

# Webpack详解

作者：小余同学

为什么没有01，那是因为01是讲谷歌V8引擎的原理，我看完了，但忘记写笔记了，不想重新看一遍写笔记了hh，大致来说也是很重要的一章节

需要全系列笔记请到[2002XiaoYu\(小余\).github.com](https://github.com/2002XiaoYu)中自行获取，觉得不错给个star，这是对作者非常大的鼓励

## 邂逅Webpack和打包过程

### (理解)npm发布自己的包和使用包的过程

#### 内置模块path

- path模块用于对路径和文件进行处理，提供了很多好用的方法。
- 我们知道在Mac OS、Linux和window上的路径时不一样的
  - window上会使用 \或者 \ 来作为文件路径的分隔符，当然目前也支持 /
  - 在Mac OS、Linux的Unix操作系统上使用 / 来作为文件路径的分隔符
- 那么如果我们在window上使用 \ 来作为分隔符开发了一个应用程序，要部署到Linux上面应该怎么办呢？
  - 显示路径会出现一些问题
  - 所以为了屏蔽他们之间的差异，在开发中对于路径的操作我们可以使用 path 模块
- 可移植操作系统接口（英语：Portable Operating System Interface，缩写为POSIX）
  - Linux和Mac OS都实现了POSIX接口
  - Window部分电脑实现了POSIX接口

#### path常见的API

- 从路径中获取信息

路径信息	
dirname	获取文件的父文件夹
basename	获取文件名
extname	获取文件扩展名

- 路径的拼接：path.join
  - 如果我们希望将多个路径进行拼接，但是不同的操作系统可能使用的是不同的分隔符
  - 这个时候我们可以使用path.join函数
- 拼接绝对路径：path.resolve
  - path.resolve() 方法会把一个路径或路径片段的序列解析为一个绝对路径
  - 给定的路径的序列是从右往左被处理的，后面每个 path 被依次解析，直到构造完成一个绝对路径
  - 如果在处理完所有给定path的段之后，还没有生成绝对路径，则使用当前工作目录
  - 生成的路径被规范化并删除尾部斜杠，零长度path段被忽略
  - 如果没有path传递段，path.resolve()将返回当前工作目录的绝对路径

```
const path = require("path")

const fileName = "D:/Desktop/Project/h5css3/h5.html"

//从路径中获取信息

//1. 获取文件的父文件夹
console.log(path.dirname(fileName)); //D:/Desktop/Project/h5css3
//2. 获取文件名
console.log(path.basename(fileName)); //h5.html
//3. 获取文件扩展名
console.log(path.extname(fileName)); //.html

//将多个路径拼接在一起
const path1 = "D:/Desktop/文件夹"
const path2 = "../Project/h5css3/h5.html" // 这里的../就是自动找到上一层，如果不加../就是单纯拼接在一起，想要往上几层就写几个../，而且这是会自动将 / 转义成适合MacOS和Linux的路径
```

```
console.log(path.join(path1,path2)); //D:\Desktop\Project\h5css3\h5.html

//将多个路径拼接在一起，最终一定返回一个绝对路径，path.resolve
console.log(path.resolve("老铁666",path1,path2)); //从右往左，形成绝对路径就不再往左推导，
D:\Desktop\Project\h5css3\h5.html

//自动推导到绝对路径(./则继续往前拼接，/则到头不再拼接，推导到的绝对路径是使用当前工作目录)，中间有空字符串会自动忽略
console.log(path.resolve("./-----", "./XiaoYu.ts")); //D:\Desktop\Project\h5css3\-----\XiaoYu.ts
```

## (掌握)webpack的介绍和环境搭建

### 认识webpack

- 事实上随着前端的快速发展，目前前端的开发已经变的越来越复杂了：
  - 比如开发过程中我们需要通过模块化的方式来开发
  - 比如也会使用一些高级的特性来加快我们的开发效率或者安全性，比如通过ES6+、TypeScript开发脚本逻辑，通过sass、less等方式来编写css样式代码
  - 比如开发过程中，我们还希望实时的监听文件的变化来并且反映到浏览器上，提高开发的效率
  - 比如开发完成后我们还需要将代码进行压缩、合并以及其他相关的优化
  - ....
- 但是对于很多的前端开发者来说，并不需要思考这些问题，日常的开发中根本就没有面临这些问题：
  - 这是因为目前前端开发我们通常会直接使用三大框架来开发：Vue、React、Angular
  - 但是事实上，这三大框架的创建过程我们都是借助于脚手架（CLI）的
  - 事实上Vue-CLI、create-react-app、Angular-CLI都是基于webpack来帮助我们支持模块化、less、TypeScript、打包优化等的

### 脚手架依赖webpack

- 事实上我们上面提到的所有脚手架都是依赖于webpack的：

Vue webpack的 `webpack.config.js`

React webpack的 `webpack.config.js`

Angular webpack的 `build-webpack`



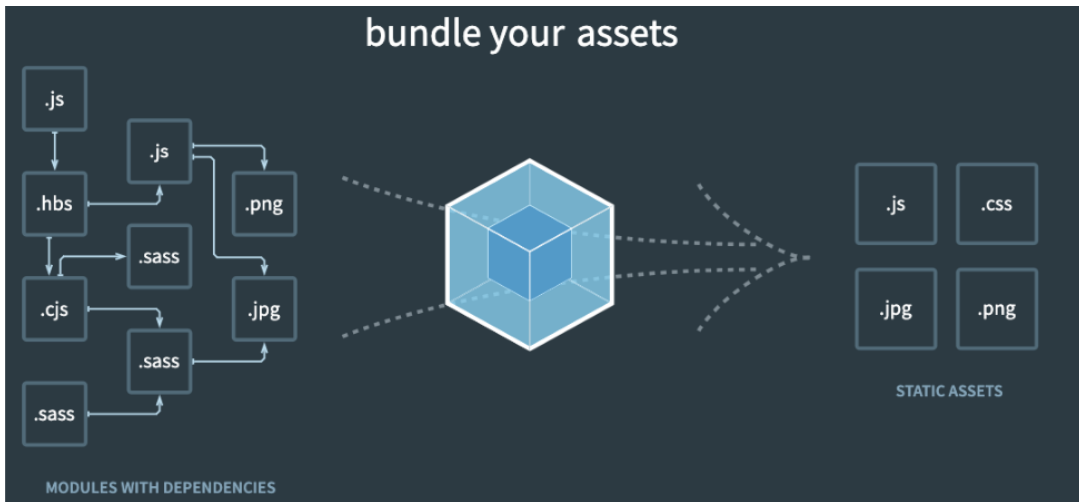
### Webpack到底是什么呢？

- 我们先来看一下官方的解释：

webpack is a static module bundler for modern JavaScript applications  
 //webpack是现代JavaScript应用程序的静态模块化打包器

- webpack是一个静态的模块化打包工具，为现代的JavaScript应用程序
- 我们来对上面的解释进行拆解：

打包bundler	webpack可以将帮助我们进行打包，所以它是一个打包工具
静态的static	这样表述的原因是我们最终可以将代码打包成最终的静态资源（部署到静态服务器）
模块化module	webpack默认支持各种模块化开发，ES Module、CommonJS、AMD等
现代的modern	我们前端说过，正是因为现代前端开发面临各种各样的问题，才催生了webpack的出现和发展

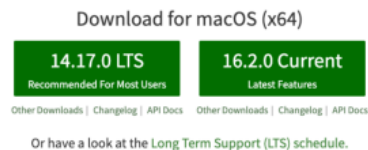


### Vue项目加载的文件有哪些呢？

- JavaScript的打包：
  - 将ES6转换成ES5的语法
  - TypeScript的处理，将其转换成JavaScript
- Css的处理：
  - CSS文件模块的加载、提取
  - Less、Sass等预处理器的处理
- 资源文件img、font：
  - 图片img文件的加载
  - 字体font文件的加载
- HTML资源的处理：
  - 打包HTML资源文件
- 处理vue项目的SFC文件.vue文件

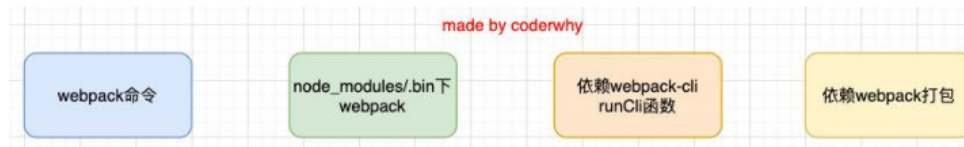
### Webpack的使用前提

- webpack的官方文档是<https://webpack.js.org/>
  - webpack的中文官方文档是<https://webpack.docschina.org/>
  - DOCUMENTATION：文档详情，也是我们最关注的
- Webpack的运行是依赖Node环境的，所以我们电脑上必须有Node环境
  - 所以我们需要先安装Node.js，并且同时会安装npm
  - Node官方网站：<https://nodejs.org/>



