



第一章 webpack4前言

第一集 webpack4入门到进阶案例实战课程介绍

简介：讲述webpack4课程大纲

第二集 webpack模块化打包概念介绍

简介：详细介绍webpack，什么是模块化打包

- webpack是什么

webpack其实就是一个JavaScript应用程序的静态模块打包器。

- webpack有什么作用

模块化打包：

webpack会将项目的资源文件当成一个一个模块，模块之间会有依赖关系，webpack将会对这些有依赖关系的文件进行处理，让浏览器能够识别，最后将应用程序需要的每个模块打包成一个或者多个bundle

第三集 webpack开发环境准备及常用打包模式介绍

简介：讲解webpack开发环境的搭建，打包模式和效果

1. 安装node

- node官网地址：<https://nodejs.org/zh-cn/>

2. 创建package.json文件

- 命令：npm init

3. 安装webpack

- 本地安装：（推荐）

npm install --save-dev webpack

npm install --save-dev webpack-cli

- 全局安装：

npm install --global webpack webpack-cli

4. 打包

默认entry入口 src/index.js

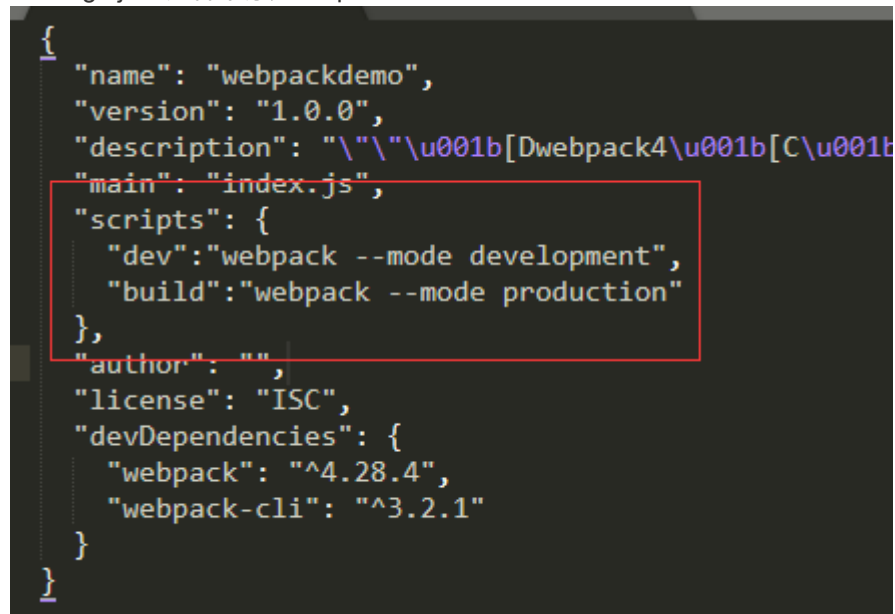
默认output出口 dist/main.js

- 打包模式：

webpack --mode development

webpack --mode production

- Package.json文件下添加scripts



```
{
  "name": "webpackdemo",
  "version": "1.0.0",
  "description": "\"\"\"\\u001b[Dwebpack4\\u001b[C\\u001b",
  "main": "index.js",
  "scripts": {
    "dev": "webpack --mode development",
    "build": "webpack --mode production"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^4.28.4",
    "webpack-cli": "^3.2.1"
  }
}
```

- 设置好后在命令程序中运行npm run dev或者npm run build来进行打包。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.17763.253]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\Watson>cd Desktop
C:\Users\Watson\Desktop>cd webpackdemo
C:\Users\Watson\Desktop\webpackdemo>npm run dev
```

第二章：webpack4配置

第一集 webpack核心配置config文件的初使用

简介：讲解webpack开发环境的搭建，打包模式和效果

1. 新建一个webpack.config.js

2. 配置入口entry(所需打包的文件路径)

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

module.exports = {
  entry: './public/index.js',
}
```

3. 配置出口output

```
module.exports = {  
  entry: './public/index.js',  
  output: {  
    path: path.resolve(__dirname, 'build'),  
    filename: "bundle.js"  
  }  
}
```

- (1) path指文件打包后的存放路径
- (2) path.resolve()方法将路径或路径片段的序列处理成绝对路径
- (3) __dirname 表示当前文件所在的目录的绝对路径
- (4) filename是打包后文件的名称

4. 命令程序运行npm run dev或者npm run build

第二集 webpack常用配置之入口entry和出口output的进阶用法

简介：讲解入口和出口的多文件用法

- 入口entry
 - 单入口
 - 单文件

例如： entry : './src/index.js'

- 多文件

在你想要多个依赖文件一起注入，并且将它们的依赖导向到一个“chunk”时，传入数组的方式就很有用。

例如：

```
entry: ['./public/index.js', './public/index2.js'],
```

- 多入口

例如：

```
entry:{
  pageOne: './public/pageOne/index.js',
  pageTwo: './public/pageTwo/index.js',
  pageThree: './public/pageThree/index.js'
},
```

