

从基础到实战 手把手带你掌握新版Webpack4.0

2019年10月8日 星期二 8:02

✓ • 第1章 课程导学 (打消你的学习疑虑)

掌握Webpack越来越成为前端工程师的标配技能，本章会对课程整体进行介绍，打消你的学习疑虑。

✓ • 1-1 课程导学 [试看](#)

✓ • 第2章 Webpack 初探

本章通过清晰易懂的例子，带你了解 Webpack 的产生背景，以及其能够解决的问题。在此基础上，完成开发环境的搭建，Webpack 的安装，并进行最基础的 Webpack 使用讲解。

✓ • 2-1 webpack 究竟是什么？

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

例子：

header.js 采用ES module 的语法 ,并且export default Header;

header.js

```
function Header() {  
  var dom = document.getElementById('root');  
  var header =document.createElement('div');  
  header.innerText = 'header';  
  dom.append(header);  
}  
export default Header;
```

index.js

```
// ES Moudule 模块引入方式  
// Webpack 模块打包工具  
import Header from './header.js';  
import Sidebar from './sidebar.js';  
import Content from './content.js';  
  
new Header();  
new Sidebar();  
new Content();
```

npm webpack index.js 会在dist 目录中生成一个main.js文件

✓ • 2-2 什么是模块打包工具？

Bundler : 模块打包工具

commonJs语法规则的打包测试

官网: <https://webpack.js.org/concepts/modules/>

← → ↻ https://webpack.js.org/concepts/modules/

Sponsor webpack and get apparel from the official shop! All proceeds go to our open collective!

webpack

DOCUMENTATION | CONTRIBUTE | VOTE | BLOG | | | | |

GUIDES | API | **CONCEPTS** | CONFIGURATION | PLUGINS | MIGRATE | LOADERS

ag-grid is proud to partner with webpack

- Output
- Loaders
- Plugins
- Configuration
- Modules**
 - What is a webpack Module**
 - Supported Module Types
- Module Resolution
- Dependency Graph
- Targets
- The Manifest
- Hot Module Replacement
- Why webpack

What is a webpack Module

In contrast to `Node.js` modules, webpack modules can express their dependencies in a variety of ways. A few examples are:

- An `ES2015` `import` statement
- A `CommonJS` `require()` statement
- An `AMD` `define` and `require` statement
- An `import` statement inside of a `css/sass/less` file.
- An image url in a stylesheet (`url(...)`) or html (``) file.

webpack 1 requires a specific loader to convert `ES2015` `import`; however, this is possible out of the box via webpack 2

Supported Module Types

webpack supports modules written in a variety of languages and preprocessors, via loaders. Loaders describe to webpack **how** to process non-JavaScript modules and include these dependencies into your bundles. The webpack community has built loaders for a wide variety of popular languages and language processors, including:

- CoffeeScript
- TypeScript
- ESNext (Babel)
- Sass
- Less
- Stylus
- Elm

webpack

DOCUMENTATION | CONTRIBUTE | VOTE | BLOG | | | | |

GUIDES | **API** | **CONCEPTS** | CONFIGURATION | PLUGINS | MIGRATE | LOADERS

- Node Interface
- Stats Data
- Hot Module Replacement
- Loader Interface
- Logger Interface

MODULES

- Module Methods**
 - ES6 (Recommended)
 - import
 - export
 - import()
 - Dynamic expressions in imp...
 - Magic Comments
 - CommonJS
 - require
 - require.resolve
 - require.cache
 - require.ensure
 - AMD
 - define
 - define
 - require
 - Labeled Modules
 - export
 - require
 - Webpack
 - require.context
 - require.include
 - require.resolveWeak
- Module Variables**

import

Statically import the exports of another module.

```
import MyModule from './my-module.js';
import { NamedExport } from './other-module.js';
```

The keyword here is **statically**. A normal `import` statement cannot be used dynamically within other logic or contain variables. See the [spec](#) for more information and `import()` below for dynamic usage.

