
```

进一步测试，这个时候 ie9 是可以运行， 但是 IE8 提示

如何解决呢？

查了些资料，发现 object.defineProperty 在 ie8 中存在，但是只用用于操作 dom 对象，难怪他会报错对象不支持该操作，安装一个 object-defineproperty-ie8 的包 试试，还是不兼容？

当我们开发的是组件库，工具库这些场景的时候，polyfill 就不适合了，因为 polyfill 是注入到全局变量，window 下 的，会污染全局环境，所以推荐闭包方式：
@babel/plugin-transform-runtime

@babel/plugin-transform-runtime

它不会造成全局污染
yarn add @babel/plugin-transform-runtime --dev

yarn add @babel/runtime --save //注意这里安装到 dependences 的依赖包

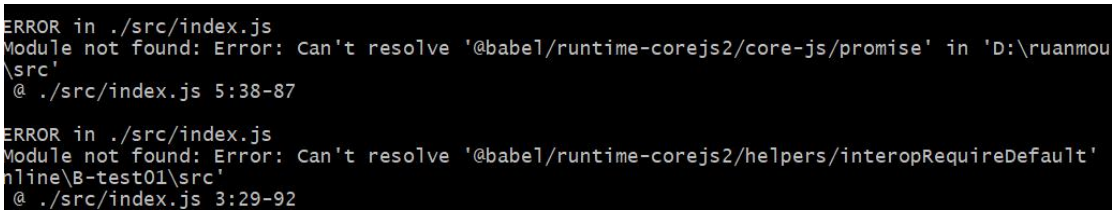
怎么使用？
先注释掉 index.js 里的 polyfill

修改配置文件：注释掉之前的 presets，添加 plugins

```
options: {
presets: [
[
```

```
"@babel/preset-env",
{
      targets: {
      edge: "17",
       firefox: "60",
      chrome: "67",
      safari: "11.1"
  },
useBuiltIns: "usage",
corejs: 2
}
  ]
],
"plugins": [
  [
"@babel/plugin-transform-runtime",
  {

       "absoluteRuntime": false,
         "corejs": 2,
         "helpers": true,
         "regenerator": true,
         "useESModules": false
  }
]
] }
```



如果出现这个错误， 记得安装 @babel/runtime-corejs2

yarn add @babel/runtime-corejs2

https://babeljs.io/docs/en/babel-runtime-corejs2

useBuiltIns 选项是 babel 7 的新功能，这个选项告诉 babel 如何配置 @babel/polyfill 。 它有三个参数可 以使用：

①entry: 需要在 webpack 的入口文件里 import "@babel/polyfill" 一次。 babel 会根据你的使用情 况导入垫片，没有使用的功能不会被导入相应的垫片。

②usage: 不需要 import ，全自动检测，但是要安装 @babel/polyfill 。（试验阶段）

③false：如果你 import "@babel/polyfill" ，它不会排除掉没有使用的垫 片，程序体积会庞大。(不推荐)

请注意： usage 的行为类似 babel-transform-runtime，不会造成全局污染，因此也会不会对类似 Array.prototype.includes() 进行 polyfill。

扩展：
babelrc 文件： 新建.babelrc 文件，把 options 部分移入到该文件中，就可以了

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

//webpack.config.js

{
        test: /\.js$/,
      exclude: /node_modules/,
      loader: "babel-loader"
 }
```

方式一：
使用 webpack，有多种方法可以包含 polyfill

- 当与 babel-preset-env 一起使用时,如果在.babelrc 中指定了 useBuiltIns: 'usage',那么在 webpack.config.js 配置的 entry 处或源代码中都不要包含 babel-polyfill。注意，babel-polyfill 仍然需要安装。
- 如果在.babelrc 中指定了 useBuiltIns: 'entry'，那么在应用程序入口点的顶部包括 babel-polyfill，方法是按照上面讨论的 require 或 import。
- 如果在.babelrc 中未指定 useBuiltIns 键，或者使用 useBuiltIns: false 设置了该键，则直接将 babel-polyfill 添加到 webpack.config.js 中的 entry 中。

方案	打包后大小	优点	缺点
babel-polyfill	259K	完整模拟ES2015+环境	体积过大；污染全局对象和内置的对象原型
babel-runtime	63K	按需引入，打包体积小	不模拟实例方法
babel-preset-env（开启useBuiltIns）	194K	按需引入，可配置性高	-

```
{
    "presets": [
     [
        "@babel/preset-env", {
            "useBuiltIns": "usage",  //entry,false
            "targets": {
                "browsers": [ "> 1%", "last 5 versions", "ie >= 8" ]
            }
        }
     ]
    ],
    "plugins": [
        "@babel/plugin-transform-runtime",
        "@babel/plugin-transform-modules-commonjs"
        ]
  }
```

在引入`@babel/preset-env`（一个帮你配置 babel 的 preset，根据配置的目标环境自动采用需要的 babel 插件）配置 `babel` 中的 `useBuiltIns: 'usage'`时，编译出现又出现这个警告提示

```
WARNING: We noticed you're using the `useBuiltIns` option without declaring a co
re-js version. Currently, we assume version 2.x when no version is passed. Since
 this default version will likely change in future versions of Babel, we recomme
nd explicitly setting the core-js version you are using via the `corejs` option.

You should also be sure that the version you pass to the `corejs` option matches
 the version specified in your `package.json`'s `dependencies` section. If it do
esn't, you need to run one of the following commands:

  npm install --save core-js@2    npm install --save core-js@3
  yarn add core-js@2              yarn add core-js@3
```

问题解释：警告在使用 useBuiltIns 选项时，需要声明 core-js 的版本

解决办法：

1) 方法一：安装 yarn add core-js@2
2) 方法二：在 .babelrc 文件中添加 corjs:”2”,申明 corejs 的版本

```
{
    "presets": [
     [
        "@babel/preset-env", {
            "useBuiltIns": "usage",
            "corejs": "2", // 声明 corejs 版本
            "targets": {
                "browsers": [ "> 1%", "last 5 versions", "ie >= 8" ]
            }
        }
     ]
    ],
    "plugins": [
    "@babel/plugin-transform-runtime",
    "@babel/plugin-transform-modules-commonjs"
    ]
  }
```

完美解决问题!