- 出口output
 - 单出口

```
output:{
  path:path.resolve(__dirname,'build'),
  filename:"bundle.js"
}
```

- 多出口

```
output:{
  path:path.resolve(__dirname,'build'),
  filename:"[name].js"
}
```

第三集 开发调试必备配置之本地服务器webpack-dev-server的搭建

简介：讲解webpack如何配置本地服务器

1. 了解webpack-dev-server

webpack-dev-server是webpack官方提供的一个小型Express服务器。使用它可以为webpack打包生成的资源文件提供web服务。

webpack-dev-server 主要提供两个功能：

- (1) 为静态文件提供服务
- (2) 自动刷新和热替换(HMR)

2. 安装webpack-dev-server

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

3. 配置webpack.config.js文件

```
devServer:{
  contentBase:'./build', //设置服务器访问的基本目录
  host:'localhost', //服务器的ip地址
  port:8080, //端口
  open:true //自动打开页面
}
```

4. 配置package.json

```
"scripts": {
  "start": "webpack-dev-server --mode development"
}
```

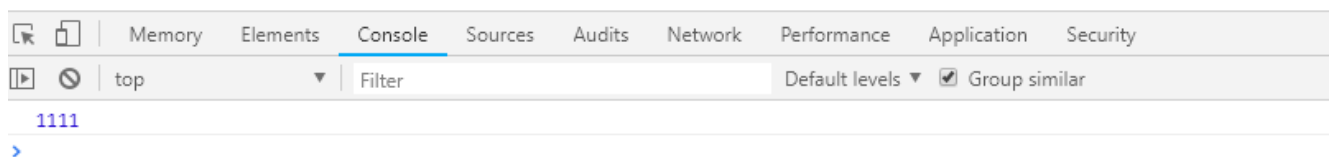
5. 在build文件夹下新建index.html文件，在html中引入bundle.js

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>

  <script src="bundle.js"></script>
</body>
</html>
```

6. 在命令行程序运行npm run start

自动弹出页面，控制台有输出信息





第三章 webpack4 loader

第一集 webpack加载css所需loader及其使用方式

简介：讲解css加载器的使用方式

- 了解loader
 - 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。
- 安装loader
 - 安装style-loader和css-loader

下载：

```
npm install style-loader css-loader --save-dev
```
- 配置loader
 - 在webpack.config.js文件里配置module中的rules

在 webpack 的配置中 loader 有两个目标：

 1. test 属性，用于标识出应该被对应的 loader 进行转换的某个或某些文件。
 2. use 属性，表示进行转换时，应该使用哪个 loader。

代码：

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

- 创建css文件并运行命令
 - 创建index.css文件并import进index.js文件中

```
import './index.css';  
  
console.log(1111);
```

第二集 webpack如何编译less和sass文件

简介：讲解webpack编译less和sass的方式

- Less
 - 处理效果

```
@width: 10px;  
@height: @width + 10px;  
  
#header {  
  width: @width;  
  height: @height;  
}
```

编译为

```
#header {  
  width: 10px;  
  height: 20px;  
}
```

- 安装loader
 - 安装less-loader和less
 - 下载：
npm install less-loader less --save-dev
- 配置loader
 - 代码：


```

module:{
  rules:[
    {
      test:/\.less$/,
      use:['style-loader','css-loader','less-loader']
    }
  ]
}

```

- Sass

- 处理效果

```

$header-color: #F90;
#header {
  $width: 100px;
  width: $width;
  color: $header-color;
}

```

编译为

```

#header {
  width: 100px;
  color: #F90;
}

```

- 安装loader

- 安装sass-loader和node-sass

下载：

npm install sass-loader node-sass --save-dev

- 配置loader

```

module:{
  rules:[
    {
      test:/\.scss$/,
      use:[{loader:'style-loader'},{loader:'css-loader'},{
        loader:'sass-loader'}]
    }
  ]
}

```

第三集 使用PostCSS自动添加css3浏览器前缀

简介：讲解PostCSS如何为css3自动添加浏览器前缀

1. 处理效果

```
#header {  
  display: flex;  
  width: 100px;  
  height: 100px;  
}
```

编译为

```
#header {  
  display: webkit-box;  
  display: webkit-flex;  
  display: moz-box;  
  display: ms-flexbox;  
  display: flex;  
  width: 100px;  
  height: 100px;  
}
```

2. 安装loader

安装postcss-loader和autoprefixer

下载：

```
npm install --save-dev postcss-loader autoprefixer
```

3. 配置loader

- 需要和autoprefixer一起用