export

Export anything as a default or named export.

```
// Named exports
export var Count = 5;
export function Multiply(a, b) {
  return a * b;
}

// Default export
export default {
  // Some data...
};
```

import()

function(string path):Promise

- ✓ 2-3 Webpack的正确安装方式
- 环境: nodejs
- npm -v 查看版本号
- Npm info webpack 查看有哪些版本号
- npm配置: `"private":true`, 表示私有项目

webpack 全局安装弊端：不同项目可能版本不同，带来麻烦，建议项目中单独安装 `npm install webpack -D`
webpack -v 全局查看版本， `npx webpack -v` 项目中查看局部安装的版本

- ☑️ • 2-4 使用Webpack的配置文件
package.js

```
"scripts": {  
  "bundle": "webpack"  
},
```

webpack.config.js

```
const path = require('path');  
module.exports = {  
  mode: "development", // development 开发模式，未被压缩， production 线上模式，被压缩了  
  
  // entry : "./src/index.js", //入口文件，放在了src目录下  
  entry : {  
    main: "./src/index.js"  
  },  
  output : {  
    filename : 'bundle.js', //输出的文件名  
    path : path.resolve(__dirname, 'dist') //配置路径  
  }  
}
```

npm run bundle

运行结果

```
> lesson2@1.0.0 bundle D:\workspace\webpack4Study\lesson2-4  
> webpack
```

```
Hash: 8739439f0b61fd188665  
Version: webpack 4.26.0  
Time: 157ms  
Built at: 2019-10-11 15:16:13  
   Asset      Size  Chunks             Chunk Names  
bundle.js  1.36 KiB       0 [emitted]  main  
Entrypoint main = bundle.js  
[0] ./src/index.js 373 bytes {0} [built]  
[1] ./src/header.js 205 bytes {0} [built]  
[2] ./src/sidebar.js 212 bytes {0} [built]  
[3] ./src/content.js 212 bytes {0} [built]
```

WARNING in configuration ('mode' option has not been set 未设置mode)

The 'mode' option has not been set, webpack will fallback to 'production' for this value. Set 'mode' option to 'development' or 'production' to enable defaults for each environment.

You can also set it to 'none' to disable any default behavior. Learn more: <https://webpack.js.org/concepts/mode/>

- ☑️ • 2-5 浅析 Webpack 打包输出内容

- ☑️ • 第3章 Webpack 的核心概念

本章讲解 Webpack 中的一些核心概念，从 Loader 与 Plugin 开始，带你学明白 Webpack 的运行机制，然后逐步深入，扩展衍生出 SourceMap，HMR，WDS 等常见概念。本章课程学习过程中，额外增加了对 Webpack 官方文档的查阅方式讲解，帮助大家学会查阅文档。...

- ☑️ • 3-1 什么是 loader 试看
- ☑️ • 3-2 使用 Loader 打包静态资源（图片篇）

npm install file-loader -D （用于打包文件，并将文件移动到dist目录中）

官网参考文档： <https://webpack.js.org/loaders/file-loader/>

例子: lesson3-1b

```
const path = require('path');
module.exports = {
  mode: "development", // development 开发模式, 未被压缩, production 线上模式, 被压缩了
  // entry : "./src/index.js", //入口文件, 放在了src目录下
  entry : {
    main: "./src/index.js"
  },
  module: {
    rules: [{
      test: /\.jpg$/, //正则, 以.jpg结尾的文件
      use: {
        loader: 'file-loader'
      }
    }]
  },
  output : {
    filename : 'bundle.js', //输出的文件名
    path : path.resolve(__dirname, 'dist') //配置路径
  }
}
```

- 3-3 使用 Loader 打包静态资源（样式篇 - 上）
 1. file-loader方式 配置 文件占位符, 配置jpg输出路径至images

```
module: {
  rules: [{
    test: /\.jpg$/, //正则, 以.jpg结尾的文件
    use: {
      loader: 'file-loader',
      options : {
        // placeholder 占位符
        // name: '[name].[ext]', //name 指原来的文件名, ext 文件后缀
        name: '[name]_[hash].[ext]', //name 指原来的文件名, ext 文件后缀
        outputPath : 'images/' // 将图片打包至images目录下
      }
    }
  }]
},
```