目前还不兼容 IE8 

项目如果是用 babel7 来转译，需要安装 @babel/core、@babel/preset-env 和 @babel/plugin-transform-runtime，而不是 babel-core、babel-preset-env 和 babel-plugin-transform-runtime，它们是用于 babel6 的

方式二： 下面的 2 种情况都不推荐

```
// 情况一：在 webpack 中引用
module.exports = {
  entry: ["babel-polyfill", "./app/js"]    //不推荐
};
```

```
// 情况二：在 js 入口顶部引入
import "babel-polyfill"；  不推荐
```

https://babeljs.io/docs/en/6.26.3/babel-polyfill

方式三： 上面已讲，下面是版本 6 的配置

runtime-transform 插件
runtime transform 也是一个插件，它与 polifill 有些类似，但它不污染全局变量，所以经常用于框架开发。
安装：

```
yarn add babel-plugin-transform-runtime --dev

yarn add babel-runtime    //注意这里不要加加--dev
```

注意安装的环境 babel-runtime 需要生产环境
用法：
将下面内容添加到.bablerc 文件中

```
{
    "plugins": ["transform-runtime"]
}
```

Step08. 常见插件 Plugins

重点插件 plugin 介绍

插件与模块解析的功能不一样，模块解析是为了导入非 es5 格式 js 或其它资源类型文件，定制了一些 loader。插件是对最后的打包文件进行处理的，也可以理解 loader 是为打包前做准备，plugin 是打包再进行处理

- **官方插件的使用步骤(内置插件 2 步)**
  ① 配置文件中导入 XxxxPlugin， const wp= require(XxxxPlugin)
  ② 在 plugins 这个数组中加入一个插件实例,new wp.XxxxPlugin({对象})

- **第三方插件的使用步骤(第 3 方 3 步，多一次安装)**
  ① 安装(第三方插件要安装) 根目录>cnpm i -D XxxxPlugin
  ② 配置文件中导入插件 const XxxxPlugin = require('xxxx-plugin')
  ③ 在 plugins 这个数组中加入一个插件实例,new XxxxPlugin({对象})

- **官方插件有**

可以在配置中打印查看

```
const webpack = require('webpack')
console.log(webpack ) //这里可以看到哪些是 webpack 内置的插件
```

待讲插件清单

01: webpack.BannerPlugin 加注释

02: terser-webpack-plugin 代码缩小

03: html-webpack-plugin 生成 html 页

04: 以前的 extract-text-webpack-plugin，mini-css-extract-plugin 提取 css 等

05. DefinePlugin //定义一个全局常量，如 new wp.DefinePlugin({BJ: JSON.stringify('北京'),})，在待打包的 js 文件中可以直接使用，如在./src/main.js 中 console.log('我是在配置文件中定义的'+BJ)

06.Dllplugins

- **BannerPlugin**

作用: 在打包的文件中添加注释

```
 //这是 webpack 内置的插件，所以不用 require 导入，但是对于第三方插件要先导入
plugins: [
        new webpack.BannerPlugin({
            banner: '永远要记得，成功的决心远胜于任何东西'
        })

    ]
```

● 打包复制

作用：在 webpack 中拷贝文件和文件夹

安装： yarn add copy-webpack-plugin --dev

```
from  定义要拷贝的源文件          from: __dirname+'/src/components'
to      定义要拷贝到的目标文件夹     to: __dirname+'/dist'
toType  file 或者 dir             可选，默认是文件
force   强制覆盖前面的插件          可选，默认是文件
context                           可选，默认 base   context 可用 specific  context
flatten  只拷贝指定的文件          可以用模糊匹配
```

```
ignore   忽略拷贝指定的文件            可以模糊匹配
```

插件引入和配置：

```
const CopyWebpackPlugin = require('copy-webpack-plugin');
new CopyWebpackPlugin([
    {
            from: path.resolve(__dirname, './static'),
            to: buildPath+'/static',
            ignore: ['.*']
            }
])
```

● js 压缩插件

作用：压缩 js 代码，减小体积
这个插件不是内置的，要先安装

不引荐运用 webpack-parallel-uglify-plugin
项目基础处于没人保护的阶段，issue 没人处置惩罚，pr 没人兼并。

引荐运用 terser-webpack-plugin

terser-webpack-plugin 是一个运用 terser 紧缩 js 的 webpack 插件。

紧缩是宣布前处置惩罚最耗时候的一个步骤，如果是你是在 webpack 4 中，只需几行代码，即可加快你的构建宣布速率。

安装：因为是第三方的 ， 所以需要安装 ， https://github.com/webpack-contrib/terser-webpack-plugin

```
yarn add terser-webpack-plugin -D
或者 npm install terser-webpack-plugin --save-dev
```