```

module:{
  rules:[
    {
      test:/\.css$/,
      use:[
        {
          loader:'style-loader'
        },
        {
          loader:'css-loader'
        },
        {
          loader:'postcss-loader',
          options:{
            plugins:[require("autoprefixer")({
              browsers: [
                'ie >= 8',
                'Firefox >= 20',
                'Safari >= 5',
                'Android >= 4',
                'Ios >= 6',
                'last 4 version'
              ]
            })]
          }
        }
      ]
    }
  ]
}

```

◦ 浏览器设置：

▪ Loader中设置：

```

{
  browsers: [
    'ie >= 8', //ie版本大于等于ie8
    'Firefox >= 20', //火狐浏览器大于20版本
    'Safari >= 5', //safari大于5版本
    'Android >= 4', //版本大于Android4
    'Ios >= 6', //版本大于ios6
    'last 4 version' //浏览器最新的四个版本
  ]
}

```

▪ 或者在package.json里加上下图设置：

```
"browserslist": [  
  "ie >= 8",  
  "Firefox >= 20",  
  "Safari >= 5",  
  "Android >= 4",  
  "Ios >= 6",  
  "last 4 version"  
]
```

第四集 webpack文件处理 (file-loader) - 图片处理

简介：讲解使用file-loader处理图片加载

- 安装loader

下载安装file-loader

```
npm install --save-dev file-loader
```

- 配置config文件

```
module:{  
  rules:[  
    {  
      test:/\.(png|jpg|gif|jpeg)$/,  
      use:'file-loader'  
    }  
  ]  
}
```

- 选项配置

```
{  
  test: /\.(png|jpg|gif)$/,  
  use: [  
    {  
      loader: 'file-loader',  
      options: {}  
    }  
  ]  
}
```

配置options：

name：为你的文件配置自定义文件名模板（默认值[hash].[ext]）

context：配置自定义文件的上下文，默认为 webpack.config.js

publicPath：为你的文件配置自定义 public 发布目录

outputPath：为你的文件配置自定义 output 输出目录

[ext]：资源扩展名

[name]：资源的基本名称

[path]：资源相对于 context 的路径

[hash]：内容的哈希值

第五集 webpack文件处理（file-loader）- 字体文件处理

简介：讲解file-loader处理字体文件的方式

- 下载字体文件
 - 以bootstrap字体文件为例子Bootstrap字体文件下载地址：<https://v3.bootcss.com/getting-started/>
- 在index.js中引入bootstrap.css，在html中使用bootstrap字体图标
- 配置config文件

```

rules:[
  {
    test:/\.css$/,
    use:['style-loader','css-loader']
  },
  {
    test:/\. (ttf|woff|woff2|eot|svg)$/,
    use:[{
      loader:'file-loader',
      options:{
        outputPath:'font/'
      }
    }]
  }
]

```

第六集 webpack文件处理 (file-loader) - 第三方js库处理

简介：讲解第三方js库的引入方式及使用方法

以jquery库为例子

- 本地导入

编写配置文件：

webpack.ProvidePlugin参数是键值对形式，键就是我们项目中使用的变量名，值就是键所指向的库。

webpack.ProvidePlugin会先从npm安装的包中查找是否有符合的库。

如果webpack配置了resolve.alias选项（理解成“别名”），那么webpack.ProvidePlugin就会顺着设置的路径一直找下去

使用webpack.ProvidePlugin前需要引入webpack

```
const webpack = require("webpack");
```

```

resolve: {
  alias: {
    jquery: path.resolve(__dirname, "public/js/aa.js")
  }
},
plugins: [
  new webpack.ProvidePlugin({
    jquery: "jquery"
  })
]

```

- npm安装模块
 - 安装jquery库：

```
npm install jquery --save-dev
```

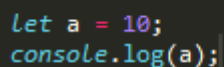
- 直接在js里import引入

```
Import $ from 'jquery'
```

第七集 使用babel-loader编译ES6语法

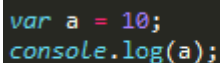
简介：讲解webpack将ES6语法编译成ES5语法的方式

- 了解babel
 - 目前，ES6（ES2015）这样的语法已经得到很大规模的应用，它具有更加简洁、功能更加强大的特点，实际项目中很可能会使用采用了ES6语法的模块，但浏览器对于ES6语法的支持并不完善。为了实现兼容，就需要使用转换工具对ES6语法转换为ES5语法，babel就是最常用的一个工具



