打包构建工具介绍

市面上现存的打包构建工具有很多中，有很多中成熟的产品，包括parcel，webpack，snowpack，rollup等等。

为什么要使用打包构建工具

原有的web网页开发的流程是通过HTML文件构建应用界面在浏览器中运行，一切HTML+CSS+JavaScript都是通过浏览器的执行引擎边解释边运行的，随着互联网技术的发展，应用规模越来越大，导致这些面向过程的代码堆叠在一起很难构建大型的复杂应用，所以便有了打包构建工具（这里可以参考webpack的工作描述图），在众多的打包构建工具中，目前最流行并且生态最广的就属webpack了。

1.webpack介绍

> 前言，webpack是大前端时代的一个核心的支撑物，也就是说如果没有webpack也就就没有大前端时代

1.1什么是webpack

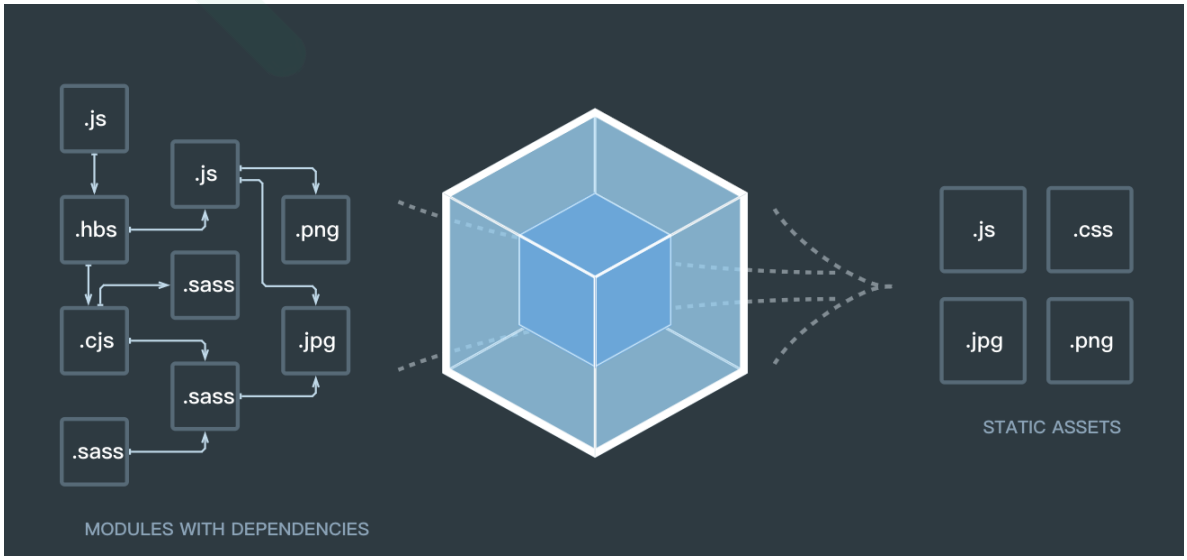
在过去的脚手架项目学习过程中，我们通过创建vue或jsx文件完成项目的开发和构建，但是这些文件本身是无法被浏览器识别并解析的，他们之所以能工作就是通过webpack环境来编译和构建的。

> 定义：
>
> webpack是一个本地的编译环境，他可以把浏览器不识别的文件类型或语法，通过loader和plugin等插件转换成浏览器能识别的html，css和javascript并实时编译和输出

1.2webpack的简介

本质上，*webpack* 是一个现代 JavaScript 应用程序的*静态模块打包器(module bundler)*。当 webpack 处理应用程序时，它会递归地构建一个*依赖关系图(dependency graph)*，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 *bundle*。

我们来通过图片理解一下webpack的核心功能



本质上，webpack就是一个加工工厂，将左侧复杂的文件类型和语法转换成右侧的web页面可用的语法和文件

1.3webpack的基本使用方式

```
npm i webpack -D #安装webpack核心文件
```

```
npm i webpack-cli -D #安装webpack脚手架工具
```

> -D的介绍，平时我们安装依赖都是使用-s来安装今天使用了-D他们的主要区别就是-s代表项目构建的必要依赖，就是硬依赖，比如我们使用 jquery开发项目，那么jquery就需要-s安装，因为打包构建之后jquery也必须出现在项目里，-D的依赖叫做开发依赖他的作用就是为了解决项目在开发过程中的一些调试编译的功能，他是用来操作项目的，也就是说构建之后webpack这种打包工具不需要存在于项目里，所以我们使用-D安装

1.4创建一个简单的webpack项目

首先在案例文件夹中创建一个文件夹`webpack01`在文件夹上右键使用命令行打开

```
npm init #将其初始化为npm项目，并声称package.json
```

然后安装webpack和webpack-cli

安装完毕之后我们创建src文件夹并且在src文件夹之下创建index.js文件结构如下

```
├── package-lock.json #package的相关锁定依赖
├── package.json #项目依赖核心描述文件
├── src #源代码文件夹
    └── index.js #项目内容
```

在index.js中创建如下内容

```
const name = 'hello webpack'
```

在项目根目录下创建webpack.config.js文件

```
//webpack.config.js
const path = require('path')//引入path模块
module.exports = {
    entry:{//entry表示webpack编译的入口文件，代表当前我们项目中使用的源代码部分的文件路径和依赖名称
        index:'./src/index.js' //index代表构建时生成的js依赖名称，他的值代表编译之前要扫描的文件路径
    },
  mode:"production",//设置当前打包的方式为生产环境构建
    output:{//output代表构建完毕后生成的依赖的配置对象
    //path代表生成的js文件要放置的文件路径
        path:path.resolve(__dirname,'dist'),
    //filename代表生成的js文件要生成的名称[name]相当于从entry中获取的
    //名称，就是entry中配置的key
        filename:'[name].bundle.js'
    }
}
```

在package.json中配置webpack编译要执行的指令

```
{
  "name": "webpack01",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    //该注释在package.json中使用时要删掉，package.json不允许使用注释
    //build为webpack运行命令，执行npm run build时会执行该命令
    //该命令的意义为通过webpack运行webpack.config.js文件的配置参数，并展示颜色和过程
    //运行后webpack就会编译entry中配置的js文件并输出到output中配置的路径内
    "build": "webpack --config webpack.config.js --color --progress"
  },
  "author": "LeoZhang",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^5.1.3",
    "webpack-cli": "^4.0.0"
  }
}
```

以上操作完成之后在项目根目录下运行

```
npm run build
```

然后我们发现在index.bundle.js中什么都没有，这时我们将index.js改造成如下效果

```
const name = 'hello webpack'
console.log(name)
```