### Webpack的安装

- webpack的安装目前分为两个：webpack、webpack-cli
- 那么它们是什么关系呢？
  - 执行webpack命令，会执行node\_modules下的.bin目录下的webpack
  - webpack在执行时是依赖webpack-cli的，如果没有安装就会报错
  - 而webpack-cli中代码执行时，才是真正利用webpack进行编译和打包的过程
  - 所以在安装webpack时，我们需要同时安装webpack-cli（第三方的脚手架事实上是没有使用webpack-cli的，而是类似于自己的vue-service-cli的东西）



```
npm install webpack webpack-cli -g # 全局安装
npm install webpack webpack-cli -D # 局部安装
```

## (掌握)webpack基本打包-配置文件-执行脚本

### Webpack的默认打包

- 我们可以通过webpack进行打包，之后运行**打包之后**的代码
- 在目录下直接执行 webpack 命令

```
webpack
```

- **生成一个dist文件夹，里面存放一个main.js的文件，就是我们打包之后的文件：**
  - 这个文件中的代码被压缩和丑化了
  - 另外我们发现代码中依然存在ES6的语法，比如箭头函数、const等，这是因为默认情况下webpack并不清楚我们打包后的文件是否需要转成ES5之前的语法，后续我们需要通过babel来进行转换和设置
- **我们发现是可以正常进行打包的，但是有一个问题，webpack是如何确定我们的入口的呢？**
  - 事实上，当我们运行webpack时，webpack会查找当前目录下的 src/index.js作为入口
  - 所以，如果当前项目中没有存在src/index.js文件，那么会报错
- **当然，我们也可以通过配置来指定入口和出口**

```
//entry:入口
npx webpack --entry ./src/main.js --output-path ./build//我指定打包的入口为main.js跟出口为build文件夹
```

### 创建局部的webpack

- 前面我们直接执行webpack命令使用的是全局的webpack，如果希望使用局部的可以按照下面的步骤来操作
- 第一步：创建package.json文件，用于管理项目的信息、库依赖等

```
npm init//使用pnpm也行
```

- 第二步：安装局部的webpack

```
npm install webpack webpack-cli -D
```

- 第三步：使用局部的webpack

```
npx webpack
```

- 第四步：在package.json中创建scripts脚本，执行脚本打包即可

```
"scripts":{
  "build":"webpack"
}
npm run build//终端执行命令
```

不喜欢打包后的文件叫做main.js，我想叫做xiaoyu.js，能不能修改，当然可以

```
npx webpack --output-filename xiaoyu.js
```

不喜欢入口文件叫做index.js，想叫做main.js，如何修改

```
npx webpack --entry ./src/main.js//入口修改为main.js
```

## Webpack配置文件

- 在通常情况下，webpack需要打包的项目是非常复杂的，并且我们需要一系列的配置来满足要求，默认配置必然是不可以的
- 我们可以在根目录下创建一个webpack.config.js文件，来作为webpack的配置文件：

这个配置文件的名字也可以进行更改，但是更改需要修改配置，否则会报错：

```
npx webpack --config wk.config.js //设置webpack的配置文件为wk.config.js
//嫌麻烦可以将他放到package.json里的scripts里面，让npm run xxx进行启动修改配置跟启动打包
"scripts": {
  "build": "webpack --config wk.config.js"
}
```

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

//导出配置信息
module.exports = {
  entry: './src/main.js', //指定打包的入口位置
  output: {
    filename: 'bundle.js' //指定打包后的出口文件
    path: path.resolve(__dirname, './build') //指定打包后的文件夹名字，使用path是因为这里必须是绝对路径，使用resolve方法，__dirname是用来获取当前的所在路径
  }
}
```

继续执行webpack命令，依然可以正常打包

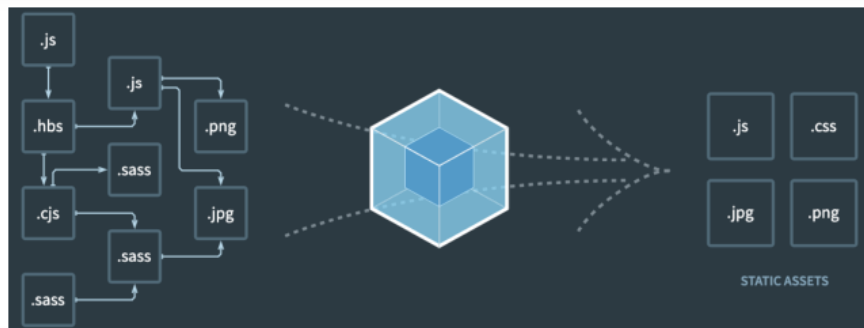
```
npm run build
```

## (理解)webpack的形成的依赖图结构

### Webpack的依赖图

- webpack到底是如何对我们的项目进行打包的呢？
  - 事实上webpack在处理应用程序时，它会根据命令或者配置文件找到入口文件
  - 从入口开始，会生成一个 依赖关系图，这个依赖关系图会包含应用程序中所需的所有模块（比如.js文件、css文件、图片、字体等）
  - 然后遍历图结构，打包一个个模块（根据文件的不同使用不同的loader来解析）

loader：webpack默认情况下是只对我们的js进行打包，如果我们想要我们文件里面也包括了css文件、图片、字体等，就需要用到loader来解析了



图结构形式的打包

## (掌握)webpack的css处理和loader的使用

### 编写案例代码

- 我们创建一个component.js
  - 通过JavaScript创建了一个元素，并且希望给它设置一些样式；

```
import "../css/stype.css";
//之所以可以不用写from，是因为我们并不使用它，而是将css导入我们的依赖图中
```

```
import "../css/style.css";

function component() {
  const element = document.createElement('div');

  element.innerHTML = ["Hello", "Webpack"].join(" ");
  element.className = "content";

  return element;
}

document.body.appendChild(component());
```

```
.content {
  color: red;
}
```

继续编译命令 npm run build

```
ERROR in ./src/css/style.css 1:0
Module parse failed: Unexpected token (1:0) 模块解析失败
You may need an appropriate loader to handle this file type, currently no
pack.js.org/concepts#loaders
> .content {
  |   color: red;
  | }
@ ./src/js/component.js 1:0-26
@ ./src/main.js 3:0-24
```

## css-loader的使用

- 上面的错误信息告诉我们需要一个loader来加载这个css文件，但是loader是什么呢？
  - loader 可以用于对模块的源代码进行转换
  - 我们可以将css文件也看成是一个模块，我们是通过import来加载这个模块的
  - 在加载这个模块时，webpack其实并不知道如何对其进行加载，我们必须制定对应的loader来完成这个功能
- 那么我们需要一个什么样的loader呢？
  - 对于加载css文件来说，我们需要一个可以读取css文件的loader
  - 这个loader最常用的是css-loader

//这些loader都是在webpack配置文件(webpack.config.js)里的module的rules  
 //可以在里面写上匹配.vue的文件就使用vue-loader去处理  
 //css, ts, tsx都是这样子处理的(这就是webpack的强大之处，不断的集成。也是loader的本质)

- css-loader的安装：

```
npm install css-loader -D
```

## css-loader的使用方案

- 如何使用这个loader来加载css文件呢？有三种方式：
  - 内联方式；
  - CLI方式（webpack5中不再使用）
  - 配置方式
- 内联方式：内联方式使用较少，因为不方便管理
  - 在引入的样式前加上使用的loader，并且使用 `!` 分割

```
import "css-loader!../css/style.css"//一般也不采用
```

- CLI方式
  - 在webpack5的文档中已经没有了--module-bind
  - 实际应用中也比较少使用，因为不方便管理(淘汰了)

## loader配置方式

- 配置方式表示的意思是在我们的webpack.config.js文件中写明配置信息：
  - module.rules中允许我们配置多个loader（因为我们会继续使用其他的loader，来完成其他文件的加载）
  - 这种方式可以更好的表示loader的配置，也方便后期的维护，同时也让你对各个Loader有一个全局的概览
- module.rules的配置如下：
- rules属性对应的值是一个数组：[Rule]
- 数组中存放的是一个一个的Rule，Rule是一个对象，对象中可以设置多个属性：
  - test属性：用于对 resource（资源）进行匹配的，通常会设置成正则表达式

- **use属性**: 对应的值是一个数组: [UseEntry]
    - UseEntry是一个对象, 可以通过对象的属性来设置一些其他属性
      1. **loader**: 必须有一个 loader属性, 对应的值是一个字符串
      2. **options**: 可选的属性, 值是一个字符串或者对象, 值会被传入到loader中
      3. **query**: 目前已经使用options来替代
- 传递字符串 (如: use: ['style-loader']) 是 loader 属性的简写方式(如: use: [{ loader: 'style-loader'}])**
- **loader属性**: Rule.use: [{ loader }] 的简写。

## Loader的配置代码