```
let a = 10;  
console.log(a);
```

转换为



```
var a = 10;  
console.log(a);
```

- babel转化语法所需依赖项：
 - babel-loader：负责 es6 语法转化
 - babel-core：babel核心包
 - babel-preset-env：告诉babel使用哪种转码规则进行文件处理
- 安装依赖
 - npm install babel-loader @babel/core @babel/preset-env --save-dev

- 配置config文件
 - exclude表示不在指定目录查找相关文件

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

- 根目录新建 .babelrc 文件配置转换规则

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

- 另一种规则配置

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

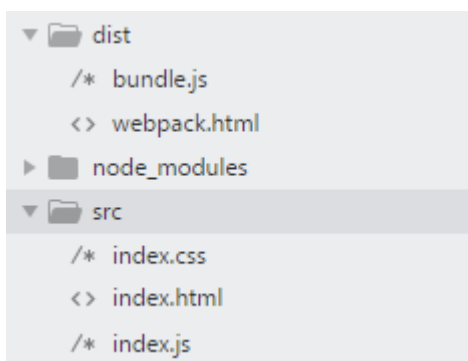


第四章 webpack4 插件

第一集 根据HTML模板自动生成HTML

简介：讲解如何使用插件根据模板自动生成html文件并关联相关文件

- 了解html-webpack-plugin
 - HtmlWebpackPlugin会自动为你生成一个HTML文件，根据指定的index.html模板生成对应的 html 文件。



根据src下的index.html自动在打包后的目录下生成html文件，相关引用关系和文件路径都会按照正确的配置被添加进生成的html里

- 安装依赖
 - npm install html-webpack-plugin --save-dev
- 配置config文件

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

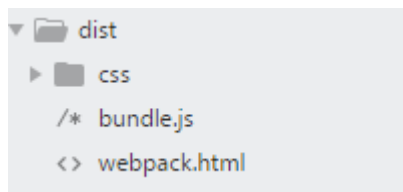
```
plugins: [  
  new HtmlWebpackPlugin({  
    template: './src/index.html', // 模板文件路径  
    filename: 'webpack.html', // 生成文件的名称  
    minify: {  
      minimize: true, // 是否打包为最小值  
      removeAttributeQuotes: true, // 去除引号  
      removeComments: true, // 去除注释  
      collapseWhitespace: true, // 去除空格  
      minifyCSS: true, // 压缩html内的样式  
      minifyJS: true, // 压缩html内的js  
      removeEmptyElements: true, // 清理内容为空的元素  
    },  
    hash: true // 引入产出资源的时候加上哈希避免缓存  
  })  
],
```

第二集 webpack提取分离css单独打包

简介：讲解如何从打包后的chunk文件中提取分离出css

- 处理效果

- 将所有的入口 chunk(entry chunks)中引用的css，移动到独立分离的 CSS 文件



- ExtractTextPlugin插件

- 安装（下载）

npm install --save-dev extract-text-webpack-plugin@next

- 配置config文件

- 引入插件：

```
const ExtractTextPlugin = require("extract-text-webpack-plugin");
```

- Rules设置：

```
module:{
  rules:[
    {
      test:/\.css$/,
      use:ExtractTextPlugin.extract({
        fallback:"style-loader",
        use:"css-loader"
      })
    }
  ],
},
```

- Plugins设置

```
plugins: [
  new ExtractTextPlugin("./css/[name].css"),//可以打包在一个文件内
]
```

- mini-css-extract-plugin插件

- 安装（下载）

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

- 配置config文件

- 引入插件

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

- Rules设置：

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

- Plugins设置

```
new MiniCssExtractPlugin({  
  filename: './css/[name].css'  
})
```

第三集 压缩css及优化css结构

简介：讲解使用插件压缩css及优化css结构

- 处理效果

```
body {  
  display: flex;  
  height: 200px;  
  border-radius: 50%;  
  background-color: red;  
}
```

处理后

```
body{display:flex;height:200px;border-radius:50%;background-color:red}
```

- optimize-css-assets-webpack-plugin插件
 - 安装（下载）

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

- 配置config文件
 - 引入插件：

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

- Plugins设置

assetNameRegExp: 正则表达式，用于匹配需要优化或者压缩的资源名。默认值是 /.css\$/g

cssProcessor: 用于压缩和优化CSS 的处理器，默认是 cssnano.

cssProcessorPluginOptions:传递给cssProcessor的插件选项，默认为{}

canPrint:表示插件能够在console中打印信息，默认值是true

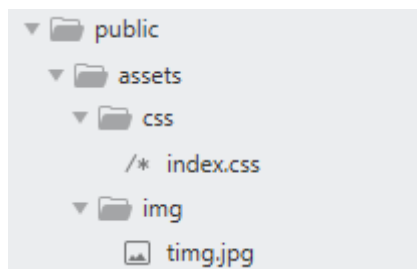
discardComments:去除注释

```
new OptimizeCSSAssetsPlugin({
  assetNameRegExp: /\.css$/g,
  cssProcessor: require("cssnano"),
  cssProcessorPluginOptions: {
    preset: ['default', {discardComments: {removeAll: true}}]
  },
  canPrint: true
})
```

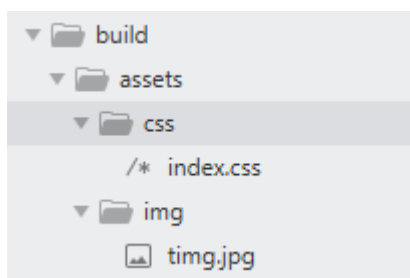
第四集 运用webpack插件拷贝静态文件

简介：讲解如何拷贝静态资源文件到打包后的目录

- 处理效果



处理后



- 安装(下载)