2. url-loader 把图片打包成base64的字符串, 设置limit

```
use: {
  // loader: 'file-loader',
  loader: 'url-loader', //把图片打包成base64的字符串
  options : {
    // placeholder 占位符
    // name: '[name].[ext]', //name 指原来的文件名, ext 文件后缀
    name: '[name]_[hash].[ext]', //name 指原来的文件名, ext 文件后缀
    outputPath : 'images/' , // 将图片打包至images目录下
    limit : 2048 // >2kb打包成文件, 否则打包成base64的字符串
  }
}
```

3. 参考官网: file-loader url-loader

- 3-4 使用 Loader 打包静态资源（样式篇 - 下）

1. Css样式文件

- 样式文件, 安装两个loader

npm install style-loader css-loader -D

css-loader 分析几个css之间的关系

style-load 把css-loader生成的内容挂载到html页面中的header部分

```
{
```

```

    test: /\.css$/, //正则, 以 .jpg结尾的文件
    use: ['style-loader', 'css-loader']
  }
}

```

- importLoaders 配置, 见第4条

2. Scss样式文件

npm install sass-loader node-sass -D

```

{
  test: /\.scss$/, //正则, 以 .jpg结尾的文件
  use: ['style-loader', 'css-loader', 'sass-loader'] //有执行顺序, 从下到上, 从右到左
}

```

3. 自动添加厂商前缀 postcss-loader 参考官网: <https://webpack.js.org/loaders/postcss-loader/>

- 安装postcss-loader npm i postcss-loader -D

```

{
  test: /\.scss$/, //正则, 以 .jpg结尾的文件
  use: [
    'style-loader',
    'css-loader',
    'sass-loader',
    'postcss-loader'
  ] //有执行顺序, 从下到上, 从右到左
}

```

- 建立postcss.config.js文件 npm i autoprefixer -D

```

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

```

4. Css-loader的importLoaders配置

index.scss 文件中又引入了其它的scss文件, 这种情况下使用importLoaders配置

```

{
  test: /\.scss$/, //正则, 以 .jpg结尾的文件
  use: [
    'style-loader',
    {
      loader: 'css-loader',
      options: {
        importLoaders: 2 // 例如scss文件, 嵌套引入scss文件, 配置importLoaders以后, 会再走postcss-loader和
        sass-loader这两个loader
      }
    },
    'sass-loader',
    'postcss-loader'
  ] //有执行顺序, 从下到上, 从右到左
}

```

5. css打包的模块化 lesson3-4c p4 00:39:43

- 样式冲突问题

index.js 中代码

```

import avatar from './avatar.jpg';
import './index.scss'; // 会影响createAvatar中的样式 (全局)
import createAvatar from './createAvatar';
createAvatar();
var img = new Image();
img.src = avatar;
img.classList.add('avatar');//avatar是class名称
var root = document.getElementById("root");
root.append(img);

```

- 解决方法:

Webpack.config.js

```

    {
      loader: 'css-loader',
      options: {
        importLoaders: 2, // 例如scss文件, 嵌套引入scss文件, 配置importLoaders以后, 会再走postcss-loader和
        sass-loader这两个loader
        modules: true // 开启css模块化打包
      }
    },
  ],
}
```

Index.js

```

import avatar from './avatar.jpg';
// import './index.scss'; // 会影响createAvatar中的样式 (全局)
import style from './index.scss'; // css模块化方式, 与其它文件里的样式不会有耦合或冲突
import createAvatar from './createAvatar';
createAvatar();
var img = new Image();
img.src = avatar;
// img.classList.add('avatar');//avatar是class名称
img.classList.add(style.avatar);//avatar是class名称
var root = document.getElementById("root");
root.append(img);
```