我们在运行 `npm run build` 查看结果这时输出的结果为

```
console.log("hello webpack");
```

然后我们在做一步操作

index.js的代码改为如下效果

```
const name = 'hello webpack'
if(name.indexOf('hello')!=-1){
  console.log(name)
}
```

然后运行并查看

然后我们可以查看命令行中的日志，并且查看项目中dist文件夹生成的index.bundle.js文件内容

> 总结：
>
> 通过以上案例我们可以将webpack理解成一个加工厂，他的工作就是将我们写的js以及其他代码进行一次修改之后再输出到指定的位置。

2.webpack的用法

2.1配合loader使用的基础

2.1.1babel-loader

了解了webpack的基本使用之后我们在js代码中增加一段逻辑

```
//src/index.js
const name = 'hello webpack'
console.log(name)
let d = new Promise((resolve,reject) => {
    setTimeout(()=>{
        resolve('hello')
    },1000)
})
d.then(res => {
    console.log(res)
})
const arr = [1,2]
arr.map(item => {
    console.log(item)
})
```

在这里我们增加了一段promise代码，然后再次执行

```
npm run build
```

查看生成的代码后我们发现Promise和es6的循环都是原样输出的只是做了代码混淆并没有其他的变化

那么这个构建对我们来说其实是一次无意义的构建。因为Promise和es6以后的语法并不是所有浏览器都支持的

webpack存在的意义不光是构建代码，还要在构建代码的过程中解决兼容性的问题

这里我们就需要学习一个新的功能loader。

首先我们看一下官方对loader的介绍：

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

本质上，webpack loader 将所有类型的文件，转换为应用程序的依赖图（和最终的 bundle）可以直接引用的模块。

> 我们在这里可以对他做一个最初步的理解，就是loader可以帮我们解决javascript兼容性的问题，还可以让浏览器本来不能执行的代码变成浏览器能执行的js代码

这里就需要使用一个叫做babel-loader的插件

1. 安装babel-loader

   ```
   npm i babel-loader -D
   ```

2. 在webpack中使用loader

   ```
   const path = require('path')
   module.exports = {
   ```

```
    entry:{
        index:'./src/index.js'
    },
    output:{
        path:path.resolve(__dirname,'dist'),
        filename:'[name].bundle.js'
    },
    module:{
        rules:[
            {
        //test代表当前的loader扫描的文件类型
                test:/\.js$/,
        //use代表对当前文件类型应用的loader是什么
                use:{loader:'babel-loader'}
            }
        ]
    }
}
```

3. 运行

```
npm run build
```

然后我们发现结果还是原样输出的**(并且这里可能会发生错误)**仅仅是混淆了一部分代码。这是因为光有babel-loader是不够的

所以接下来我们要把babel的核心包配置起来

```
npm i @babel/core -D
```

```
npm i @babel/preset-env -D
```

```
npm i core-js -s
```

执行完以下操作之后我们在项目根目录创建一个名为.babelrc的文件

> 该文件的作用是用来让babel可以根据我们代码中使用的语法来决定是否将代码做兼容性处理

```
//.babelrc 配置babel/preset-env用来驱动core-js来解析Promise等包
{
    "presets":[
        [
            "@babel/preset-env",
            {
                "useBuiltIns": "usage",//设置为自动检测es6以后的语法并按需引入相关库
                "corejs":3//使用core-js3版本的源码库
            }
        ]
    ]
}
```

在创建.browserslistrc文件

> browserslistrc文件用来描述我们当前的项目要兼容的浏览器的范围，babel会根据需要兼容的浏览器范围自动的进行代码的转译，这样我们就不需要考虑当前写的js是否在我们想要兼容的浏览器中可以使用，因为我们设置浏览器范围之后babel就会输出成这个范围内的浏览器都能正常执行的代码

```
> 0.25%
last 2 versions
```

browserlist配置说明

例子	说明
> 1%	全球超过1%人使用的浏览器
> 5% in US	指定国家使用率覆盖
last 2 versions	所有浏览器兼容到最后两个版本根据 CanIUse.com 追踪的版本
Firefox ESR	火狐最新版本
Firefox > 20	指定浏览器的版本范围
not ie <=8	方向排除部分版本
Firefox 12.1	指定浏览器的兼容到指定版本
unreleased versions	所有浏览器的beta测试版本
unreleased Chrome versions	指定浏览器的测试版本
since 2013	2013年之后发布的所有版本

使用了如下配置之后我们在执行

```
npm run build
```

这时我们查看编译后的文件，发现代码混合进来很多的混淆函数，并且我们查看最下方可以找到我们自己编写的部分代码，然后我们查看webpack构建时输出的日志

```
asset index.bundle.js 20.2 KiB [emitted] [minimized] (name: index)
runtime modules 939 bytes 4 modules
modules by path ./node_modules/core-js/internals/*.js 41.7 KiB 79 modules
modules by path ./node_modules/core-js/modules/*.js 14.5 KiB
  ./node_modules/core-js/modules/es.array.map.js 790 bytes [built] [code
generated]
  ./node_modules/core-js/modules/es.object.to-string.js 395 bytes [built] [code
generated]
  ./node_modules/core-js/modules/es.promise.js 13.4 KiB [built] [code generated]
./src/index.js 399 bytes [built] [code generated]
webpack 5.1.3 compiled successfully in 1601 ms
```

这里我们看他从core-js中拿出了map和promise的包，我们可以在构建出来的代码中搜索promise和map发现有很多地方对promise和map做了声明。

这就是babel根据我们在browserslistrc中定义的兼容范围来决定的构建结果，为了兼容> 0.25%使用率的浏览器并且兼容到最后两个版本，babel为了防止部分浏览器对Promise和map循环不兼容，他引用了core-js中单独对map循环和Promise库的代码这样就算浏览器不兼容map循环和promise对象，也能直接调用core-js中实现的promise和map循环，这个就是babel的作用。

接下来我们在学习一下如何使用browserslist，我们可以在当前的目录下在命令行执行

```
npx browserslist #用来查看当前项目的.browserslistrc文件中的配置描述所兼容的浏览器列表
```

执行之后控制台就会出现一系列的浏览器名称，这个就是我们当前配置的这个范围所兼容的所有浏览器，我们可以改造一下browserslist将它设置为

```
chrome > 80
```

这个配置代表只兼容chrome浏览器80版本以上的，然后我们执行

```
npx browserslist
```

看一下当前输出的浏览器列表

```
zhangyunpeng@zhangyunpengdeMacBook-Pro webpack01 % npx browserslist
chrome 86
chrome 85
chrome 84
chrome 83
chrome 81
```