```
//webpack.config.js文件继续深入(将去掉之前写过的注释, 强调现有的注释)

const path = require('path')

//导出配置信息
module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'bundle.js'
    path: path.resolve(__dirname, './build')
  },
  module: { //模块配置:
    rules: [ //规则很多, 所以是数组类型里用对象用来存放
      // {}, {}, {} 用来存放多种类型的loader
      {
        //告诉webpack, 我们要匹配什么文件
        test: /\.css$/, //这里填入的是正则, 用来匹配我们的后缀文件
        //use中多个loader的使用顺序是从后往前的
        use: [ //告诉webpack用什么loader来处理, 为什么用数组, 是因为有时候一个loader是处理不完的
          { loader: 'css-loader' } //这时候你需要npm下载css-loader -D, 在开发的时候才会用到
        ]
      }
    ]
  }
}
```

## 认识style-loader

- 我们已经可以通过css-loader来加载css文件了
  - 但是你会发现这个css在我们的代码中并没有生效 (页面没有效果)
- 这是为什么呢?
  - 因为css-loader只是负责将.css文件进行解析, 并不会将解析之后的css插入到页面中
  - 如果我们希望再完成插入style的操作, 那么我们还需要另外一个loader, 就是style-loader
- 安装style-loader:

```
npm install style-loader -D
```

## 配置style-loader

- 那么我们应该如何使用style-loader:
  - 在配置文件中, 添加style-loader
  - 注意: 因为loader的执行顺序是从右向左 (或者说从下到上, 或者说从后到前的), 所以我们需要将style-loader写到css-loader的前面

```
use: [ //告诉webpack用什么loader来处理, 为什么用数组, 是因为有时候一个loader是处理不完的
  { loader: 'style-loader' },
  { loader: 'css-loader' } //这时候你需要npm下载css-loader -D, 在开发的时候才会用到
]
```

- 重新执行编译npm run build, 可以发现打包后的css已经生效了:
  - 当前目前我们的css是通过页内样式的方式添加进来的
  - 后续我们也会讲如何将css抽取到单独的文件中, 并且进行压缩等操作

## (掌握)webpack的less文件处理

刚刚rules里面的 `use` 简写方式

```
//简写1: 如果loader只有1个, 可以直接省略use
loader:"css-loader"
//简写2:
use:[
  {loader:"css-loader",options:{}}//之所以写对象, 这是因为这里面除了写loader, 还能写其他配置, 比如options之类
  //如果不写其他配置, 那也可以省略掉
  loader:"style-loader"
]
```

### 如何处理less文件?

- 在我们开发中, 我们可能会使用less、sass、stylus的预处理器来编写css样式, 效率会更高
- 那么, 如何可以让我们的环境支持这些预处理器呢?
  - 首先我们需要确定, less、sass等编写的css需要通过工具转换成普通的css
- 比如我们编写如下的less样式:

```
@fontSize:30px;
@fontWeight:700;

.content {
  font-size:@fontSize;
  font-weight:@fontWeight;
}
```

### Less工具处理

- 我们可以使用less工具来完成它的编译转换:

```
npm install less -D
```

- 执行如下命令:

```
npx lessc ./src/css/title.less title.css
```

### less-loader处理

- 但是在项目中我们会编写大量的css, 它们如何可以自动转换呢?
  - 这个时候我们就可以使用less-loader, 来自动使用less工具转换less到css

```
npm install less-loader -D
```

- 配置webpack.config.js

我们配置的时候除了写less之外, 我们还写了css跟style是因为, less-loader只是将less语法转化成css语法, 但是css语法还得进行处理的, 使用css的loader处理css语法, 最后使用style的loader将其引入使用

```
{
  test:/\.less$/,//处理less文件
  use:[
    {loader:"style-loader"},
    {loader:"css-loader"},
    {loader:"less-loader"},
  ]
}
```

## (理解)webpack中postcss-loader的

### 认识PostCSS工具

- 什么是PostCSS呢?
  - PostCSS是一个通过JavaScript来转换样式的工具
  - 这个工具可以帮助我们进行一些CSS的转换和适配(有些属性是由兼容问题的), 比如自动添加浏览器前缀、css样式的重置
  - 但是实现这些功能, 我们需要借助于PostCSS对应的插件(不安装的话PostCSS不会生效)



- 如何使用PostCSS呢？主要就是两个步骤：
  - 第一步：查找PostCSS在构建工具中的扩展，比如webpack中的postcss-loader
  - 第二步：选择可以添加你需要的PostCSS相关的插件

## postcss-loader

- 我们可以借助于构建工具：
  - 在webpack中使用postcss就是使用postcss-loader来处理的
- 我们来安装postcss-loader：

```
npm install postcss-loader -D
```

- 我们修改加载css的loader：（配置文件已经过多，给出一部分了）
  - 注意：因为postcss需要有对应的插件才会起效果，所以我们需要配置它的plugin；

```
{
  test: /\.css$/, //处理less文件
  use: [
    {loader: "style-loader"},
    {loader: "css-loader"},
    {loader: "postcss-loader",
      options: { //loader会读取这里面的内容，options是选项们的意思
        postcssOptions: {
          plugins: [ //插件们
            "autoprefixer" //使用这个插件对loader做出某种转换(添加前缀的插件)
          ]
        }
      }
    ]
  }
}
```

## 单独的postcss配置文件

- 因为我们需要添加前缀，所以要安装autoprefixer：

```
npm install autoprefixer -D
```

- 我们可以将这些配置信息放到一个单独的文件中进行管理：（就不会让上面配置文件里面套太多层了，看着都密密麻麻的）
  - 在根目录下创建postcss.config.js(名字不能乱改，会自动寻找到这个名字的文件，就我们前面说的从.js开始找，没找到就找.json，node文件)

```
//postcss.config.js文件
module.exports = { //暴露出去
  //不用写postcssOptions，因为这个不是独立起来的(没有独立于webpack)，postcss内部会自动寻找到
  plugins: [
    require("autoprefixer") //引入
  ]
}
```

```
{
  test: /\.css$/, //处理less文件
  use: [
    {loader: "style-loader"},
    {loader: "css-loader"},
    {loader: "postcss-loader"} //然后默认填写就行了，因为我们在另外的配置文件中已经写了plugins的东西了，会自动去读取，less如果也想要处理的话，也要加上
  ]
}
```

## postcss-preset-env

- 事实上，在配置postcss-loader时，我们配置插件并不需要使用autoprefixer
- 我们可以使用另外一个插件：postcss-preset-env
  - postcss-preset-env也是一个postcss的插件(因为有一些插件是使用postcss基本上都会使用的，所以postcss就将他集成了，叫做预设环境，已经包含了常用插件跟插件环境的配置了)
  - 它可以帮助我们一些现代的CSS特性，转成大多数浏览器认识的CSS，并且会根据目标浏览器或者运行时环境添加所需的polyfill
  - 也包括会自动帮助我们添加autoprefixer（所以相当于已经内置了autoprefixer）
- 首先，我们需要安装postcss-preset-env：

```
npm install postcss-preset-env -D
```

- 之后，我们直接修改掉之前的autoprefixer即可：

```
plugins:[  
  requires("postcss-preset-env")  
]
```

- 注意：我们在使用某些postcss插件时，也可以直接传入字符串

```
module.exports = {  
  plugins:[  
    "postcss-preset-env"  
  ]  
}
```

## Webpack打包图片-JS-Vue

### (掌握)webpack对图片资源的基本处理

#### 加载图片案例准备

- 为了演示我们项目中可以加载图片，我们需要在项目中使用图片，比较常见的使用图片的方式是两种：
  - **img元素**，设置**src属性**
  - **其他元素**（比如div），设置**background-image的css属性**

```
const zznImage = new Image();//创建一个image元素  
zznImage.src = zznImg;  
  
element.appendChild(zznImage);  
  