6. 打包字体文件 lesson3-4d

Iconfont 添加图标至我的项目中, 并下载图标至本地。

Webpack.config.js 配置

```

module: {
  rules: [
    {
      test: /\. (jpg|png|gif)$/ , //正则, 以. jpg结尾的文件
      use: {
        // loader: 'file-loader',
        loader: 'url-loader', //把图片打包成base64的字符串
        options: {
          // placeholder 占位符
          // name: '[name].[ext]', //name 指原来的文件名, ext 文件后缀
          name: '[name]_[hash].[ext]', //name 指原来的文件名, ext 文件后缀
          outputPath: 'images/' , // 将图片打包至images目录下
          limit: 2048 // >2kb打包成文件, 否则打包成base64的字符串
        }
      }
    },
    {
      test: /\.scss$/ , //正则, 以. jpg结尾的文件
      use: [
        'style-loader',
        {
          loader: 'css-loader',
          options: {
            importLoaders: 2, // 例如scss文件, 嵌套引入scss文件, 配置importLoaders以后, 会再走postcss-loader和
            sass-loader这两个loader
            // modules: true // 开启css模块化打包
          }
        },
        'sass-loader',
        'postcss-loader'
      ] //有执行顺序, 从下到上, 从右到左
    },
    {
      test: /\. (eot|ttf|svg|woff2|woff)$/ , //正则, 以. jpg结尾的文件
      use: {
```

```

        loader: 'file-loader',
      }
    }
  ],
},

```

其实还是利用file-loader将字体文件copy至dist目录中

7. csv、excel等文件的打包方式 官网：<https://webpack.js.org/guides/asset-management/#loading-data>

☑️ 3-5 使用 plugins 让打包更便捷

1. Html-webpack-plugin

- npm install --save-dev html-webpack-plugin

- webpack.config.js 文件中

```

const HtmlWebpackPlugin = require('html-webpack-plugin');
.....
plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }) //htmlWebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
],

```

2. clean-webpack-plugin 参考文档：<https://www.npmjs.com/package/clean-webpack-plugin>

- npm i -D clean-webpack-plugin

- webpack.config.js文件

```

const {CleanWebpackPlugin }= require('clean-webpack-plugin');
.....
plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlWebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin() //打包前先清除dist目录
],

```

☑️ 3-6 Entry 与 Output 的基础配置

1. 打包出两个文件:

```

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

output: {
  filename: '[name].js', //输出的文件名
  path: path.resolve(__dirname, 'dist') //配置路径
}

```

Dist/Index.html

```
<script type="text/javascript" src="main.js"></script><script type="text/javascript" src="sub.js"></script></body>
```

2. Index.html Src 中添加<http://cdn.com.cn/main.js>

Webpack.config.js 配置

```

output: {
  publicPath: 'http://cdn.com.cn',
  filename: '[name].js', //输出的文件名
  path: path.resolve(__dirname, 'dist') //配置路径
}

```

```
}
```

Dist/Index.html 文件中

```
<script type="text/javascript" src="http://cdn.com.cn/main.js"></script>
<script type="text/javascript" src="http://cdn.com.cn/sub.js"></script></body>
```

3. output配置项 参考官网

3-7 SourceMap 的配置 lesson3-7

//sourceMap它是一个映射关系，它知道dist目录下main.js文件line96有问题，实际上对应的是src目录下index.js文件中的第一行
//当前其实是index.js中第一行代码出错了

```
devtool : 'source-map', //源码映射，默认mode:"development"其实已经有默认配置了
devtool : 'eval', // 通过eval的这种执行形式来形成source map的对应关系的，效率最快，但有时不全面
devtool : 'inline-source-map', // 以url的方式变成一个base64的字符串写在main.js文件中
devtool : 'cheap-inline-source-map', // cheap具体到行，性能提升。而 inline具体到行列
// 最佳实践
devtool : 'cheap-module-eval-source-map', // mode:"development" 模式下推荐使用
devtool : 'cheap-module-source-map', // mode:"production" 模式下推荐使用
```