只有chrome81-86，这代表我们当前的项目只需要支持chrome浏览器81-86版本就可以

这时我们再执行一次

```
npm run build
```

在看生成的index.bundle.js

发现代码又变成原样输出了，这是因为如果只支持chrome80以上版本，他的浏览器本身就支持es6以后的js语法所以代码不需要对他进行依赖库的补充。

以上就是在webpack中应用babel-loader并且根据想要兼容的浏览器范围自动构建兼容性代码的方法。

2.2配合plugin使用的基础

当我们已经解决了js的兼容性问题之后再考虑一个问题，我们使用webpack一定会配合项目去使用，这样的话如果只能编译js文件是完全没有用的，因为只有js文件我们没法在本地直接运行，需要依赖html容器，这样我们才能实现边开发边看结果。所以想要让我们写的js不光能编译还能执行就需要使用webpack另一个功能：plugins

> plugins简介：
>
> 插件是 webpack 的支柱功能。webpack 自身也是构建于，你在 webpack 配置中用到的**相同的插件系统**之上！
>
> 插件目的在于解决 loader 无法实现的**其他事**。

> loader能干的事情仅限于，将webpack中的js或其他文件做一个加工并输出成js模块文件，让webpack能认识，所以想要将结果输出成其他类型的文件我们还得借助plugin

2.2.1html-webpack-plugin

我们学习第一个plugin，html-webpack-plugin，他可以将我们定义的js文件打包构建生成html与js混合的项目。首先我们要安装他

```
npm i html-webpack-plugin -D
```

安装完成之后我们需要做的就是在webpack.config.js文件中引用他并且配置到plugins中

```
//webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin')
```

然后我们在项目根目录下创建public文件夹

并在public文件夹内部创建index.html文

然后在module,entry,output的同级别位置添加

```
plugins:[
  new HtmlWebpackPlugin({
    template:'./public/index.html',//html原始模版位置
    filename:'index.html',//生成的html文件名
    //生成的html文件中引用的js依赖模块，这里填写的就是在entry中定义的key
    //也就是说如下配置相当于在index.html中引用了src/index.js
    chunks:['index']
  })
]
```

做了如下操作之后我们再执行

```
npm run build
```

查看结果，这次我们的dist文件夹中多生成了一个html文件，我们查看他的源代码，发现这个html文件自动引入了index.bundle.js并且运行他可以自动执行index.js文件。

完成此步骤之后我们回想vue，react等脚手架的结构。我们这里的index.html就是当时我们项目中的index.html这里的index.js就是项目中的main.js，学习这里我们就能理解为什么当时脚手架中不需要在index.html中引入任何依赖就可以关联到js文件了。因为他们都是通过html-webpack-plugin实现的。

2.3resolve的使用

resolve是webpack提供的一个用来解析文件以及引用路径的模块。

比如我们在过去的脚手架中使用import引用文件时，不需要写文件的后缀就可以实现引用对应的模块

并且如果引用的文件是某个文件夹中的index.js如test/index.js，我们可以直接引用到该文件夹即可，如import xxx from './test'。

还有我们可以通过

```
import xxx from '@/components/xxx'
```

这种@的形式来当作绝对路径进行模块的引用。

这些能力都是通过resolve对象进行解析的。

他的配置方式如下,在webpack.config.js中添加如下代码,

```
//resolve是与entry，plugins等平级的属性，是webpack的解析模块
resolve:{
  //extensions用来定义后缀列表，在这里定义的后缀的文件类型在import引用时就不需要填写后缀
  extensions:['.js','.jsx','.vue','.less','.sass'],
  //alias代表引用路径的别名，如下配置代表import中如果存在@符号的路径
  //@符号就会被解释为从电脑根目录到当前项目的src，所以如果引用models/user-model.js
  //只需要import userModel from '@/models/user-model.js'就相当于从根目录到user-
model.js的全路径
  alias:{
    '@':path.resolve(__dirname,'src')
  }
}
```

我们现在配置完resolve之后在src中创建一个model文件夹并且在其中创建一个user-model.js文件

```
//model/user-model.js
export default {
    namespaced:true,
    state:{
        list:[]
    }
}
```

然后我们在index.js中进行引用，增加如下代码

```
//index.js
import userModel from  '@/model/user-model'
console.log(userModel)
```

然后我们通过

```
npm run build
```

进行打包，这回我们直接运行生成的index.html文件，查看控制台是否打印了userModel对象的内容，如果打印出了结果就说明我们定义的别名和全路径的代号都生效了

2.4样式处理

完成了上述功能之后我们的webpack环境已经可以解决开发和代码的初步编译了，下面我们学习一下如何在webpack环境中使用css样式。

首先我们在src中创建一个index.css文件

```
/*index.css*/
.test{
    display: flex;
    flex-direction: column;
    justify-content: center;
}
```

然后我们尝试在index.js中通过import引用css文件

```
import '@/index.css'
```

然后执行

```
npm run build
```

这里会产生如下错误

```
Module parse failed: Unexpected token (1:0)
You may need an appropriate loader to handle this file type, currently no
loaders are configured to process this file. See
https://webpack.js.org/concepts#loaders
> .test{
|   display: flex;
|   flex-direction: column;
 @ ./src/index.js 2:0-21
```

这是因为我们的webpack目前只认识js文件，因为我们在loader中仅仅对js文件做了兼容性处理。

他还不能识别css文件并加以处理。

如果想在js中引入css样式需要通过style-loader和css-loader

```
npm i style-loader -D #安装style-loader，style-loader是处理style样式的，他用来将样式文件通过js动态的添加到html的head标签中来实现样式加载
npm i css-loader -D #安装css-loader，css-loader是处理css样式文件的
```

安装完毕之后我们需要在webpack的module中的rules内部添加css文件的loader

```
//webpack.config.js
//将这段代码添加到module中的rules中
{
  test:/\.css$/,//检测css结尾的文件
    //这里的loader是有顺序的，固定写法，先放style-loader，后放css-loader,loader的加载顺序是倒着加载的，所以会先执行css-loader将css样式代码转成js代码，然后执行style-loader将js中的样式动态添加到网页中
    use:[
      {loader:'style-loader'}, //使用style-loader用来将style样式追加到js代码中
      {loader:'css-loader'} //使用css-loader用来解析css文件
    ]
}
```

完成之后我们在执行

```
npm run build
```

这里我们在看打包构建的index.bundle.js，发现内部有处理样式的js文件

但是这个结果并不是我们预期想要的结果因为css样式文件我们想要通过单独的方式抽取到外面

与js混合的话会造成index.bundle.js单文件过大。

这里我们还需要使用一个plugin来处理css的分离：mini-css-extract-plugin

```
npm i mini-css-extract-plugin -D
```