用法

```
module.exports = {
  optimization: {
    minimize: true,
    minimizer: [
      new TerserPlugin({
        parallel: 4,     //多线程
      }),
```

```
    ],
  },
};
```

- **抽离 css 样式,**

抽离 css 的目的是防止将样式打包在 js 中方便缓存静态资源

方式一：extract-text-webpack-plugin

安装： yarn add extract-text-webpack-plugin^4.0.0-beta.0 --dev

注意：如果用这个 需要 指定版本："extract-text-webpack-plugin": "^4.0.0-beta.0"

配置：

```
const ExtractTextPlugin = require("extract-text-webpack-plugin");
//module 部分
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ExtractTextPlugin.extract({
          fallback: "style-loader",
          use: "css-loader"
        })
      }
    ]
  },
//插件部分
  plugins: [
    new ExtractTextPlugin("styles.css"),
  ]
}
```

use:指需要什么样的 loader 去编译文件,这里由于源文件是.css 所以选择 css-loader

fallback:编译后用什么 loader 来提取 css 文件

方式二：mini-css-extract-plugin

相比 extract-text-webpack-plugin：

- 异步加载
- 没有重复的编译（性能）
- 更容易使用
- 特定于 CSS

安装： yarn add mini-css-extract-plugin -D

抽出 css ：

```
const MiniCssExtractPlugin= require("mini-css-extract-plugin");
```

modle 部分

```
//mini-css-extract-plugin 示例
 {
 test: /\.(sa|sc|c)ss$/,
 use: [
 isProduction ? MiniCssExtractPlugin.loader:'style-loader',
 'css-loader',
 'postcss-loader',
 'sass-loader',
 ],
 },
```

插件部分：

```
// 这里的配置和 webpackOptions.output 中的配置相似
// 即可以通过在名字前加路径，来决定打包后的文件存在的路径

new MiniCssExtractPlugin({
 filename: isProduction ? 'styles/'+outFilename+'.css' : 'styles/[name].[contenthash].css',
 chunkFilename: isProduction ? 'styles/[id].[hash].css' : 'styles/[id].css',
 });
```

[id]和[name]在 webpack 中被称做 placeholder
用来在 webpack 构建之后动态得替换内容的（本质上是正则替换）。
chunkFilename 是构建应用的时候生成的。

- **压缩 css: optimize-css-assets-webpack-plugin**

- ·生产环境的配置，默认开启 tree-shaking 和 js 代码压缩；
- ·通过 optimize-css-assets-webpack-plugin 插件可以对 css 进行压缩，与此同时，必须指定 js 压缩插件（例子中使用 terser-webpack-plugin 插件），否则 webpack 不再对 js 文件进行压缩；
- 设置 optimization.splitChunks.cacheGroups，可以将 css 代码块提取到单独的文件中。

安装：yarn add optimize-css-assets-webpack-plugin -D

引入插件与配置：

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

module.exports = merge(common, {
  mode: 'production',
  optimization: {
    minimizer: [new TerserJSPlugin({}), new OptimizeCSSAssetsPlugin({})],
    splitChunks: {
      cacheGroups: {
        styles: {
          name: 'styles',
          test: /\.css$/,
          chunks: 'all',
          enforce: true,
        },
      }
    },
  }})
```

总结

① 不同环境下的打包，如果出现图片显示不了时（特别是 css 中的图片），请检查 publicPath 的配置。

② mode: 'production'会开启 tree-shaking 和 js 代码压缩，但配置 optimization.minimizer 会使默认的压缩功能失效。所以，指定 css 压缩插件的同时，务必指定 js 的压缩插件。

③ mini-css-extract-plugin 插件，结合 optimization.splitChunks.cacheGroups 配置，可以把 css 代码打包到单独的 css 文件，且可以设置存放路径（通过设置插件的 filename 和 chunkFilename）。

- **生成 json 文件的列表索引插件**

安装： yarn add assets-webpack-plugin --dev

```
var AssetsPlugin = require('assets-webpack-plugin');
//生成 json 文件的列表索引插件
new AssetsPlugin({
    filename: 'assets-resources.json',
    fullPath: false,
    includeManifest: 'manifest',
    prettyPrint: true,
    update: true,
    path: buildPath,
    metadata: {version: 123}
});
```

sourceMap

源代码与打包后的代码的映射关系
在 dev 模式中，默认开启，关闭的话 可以在配置文件里

devtool:"none"

devtool 的介绍：https://webpack.js.org/configuration/devtool#devtool
eval:速度最快
cheap:较快，不用用管列的报错

Module：第三方模块
开发环境推荐
devtool:"cheap-module-eval-source-map"

线上环境可以不不开启：如果要看到一一些错误信息，推荐；
devtool:"cheap-module-source-map"

配置别名快捷方式 resolve

```
resolve: {
       extensions: ['.js', '.vue', '.json'],
       alias: {
         'src': srcPath,
         'styles': srcPath+'/styles',
         'images':srcPath+'/images',
         'config':path.resolve(srcPath,'js/config.js')
       }
     },
```

resolve.alias ： 设置别名
resolve.enforceExtension :默认是 false
如果是 true，将不允许无扩展名(extension-less)文件。如果启用此选项，只有 require('./foo.js') 能够正常工作。
resolve.extensions　自动解析确定的扩展。

设置参数　通过 script

很久以前这样用过：