```
npm install --save-dev copy-webpack-plugin
```

- 配置config文件
 - 引入插件

```
const CopyWebpackPlugin = require("copy-webpack-plugin");
```

- Plugins设置

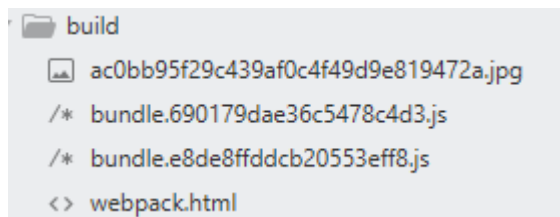
```
plugins:[  
  new CopyWebpackPlugin(  
    {  
      from: __dirname+'/public/assets',  
      to: __dirname+'/build/assets'  
    }  
  )  
]
```

第五集 webpack插件之clean-webpack-plugin清除文件

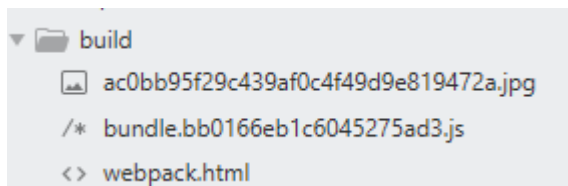
简介：讲解如何清除之前打包的旧文件

- 处理效果

当我们修改带hash的文件并进行打包时，每打包一次就会生成一个新的文件，而旧的文件并没有删除。



为了解决这种情况，我们可以使用clean-webpack-plugin在打包之前将文件先清除，之后再打包出最新的文件



- 安装

npm install --save-dev clean-webpack-plugin

- 配置config文件
 - 引入插件

```
const CleanWebpackPlugin = require("clean-webpack-plugin");
```

- Plugin配置

```
plugins:[  
  new CleanWebpackPlugin(['build']),
```

注意：clean-webpack-plugin 2.0版本需改成以下配置

```
plugins:[  
  new CleanWebpackPlugin({  
    dry:false  
  })
```

dry是否模拟删除文件，true是模拟删除，不会移除文件，false会移除文件再重新创建

插件地址：<https://www.npmjs.com/package/clean-webpack-plugin>



小D课堂 愿景："让编程不在难学，让技术与生活更加有趣" 更多教程请访问 xdclass.net

第五章 webpack4 其他配置

第一集 webpack处理HTML内嵌图片

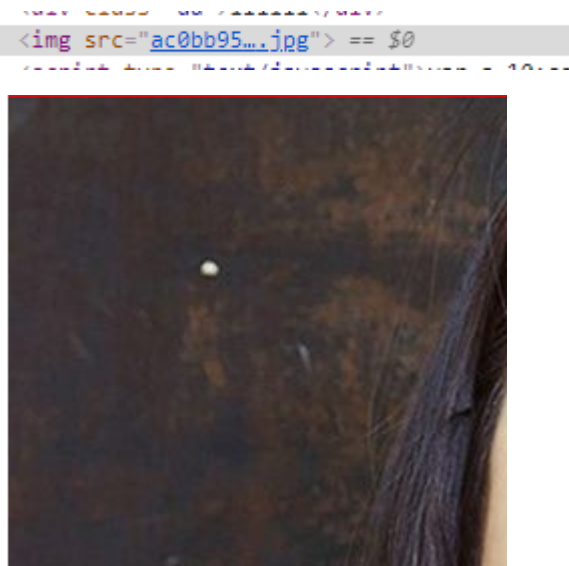
简介：讲解webpack如何处理html中引入的图片

- 处理效果 未使用loader时，会出现路径错误，图片不显示的情况



```
GET file:///C:/Users/Watson/Desktop/webpackdemo/build/assets/img/timg.jpg 0 ()
```

经过loader处理后，图片能正常显示



- 安装
cnpm install --save-dev html-loader
- 配置config文件
 - Rules中配置

```
{  
  test: /\.html$/,  
  use: {  
    loader: 'html-loader',  
    options: {  
      attrs: ['img:src', 'img:data-src']  
    }  
  }  
}
```

第二集 webpack调试必备配置之SourceMap的类型介绍及其使用方式

简介：讲解sourcemap的作用和调试方式

- 了解sourcemap

SourceMap是为了解决实际运行代码（打包后的）出现问题时无法定位到开发环境中的源代码的问题。

devtool选项

devtool	构建速度	重新构建速度	生产环境	品质(quality)
(none)	+++	+++	yes	打包后的代码
eval	+++	+++	no	生成后的代码
cheap-eval-source-map	+	++	no	转换过的代码（仅限行）
cheap-module-eval-source-map	o	++	no	原始源代码（仅限行）
eval-source-map	--	+	no	原始源代码
cheap-source-map	+	o	no	转换过的代码（仅限行）
cheap-module-source-map	o	-	no	原始源代码（仅限行）
inline-cheap-source-map	+	o	no	转换过的代码（仅限行）
inline-cheap-module-source-map	o	-	no	原始源代码（仅限行）
source-map	--	--	yes	原始源代码
inline-source-map	--	--	no	原始源代码
hidden-source-map	--	--	yes	原始源代码
nosources-source-map	--	--	yes	无源代码内容

+++ 非常快速, ++ 快速, + 比较快, o 中等, - 比较慢, -- 慢

5个基本类型：

(1)eval

每个模块都使用 eval() 执行，每一个模块后会增加sourceURL来关联模块处理前后的对应关系。如下

图