然后我们在webpack.config.js中引用

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

然后需要改造css-loader部分的代码改造为如下

```
{
  test:/\.css$/,
    use:[

      //{loader:'style-loader'},
      MiniCssExtractPlugin.loader,//这里必须将MiniCssExtractPlugin放在这个位置并且提取css之后不需要使用style-loader
      {loader:'css-loader'},

    ]
}
```

最后一步是将MiniCssExtractPlugin注册到plugins中

```
//将它放入plugins中
new MiniCssExtractPlugin({
  filename:'[name].css'//这里的[name]获取的还是entry中的key
})
```

完成之后我们执行

```
npm run build
```

然后发现打包构建出来的文件多了一个index.css并且查看html文件发现已经将css自动引入了。

但是这里还缺少一个功能就是css在各浏览器的兼容性也是存在差异的，有些浏览器处理的样式

需要单独的增加浏览厂商的前缀才能展示正确的样式，这里我们可以通过一个叫做postcss-loader的loader来处理css兼容性问题

所以我们需要安装postcss-loader和postcss 以及postcss-preset-env三个组件

```
npm i postcss-loader -D
npm i postcss -D
npm i postcss-preset-env -D
```

然后在webpack.config.js中配置postcss-loader，改造module如下

```
{
  test:/\.css$/,
    use:[
      //{loader:'style-loader'},
      MiniCssExtractPlugin.loader,
      {loader:'css-loader'},
      {loader:'postcss-loader'}//postcss-loader安装在最后
    ]
}
```

然后在根目录创建postcss.config.js

```
//postcss.config.js这里默认插件使用postcss-preset-env
module.exports = {
    plugins: {
        'postcss-preset-env': {},
    }
}
```

然后我们运行

```
npm run build
```

然后查看index.css，发现并没有任何变化，这是因为我们刚才把browserslistrc改成了chrome > 80

postcss-loader同样是根据浏览器兼容范围决定的css补全，这个范围的css不需要考虑兼容性

所以我们将browserslistrc改成

```
> 0.25%
last 2 versions
```

让他变成兼容使用率大于0.25%的浏览器并且兼容到每款浏览器的最后两个版本，这里可以使用 `npx browserslist` 来看一下当前的浏览器列表

这回我们在通过npm run build来查看得到的index.css这里我们发现

```
.test{
    display: -webkit-box;
    display: -ms-flexbox;
    display: flex;
    -webkit-box-orient: vertical;
    -webkit-box-direction: normal;
        -ms-flex-direction: column;
            flex-direction: column;
    -webkit-box-pack: center;
        -ms-flex-pack: center;
            justify-content: center;
}
```

生成的css样式变成了带前缀的样式表，这样我们就可以实现兼容更多的浏览器了。

但是还有一点不足的地方，我们来看一下当前的css打包的结果并不是压缩效果，而是纵向展示的，这样我们的css文件会比压缩文件大

所以我们应该使用css压缩技术来让他和js效果一样

这里需要安装postcss的插件cssnano

```
npm i cssnano -D
```

安装完成后改造postcss.config.js

```
module.exports = {
    plugins: {
        'postcss-preset-env': {},
        'cssnano':{}
    }
}
```

然后我们接着打包构建

```
npm run build
```

完成后发现现在的css可以压缩了

2.5样式处理sass处理

首先我们在src中创建一个index.scss

```
.p-test{
    display: flex;
    justify-content: center;
    .p-item{
        flex-grow: 1;
    }
}
```

然后我们在index.js中引入index.scss

```
import '@/index.scss'
```

然后执行

```
npm run build
```

执行之后发现报错。这是因为我们只处理了css样式但是scss文件webpack仍然不认识

那么这时候我们就需要将scss也添加到module中如下,与css处理完全一样

```
{
  test:/\.scss$/,
    use:[
      //{loader:'style-loader'},
      MiniCssExtractPlugin.loader,
      {loader:'css-loader'},
      {loader:'postcss-loader'}
    ]
}
```

然后运行

```
npm run build
```

当我们运行完毕发现，打包构建没有问题，但是生成的index.css中的css样式p-test还是嵌套语法，这样的嵌套格式网页是无法直接解析的。

所以这里我们需要引入另外一个loader：sass-loader

并且还需要一同安装sass

```
npm i sass-loader -D
npm i sass -D
#或者
yarn add sass-loader -D
yarn add sass -D
```

然后我们改造scss文件的处理

```
{
  test:/\.scss$/,
    use:[
      //{loader:'style-loader'},
      MiniCssExtractPlugin.loader,
      {loader:'css-loader'},
      {loader:'postcss-loader'},
      {loader:'sass-loader'}//使用sass-loader来编译sass语法
    ]
}
```

这样操作完毕之后我们再执行一次

```
npm run build
```

再次查看生成的index.css文件，这时p-test的嵌套语法被解析成了css的子代选择器形式并且也补全了兼容性样式。这就是

sass-loader处理scss文件的办法。

3.loader的运行原理

loader其实就是在webpack开始工作的时候读取了我们配置loader的文件并且将它们的源码输入到loader中，在loader中做的操作就是拿到源码，做简单的处理并返回，从而交给下一个loader

我们打开webpack01的项目，查看webpack.config.js

这里我们对js文件使用了两个loader，这次使用的loader不是下载的，而是本地文件路径，所以我们找到loader中的代码

查看两个loader中的代码。

这两个loader中的代码就是自定义loader的最基本的格式，

loader是一个函数，参数就是配置的文件的源代码，会以字符串的形式得到，我们可以通过处理字符串来改变代码的内容

return之后源代码就交给下一个loader处理直到loader全处理完最后交给webpack，生成到output配置的路径中。

这里loader执行的顺序是按照数组配置的倒序执行的所以先执行p-loader1在执行p-loader

我们可以看到在p-loader1中我们对代码进行了替换，将所有的let和const替换成了var，然后第二个loader执行的时候得到的

source就是替换完之后的样子。这就是loader处理代码的过程。

关于AST抽象语法树和gogocode

AST介绍

抽象语法树（abstract syntax code，AST）是源代码的抽象语法结构的树状表示。这里特指编程语言的源代码。

树上的每个节点都表示源代码中的一种结构，之所以说是抽象的，是因为抽象语法树并不会表示出真实语法出现的每一个细节，比如说，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现。