3-8 使用 WebpackDevServer 提升开发效率

1. "watch":"webpack --watch",

2. webpackDevServer

"start":"webpack-dev-server"

npm i -D webpack-dev-server

webpack.config.js

```
devServer : {
  port: 8090, // 端口号
  contentBase: './dist',
  open:true, // 直接打开浏览器,
  proxy : {
    '/api' : 'http://localhost:3000'
  } //跨域代理
},
```

3. 自己写个类似webpack-dev-server的服务

npm i -D express webpack-dev-middleware

Webpack.config.js

```
output: {
  publicPath : '/', // 设置根路径
  filename: '[name].js', //输出的文件名
  path: path.resolve(__dirname, 'dist') //配置路径
}
```

Server.js

```
const express = require('express');
const webpack = require('webpack');
const webpackDevMiddleware = require('webpack-dev-middleware');
const config = require('./webpack.config');
const compiler = webpack(config); // 返回一个编译器
// 在node中直接使用webpack
// 在命令行里使用webpack
const app = express();
app.use(webpackDevMiddleware(compiler, {
```



```

    publicPath: config.output.publicPath
  )))
  app.listen(3000, ()=>{
    console.log('server is running');
  })
}

```

Commad Line Interface 命令行方式用法 <https://webpack.js.org/api/cli/>

☑️ • 3-9 Hot Module Replacement 热模块更新 (1)

热模块替换 HMR

Css 中使用, 3-10的index.js一段代码其实css-loader底层已经实现了

Webpack.config.js

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

.....

```

devServer : {
  port: 8090, // 端口号
  contentBase: './dist',
  open: true, // 直接打开浏览器,
  hot: true, // 热模块更新
  hotOnly: true // 不让浏览器重新刷新
},

```

.....

```

plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlwebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin(), //打包前先清除dist目录
  new webpack.HotModuleReplacementPlugin() //开启webpack HMR 功能
],

```

☑️ • 3-10 Hot Module Replacement 热模块更新 (2)

js中使用, 配置参考上面

js中自己写逻辑, 进行控制。

- Index.js

```

// import './style.css';
//
// var btn = document.createElement('button');
// btn.innerHTML = '新增';
// document.body.appendChild(btn);
// btn.onclick = function() {
//   var div = document.createElement('div');
//   div.innerHTML = 'item';
//   document.body.appendChild(div);
// }
import counter from './counter';
import number from './number';
counter();
number();
if(module.hot){
  module.hot.accept('./number', ()=>{
    document.body.removeChild(document.getElementById('number'));
    number();
  })
}

```

- Counter.js

```

function counter() {
  var div = document.createElement('div');
  div.setAttribute('id', 'counter');
  div.innerHTML = 1;
}

```

```

div.onclick = function () {
  div.innerHTML = parseInt(div.innerHTML, 10) + 1
}
document.body.appendChild(div);
}
export default counter;
- Number.js
function number() {
  var div = document.createElement('div');
  div.setAttribute('id', 'number');
  div.innerHTML = 4000;
  document.body.appendChild(div);
}
export default number;

```

3-11 使用 Babel 处理 ES6 语法 (1)

每个Babel编译后的脚本文件，都以导入的方式使用Babel的帮助函数，而不是每个文件都复制一份帮助函数的代码。

优点

- (1) 提高代码重用性，缩小编译后的代码体积。
- (2) 防止污染全局作用域。（启用corejs配置）

babel-polyfill会将Promise等添加成全局变量，污染全局空间。

① 默认使用@babel/runtime，corejs配置为2时，改为使用@babel/runtime-corejs2。

② 几个包的包含关系。

babel-polyfill仅仅是引用core-js、regenerator-runtime这两个npm包。

@babel/runtime包含两个文件夹：helpers（定义了一些处理新的语法关键字的帮助函数）、regenerator（仅仅是引用regenerator-runtime这个npm包）。