```
F (true) {\n  module.hot.accept();\n}\n\n// # sourceMappingURL=webpack:///./public/index.js?");
```

(2) source-map

```
/***/ });  
//# sourceMappingURL=bundle534fbbd22283cea69854.js.map
```

- build
 - assets
 - img
 - /* bundle534fbbd22283cea69854.js
 - bundle534fbbd22283cea69854.js.map
 - <> webpack.html

```
/**/ })

/***/ });
// # sourceMappingURL=data:application/json;charset=utf-8;base64,eyJ2ZXJzaW9uIjozLCJzb3VyY2VzIjpjbG91bnBh
```

(4) cheap

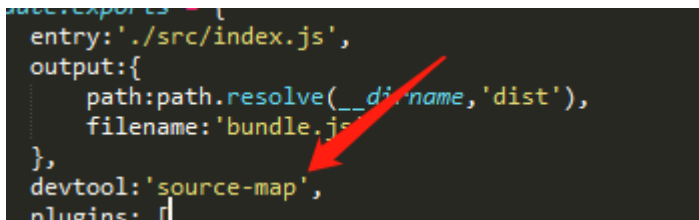
cheap属性在打包后同样会为每一个模块生成.map文件,但是与source-map的区别在于cheap生成的.map文件会忽略原始代码中的列信息,也不包含loader的sourcemap。

(5) module

包含了loader模块之间的sourcemap,将 loader source map 简化为每行一个映射。

- 使用sourcemap调试

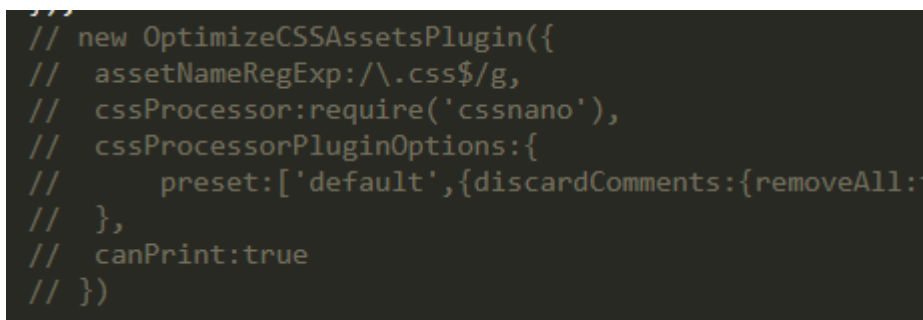
- js调试



```
module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  devtool: 'source-map',
  plugins: []
}
```

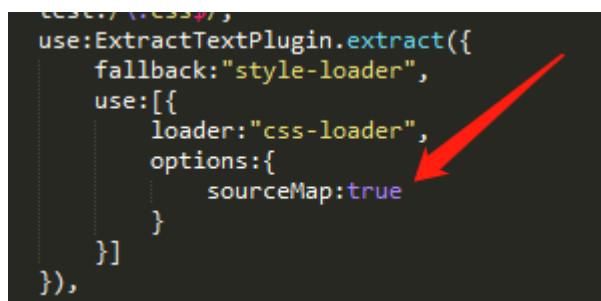
- css调试

调试css时需要将压缩css的插件注释掉



```
// new OptimizeCSSAssetsPlugin({
//   assetNameRegExp: /\.css$/g,
//   cssProcessor: require('cssnano'),
//   cssProcessorPluginOptions: {
//     preset: ['default', {discardComments: {removeAll:
//   }},
//   canPrint: true
// })
```

Css sourcemap设置



```
use: ExtractTextPlugin.extract({
  fallback: "style-loader",
  use: [{
    loader: "css-loader",
    options: {
      sourceMap: true
    }
  }],
}),
```

第三集 webpack开发调试必备功能之模块热替换HMR

简介：讲解webpack核心知识点模块热替换的作用

- 了解模块热替换

模块热替换(HMR - Hot Module Replacement)功能会在应用程序运行过程中替换、添加或删除模块，而无需重新加载整个页面。主要是通过以下几种方式，来显著加快开发速度：

- ！ 保留在完全重新加载页面时丢失的应用程序状态。
- ！ 只更新变更内容，以节省宝贵的开发时间。
- ！ 调整样式更加快速 - 几乎相当于在浏览器调试器中更改样式。

- 配置config文件