```
script:{
    "start": "export NODE_ENV='development' && node app.js"
    // 在 Mac 和 Linux 上使用 export，　在 windows 上 export 要换成 set
}
```

目前推荐 cross-env

cross-env

安装：yarn add cross-env --dev

注意：cross-env scene=dev　cross-env scene=prod　后面不能添加&&

"scripts": {

"dev": "cross-env scene=dev webpack-dev-server --config webpack.config.js",

"build": "cross-env scene=prod webpack --mode production --config webpack.config.js"

},

需求： 根部不同的开发环境，来设置一些变量

```
console.log(process.env.scene);
var isProduction = (process.env.scene=='prod');
```

DefinePlugin

DefinePlugin 允许创建一个在编译时可以配置的全局常量

在配置文件里获取变量并把值设置，DefinePlugin 是 webpack 自带的插件，无法安装

```
new webpack.DefinePlugin({
  'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV),
  'process.env.DEBUG': JSON.stringify(process.env.DEBUG)
})
```

```
pluginsAll.push(new webpack.DefinePlugin({
    'sceneParam': JSON.stringify(process.env.scene),
    'laney':JSON.stringify('laney'),
    'test':'"kkkkk"'
}));
```

然后在项目中，根据不同的环境来调用不是的样式

```
if(sceneParam=='prod') {
  import './styles/test03.scss';
} else if(sceneParam=='dev'){
  import './styles/test02.less';
}
```

如果这么使用会报错：

`import` 和 `export` 只能在顶级用，不能在代码块中用。否则会报 `'import' and 'export' may only appear at the top level`。

下面的写法可以

```
switch(sceneParam){
   case 'prod':
     console.log('pppp11');
     import('styles/test03.scss');
     break;
   case 'dev':
     import('styles/login.scss');
     break;
 }
```

Require 按需加载

https://segmentfault.com/a/1190000013630936
https://github.com/jiang43605/multiple-mini-css-extract-plugin

EnvironmentPlugin

EnvironmentPlugin 是一个通过 DefinePlugin 来设置 process.env 环境变量的快捷方式。
用法：
new webpack.EnvironmentPlugin(['NODE_ENV', 'DEBUG'])
不同于 DefinePlugin，默认值将被 EnvironmentPlugin 执行 JSON.stringify。
上面的写法和下面这样使用 DefinePlugin 的效果相同：

```
new webpack.DefinePlugin({
   'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV),
   'process.env.DEBUG': JSON.stringify(process.env.DEBUG)
})

new webpack.EnvironmentPlugin({
   NODE_ENV: 'development',
// 除非有定义 process.env.NODE_ENV，否则就使用 'development'
   DEBUG: false})
```

配置全局可引用的配置文件 webpack.ProvidePlugin

在 src/js 里添加一个 配置文件，或者全局文件 configCommon，然后在 build/webpack.common.js 的插件里修改如下：

```
plugins.push(new webpack.ProvidePlugin({
    // config:path.resolve(srcPath,'js/config.js')
    // config:'config'
    // config:['config','default'],
    configCommon:[path.resolve(srcPath,'js/config.js'),'defaul
t'],
    // $:'jquery',
    // jQuery:'jquery'
  }));
```

重启项目后，在页面中就可以在项目文件中直接用 configCommon,包括里面的方法和 各种参数，非常爽

使用场景微服务架构：

方式一：webpack.ProvidePlugin

为了保证一个服务坍塌了， 不影响别的服务，需要有多个服务，多个域名，

如果是项目中可能有多个接口的域名， 不同的环境， 这些 接口的 域名都不一样， 这个时候 ，需要根据 开发 环境， 测试环境，以及线上分别配置，比如：

```
commonFun.environment = {
    // 开发环境：
    dev:{
        service:'http://localhost:8081',
        setting:'http://localhost:8082',
        manage:'http://localhost:8083'
    },
```

```
    // 测试环境:
    test:{
        service:'http://192.168.0.1:8081',
        setting:'http://192.168.0.1:8082',
        manage:'http://192.168.0.1:8083'
    },
    // 线上环境:
    prod:{
        service:'http://test1.ruanmou.com',
        setting:'http://test2.ruanmou.com',
        manage:'http://test3.ruanmou.com'
    }
}
```

如何去根据不同的环境调用这些 api 的域呢，第一种方法可以用 webpack.ProvidePlugin，在 src\js\config.js 加上以上配置，在其他页面就可以直接通过这个 configCommon.environment[sceneParam] 获取到当前所有接口需要的域

方式二：在根目录添加配置，根据环境写入到 index.html

但是这个 方法 是每次都需要重新打包才可以生效， 有没有有一种方法可以 修改配置后，不需要打包的呢，那这个时候可以 把配置文件在 index.html 里直接通过 script 引入相应环境的配置。

在根目录添加 service.local.js ，用于本地开发

配置如下:

```
window.serviceIpConfig = {
  dev:{
    Product:'http://10.0.1.28:888',//商品服务
    Order:'http://10.0.1.28:888',//订单服务
    Member:'http://10.0.1.28:999',//会员服务
    Auth:'http://10.0.1.28:222',//授权服务
  },
  debug:{
    Product:'http://127.0.0.1:888',//商品服务
    Order:'http://127.0.0.1:888',//订单服务
```

```
    Member:'http://127.0.0.1:999',//会员服务
    Auth:'http://127.0.0.1:222',//授权服务
  }
}
```

在根目录添加 service.online.js ，用于线上开发，

这个文件通过复制的方式把文件复制到打包的 static 的文进件

在 wenpack.common.js 里添加