抽象语法树并不依赖于源语言的语法，也就是说语法分析阶段所采用的上下文无文文法，因为在写文法时，经常会对文法进行等价的转换（消除左递归，回溯，二义性等），这样会给文法分析引入一些多余的成分，对后续阶段造成不利影响，甚至会使合个阶段变得混乱。因些，很多编译器经常要独立地构造语法分析树，为前端，后端建立一个清晰的接口。

抽象语法树在很多领域有广泛的应用，比如浏览器，智能编辑器，编译器等。为何需要抽象语法树（抽象语法树作用）

为什么需要抽象语法树

编程语言太多，需要一个统一的结构让计算机识别。

作用：比如 `typescript` 的类型检查，IDE的语法高亮，代码检查，转译等等，都是需要先将代码转化成AST在进行后续的操作。

抽象语法树的生成过程（编译）

js为例：

词法分析（lexical analysis）：进行词法分析的程序或者函数叫作词法分析器（Lexical analyzer，简称Lexer），也叫扫描器（Scanner，例如typescript源码中的scanner.ts），字符流转换成对应的Token流。

tokenize：tokenize就是按照一定的规则，例如token令牌（通常代表关键字，变量名，语法符号等），将代码分割为一个个的“串”，也就是语法单元）。涉及到词法解析的时候，常会用到tokennize。

语法分析（parse analysis）：是编译过程的一个逻辑阶段。语法分析的任务是在词法分析的基础上将单词序列组合成语法树，如“程序”，“语句”，“表达式”等等.语法分析程序判断源程序在结构上是否正确。源程序的结构由上下文无关文法描述。

词法分析的示例

```
const a = 1;
const b = a + 1;
```

词法分析器在解释这段语法的时候会把他们整理成一个一维的线性表如下：

```
[
  {
    type:'Keyword',
    value:'const'
  },
  {
    type:'Identifier',
    value:'a'
  },
  {
    type:'Punctuator',
    value:'='
  },
  {
    type:'Numeric',
    value:'1'
  },
  {
    type:'Punctuator',
    value:';'
  },
  {
    type:'Keyword',
    value:'const'
  },
  {
    type:'Identifier',
    value:'b'
  },
  {
    type:'Punctuator',
    value:'='
  },
  {
    type:'Identifier',
    value:'a'
  },
  {
    type:'Punctuator',
    value:'+'
  },
  {
    type:'Numeric',
    value:'1'
  },
  {
```

```
    type:'Punctuator',
    value:';'
  }
]
```

词法分析主要是通过scanner扫描的过程，通过正则，有穷自动机等方式识别关键词，分隔符，运算符等内容，最终将识别的结构装入到结果集中，这样便实现了编译的第一个步骤。

语法分析的示例

刚才的案例得到的线性表可以通过语法分析最终转换成AST结构，这个结构通常以树的形式变现，可以通过recast框架进行抽象语法树的查看。

随意创建一个node项目，在项目中安装recast工具

```
npm i recast -s
```

在项目中的index.js中输入

```
const recast = require('recast')
const util = require('util')
const code = `
    const a = 1;
    const b = a + 1;
`
let ast = recast.parse(code)
console.log(util.inspect(ast.program.body, { showHidden: false, depth: null }))
```

在命令行工具中输出该内容，会出现如下对象展示在控制台中，这个就是经过了语法分析转换之后的ast语法树的结构

```
[
  VariableDeclaration {
    type: 'VariableDeclaration',
    declarations: [
      VariableDeclarator {
        type: 'VariableDeclarator',
        id: Identifier {
          type: 'Identifier',
          name: 'a',
          loc: {
            start: { line: 2, column: 10, token: 1 },
            end: { line: 2, column: 11, token: 2 },
            lines: {...},
                        tokens: {...},
                        indent: 4
          }
        },
        init: Literal {
          type: 'Literal',
          value: 1,
          raw: '1',
          loc: {
            start: { line: 2, column: 14, token: 3 },
            end: { line: 2, column: 15, token: 4 },
            lines:{...},
```

```
            tokens: [...],
                        indent: 4
          }
        },
        loc: {
          start: { line: 2, column: 10, token: 1 },
          end: { line: 2, column: 15, token: 4 },
          lines: {...},
                    tokens: [...],
                    indent: 4
        }
      }
    ],
    kind: 'const',
    loc: {
      start: { line: 2, column: 4, token: 0 },
      end: { line: 2, column: 16, token: 5 },
      lines: {...},
            tokens: [...],
            indent: 4
    }
  },
  VariableDeclaration {
    type: 'VariableDeclaration',
    declarations: [
      VariableDeclarator {
        type: 'VariableDeclarator',
        id: Identifier {
          type: 'Identifier',
          name: 'b',
          loc: {
            start: { line: 3, column: 10, token: 6 },
            end: { line: 3, column: 11, token: 7 },
            lines: {...},
                        tokens: [...],
                        indent: 4
          }
        },
        init: BinaryExpression {
          type: 'BinaryExpression',
          operator: '+',
          left: Identifier {
            type: 'Identifier',
            name: 'a',
            loc: {
              start: { line: 3, column: 14, token: 8 },
              end: { line: 3, column: 15, token: 9 },
              lines: {...},
                            tokens: [...],
                            indent: 4
            }
          },
          right: Literal {
            type: 'Literal',
            value: 1,
            raw: '1',
            loc: {
              start: { line: 3, column: 18, token: 10 },
```

```
              end: { line: 3, column: 19, token: 11 },
              lines: {...},
                            tokens: [...]
                            indent: 4
            }
          },
          loc: {
            start: { line: 3, column: 14, token: 8 },
            end: { line: 3, column: 19, token: 11 },
            lines: {...},
                        tokens:[...],
                        indent: 4
          }
        },
        loc: {...}
      }
    ],
    kind: 'const',
    loc: {...},
    }
]
```

简化操作AST的框架gogocode

GoGoCode 是一个基于 AST 的 JavaScript/Typescript/HTML 代码转换工具，你可以用它来构建一个代码转换程序来帮你自动化完成如框架升级、代码重构、多平台转换等工作。