```
//  
devServer:{  
  contentBase:'./build', //设置服务器访问的基本目录  
  host:'localhost', //服务器的ip地址  
  port:8080, //端口  
  open:true, //自动打开页面  
  hot:true  
},
```

Plugin设置

```
//  
new webpack.NamedModulesPlugin(),  
new webpack.HotModuleReplacementPlugin()
```

NamedModulesPlugin：当开启 [HMR](#) 的时候使用该插件会显示模块的相对路径

- 其他配置

devServer中加入hotOnly表示只有热更新，不会自动刷新页面

```
devServer:{  
  contentBase:'./build', //设置服务器访问的基本目录  
  host:'localhost', //服务器的ip地址  
  port:8080, //端口  
  open:true, //自动打开页面  
  hot:true,  
  hotOnly:true  
},
```

修改js文件时代码不会自动热更新，需加入以下代码可以告诉 webpack 接受更新的模块

```
if (module.hot) {  
  module.hot.accept()  
}
```

第四集 区分生产环境和开发环境的配置

简介：讲解如何区分生产环境和开发环境的配置

- 简单了解

开发环境和生产环境的构建目标差异很大。在开发环境中，我们需要具有强大的、具有实时重新加载或热模块替换能力和localhost server。而在生产环境中，我们的目标则转向于关注更小的 bundle，以及资源的优化，以改善加载时间。所以我们通常建议为每个环境编写彼此独立的webpack 配置。

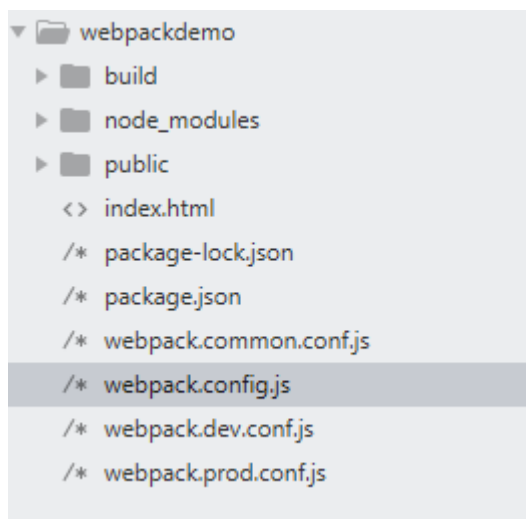
- 安装

`npm install --save-dev webpack-merge`

- 配置

- 拆分文件

- 在这里我们可以将webpack.config.js拆分为三个文件，分别是webpack.common.conf.js、webpack.dev.conf.js和webpack.prod.conf.js。



webpack.common.conf.js是放一些我们公用的配置，比如入口entry、出口output、常用loader以及插件等。

webpack.dev.conf.js是在开发环境上的配置，比如devServer配置、模块热替换等方便开发的配置

webpack.prod.conf.js是在生产环境上的配置，比如提取分离css、压缩css和js等

- webpack.common.conf.js

```

1  const path = require('path');
2  const webpack = require('webpack');
3  const HtmlWebpackPlugin = require("html-webpack-plugin");
4  const CopyWebpackPlugin = require("copy-webpack-plugin");
5  const CleanWebpackPlugin = require("clean-webpack-plugin");
6
7  module.exports = {
8    entry: './public/index.js',
9    output: {
10     }
11    module: {
12     },
13    plugins: [
14     ]
15  }

```

- Webpack.dev.conf.js

```

const merge = require('webpack-merge');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './build', //设置服务器访问的基本目录
    host: 'localhost', //服务器的ip地址
    port: 8080, //端口
    open: true, //自动打开页面
    hot: true,
    hotOnly: true
  },
  plugins: [
    new webpack.NamedModulesPlugin(),
    new webpack.HotModuleReplacementPlugin()
  ]
});

```

- Webpack.prod.conf.js

```

const path = require('path');
const webpack = require('webpack');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const OptimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');
const CopyWebpackPlugin = require('copy-webpack-plugin');
const common = require('./webpack.common.conf.js');
const merge = require('webpack-merge');

module.exports = merge(common, {
  devtool: 'source-map',
  plugins: [
    new CopyWebpackPlugin([
      {
        from: __dirname + '/public/assets',
        to: __dirname + '/build/assets'
      }
    ]),
    // 分离css
    // new ExtractTextPlugin("./css/[name].css")
    new MiniCssExtractPlugin({
      filename: "./css/[name].css"
    }),
    // 压缩css
    new OptimizeCSSAssetsPlugin({
      assetNameRegExp: /\.css$/g,
      cssProcessor: require('cssnano'),
      cssProcessorPluginOptions: {
        preset: ['default', {discardComments: {removeAll: true}}]
      },
      canPrint: true
    })
  ]
})

```

- 修改script

修改package.json文件中的script