```
var WEBPACK_ENV = process.env.scene.trim();
```

在通过模板 htmlWebpackPlugin 打包的时候，选择不同的模板，本地开发用 template/index-dev.html.,开发的时候用 template/index-prod.html，分别写入不同的配置文件就行， 就达到了 即使修改 接口得 ip 也可以及时生效， 不需要重新打包的需求了。

Step09 代码分割 code Splitting

import _ from "lodash";

console.log(_.join(['a','b','c','****']))

假如我们引入一个第三方的工具库，体积为 1mb，而我们的业务逻辑代码也有 1mb，那么打包出来的体积大小会在 2mb

导致问题：
- ✧ 体积大，加载时间长
- ✧ 业务逻辑会变化，第三方工具库不会，所以业务逻辑一变更，第三方工具库也要跟着变。

引入代码分割的概念
//lodash.js
import _ from "lodash";
window._ = _;

//index.js

注释掉 lodash 引用 //import _ from "lodash";

```
console.log(_.join(['a','b','c','****']))

//webpack.config.js
entry: {
lodash: "./lodash.js",
index: "./index.js"    },
//指定打包后的资源位置
output: {
path: path.resolve(__dirname, "./build"),
filename: "[name].js"
}
```

其实 code Splitting 概念 与 webpack 并没有直接的关系，只不过 webpack 中提供了一种更加方便的方法供我们实现 代码分割

基于 https://webpack.js.org/plugins/split-chunks-plugin/

参考：https://segmentfault.com/a/1190000017066322?utm_source=tag-newest

```
optimization: {
      splitChunks: {
            chunks: 'async',//对同步，异步，所有的模块有效
            minSize: 30000,//当模块大于 30kb
            maxSize: 0,//对模块进行二次分割时使用，不推荐使用
            minChunks: 1,//打包生成的 chunk 文件最少有几个 chunk 引用了这个模块
            maxAsyncRequests: 5,//模块请求 5 次
            maxInitialRequests: 3,//入口文件同步请求 3 次
            automaticNameDelimiter: '~',
            name: true,
            cacheGroups: {
                  vendors: {
                        test: /[\\/]node_modules[\\/]/,
                        priority: -10//优先级 数字越大，优先级越高
                  },
                  default: {
                        minChunks: 2,
                        priority: -20,
                        reuseExistingChunk: true
                   }
             }
       }
 }
```

使用下面配置即可：

```
optimization:{
    //帮我们自动做代码分割
    splitChunks:{
        chunks:"all",//默认是支持异步，我们使用 all
        //chunks:"async",//默认是支持异
     }
 }
```

打包分析

https://github.com/webpack/analyse

通过命令 webpack --profile --json > stats.json
产生一个 json 文件 ，然后上传到下面链接的网址进行分析

生成在文件加入 https://webpack.github.io/analyse/

官方推荐工具
https://webpack.js.org/guides/code-splitting/#bundle-analysis

代码利用率的问题

测试代码：
//index.js

```
document.addEventListener("click", () => {
    const element = document.createElement("div");
    element.innerHTML = "welcome to webpack4.x";
    document.body.appendChild(element);
});
```

通过控制台看看代码利用率

把里面异步代码抽离出来
//index.js

```
document.addEventListener("click", () => {
import("./click.js").then(({ default: func }) => {
//需要用到 npm install --save-dev @babel/plugin-syntax-dynamic-import

    func();
```

```
  });
});

//click.js
function handleClick() {
    const element = document.createElement("div");
    element.innerHTML = "welcome to webpack4.x";
    document.body.appendChild(element);
}

export default handleClick;
```

以上情况是在点击 按钮以后才会动态去加载相应的 js 文件， 但是有可能你在点击的时候，会出现用户用等待的情况 而出现延时， 可否让浏览器把 主要的代码加载完毕后， 在空余的时间 去加载异步点击的代码呢？

使用预先拉取和预先加载提升性能

Webpack 4.6.0 为我们提供了**预先拉取（prefetching）和预先加载（preloading）**的功能。使用这些声明可以修改浏览器处理异步 chunk 的方式。

✧ 预先拉取

使用**预先拉取**，你表示该模块可能以后会用到。浏览器会在空闲时间下载该模块，且下载是发生在父级 chunk 加载完成之后。

```
import(
 `./utilities/divide`
 /* webpackPrefetch: true */
 /* webpackChunkName: "utilities" */
)
```

以上的导入会让<link rel="prefetch" as="script" href="utilities.js">被添加至页面的头部。因此浏览器会在空闲时间预先拉取该文件。

prefetch 概念

为了帮助其它浏览器对某些域名进行预解析，你可以在页面的 html 标签中添加 dns-prefetch 告诉浏览器对指定域名预解析，如下：

<link rel="dns-prefetch" href="//domain.com">

DNS Prefetch，即 DNS 预获取，是前端优化的一部分。一般来说，在前端优化中与 DNS 有

关的有两点：

一个是减少 DNS 的请求次数，

另一个就是进行 DNS 预获取 。

✧ 预先加载

在资源上添加预先加载的注释，你指明该模块需要立即被使用。**异步 chunk** 会和**父级 chunk** 并行加载。如果**父级 chunk** 先下载好，页面就已可显示了，同时等待**异步 chunk** 的下载。这能大幅提升性能。