//3.增加一个div用来存放图片  
const bgDiv = document.createElement('div')  
bgDiv.style.width = 200 + 'px'  
bgDiv.style.height = 200 + 'px'  
bgDiv.style.display = 'inline-block'  
bgDiv.className = 'bg-image'  
bgDiv.style.backgroundColor = 'red'  
element.appendChild(bgDiv)
```

#### 认识asset module type

- 我们当前使用的webpack版本是webpack5：
  - 在webpack5之前，加载这些资源我们需要使用一些loader，比如raw-loader、url-loader、file-loader
  - 在webpack5开始，我们可以直接使用资源模块类型（asset module type），来替代上面的这些loader
- 资源模块类型(asset module type)，通过添加 4 种新的模块类型，来替换所有这些 loader：
  - **asset/resource** 发送一个单独的文件并导出 URL  
之前通过使用 file-loader 实现  
asset/resource = 资产里的资源
  - **asset/inline** 导出一个资源的 data URI  
之前通过使用 url-loader 实现
  - **asset/source** 导出资源的源代码(不常用)  
之前通过使用 raw-loader 实现
  - **asset** 在导出一个 data URI 和发送一个单独的文件之间自动选择  
之前通过使用 url-loader，并且配置资源体积限制实现

```
//引入图片模块，现在已经不再使用loader的方式来执行这些部分了，以前是使用file-loader。现在新版本使用处理还可能报错。但现在webpack已经内置如何处理文件了。但如果你直接运行还是会报错的，因为webpack将所有图片格式的文件也当作了js模块去处理了，毕竟webpack也不知道怎么处理，需要我们去告诉webpack  
imports zznImage from "./xxx/你的图片地址"  
  
//image元素  
//const imgEl = document.createElement('img')
```

## asset module type的使用

- 比如加载图片，我们可以使用下面的方式：(没错，我们要来处理图片了，要告诉webpack怎么去做)

```
{
  test: /\. (img|svg|ipg|jpe?g|gif)$/ //识别各种图片的后缀格式，让webpack找到这些文件
  type: "asset/resource" //告诉webpack这是资源类型，将找到的这些文件当作资源类型处理(资源类型是什么在上面有写出来)
}
```

- 但是，如何可以自定义文件的输出路径和文件名呢？
  - 方式一：修改output，添加assetModuleFilename属性

好处举例：这样自己制定规则就可以防止，打包后的图片文件名字跟原来的文件名字已经牛头不对马嘴，完全对不上了。你可以自己设定容易让自己区分的规则来打包。但一般很少在这里配置，我们一般采用第二种配置方法

- 方式二：在Rule中，添加一个generator属性，并且设置filename

```
output: {
  filename: "js/bundle.js",
  path: path.resolve(__dirname, "./dist"),
  assetModuleFilename: "img/[name].[hash:6][ext]"
},

{
  test: /\. (png|svg|jpg|jpeg|gif)$/i,
  type: "asset/resource",
  generator: {
    filename: "img/[name].[hash:6][ext]"
  }
},
```

- 我们这里介绍几个最常用的placeholder：
    - [ext]: 处理文件的扩展名(后缀名，比如.svg，.js，.html之类的)
    - [name]: 处理文件的名称(原来图片的名字)
    - [hash]: 文件的内容，使用MD4的散列函数处理，生成的一个128位的hash值（32个十六进制）

哈希值最好保存下来，哈希值是唯一的，防止图片不小心重复，[hash:6]就是只使用前6位哈希值(基本上也够用了)  
最前面写上 `img/`，就会自己生成一个img文件夹，然后将图片放进去

## (理解)webpack对图片资源的特殊处理

```
import "css文件的地址" //css文件引入js跟webpack形成依赖图(不需要额外配置)
//创建div元素，设置背景
const divBgEl = document.createElement("div")
divBgEl.classList.add('img-bg')
document.body.append(divBgEl)

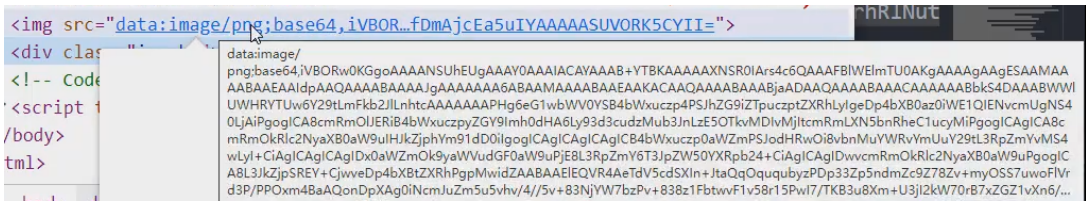
//css文件
.bg-image{
  background-image: url("../img/nhlt.jpg"); //就是填入图片
  width: 500px;
  height: 500px
}
```

进行打包，图片会被重命名，是基于哈希算法生成的哈希值命名

```
//打包图片，并且这图片有自己的地址，将地址设置到img/bgi中(打包图片命名是哈希算法生成)
type: "asset/resource"
//对图片使用了base64编码，图片编译形成的源码内容放到了行内(也就是放到了js文件)，所以打包的文件里面就看不到图片了，但js的文件就变得很大
type: "asset/inline"

//base64编码优势：少发送 当前图片数量(比如两张图片就两次网络请求) 的网络请求，因为第一种方式图片放在服务器上后，要进行http请求图片，而base64就跟着js文件一起下载下来了
//base64编码的劣势：js文件变大，下载文件本身时间过长，造成js代码的下载和解析/执行时间过长。用户长时间都只能看着空白界面等待文件的下载

//合理的规范
//1. 对于小一点图片，可以进行base64编码
//2. 对于大一点图片，单独的图片打包，形成url地址，单独的请求这个url图片
```



## url-loader的limit效果

- 开发中我们往往是小的图片需要转换，但是大的图片直接使用图片即可
  - 这是因为小的图片转换base64之后可以和页面一起被请求，减少不必要的请求过程
  - 而大的图片也进行转换，反而会影响页面的请求速度
- 我们需要两个步骤来实现：
  - 步骤一：将type修改为asset；
  - 步骤二：添加一个parser(解析)属性，并且制定dataUrl(地址数据)的条件，添加maxSize(最大的大小)属性

```
rules:[
  {
    test: /\. (png|svg|jpg|jpeg?|gif)$/ ,
    type: "asset" , //这是最好的选择，会自动选择合理的规范
    generator:{
      filename: "img/{name}.{hash:6}{ext}"
    },
    parser:{
      dataUrlCondition:{//数据地址条件
        maxSize:100 * 1024//以什么尺寸作为分界线(单位是byte字节)
      }
    }
  }
]
```

## (掌握)webpack对JS代码的babel处理

### 为什么需要babel?

- 事实上，在开发中我们很少直接去接触babel，但是babel对于前端开发来说，目前是不可缺少的一部分：
  - 开发中，我们想要使用ES6+的语法，想要使用TypeScript，开发React项目，它们都是离不开Babel的
  - 所以，学习Babel对于我们理解代码从编写到线上的转变过程至关重要
- 那么，Babel到底是什么呢？
  - Babel是一个工具链，主要用于旧浏览器或者环境中将ECMAScript 2015+代码转换为向后兼容版本的JavaScript
  - 包括：语法转换、源代码转换等

### Babel命令行使用

- babel本身可以作为一个独立的工具（和postcss一样），不和webpack等构建工具配置来单独使用
- 如果我们希望在命令行尝试使用babel(也就是独立使用，不配合webpack使用)，需要安装如下库：
  - @babel/core: babel的核心代码，必须安装
  - @babel/cli: 可以让我们在命令行使用babel
- 使用babel来处理我们的源代码：
  - src: 是源文件的目录
  - --out-dir: 指定要输出的文件夹dist

```
npx babel src --out-dir dist
```

### 插件的使用

- 比如我们需要转换箭头函数，那么我们就可以使用箭头函数转换相关的插件：

```
npm install @babel/plugin-transform-arrow-functions -D
npx babel src --out-dir dist --plugins=@babel/plugin-transform-arrow-functions
```

npx	npx 是 npm 5.2.0 版本以上提供的命令行工具，用于在命令行中运行安装在本地的 node 模块。这里使用 npx 来运行 babel 命令，避免在全局安装 Babel 的情况下产生冲突
babel	babel 是一个 JavaScript 编译器，可以将新版本的 JavaScript 代码转换为旧版本的语法，以确保代码能够在更多的浏览器或环境中运行。这里使用 babel 命令对指定目录下的 JavaScript 代码进行转换

npx	npx 是 npm 5.2.0 版本以上提供的命令行工具，用于在命令行中运行安装在本地的 node 模块。这里使用 npx 来运行 babel 命令，避免在全局安装 Babel 的情况下产生冲突
src	要转换的源代码目录
--out-dir dist	指定转换后的代码输出到 dist 目录
@babel/plugin-transform-arrow-functions	指定使用 @babel/plugin-transform-arrow-functions 插件来转换 ES6 箭头函数语法。该插件的作用是将箭头函数转换为普通的函数表达式，以确保代码能够在不支持箭头函数的环境中正常运行

- 查看转换后的结果：我们会发现 const 并没有转成 var
  - 这是因为 plugin-transform-arrow-functions，并没有提供这样的功能
  - 我们需要使用 plugin-transform-block-scoping 来完成这样的功能