```

"scripts": {
  "dev": "webpack --mode development --config webpack.dev.js",
  "build": "webpack --mode production --config webpack.prod.js",
  "start": "webpack-dev-server --mode development --config webpack.dev.js"
},

```

--config可以指定使用的配置文件

第五集 webpack打包优化技巧(1)

简介：介绍如何优化webpack打包速度的常用方法

第六集 webpack打包优化技巧(2)

简介：案例实战之使用resolve、happypack等方法优化打包速度

- 减少文件搜索范围
 - 优化resolve.extensions配置

在导入语句没带文件后缀时，Webpack 会自动带上后缀后去尝试询问文件是否存在。

在配置 resolve.extensions 时你需要遵守以下几点，以做到尽可能的优化构建性能：

！ 后缀尝试列表要尽可能的小，不要把项目中不可能存在的情况写到后缀尝试列表中。

！ 频率出现最高的文件后缀要优先放在最前面，以做到尽快的退出寻找过程。

！ 在源码中写导入语句时，要尽可能的带上后缀，从而可以避免寻找过程。例如在你确定的情况下把 require('./data') 写成 require('./data.json')。

```
resolve:{  
  extensions:['.js'],
```

- 优化 resolve.modules配置

resolve.modules 用于配置 Webpack 去哪些目录下寻找第三方模块。

resolve.modules 的默认值是 ['node_modules']，会采用向上递归搜索的方式查找

```
function resolve (dir) {  
  return path.join(__dirname, '..', dir)  
}
```

```
resolve:{  
  extensions:['.js'],  
  modules:[  
    resolve('public'),  
    resolve('node_modules')  
  ],
```

- 优化resolve.alias配置

resolve.alias配置项通过别名来把原导入路径映射成一个新的导入路径。

```
// 使用 alias 把导入 react 的语句换成直接使用单独完整的 react.min.js 文件,
// 减少耗时的递归解析操作
alias: {
  'react': resolve('./node_modules/react/dist/react.min.js'),
  'assets': resolve('./public/assets')
}
```

- 缩小文件匹配范围

Include：需要处理的文件的位置

Exclude：排除掉不需要处理的文件的位置

```
{
  test: /\.js$/,
  include: [resolve('src')],
  exclude: /node_modules/,
  use: {
    loader: 'babel-loader',
    options: {
      presets: ["@babel/preset-env"]
    }
  }
},
```

- 设置noParse

防止 webpack 解析那些任何与给定正则表达式相匹配的文件。忽略的文件中不应该含有import, require, define 的调用, 或任何其他导入机制。忽略大型的 library 可以提高构建性能。比如jquery、elementUI等库

```
module: {
  noParse: /node_modules\/(element-ui\.js)/,
  rules: [
    {

```

- 给babel-loader设置缓存

babel-loader提供了 cacheDirectory特定选项（默认 false）：设置时，给定的目录将用于缓存加载器的结果。


```
{
  test: /\.js$/,
  include: [resolve('src')],
  exclude: /node_modules/,
  use: {
    loader: 'babel-loader?cacheDirectory=true',
    options: {
      presets: ["@babel/preset-env"]
    }
  }
},
```

- 使用happyPack

HappyPack的基本原理：在webpack构建过程中，我们需要使用Loader对js，css，图片，字体等文件做转换操作，并且转换的文件数据量也是非常大的，且这些转换操作不能并发处理文件，而是需要一个个文件进行处理，HappyPack的基本原理是将

这部分任务分解到多个子进程中去并行处理，子进程处理完成后把结果发送到主进程中，从而减少总的构建时间。

(1) 安装

```
cnpm install happypack --save-dev
```

(2) 配置webpack.common.conf.js文件

引入happypack

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

Rules设置

```
{
  test: /\.js$/,
  exclude: /node_modules/,
  loader: 'happypack/loader?id=happyBabel',
},
```

Plugins设置

```
new HappyPack({
  //用id来标识 happypack处理那里类文件
  id: 'happyBabel',
  //如何处理 用法和loader 的配置一样
  loaders: [{
    loader: 'babel-loader?cacheDirectory=true',
  }]
})
```

(3) npm run build打包



小D课堂 愿景：“让编程不在难学，让技术与生活更加有趣” 更多教程请访问 xdclass.net

第六章 webpack4 课程总结

第一集 课程核心知识回顾和总结

简介：回顾总结课程核心知识点

小D课堂，愿景：让编程不在难学，让技术与生活更加有趣

相信我们，这个是可以让你学习更加轻松的平台，里面的课程绝对会让你技术不断提升

欢迎加小D讲师的微信：jack794666918

我们官方网站：<https://xdclass.net>

千人IT技术交流QQ群：718617859

重点来啦：免费赠送你干货文档大集合，包含前端，后端，测试，大数据，运维主流技术文档（持续更新）

<https://mp.weixin.qq.com/s/qYnjcDYGFDQorWmSfE7lpQ>