```
import(
  `./utilities/divide`
  /* webpackPreload: true */
  /* webpackChunkName: "utilities" */
)
```

以上代码的效果是让<link rel="preload" as="script" href="utilities.js">起作用。不当地使用 wepbackPreload 会损害性能，所以使用的时候要小心。

DLLPlugin 和 DLLReferencePlugin 的使用

DLLPlugin 和 DLLReferencePlugin 用某种方法实现了拆分 bundles，同时还大大提升了构建的速度。

1.首先 build 文件夹添加----webpack.dll.config.js:

```
var path = require("path");
var webpack = require("webpack");
module.exports = {
  // 要打包的模块的数组
  entry: {
    vendor: ['vue/dist/vue.esm.js','vue-router']
  },
  output: {
    path: path.join(__dirname, '../static/js'), // 打包后文件输出的位置
    filename: '[name].dll.js',// vendor.dll.js 中暴露出的全局变量名。
    library: '[name]_library' // 与 webpack.DllPlugin 中的`name: '[name]_library',`保持一致。
  },
  plugins: [
```

```
        new webpack.DllPlugin({
            path: path.join(__dirname, '.', '[name]-manifest.json'),
            name: '[name]_library',
            context: __dirname
        }),
    ]
};
```

2.在 package.json 的 scripts 里加上：

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

3.运行 npm run dll 在 static/js 下生成 vendor-manifest.json;

4.在 build/webpack.base.conf.js 里加上：

// 添加 DllReferencePlugin 插件

```
    plugins: [
        new webpack.DllReferencePlugin({
            context: __dirname,
            manifest: require('./vendor-manifest.json')
        })
    ],
```

5.然后在 index.html 中引入 vendor.dll.js：

```
<div id="app"></div><script src="./static/js/vendor.dll.js"></script>
```

至此，配置之后的：

可以看到 npm run build 后的时间大幅度减少，在 dist 打包体积上也比之前的小。在项目优化中，可以很大程度上加快项目的构建速度和减少项目的打包体积。

Step09. 配置 vue 环境

不管是改变 vue 引用 js 还是使用 render 函数都是不方便的，我们更希望页面组件能以.vue 文件加载到 html 文件中，在 src 目录下添加 vue 的配置文件 main.js

安装 vue: yarn add vue -D

1.在 index.html 中添加挂载点 dom（#app）

在 main.js 中引入 vue

```
import Vue from 'vue'
var app = new Vue({
   el: "#app",
   data: {
      message: 'hello webpack!!'
   }
})
```

启动 server，会出现
[Vue warn]: You are using the runtime-only build of Vue where the template compiler is not available. Either pre-compile the templates into render functions, or use the compiler-included build.

原因

vue 默认使用的是 vue.runtime.js,

vue 有两种形式的代码 compiler（模板）模式和 runtime 模式（运行时），vue 模块的 package.json 的 main 字段默认为 runtime 模式， 指向了"dist/vue.runtime.common.js"位置。
https://cn.vuejs.org/v2/guide/installation.html#%E8%BF%90%E8%A1%8C%E6%97%B6-%E7%BC%96%E8%AF%91%E5%99%A8-vs-%E5%8F%AA%E5%8C%85%E5%90%AB%E8%BF%90%E8%A1%8C%E6%97%B6

这是 vue 升级到 2.0 之后就有的特点。

而我的 main.js 文件中，初始化 vue 却是这么写的，这种形式为 compiler 模式的，所以就会出现上面的错误信息

```
// compiler
new Vue({
   el: '#app',
   router: router,
   store: store,
   template: '<App/>',
   components: { App }
})
```

解决办法一：

将 main.js 中的代码修改如下就可以

```
//runtime
new Vue({
  router,
  store,
  render: h => h(App)
}).$mount("#app")
```

解决办法二：
因为之前我们的 webpack 配置文件里有个别名配置，具体如下

```
resolve: {
      alias: {
            'vue$': 'vue/dist/vue.esm.js' //内部为正则表达式   vue 结尾的
      }
}
```

也就是说，import Vue from ‘vue’ 这行代码被解析为 import Vue from ‘vue/dist/vue.esm.js’，直接指定了文件的位置，没有使用 main 字段默认的文件位置

解决办法三：
既然到了这里我想很多人也会想到第三中解决方法，那就是在引用 vue 时，直接写成如下即可

```
import Vue from 'vue/dist/vue.esm.js'
```

这时，在 index-dev.html 修改成 {{message}}
，可以预览到页面显示 'hello webpack!!'。

2. 在项目中使用、引入样式（测试）

修改 styles/test03.scss 的样式 如下：

```
$appColor:red;
body {
  h2 {
    color:$appColor;
  }
}
```

在 main.js 里 import 'styles/test03.scss';

检测 es6 语法（正常运行说明 babel 配置成功）

在 src 新建 providers 文件夹，并添加 util.js

```
export default function getData() {
        return new Promise((resolve, reject) => {
                resolve('ok');
        })
}
```

main.js

```
//引入组件
import getData from './providers/util';
var app = new Vue({
    el: "#app",
    data: {
        message: 'hello webpack!!'
    },
    methods: {
        async fetchData() {
            const data = await getData();
            this.message = data;
        }
    },
    created() {
        this.fetchData();
    }
})
```

IE9 可以正常显示

3. 使用 Vue 单文件组件

安装 vue-loader 及依赖

yarn add vue vue-loader vue-template-compiler --dev

vue template compiler 模版解析器

简单来说就是使用这个插件就会将 template 语法转为 render 函数,那么常见的操作就是这样的情况:

```
import Vue from 'vue'
import App from './app.vue';
new Vue({
   el:'#root',
   render:h=>h(App)
})
```

配置 webpack.config.js

首先，在 Webpack 4.x 中我们必须显性的引入 vue-loader 插件

```
// webpack.config.js
const VueLoaderPlugin = require('vue-loader/lib/plugin')
module.exports = {
  module: {
    rules: [
      // ... 其它规则
      {
        test: /\.vue$/,
        loader: 'vue-loader'
      }
    ]
  },
  plugins: [
    // 请确保引入这个插件！
    new VueLoaderPlugin()
  ]
}
```

模板预编译 vue-template-compiler

当使用 DOM 内模板或 JavaScript 内的字符串模板时，模板会在运行时被编译为渲染函数。通常情况下这个过程已经足够快了，但对性能敏感的应用还是最好避免这种用法。

预编译模板最简单的方式就是使用单文件组件——相关的构建设置会自动把预编译处理好，所以构建好的代码已经包含了编译出来的渲染函数而不是原始的模板字符串。

如果你使用 webpack，并且喜欢分离 JavaScript 和模板文件，你可以使用 vue-template-loader，它也可以在构建过程中把模板文件转换成为 JavaScript 渲染函数。

如果你需要在客户端编译模板 (比如传入一个字符串给 template 选项，或挂载到一个元素上并以其 DOM 内部的 HTML 作为模板)，就将需要加上编译器，即完整版：

vue 的 template 是在运行时转换为 render 函数的

单文件模板中的 template 会在打包运行时 ，即编译阶段编译成 render 函数.

vue 底层-template 模板编译

https://blog.csdn.net/wang729506596/article/details/90947583

```
// 需要编译器
new Vue({
  template: '<div>{{ hi }}</div>'
})

// 不需要编译器
new Vue({
  render (h) {
    return h('div', this.hi)
  }
```

})

Vue2.0 render:h => h(App)的理解

render: h => h(App) 是下面内容的缩写：

render: function (createElement) {

    return createElement(App);

}

进一步缩写为(ES6 语法)：

render (createElement) {

    return createElement(App);

}

再进一步缩写为：

render (h){

    return h(App);

}

按照 ES6 箭头函数的写法，就得到了：

render: h => h(App);

```
new Vue({
  router,
  store,
  //components: { App }   vue1.0 的写法
  render: h => h(App)      vue2.0 的写法
}).$mount('#app')
```

render 函数是渲染一个视图，然后提供给 el 挂载，如果没有 render 那页面什么都不会出来

vue.2.0 的渲染过程：

1.首先需要了解这是 es 6 的语法，表示 Vue 实例选项对象的 render 方法作为一个函数，接受传入的参数 h 函数，返回 h(App) 的函数调用结果。

2.其次，Vue 在创建 Vue 实例时，通过调用 render 方法来渲染实例的 DOM 树。

3.最后，Vue 在调用 render 方法时，会传入一个 createElement 函数作为参数，也就是这里的 h 的实参是 createElement 函数，然后 createElement 会以 APP 为参数进行调用

结合一下官方文档的代码便可以很清晰的了解 Vue2.0 render:h => h(App)的渲染过程。

// ES5

(function (h) {

```
  return h(App);
});
// ES6
h => h(App);
//官方
render: function (createElement) {
    return createElement(
      'h' + this.level,    // tag name 标签名称
      this.$slots.default // 子组件中的阵列
    )
  }
```

选项/DOM

#el

为 Vue 实例提供 DOM 元素挂载。值可以是 CSS 选择符，或实际 HTML 元素。

在实例挂载之后， 元素可以用 vm.$el 访问。

如果这个选项在实例化时有作用，实例将立即进入编译过程，否则，需要显式调用 vm.$mount() 手动开启编译。

提供的元素只能作为挂载点。所有的挂载元素会被 Vue 生成的 DOM 替换。因此不推荐挂载 root 实例到 <html> 或者 <body> 上。

4. 准备单文件组件

在 src 目录下创建文件 App.vue
在 src 目录下新建目录 components 用来存放单文件组件，新建文件 header.vue

App.vue

```
<template>
  <div>
    <home-header user="vict"></home-header>
  </div>
```

```
</template>
<script>
  import homeHeader from '@/components/header.vue'
  export default {
    name:'app',
    components:{
      homeHeader
    }
  }</script>
<style lang="scss" scoped>
  $txtColor:red;
  .sg{
    color:$txtColor
  }
</style>
```

header.vue（子组件）

```
<template>
  <div>
    header
    <span>color</span>
    <img src="@/images/logo02.png">
    {{this.message}} - {{this.user}}
  </div></template>
<script>
  export default {
    name:'homeHeader',
    data () {
      return {
        message:'hello world'
      }
    },
    props: {
      user:String
    }
  }</script>
<style lang="less" scoped>
  span{
    color:green
  }
</style>
```

main.js

```
import Vue from 'vue'
import App from './App.vue'
import 'styles/index.scss'
new Vue({
  render:h => h(App)
}).$mount('#app')
```

5. 注入 Vue Router

1. 安装 VueRouter

yarn add vue-router --dev

2.构建 router 组件

按照路径创建 router 文件：src/router/router.js

router.js

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Page1 from '@/pages/page1'
import Page2 from '@/pages/page2'

Vue.use(VueRouter)
export default new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [{
    path: '/',
    name:'home',
    component: Page1,
    beforeEnter(from,to,next){
        console.log(`beforeEnterhome from ${from} to ${to}`);
       setTimeout(()=>{
                next();
        },1000)
    }
  },{
```

```
    path: '/page2',
    name:'page2',
    component: Page2
  }]
})
```