```
npm install @babel/plugin-transform-block-scoping -D //转化const、let这些ES6语法
npx babel src --out-dir dist --plugins=@babel/plugin-transform-block-scoping
,@babel/plugin-transform-arrow-functions
//es6的语法很多，要转化一个个的安装会很麻烦，而且到时候写进rules也会很多，所以我们除了等等会使用预设之外，我们还要将他跟刚刚一样抽取出去，让webpack.config.js更简洁一点。
//抽取到哪？当然是我们自己创建一个babel.config.js，然后放进去啦
```

```
//babel.config.js
module.exports = {
  plugin: [
    "@babel/plugin-transform-arrow-functions"
    "@babel/plugin-transform-block-scoping"
  ]
}
```

然后就可以省略掉下面这部分

```
{
  test: /\.js$/,
  use: [
    {
      loader: "babel-loader",
      options: {
        plugins: [
          "@babel/plugin-transform-arrow-functions",
          "@babel/plugin-transform-block-scoping"
        ]
      }
    }
  ]
}
```

## Babel的预设preset

- 但是如果要转换的内容过多，一个个设置是比较麻烦的，我们可以使用预设（preset）：
  - 安装 @babel/preset-env 预设，是指安装一个 Babel 预设模块，用于根据所配置的目标环境，自动选择并使用一组适用于目标环境的 Babel 插件。该预设包含了所有与 JavaScript 新语法相关的转换插件，并根据所配置的目标环境，自动选择需要的插件进行转换
  - 后面我们再具体来写预设代表的含义
- 安装@babel/preset-env预设：

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

- 执行如下命令：

```
npx babel src --out-dir dist --presets=@babel/preset-env
```

- 安装完预设之后，我们刚刚创建的babel.config.js就需要做出一点转变了，因为预设(presets)跟配置(plugins)是分开的

```
module.exports = {
  //没有plugins了，有预设就不需要plugin了。他们是分开算的，但是预设里面已经基本包含了
  presets: [
    "@babel/preset-env"
  ]
}
```

## babel-loader

- 在实际开发中，我们通常会在构建工具中通过配置babel来对其进行使用的，比如在webpack中
- 那么我们就需要去安装相关的依赖：
  - 如果之前已经安装了@babel/core，那么这里不需要再次安装

```
npm install babel-loader -D
```

- 我们可以设置一个规则，在加载js文件时，使用我们的babel

```
module: [
  rules: [
    {
      test: /\.js$/,
      use: {
        loader: "babel-loader"
      }
    }
  ]
}
```

## babel-preset

- 如果我们一个个去安装使用插件，那么需要手动来管理大量的babel插件，我们可以直接给webpack提供一个preset，webpack会根据我们的预设来加载对应的插件列表，并且将其传递给babel
- 比如常见的预设有三个：
  - env(主要针对ES6语法)
  - react
  - TypeScript
- 安装preset-env：

```
npm install @babel/preset-env
```

## (掌握)webpack对vue文件的处理打包

### 编写App.vue代码

- 在开发中我们会编写Vue相关的代码，webpack可以对Vue代码进行解析：
  - 接下来我们编写自己的App.vue代码

```
<template>
  <h2>{{title}}</h2>
  <p>{{content}}</p>
</template>

<script>
export default {
  data() {
    return {
      title: "我是App标题",
      content: "我是App的内容，哈哈哈"
    }
  }
}
</script>

<style>
h2 {
  color: red;
}
p {
  color: blue;
}
</style>
```

## App.vue的打包过程

- 我们对代码打包会报错：我们需要合适的Loader来处理文件

```
ERROR in ./src/vue/App.vue 1:0
Module parse failed: Unexpected token (1:0)
You may need an appropriate loader to handle this file type,
see https://webpack.js.org/concepts/loaders
```

- 这个时候我们需要使用vue-loader：

```
npm install vue-loader -D
```

- 在webpack的模板规则中进行配置：

```
{
  test: /\.js$/,
  loader: "vue-loader"
}
```

## @vue/compiler-sfc

- 打包依然会报错，这是因为我们必须添加@vue/compiler-sfc来对template进行解析：

```
npm install @vue/compiler-sfc -D
```

- 另外我们需要配置对应的Vue插件：

```
const {VueLoaderPlugin} = require('vue-loader/dist/index')
new VueLoaderPlugin()
```

- 重新打包即可支持App.vue的写法
- 另外，我们也可以编写其他的.vue文件来编写自己的组件

## (掌握)webpack对文件路径的解析和配置

### resolve模块解析

- resolve用于设置模块如何被解析：
  - 在开发中我们会有各种各样的模块依赖，这些模块可能来自于自己编写的代码，也可能来自第三方库
  - resolve可以帮助webpack从每个 require/import 语句中，找到需要引入到合适的模块代码
  - webpack 使用 enhanced-resolve 来解析文件路径
- webpack能解析三种文件路径：
  - 绝对路径
    - 由于已经获得文件的绝对路径，因此不需要再做进一步解析
  - 相对路径
    - 在这种情况下，使用 import 或 require 的资源文件所处的目录，被认为是上下文目录
    - 在 import/require 中给定的相对路径，会拼接此上下文路径，来生成模块的绝对路径
  - 模块路径(比如 `import {createApp} from 'vue'` 的vue就是一个模块路径，会去node\_modules查找)
    - 在 resolve.modules中指定的所有目录检索模块
    - 默认值是 ['node\_modules']，所以默认会从node\_modules中查找文件
    - 我们可以通过设置别名的方式来替换初始模块路径，具体后面讲解alias的配置

### 确实文件还是文件夹

- 如果是一个文件：
  - 如果文件具有扩展名(js, .ts, .json这些)，则直接打包文件；
  - 否则，将使用 resolve.extensions选项作为文件扩展名解析（extensions的中文意思是扩展）
- 如果是一个文件夹：
  - 会在文件夹中根据 resolve.mainFiles配置选项中指定的文件顺序查找
  - resolve.mainFiles的默认值是 ['index']（这就是我们引入文件夹下的index能够省略index的原因，因为这里默认找index)
  - 再根据 resolve.extensions来解析扩展名

## extensions和alias配置

- extensions是解析到文件时自动添加扩展名：
  - 默认值是 ['.wasm', '.mjs', '.js', '.json']
  - 所以如果我们代码中想要添加加载 .vue 或者 jsx 或者 ts 等文件时，我们必须自己写上扩展名
  - 我们之前浅尝试了一下 `import hello from './xxx/Hello.vue'` 的后缀.vue不能去掉就是因为我们没有配置extensions，如果配置了，下次我们就不需要加上.vue这个后缀了
- 另一个非常好用的功能是配置别名alias：
  - 特别是当我们项目的目录结构比较深的时候，或者一个文件的路径可能需要 ../../../这种路径片段
  - 我们可以给某些常见的路径起一个别名