当前 GoGoCode 支持解析和操作如下类型的代码：

- JavaScript(JSX)
- Typescript(TSX)
- HTML
- Vue

一次代码转换的基本流程

上图概述了一次代码转换的四个流程，我们接下来的教程也会按照这四步依次进行：

1. 把代码解析成抽象语法树（AST）
2. 找到我们要改动的代码
3. 把它修改成我们想要的样子
4. 把它再生成回字符串形式的代码

4.plugin的运行原理

plugin与loader的功能不同，他并不是用于处理不同文件类型的代码存在的，他主要用于在webpack不同的生命周期阶段

处理一些复杂的操作，比如html-webpack-plugin就是在webpack处理完js和css之后将生成的文件，动态的添加到template文件的结构中

并输出新的html文件。具体的生命周期拦截方式以及创建方式我们可以参考webpack02

5.开发环境和生产环境的定义

webpack对开发和生产环境的定义很清楚。

webpack在配置对象中提供了一个mode参数

mode:production/development两个结果

production:代表生产环境，当配置为此结果时，webpack会对所有的js和html进行压缩处理，并且将构建结果输出到指定文件结构。用于发布到生产服务器

development：代表开发环境，当配置为此结果时，webpack要配合webpack-dev-server插件来使用，此时会启动本地服务器，用来调试和开发前端项目的代码，还需要配合devtools来实现编译代码和开发代码的映射来保证调试的准确性

6.webpack搭建开发和生产环境

6.1创建一个生产和开发环境分开的结构

首先在案例中创建一个`p-cli`的目录，右键使用控制台打开。在命令行中输入

```
npm init
```

一路确认，填写初始文件。

然后我们安装基本插件:

1. webpack核心包

   ```
   npm i webpack -D
   ```

2. webpack-cli包

   ```
   npm i webpack-cli -D
   ```

3. webpack-dev-server本地服务插件

   ```
   npm i webpack-dev-server -D
   ```

然后我们在项目中创建三个文件

1. webpack.base.js (他代表webpack中公用部分的配置信息)
2. webpack.dev.js （他代表开发环境的配置文件部分，dev代表development）
3. webpack.prod.js （他代表生产环境的配置文件部分，prod代表production）

接下来创建两个文件夹并创建对应两个文件

1. public
    1. index.html
2. src

1. index.js，这个文件内部可以随便用console输出点内容

6.2配置entry和output

安装html-webpack-plugin

```
npm i html-webpack-plugin -D
```

在webpack.base.js中输入如下代码

```
const path = require('path')
//html处理插件
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
    entry:{
        index:'./src/index.js'
    },
    output:{
        path:path.resolve(__dirname,'dist'),
        filename:'[name].bundle.js',
        publicPath:''//publicPath是生成的dist中的html文件中自动引入js和css文件时在最前面拼的一部分字符串
    },
    plugins:[//html处理插件
        new HtmlWebpackPlugin({
            template:'./public/index.html',//html模版文件位置
            filename:'index.html',//生成的html文件名，生成的html文件路径会整合base中配置的path生成到目标位置
      chunks:['index']//生成的index.html中自动引入的组件，这里设置的是entry中定义的key
        })
    ]
}
```

6.3配置dev环境的基本配置

我们今天将webpack分成两个环境分别对应的就是dev和prod

dev环境就是本地的开发环境，prod就是构建的生产环境

base相当于dev和prod中共同部分的内容我们单独抽取成一个根文件

所以这里涉及到一个中间件webpack-merge

```
npm i webpack-merge -D
```

安装成功之后我们将webpack.dev.js文件做如下改造

```
//webpack.dev.js
//引入webpack-merge用来合并配置到webpack.base.js中
const { merge } = require('webpack-merge');
//引入webpack.base.js
const base = require('./webpack.base.js')
const path = require('path')
```

```
//merge用法用来将配置内容合并到webpack.base.js中
//第一个参数是原始的webpack的配置json对象
//第二个参数是我们要合并的单独的配置对象
//他们最终会形成一个整体的大json
module.exports = merge(base,{
  //定义环境为开发环境
  mode:'development',
  //配置本地服务
    devServer:{
    //配置本地的静态资源文件夹，用来让这两个文件夹内部的文件可以通过访问http地址直接展示
        static:[
            path.resolve(__dirname,'dist'),//这里是构建目标路径
            path.resolve(__dirname,'public')//这里是public部分的内容
        ],
        host:'localhost',//本地服务启动后的ip地址
        port:8080,//本地服务启动的端口号
    open:true,//启动时自动打开默认浏览器
    },
})
```

然后我们在package.json中定义启动命令.

> 注意：这里是通过webpack-cli命令去启动，并不是通过webpack来启动，才能实现在构建时结合本地服务器自动构建，并且在本地可访问。通过webpack-cli的方式启动本地服务是使用webpack5+webpack-cli4版本之后的配置方式，旧版本的使用方式请自行百度学习

```
#该方式已弃用
#"serve": "webpack-cli serve --config webpack.dev.js --color --progress --hot"
```

```
"serve":"webpack serve --config webpack.dev.js --color --progress --hot"
```

配置完启动命令，并确保所有相关插件已经安装完毕我们可以尝试启动本地服务

```
npm run serve
```

运行后如果打开了浏览器并且在控制台打印出了我们在index.js中随便打印的信息就说明配置成功

6.4配置prod环境

配置prod环境我们需要改造webpack.prod.js文件的内容如下

```
//webpack.prod.js
const { merge } = require('webpack-merge')
const base = require('./webpack.base.js')
//清理dist文件夹的插件，用来在每次执行构建的时候清空上次构建的结果防止文件缓存
const {CleanWebpackPlugin} = require('clean-webpack-plugin')
module.exports = merge(base,{
  //定义环境为生产环境
    mode:'production',
    plugins:[
        new CleanWebpackPlugin()
    ]
})
```

所以这里我们需要安装一下clean-webpack-plugin

```
npm i clean-webpack-plugin -D
```

然后我们需要在package.json中添加构建指令

```
"build": "webpack --config webpack.prod.js --color --progress"
```

添加完毕后我们执行

```
npm run build
```

完毕后我们可以查看项目目录中的dist文件夹查看是否生成了index.html以及index.js

6.5配置babel

我们需要在webpack.base.js中配置babel-loader

这里配置在base中是因为无论生产环境还是开发环境都需要对js进行解析并且编译

```
//webpack.base.js
const path = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
    entry:{
        index:'./src/index.js'
    },
    output:{
        path:path.resolve(__dirname,'dist'),
        filename:'[name].bundle.js',
        publicPath:''
    },
    module:{
        rules:[
            {//配置babel-loader用来编译和解析js
                test:/\.js$/,
                use:{loader:'babel-loader'}
            }
        ]
    },
    plugins:[
        new HtmlWebpackPlugin({
            template:'./public/index.html',
            filename:'index.html',
            chunks:['index']
        })
    ]
}
```

这一步操作完毕之后我们还需要在项目中创建babel的核心配置文件

.babelrc文件

```
{
    "presets":[
        [
            "@babel/preset-env", //应用@babel/preset-env解析js
            {
                "useBuiltIns": "usage", //使用动态解析语法，根据兼容性转义
                "corejs":3//使用core-js3版本的js库来对低版本浏览器做兼容
            }
        ]
    ]
}
```