Vue.use

用户执行 Vue.use 的时候，实际执行的是模块的 install 方法，会把 vue 的实例传递进去

3. 在 main.js 的 Vue 实例中注入 Router

```
import Vue from 'vue'
import App from './App'//引入 router 组件
import router from './router/router'
import './assets/styles/reset.css'
new Vue({
  //注入 router
  router,
  render:h => h(App)
}).$mount('#app')
```

5. 在 app.vue 页面添加路由外联

```
<template>
  <div>
    <home-header user="vict"></home-header>
    <!-- 路由外链 -->
    <router-view></router-view>
  </div>
</template>
<script>
  import homeHeader from '@/components/header.vue'
```

常见的渲染失败的原因：

- router 实例的路由配置项 routes，不要误写成 routers

- routes 配置项中必须要有路径为‘ /’，否则渲染失败

6. 使用 router

在 App.vue 中可以直接使用<router-view>和<router-link>标签

router-link

https://router.vuejs.org/zh/api/ ， 在 header.vue 中添加

```
<div class="page-head">
          <h1>Hello App!</h1>
          <div class="top-menu">
              <!-- router-link 进行导航。 -->
              <router-link to="/page1">page1</router-link>
              <router-link to="/page2">page2</router-link>
          </div>
      </div>
```

并调整相应的样式

这时路由已经可以正常工作了！

只是被激活的菜单的默认连接是 router-link-exact-active router-link-active

```
<h1 data-v-29e8c3c6>Hello App!</h1>
▼<div data-v-29e8c3c6 class="top-menu">
  <a data-v-29e8c3c6 href="/page1" class>page1</a>
  <a data-v-29e8c3c6 href="/page2" class="router-link-exact-active router-link-active">page2</a>
 </div>
</div>
```

对 class 进行配置

```
// router.js

export default () => {

  return new Router({

    routes,

    mode: 'history',

    base: '/base/',
```

```
  linkActiveClass: 'active-link',

  linkExactActiveClass: 'exact-active-link'

})}
```

vue-router 认为只有路由真正匹配时，才会加上 exact-active-link 这个 class，如果只有一部分重合，就会加上 active-link。

scrollBehavior

页面跳转后，页面是否滚动。

```
// router.js
export default () => {
  return new Router({
    scrollBehavior (to, from, savedPostion) {
      if (savedPostion) {
        return savedPostion
      } else {
        return { x: 0, y: 0 }
      }
    }
  })}
```

vue-router 传参的坑（query 和 params）

1. query 方式传参和接收参数

```
传参:
this.$router.push({
       path:'/xxx',
       query:{
         id:id
       }
     })

接收参数:
this.$route.query.id
```

注意:传参是 this.$router,接收参数是 this.$route,这里千万要看清了!!!

1.$router 为 VueRouter 实例，想要导航到不同 URL，则使用$router.push 方法
2.$route 为当前 router 跳转对象，里面可以获取 name、path、query、params 等

2. params 方式传参和接收参数

```
传参:
this.$router.push({
        name:'xxx',
        params:{
          id:id
        }
      })

接收参数:
this.$route.params.id
```

注意:params 传参，push 里面只能是 name:'xxxx',不能是 path:'/xxx',因为 params 只能用 name 来引入路由，如果这里写成了 path，接收参数页面会是 undefined!!!

另外，二者还有点区别，直白的来说 query 相当于 get 请求，页面跳转的时候，可以在地址栏看到请求参数，而 params 相当于 post 请求，参数不会再地址栏中显示

子路由 children，路由的嵌套

使用 children 属性实现路由嵌套,子路由 path 前不要加/，否则永远以根路径开始请求

--- 在 page1 页面添加子路由，

```
{
      path: '/page1',
      name:'page1',
      component: Page1,
      // 路由元信息 meta
      meta: {
          title: 'this is app',
          description: 'xxx'
       },
      children: [
          // 在 page1 路由下，添加子路由
          //// 子路由使用，需要再上一级路由页面加上 router-view 显示。
          {
            path: 'test',
            component: page1Con
          }
       ]
},{
      path: '/page2',
```

----在子路由的上一级添加 router-view

```
  <h1>this is page1</h1>

  <ul @click="toChildPage($event)">
      <li tag='1'>子页面1</li>
      <li tag='2'>子页面2</li>
      <li tag='3'>子页面3</li>
  </ul>
  <router-view></router-view>
  <button type="button" @click="jumpPage()" style="margin-top:300px">
</div>
plate>
```

给 router-view 加 transition

给所有路由切换加个过渡效果。

```
<template>
  <div id="app">
    <transition name="fade"> // 使用 name
      <router-view />
    </transition>
  </div>
</template>
```

命名视图

命名视图在 components（这时会多个 s）后加属性再在使用

<router-view> </router-view>的时候用 name 引入,可以使一个页面中存在多个路由

```
  <router-view></router-view>
  <router-view name="left"></router-view>
  <router-view name="main"></router-view>
```

```
var router=new VueRouter({
              routes:[
                  {path:'/',components:{
                      default:header,
                      left:leftBox,
                      main:mainBox
                  }},
              ]
          })
```