```
//webpack.config.js文件
resolve:{//resolve用来解析模块
  extensions:['.wasm','.mjs','.js','.vue','tsx'],//解析到对应文件就加上这些扩展名(这也是我们为什么可以引入的时候省略.vue的原因)
  alias:{//给路径起别名，下次写的时候就能用@替代掉路径(在小满zs的Router4视频就有使用到这种方法)
    "@":resolveApp(__dirname,'./src/utis')//__dirname是用来获取当前的所在路径
    pages:resolveApp('./src/pages')
  }
}
```

## Webpack常见的插件和模式

### (理解)webpack中plugin的作用和loader

#### 认识Plugin

- Webpack的另一个核心是Plugin，官方有这样一段对Plugin的描述：

- While loaders are used to transform certain types of modules, plugins can be leveraged to perform a wider range of tasks like bundle optimization, asset management and injection of environment variables
- 翻译：虽然加载器用于转换某些类型的模块，但可以利用插件执行更广泛的任務，如包优化、资产管理和环境变量注入

- 上面表达的含义翻译过来就是：

- Loader是用于**特定的模块类型**进行转换(但也仅此而已了，所以Loader跟plugin是很不一样的)

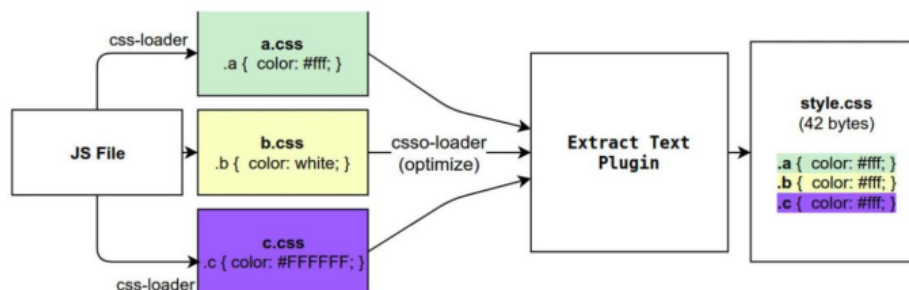
然后复习下Loader的概念

`npm install sass` 安装的是 Sass 编译器，它是一个独立的工具，用于将 Sass/Scss 样式文件编译成 CSS 文件。Sass 编译器能够将 Sass 语法或者 Scss 语法转换为 CSS 语法，并输出为 CSS 文件，方便浏览器或者其他设备渲染。在使用 Sass 编译器时，需要手动运行命令才能将 Sass/Scss 文件编译成 CSS 文件，而无法自动集成到 Webpack 构建流程中。

相比之下，Webpack 中的 `sass-loader` 是一个 Loader，可以集成到 Webpack 构建流程中，让 Webpack 能够将 Sass/Scss 样式文件编译成 CSS 文件。`sass-loader` 能够自动地将 Sass/Scss 文件编译成 CSS 文件，并将其打包进最终的 JavaScript 文件中，使得我们可以直接在浏览器中使用编译后的 CSS 文件。

在使用 Webpack 进行项目构建时，使用 `sass-loader` 能够使我们更方便地处理 Sass/Scss 样式文件，将其转换成能够直接在浏览器中使用的 CSS 文件，并能够将其集成到构建流程中，自动化地处理样式文件

- Plugin可以用于**执行更加广泛的任務(能够贯穿整个webpack的周期)**，比如打包优化、资源管理、环境变量注入等



### (掌握)webpack插件-Clean插件



## CleanWebpackPlugin

- 前面我们演示的过程中，每次修改了一些配置，重新打包时，都需要手动删除dist文件夹：
  - 我们可以借助于一个插件来帮助我们完成，这个插件就是CleanWebpackPlugin
  - CleanwebpackPlugin 是一个Webpack插件，用于在每次构建前清空输出目录。通过使用 new 关键字来创建一个 CleanwebpackPlugin 的实例，我们可以将其作为一个插件加入到Webpack构建流程中，让Webpack在每次构建之前自动清空输出目录。这样我们就能够保证输出目录中只包含最新的构建结果，而不会留下过时的文件
- 首先，我们先安装这个插件：

```
npm install clean-webpack-plugin -D
```

- 之后在插件中配置：

```
const {CleanWebpackPlugin} = require('clean-webpack-plugin')

module.exports = {
  //其他省略
  plugins:[
    //webpack的配置中，我们通常使用 new 关键字来创建一个Plugin实例
    //在webpack的配置中，Plugins通常是以实例化的形式添加到配置中的，这是因为插件实例化后可以传递一些参数，从而达到更加灵活的配置目的。同时，一个插件实例通常只能使用一次，因此需要在每个插件前使用 new 关键字来创建新的实例
    new CleanWebpackPlugin()
  ]
}
```

## (掌握)webpack插件-Html插件

### HtmlWebpackPlugin

- 另外还有一个不太规范的地方：
  - 我们的HTML文件是编写在根目录下的，而最终打包的dist文件夹中是没有index.html文件的
  - 在进行项目部署的时，必然也是需要对应的入口文件index.html
  - 所以我們也需要对index.html进行打包处理
- 对HTML进行打包处理我们可以使用另外一个插件：HtmlWebpackPlugin

```
npm install html-webpack-plugin -D
```

```
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  //其他省略
  plugins:[
    new HtmlWebpackPlugin({//使用之后，打包的时候，入口文件index.html也会被打包进去
      title:"webpack案例">//这样可以生成index.html里面的标题<title>标题</title>
      template:"路径">//生成你自己指定的html模板(也就是下面的自定义模板)
    })
  ]
}
```

### 生成index.html分析

- 我们会发现，现在自动在dist文件夹中，生成了一个index.html的文件：
  - 该文件中也自动添加了我们打包的bundle.js文件

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>webpack案例</title>
    <meta name="viewport" content="width=device-width, initial-scale=1"></head>
    <body>
      <script src="bundle.js"></script></body>
</html>
```

- 这个文件是如何生成的呢？
  - 默认情况下是根据ejs的一个模板来生成的
  - 在html-webpack-plugin的源码中，有一个default\_index.ejs模块

## 自定义HTML模板

- 如果我们想在自己的模块中加入一些比较特别的内容：
  - 比如添加一个`noscript`标签，在用户的JavaScript被关闭时，给予响应的提示
  - 比如在开发vue或者react项目时，我们需要一个可以挂载后续组件的根标签
- 这个我们需要一个属于自己的index.html模块：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="%= BASE_URL %>favicon.ico">
    <title>%= htmlWebpackPlugin.options.title %</title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work properly
      without JavaScript enabled. Please enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

## 自定义模板数据填充

- 上面的代码中，会有一些类似这样的语法`<% 变量 %>`，这个是EJS模块填充数据的方式
- 在配置HtmlWebpackPlugin时，我们可以添加如下配置：
  - `template`：指定我们要使用的模块所在的路径
  - `title`：在进行htmlWebpackPlugin.options.title读取时，就会读到该信息(代码块在上面了)

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  // 其他省略
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'webpack项目',
      template: './public/index.html'
    })
  ]
}
```

## (掌握)webpack插件-Define插件

### DefinePlugin的介绍

- 但是，这个时候编译还是会报错，因为在我们的模块中还使用到一个BASE\_URL的常量：

```
ERROR in Template execution failed: ReferenceError: BASE_URL is not defined
ERROR in ReferenceError: BASE_URL is not defined
```

- 这是因为在编译template模块时，有一个BASE\_URL：
  - `<link rel="icon" href="%= BASE_URL %>favicon.ico">`
  - 但是我们并没有设置过这个常量值，所以会出现没有定义的错误
- 这个时候我们可以使用DefinePlugin插件

### DefinePlugin的使用

- DefinePlugin允许在编译时创建配置的全局常量，是一个webpack内置的插件（不需要单独安装）：

```
const {DefinePlugin} = require('webpack');

module.exports = {
  //其他省略
  plugins:[
    new DefinePlugin({
      //作用：直接注入全局，你可以在任何一个地方使用它
      BASE_URL: "'./'"//因为是将里面的内容直接当作一个代码来执行的，所以内部还需要""将./括起来才表示字符串
      xiaoYu: "'小余全局都能看到'"
    })//因为插件都是类，所以需要new来构造使用
  ]
}
```

```
}

//随意一个文件，使用用DefinePlugin注入的变量
console.log(XiaoYu)//小余全局都能看到，不需要声明就可以直接打印，因为我们已经注入全局了

//判断是开发环境还是生产环境
console.log(process.env.NODE_DNV)
```

- 这个时候，编译template就可以正确的编译了，会读取到BASE\_URL的值；

## (理解)webpack模式-不同模式的作用

### Mode配置

- 前面我们一直没有讲mode(模式的配置)
- Mode配置选项，可以告知webpack使用相应模式的内置优化：
  - 默认值是production（什么都不设置的情况下）
  - 可选值有：'none(什么都没)' | 'development(开发模式)' | 'production(生产模式)'
- 这几个选项有什么样的区别呢？

选项	描述
development	会将 DefinePlugin 中 process.env.NODE_ENV 的值设置位 development 为模块(分包的)和chunk启用有效的名(能让你看懂的名字)
production	会将 DefinePlugin 中 process.env.NODE_ENV 的值设置位 production 。为模块和chunk启用确定性的混淆名称， FlagDependencyUsageP FlagIncludedChunksPlugin ， ModuleConcatenationPlugin ， NoEmitOnErrorsPlugin 和 TerserPlugin(代码压缩的)
none	不使用任何默认优化选项

### Mode配置代表更多

设置mode为'development'相当于设置了这些红色的部分


```
// webpack.development.config.js
module.exports = {
+ mode: 'development'
- devtool: 'eval',
- cache: true,
- performance: {
-   hints: false
- },
- output: {
-   pathinfo: true
- },
- optimization: {
-   moduleIds: 'named',
-   chunkIds: 'named',
-   mangleExports: false,
-   nodeEnv: 'development',
-   flagIncludedChunks: false,
-   occurrenceOrder: false,
-   concatenateModules: false,
-   splitChunks: {
-     hidePathInfo: false,
-     minSize: 10000,
-     maxAsyncRequests: Infinity,
-     maxInitialRequests: Infinity,
-   },
-   emitOnErrors: true,
-   checkWasmTypes: false,
-   minimize: false,
-   removeAvailableModules: false
- },
- plugins: [
-   new webpack.DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("development") })
- ]
}
```

而当mode设置了'production'的时候，也相当设置了下面这些红色部分，是非常多的选项的