然后我们需要定义该项目可以兼容浏览器的范围

创建.browserslistrc文件（webpack5版本配置browserslist之后会造成热更新失效的问题，有待官方解决）

```
> 0.25%
last 2 versions
```

定义完成之后我们可以通过

```
npx browserslist
```

来查看当前项目可兼容浏览器的列表

完成之后我们可以在index.js中定义一些es6的map循环，Promise等来测试构建是否被转译

6.6样式的处理

这里有一点需要注意，再区分了生产和开发环境之后。我们的css样式处理也要分两种情况，在开发环境中由于启动项目是通过webpack-cli启动的本地服务器，所以css样式在开发的过程中不需要抽取到单独文件中，这里我们在dev环境中只需要使用postcss以及普通样式loader来处理就可以，在prod文件中需要对css样式文件进行单独的解析

样式的处理和js不同，样式在开发和生产环境的要求是不同的，我们在开发环境中由于启动本地服务实时编写代码并且调试，所以不需要将css抽取到外部，而是整合到js中即可，发布到生产环境时需要将css抽取到外部，所以两个处理是有差异的

所以安装成功之后我们首先处理dev文件，在webpack.dev.js中使用loader解析样式文件

```
const { merge } = require('webpack-merge');
const base = require('./webpack.base.js')
const path = require('path')
module.exports = merge(base,{
    mode:'development',
    devServer:{
        contentBase:[
            path.resolve(__dirname,'dist'),
            path.resolve(__dirname,'public')
        ],
        host:'localhost',
        port:8080,
        open:true,
        openPage:''
    },
```

```
    module:{
        rules:[
            { //用来编译css代码
                test:/\.css$/,
                use:[
                    {loader:'style-loader'},
                    {loader:'css-loader'},
                    {loader:'postcss-loader'},
                ]
            },
            { //用来编译sass代码
                test:/\.scss$/,
                use:[
                    {loader:'style-loader'},
                    {loader:'css-loader'},
                    {loader:'postcss-loader'},
                    {loader:'sass-loader'}
                ]
            }
        ]
    }
})
```

然后我们需要在项目中定义postcss的配置文件，这里需要安装

postcss-preset-env，postcss,cssnano三个插件

```
npm i postcss-preset-env postcss cssnano -D
```

然后我们定义postcss.config.js文件

```
//postcss.config.js
module.exports = {
    plugins: {
        'postcss-preset-env': {},//处理兼容性
        'cssnano':{}//压缩样式
    }
}
```

完毕之后我们通过

```
npm run dev
```

来启动项目，并且在src中创建index.scss文件，内容如下

```scss
/*index.scss*/
.test{
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    .item{
        width: 100px;
        height: 100px;
        background-color: #333;
    }
}
```

然后我们在public/index.html中书写如下代码

```html
<!--index.html-->
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title></title>
    </head>
    <body>
        <div class="test">
            我是一个测试元素
            <div class="item">

            </div>
            <div class="item">

            </div>
        </div>
    </body>
</html>
```

然后在index.js中通过import引入index.scss文件

然后我们查看首页的样式，可以右键查看控制台的查看器来看定义的样式是否增加了兼容性前缀，如果补全了前缀说明我们的postcss已经根据browserslist进行了兼容处理。

到这里dev环境的css样式处理已经完成了。

接下来我们需要处理prod环境的样式处理。在生产环境中样式不光要做兼容性处理还需要将css部分的代码抽取到css文件中，这里就需要使用mini-css-extract-plugin来实现了

```
npm i mini-css-extract-plugin -D
```

然后我们需要在webpack.prod.js中做如下改造

```
const { merge } = require('webpack-merge')
const base = require('./webpack.base.js')
const {CleanWebpackPlugin} = require('clean-webpack-plugin')
//引入抽取css样式插件
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
module.exports = merge(base,{
```

```
    mode:'production',
    module:{
        rules:[
            {
                test:/\.css$/,
                use:[
                    MiniCssExtractPlugin.loader,//抽取css样式文件
                    {loader:'css-loader'},
                    {loader:'postcss-loader'},
                ]
            },
            {
                test:/\.scss$/,
                use:[
                    MiniCssExtractPlugin.loader,//抽取css样式文件
                    {loader:'css-loader'},
                    {loader:'postcss-loader'},
                    {loader:'sass-loader'}
                ]
            }
        ]
    },
    plugins:[
        new CleanWebpackPlugin(),
    //配置样式抽取插件，生成的css文件名称为[name],[name]为entry中定义的key
        new MiniCssExtractPlugin({
            filename:'[name].css'
        })
    ]
})
```

然后我们运行

```
npm run build
```

查看生成的dist文件夹是否生成了css的样式文件。

6.7处理source-map

6.7.1什么是source-map

由于在webpack环境中进行代码开发时实际运行在浏览器中的代码是通过babel解析后的混淆过的并且处理过兼容性的js代码，这样我们在开发环境中调试开发时如果有代码报错，那么浏览器返回给我们的错误信息会与我们在src中编写的代码从行数到函数名称全部无法对应，这样就造成了调试困难。webpack提供的source-map就是相当于在构建后的代码和源代码中间做了一个简单的映射用来在出现语法错误时可以还原到源代码的结构提示错误的位置和内容。

6.7.2如何配置source-map

在生产环境中我们需要在webpack.prod.js中增加一个属性devtool

```
{
  devtool:'source-map'//独立配置源码映射
}
```

在开发环境中我们需要在webpack.dev.js中增加一个同样的devtool

```
{
  devtool:'inline-source-map'//内联配置源码映射
}
```

它们两个的区别就是生产环境会对每一个js文件生成一个.map后缀的映射文件，而开发环境映射内容会直接构建在js代码中

6.8路径解析处理

我们可以通过昨天学习的resolve属性来配置项目的引用。

由于生产和开发环境都需要通过import来引用依赖文件，所以我们直接将resolve配置到webpack.base.js中

```
//webpack.base.j追加如下代码
module.exports = {
  resolve:{
    //配置免后缀的文件类型
    extensions:['.js','.jsx','.vue','.css','.less','.scss'],
    //为全路径配置缩写@
        alias:{
            '@':path.resolve(__dirname,'src')
        }
  }
}
```

完成如下配置之后我们在src下创建css文件夹，在css文件夹内部创建test.scss文件

并且将index.scss的内容复制到里面

然后删除index.scss

之后我们将index.js中的代码引用改成

```
import '@/css/test'
```

这样我们分别运行生产环境和开发环境来验证test样式是否可以正常的加载

6.9文件处理

我们现在已经成功的进行了webpack的脚手架的基本功能的设置。不过还有一个比较重要的环节我们没有处理，这个就是当我们在脚手架中使用图片或者其他文件的时候我们需要对文件和图片进行引入。下面我们来做一个实验。

在src下创建assets文件夹，在其中随便放一个图片.