```
// webpack.production.config.js
module.exports = {
  + mode: 'production',
  - performance: {
    - hints: 'warning'
  },
  - output: {
    - pathinfo: false
  },
  - optimization: {
    - moduleIds: 'deterministic',
    - chunkIds: 'deterministic',
    - mangleExports: 'deterministic',
    - nodeEnv: 'production',
    - flagIncludedChunks: true,
    - occurrenceOrder: true,
    - concatenateModules: true,
    - splitChunks: {
      - hidePathInfo: true,
      - minSize: 30000,
      - maxAsyncRequests: 5,
      - maxInitialRequests: 3,
    },
    - emitOnErrors: false,
    - checkWasmTypes: true,
    - minimize: true,
  },
  - plugins: [
    - new TerserPlugin(/* ... */),
    - new webpack.DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("production") }),
    - new webpack.optimize.ModuleConcatenationPlugin(),
    - new webpack.NoEmitOnErrorsPlugin()
  ]
}
```

## Webpack搭建本地服务器

补充打包dist前会将之前的内容先清空的一种新的方式


**webpack**

[中文文档](#)
[参与贡献](#)
[投票](#)
[博客](#)
[印记中文](#)

[API](#)
[概念](#)
[配置](#)
[指南](#)
[Loader](#)
[迁移](#)
[Plugin](#)

> 配置
 > Configuration Languages
 > Configuration Types
 > 入口和上下文
 > 模式(Mode)
 > **Output**

- assetModuleFilename
- asyncChunks
- auxiliaryComment
- charset
- chunkFilename
- chunkFormat
- chunkLoadTimeout \$#outpu...
- chunkLoadingGlobal
- chunkLoading
- clean
- compareBeforeEmit
- crossOriginLoading
- devtoolFallbackModuleFilen...
- devtoolModuleFilenameTem...
- devtoolNamespace
- enabledChunkLoadingTypes
- enabledLibraryTypes
- enabledWasmLoadingTypes
- filename
  - Template strings
- globalObject
- hashDigest
- hashDigestLength

### output.clean

5.20.0+

```
boolean { dry?: boolean, keep?: RegExp | string | ((filename: string) => boolean) }
```

```
module.exports = {
  //...
  output: {
    clean: true, // 在生成文件之前清空 output 目录
  },
};
```

```
module.exports = {
  //...
  output: {
    clean: {
      dry: true, // 打印而不是删除应该移除的静态资源
    },
  },
};
```

```
module.exports = {
  //...
  output: {
    clean: {
      keep: /ignored\/dir\/, // 保留 'ignored/dir' 下的静态资源
    },
  },
};

// 或者

module.exports = {
  //...
  output: {
    clean: {
      keep(asset) {
```

## (掌握)webpack开启本地服务器

### 为什么要搭建本地服务器？

- 目前我们开发的代码，为了运行需要有两个操作：
  - 操作一：npm run build，编译相关的代码
  - 操作二：通过live server或者直接通过浏览器，打开index.html代码，查看效果
- 这个过程经常操作会影响我们的开发效率(live server搭建的本地服务器在修改完代码需要重新打包才能够看到效果)，我们希望能够做到，当文件发生变化时，可以自动的完成 编译 和 展示
- 为了完成自动编译，webpack提供了几种可选的方式：
  - webpack watch mode(让webpack自动观察代码的变化)
  - webpack-dev-server（常用，搭建本地服务，一旦代码发生变化就会自动的做一个编译，编译完直接在浏览器做一个刷新，就能够直接看到效果）
  - webpack-dev-middleware

### webpack-dev-server

- 上面的方式可以监听到文件的变化，但是事实上它本身是没有自动刷新浏览器的功能的：
  - 当然，目前我们可以在VSCode中使用live-server来完成这样的功能
  - 但是，我们希望在不用live-server的情况下，可以具备live reloading（实时重新加载）的功能
- 安装webpack-dev-server

```
npm install webpack-dev-server -D
```

- 修改配置文件，启动时加上serve参数：

```
//在package.json文件中
"scripts": {
  "serve": "webpack serve --config webpack.config.js" //命令作用就是搭建本地服务，我们放在scripts里面，方便使用npm run
serve在终端快捷访问
}
```

webpack是怎么搭建本地服务器的：

1. 不再对dits文件做单独的打包，因为webpack搭建出来的本地服务器能够自动的对src(也就是我们写的代码)进行打包，打包后不会生成对应的本地文件的(因为生成本地文件还需要写入本地文件外加读取本地文件，效率太低)。
2. webpack打包好的东西如果不生成对应的本地文件的话，会在哪里？在内存里，是直接放到了内存里面，然后搭建的服务器直接从内存中读取，读取后放到服务器上面，然后浏览器对服务器做出请求就行了

- webpack-dev-server 在编译之后不会写入到任何输出文件，而是将 bundle 文件保留在内存中：
  - 事实上webpack-dev-server使用了一个库叫memfs（memory-fs webpack自己写的）

## (理解)webpack的热模块替换HMR

我们刚刚自动更新的方式是当我们修改了一个地方，会整个浏览器都更新了一遍，但这其实是没有必要的，会浪费性能而且保存的数据也会丢失，我们只需要更新我们修改的地方

### 认识模块热替换（HMR）

- 什么是HMR呢？
  - HMR的全称是Hot Module Replacement，翻译为模块热替换
  - 模块热替换是指在 应用程序运行过程中，替换、添加、删除模块，而无需重新刷新整个页面
- HMR通过如下几种方式，来提高开发的速度：
  - 不重新加载整个页面，这样可以保留某些应用程序的状态不丢失
  - 只更新需要变化的内容，节省开发的时间
  - 修改了css、js源代码，会立即在浏览器更新，相当于直接在浏览器的devtools中直接修改样式
- 如何使用HMR呢？
  - 默认情况下，webpack-dev-server已经支持HMR，我们只需要开启即可（默认已经开启）

## 开启HMR

- 修改webpack.config.js的配置：

```
module.exports = {  
  //其他配置忽略  
  devServer:{  
    hot:true  
  }  
}
```

- 浏览器可以看到如下效果：

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

- 但是你会发现，当我们修改了某一个模块的代码时，依然是刷新的整个页面：
  - 这是因为我们需要去指定哪些模块发生更新时，进行HMR

```
if(module.hot){//判断有没有hot  
  //指定哪一个模块需要HMR  
  module.hot.accept("./util/math.js", ()=>{//触发热更新  
    console.log("math更新了")  
  })  
}  
//此时还是会大部分刷新，那是因为你如果在math文件中引入其他文件(其中包括了入口文件，那差不多相当于全刷新了)，webpack也会认为引入相关联的文件也要刷新，那这种情况怎么解决？  
//在util文件夹下创建一个demo.js文件，里面就啥都不写，然后将demo引入math文件中。然后将上面内容修改一下，改成热更新demo部分，这样demo带动math文件的更新  
  
if(module.hot){//判断有没有hot  
  //指定哪一个模块需要HMR  
  module.hot.accept("./util/demo.js", ()=>{  
    console.log("demo更新了")  
  })  
}
```

## (了解)webpack的devServer配置信息

### 框架的HMR

- 有一个问题：在开发其他项目时，我们是否需要经常手动去写入 module.hot.accept相关的API呢？
  - 比如开发Vue、React项目，我们修改了组件，希望进行热更新，这个时候应该如何去操作呢？
- 事实上社区已经针对这些有很成熟的解决方案了：
  - 比如vue开发中，我们使用vue-loader，此loader支持vue组件的HMR，提供开箱即用的体验
  - 比如react开发中，有React Hot Loader，实时调整react组件（目前React官方已经弃用了，改成使用react-refresh）

### host配置

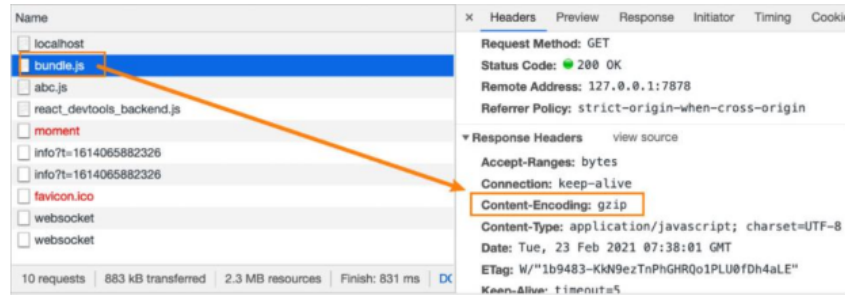
- host设置主机地址：
  - 默认值是localhost
  - 如果希望其他地方也可以访问，可以设置为 0.0.0.0