然后我们思考如果在src中引用图片并通过js动态设置到网页中，我们必须确保图片在浏览器中可以直接访问所以我们先做一个简单的小实验

在index.html中创建一个id为img的img标签

然后我们在index.js中

```
document.querySelector('#img').src = assets中的图片路径
```

查看浏览器中是否能展示图片。

运行结果我们发现在浏览器中无法直接访问src中的图片

因为我们在项目运行的时候所有的浏览器可访问到的文件夹只有dist和public

本地服务启动时本地的dist没有实体文件只有public可以存放静态资源

不过如果我们一定要使用src中的图片就需要引用一个新的loader

```
npm i file-loader -D
```

```
{ //在webpack.base.js中增加file-loader用来解析文件
  test:/\.(png|jpg|jpeg|gif)$/,
    use:[
      {loader:'file-loader'}
    ]
}
```

以上操作完成之后我们在index.js中做如下操作

```
import img from '@/assets/p1.png'
console.log(img)
document.querySelector('#img').src = img
```

我们可以输出img对象，发现他就是一串乱码名称的图片，并且我们可以直接将他拼在浏览器localhost:8080/的后面尝试是否可以访问

file-loader主要解决的问题就是将src中的文件类型的数据动态的追加到dev-server的内存中这样在本地的开发环境就可以直接的访问到图片了，并且在打包构建之后通过import引入的图片也会构建到生成的dist文件夹中。

7.个性化功能

7.1gzip压缩代码

引入compression-webpack-plugin组件来实现对vue项目的代码进一步压缩

```
npm i compression-webpack-plugin -D
```

安装成功之后我们在webpack.prod.js中引入这个插件

```
const CompressionPlugin = require("compression-webpack-plugin")
plugins:[
  new CompressionPlugin({
    algorithm: "gzip",
    test: /\.js$|\.html$|\.css$/,
    threshold: 10240,
    minRatio: 0.8
  })
]
```

完成之后我们打包构建这个项目

```
npm run build
```

查看构建的代码如果有超过10kb的代码就会被压缩成.gz为后缀的压缩包

这样将项目放到服务器上之后访问项目时如果哪个文件有gz的压缩包就会优先加载gz文件下载到本地再解压这样就能提升加载速度。

7.2public中的静态资源处理

我们在项目中有可能会引用public中的静态资源。

所以我们先启动项目

然后改造Index.vue，这里需要在对应的public文件夹中创建imgs文件夹并且放入一个名为bg1.png的图片然后在src中创建一个名为assets的文件夹放入bg.png图片

```
<template>
    <div>
        我是index.vue
        <router-link to="/test">跳转到test</router-link>
        <img style="display: block;" width="200" :src="src" alt="">
        <img width="200" height="200" src="imgs/bg1.png" alt="">
    </div>
</template>

<script>
    import img from '../assets/bg.png'
    export default{
        data(){
            return {
                src:img
            }
        }
    }
</script>

<style>
</style>
```

这里是两种引入静态资源的方式，直接在img上编写路径是通过public文件夹引入图片，通过import的方式是在src中引入图片，我们现在运行

```
npm run build
```

然后我们打开dist中的index.html文件查看内容，发现只有一张图片展示

这是因为我们在public中使用的图片地址是在html标签上直接写的webpack并不知道我们依赖了这张图片所以没有将他输出到dist中，所以这里我们需要通过copy-webpack-plugin这个插件来实现将public的内容合并到dist中这样所有的静态资源在构建之后才能正常使用

```
npm i copy-webpack-plugin -D
```

然后改造webpack.prod.js

```
const { merge } = require('webpack-merge')
const base = require('./webpack.base.js')
const {CleanWebpackPlugin} = require('clean-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CompressionPlugin = require("compression-webpack-plugin");
const CopyPlugin = require('copy-webpack-plugin');
const path = require('path')
module.exports = merge(base,{
    mode:'production',
    devtool:'source-map',
    module:{
        rules:[
            {
                test:/\.css$/,
                use:[
                    MiniCssExtractPlugin.loader,//抽取css样式文件
                    {loader:'css-loader'},
                    {loader:'postcss-loader'},
                ]
            },
            {
                test:/\.scss$/,
                use:[
                    MiniCssExtractPlugin.loader,//抽取css样式文件
                    {loader:'css-loader'},
                    {loader:'postcss-loader'},
                    {loader:'sass-loader'}
                ]
            }
        ]
    },
    plugins:[
        new CleanWebpackPlugin(),
        new MiniCssExtractPlugin({
            filename:'[name].css'
        }),
        new CompressionPlugin({
            algorithm: "gzip",
            test: /\.js$|\.html$|\.css$/,
            threshold: 10240,
            minRatio: 0.8
        }),
        new CopyPlugin({
            patterns: [
                {
                    from: path.resolve(__dirname,'public'),
                    to: path.resolve(__dirname,'dist'),
          globOptions:{
                        ignore:['**/index.html']//忽略index.html防止重名文件报错
                    }
                },
            ],
            options: {
                concurrency: 100,
            },

        })
    ]
```

```
})
```

然后运行

```
npm run build
```

我们查看结果，dist文件夹中集成了public中创建的imgs文件夹，这样静态资源就可以全部使用了。

以上就是我们模拟vue脚手架的常用功能。

7.3 依赖图谱展示

依赖图谱展示需要使用的插件为`webpack-bundle-analyzer`

需要在webpack的配置文件中加入如下代码

```
const BundleAnalyzerPlugin = require('webpack-bundle-
analyzer').BundleAnalyzerPlugin;
```

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

他的文档地址如下：https://www.npmjs.com/package/webpack-bundle-analyzer

使用后可以通过他来可视化追踪构建项目的依赖图谱和各个组件间的依赖关系

7.4公共依赖提取

在webpack.prod.js中追加如下配置，可以实现公共依赖的提取

```
{
  optimization: {
        splitChunks: {
            cacheGroups: {
                    commons: {
                            name: "commons",
                            chunks: "initial",
                            minChunks: 2
                    },
                    vendor: {
                            chunks: "all",
                            test: /[\\/]node_modules[\\/]/,
                            name: "vendor",
                            minChunks: 1,
                            maxInitialRequests: 5,
                            minSize: 0,
                            priority: 100
                    }
            }
        },
    }
}
```

关于公共依赖提取的介绍可以参考官方文档