```
//在webpack.config.js中进行设置  
devServer:{  
  hot:true//就刚刚设置的开启热更新  
  port:8888//看你想要开启哪个端口，你也可以设置0.0.0.0(IP地址)  
}
```

- localhost 和 0.0.0.0 的区别：
  - localhost：本质上是一个域名，通常情况下会被解析成127.0.0.1
  - 127.0.0.1：回环地址(Loop Back Address)，表达的意思其实是我们主机自己发出去的包，直接被自己接收；  
正常的数据库包经常 应用层 - 传输层 - 网络层 - 数据链路层 - 物理层  
而回环地址，是在网络层直接就被获取到了(会被自己捕获到)，是不会经常数据链路层和物理层的  
比如我们监听 127.0.0.1时，在同一个网段下的主机中，通过ip地址是不能访问的
  - 0.0.0.0：监听IPV4上所有的地址，再根据端口找到不同的应用程序  
比如我们监听 0.0.0.0时，在同一个网段下的主机中，通过ip地址是可以访问的

## port、open、compress

- port设置监听的端口，默认情况下是8080
- open是否打开浏览器：
  - 默认值是false，设置为true会打开浏览器(就是当你运行后自己打开浏览器)
  - 也可以设置为类似于 Google Chrome等值
- compress是否为静态文件开启gzip compression：
  - 默认值是false，可以设置为true(压缩文件的)



## Proxy (Vue项目学习)

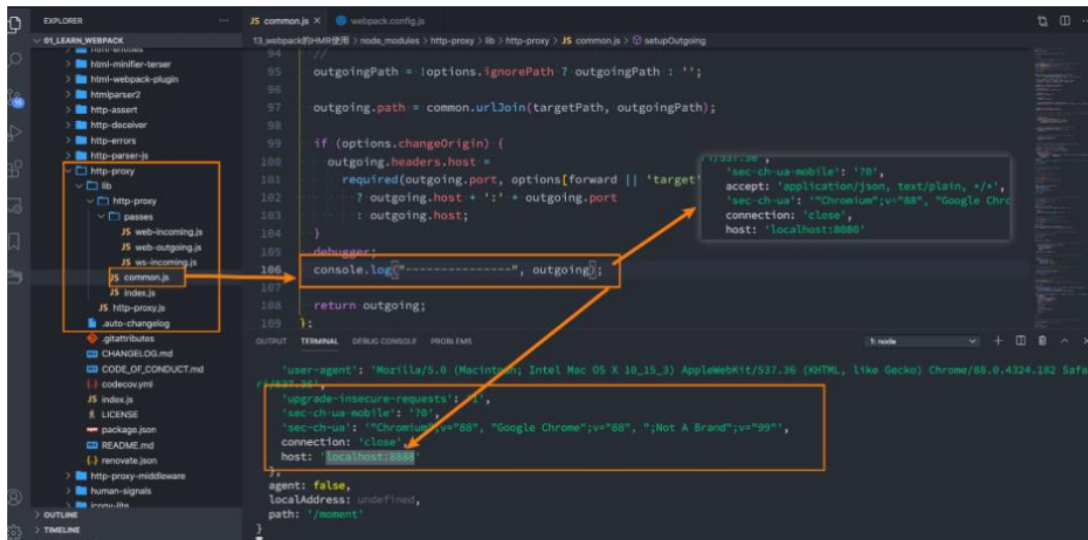
在Vue项目中再写详细的，暂时跳过

- proxy是我们开发中非常常用的一个配置选项，它的目的设置代理来解决跨域访问的问题：
  - 比如我们的一个api请求是 <http://localhost:8888>，但是本地启动服务器的域名是 <http://localhost:8000>，这个时候发送网络请求就会出现跨域的问题；
  - 那么我们可以将请求先发送到一个代理服务器，代理服务器和API服务器没有跨域的问题，就可以解决我们的跨域问题了
- 我们可以进行如下的设置：
  - target: 表示的是代理到的目标地址，比如 /api-hy/moment会被代理到 <http://localhost:8888/api-hy/moment>
  - pathRewrite: 默认情况下，我们的 /api-hy 也会被写入到URL中，如果希望删除，可以使用pathRewrite
  - secure: 默认情况下不接收转发到https的服务器上，如果希望支持，可以设置为false
  - changeOrigin: 它表示是否更新代理后请求的headers中host地址

## changeOrigin的解析 (Vue项目学习)

在Vue项目中再写详细的，暂时跳过

- 这个 changeOrigin官方说的非常模糊，通过查看源码我发现其实是要修改代理请求中的headers中的host属性：
  - 因为我们真实的请求，其实是需要通过 <http://localhost:8888>来请求的
  - 但是因为使用了代码，默认情况下它的值时 <http://localhost:8000>
  - 如果我们需要修改，那么可以将changeOrigin设置为true即可





## historyApiFallback (Vue项目学习)

- **historyApiFallback**是开发中一个非常常见的属性，它主要的作用是解决SPA页面在路由跳转之后，进行页面刷新时，返回404的错误
- **boolean值**：默认是false
  - 如果设置为true，那么在刷新时，返回404错误时，会自动返回 index.html 的内容
- **object类型的值**，可以配置rewrites属性：
  - 可以配置from来匹配路径，决定要跳转到哪一个页面
- 事实上devServer中实现historyApiFallback功能是通过connect-history-api-fallback库的：
  - 可以查看connect-history-api-fallback 文档

## 如何区分开发环境

- 目前我们所有的webpack配置信息都是放到一个配置文件中的：**webpack.config.js**
  - 当配置越来越多时，这个文件会变得越来越不容易维护
  - 并且某些配置是在开发环境需要使用的，某些配置是在生成环境需要使用的，当然某些配置是在开发和生成环境都会使用的
  - 所以，我们最好对配置进行划分，方便我们维护和管理
- 那么，在启动时如何可以区分不同的配置呢？
  - 方案一：编写两个不同的配置文件，开发和生成时，分别加载不同的配置文件即可
  - 方式二：使用相同的一个入口配置文件，通过设置参数来区分它们

```
"scripts": {
  "build": "webpack --config ./config/common.config --env production",
  "serve": "webpack serve --config ./config/common.config"
}
```

## 入口文件解析

- 我们之前编写入口文件的规则是这样的：**./src/index.js**，但是如果我们的配置文件所在的位置变成了 **config** 目录，我们是否应该变成 **../src/index.js**呢？
  - 如果我们这样编写，会发现是报错的，依然要写成 **./src/index.js**
  - 这是因为入口文件其实是和另一个属性时有关的 **context**
- **context**的作用是用于解析入口（entry point）和加载器（loader）
  - 官方说法：默认是当前路径（但是经过coderwhy测试，默认应该是webpack的启动目录）
  - 另外推荐在配置中传入一个值

```
//context是配置文件所在目录
module.exports = {
  context: path.resolve(__dirname, "./"),
  entry: "../src/index.js"
}

//context是上个目录
module.exports = {
  context: path.resolve(__dirname, "../")
  entry: "../src/index.js"
}
```

## 区分开发和生成环境配置

- 这里我们创建三个文件：
  - webpack.comm.conf.js(公共)
  - webpack.dev.conf.js(开发)
  - webpack.prod.conf.js(生产)分别针对不同的情况去准备

```
//我们使用以下命令用来合并公共部分(对配置进行合并)
npm install webpack-merge -D
//然后就可以使用CommonJS(因为webpack支持这个引入方式)来引用公共部分，引到webpack.dev.config.js跟...prod.config.js
```

coderwhy留下的题目，我在想要不要补充在这里。那就补充一点点好了

- webpack开发环境跟生产环境有什么区别？



	开发环境-development	生产环境-production
模式 (mode)	在开发模式下，Webpack会更关注速度和开发体验，如输出的代码更容易阅读和调试	在生产模式下，Webpack会更关注文件大小和性能优化，如代码压缩和代码分离等
插件 (plugins)	在生产环境下，需要使用一些插件来对代码进行优化，如UglifyJsPlugin用于压缩代码，ExtractTextPlugin用于提取CSS文件等	
输出文件 (output)	在开发环境下，Webpack通常会将所有的代码打包成一个或多个bundle.js文件	在生产环境下，Webpack通常会将不同的代码块(chunk)分开打包，以便于浏览器缓存和更好的加载性能
调试工具 (devtool)	在开发环境下，需要方便地调试代码，可以使用一些调试工具，如eval-source-map	在生产环境下，可以使用一些轻量级的调试工具，如source-map

- 总之，开发环境和生产环境在Webpack配置上的区别主要是在于性能和调试方面。开发环境更注重开发效率和调试方便，而生产环境更注重性能和文件大小优化