@babel/runtime-corejs2包含三个文件夹：core-js（引用core-js这个npm包）、helpers（定义了一些处理新的语法关键字的帮助函数）、regenerator（仅仅是引用regenerator-runtime这个npm包）。

可以看出，@babel/runtime-corejs2 ≈ @babel/runtime + babel-polyfill：

@babel/runtime只能处理语法关键字，而@babel/runtime-corejs2还能处理新的全局变量（例如，Promise）、新的原生方法（例如，String.padStart）；

使用了@babel/runtime-corejs2，就无需再使用@babel/runtime了。

官网：<https://babeljs.io/>

npm install --save-dev babel-loader @babel/core

```

{
  test: /\.js$/,
  exclude: /node_modules/,
  loader: "babel-loader", // 只是桥梁作用，还需要 @babel/preset-env
  options: {
    presets: ["@babel/preset-env"],
  }
},

```

npm i -D @babel/preset-env //which enables transforms for ES2015+

npm i -D @babel/polyfill //语法补充，弥补一些低版本的实现

Index.js 中

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

Webpack.config.js 中

```
{
  test: /\.js$/,
  exclude: /node_modules/,
  loader: "babel-loader", // 只是桥梁作用, 还需要 @babel/preset-env
  options: {
    presets: [["@babel/preset-env", {
      targets: {
        chrome: "67" // chrome>67版本的, 就不要再进行babel的es6->es6转换了
      },
      useBuiltIns: 'usage' //babel-playfill 填充的时候, 根据业务代码来决定加什么, 从而减小打包文件的大小
    }]],
  },
}
```

userBuiltIns配置后main.js文件大小明显变小了。

注意: babel-playfill 不适合开发library, 会影响全局的变量和库。

3-12 使用 Babel 处理 ES6 语法 (2)

Library 库项目的时候的配置:

`npm i -D @babel/plugin-transform-runtime`

`npm i --save @babel/runtime-corejs2 @babel/runtime`

Webpack.config.js 配置: (注意plugins中["@babel/xxx",{abc:123}],)

```
options: {
  // presets: [["@babel/preset-env", {
  //   targets: {
  //     chrome: "67" // chrome>67版本的, 就不要再进行babel的es6->es6转换了
  //   },
  //   useBuiltIns: 'usage' //babel-playfill 填充的时候, 根据业务代码来决定加什么, 从而减小打包文件的大小
  // }]],
  "plugins": [
    ["@babel/plugin-transform-runtime",
    {
      // "absoluteRuntime": false,
      "corejs": 2,
      "helpers": true,
      "regenerator": true,
      "useESModules": false
    }
  ]
}
```

.babelrc文件方式

.babelrc文件中放置:

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

3-13 Webpack 实现对React框架代码的打包

`npm i react react-dom --save //安装react的包`

`npm install --save-dev @babel/preset-react`

.babelrc文件:

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        targets: {
          chrome: "67"
        },
        useBuiltIns: 'usage'
      }
    ],
    "@babel/preset-react"
  ]
}
```

Index.js示例:

```
import "@babel/polyfill";
import React, {Component} from 'react';
import ReactDOM from 'react-dom';
class App extends Component {
  render() {
    return <div>Hello World</div>
  }
}
ReactDOM.render(<App />, document.getElementById('root'));
```

✓ • 第4章 Webpack 的高级概念

本章接第三章内容, 继续讲解 Webpack 中难度更大的知识点, 包含了 Tree Shaking, Code Splitting, 打包环境区分, 缓存, shimming 等内容, 帮助你继续扩展 Webpack 的基础知识面。

✓ • 4-1 Tree Shaking 概念详解

1. When setting `useBuiltIns: 'usage'`, polyfills are automatically imported when needed.

Please remove the `import '@babel/polyfill'` call or use `useBuiltIns: 'entry'` instead.

使用了`useBuiltIns: 'usage'`, 可以把`import '@babel/polyfill'`去除

2. Lesson4-1 打包后

现象: `/*! exports provided: add, minus */` ,add minus都被打包, 而minus没有用到, 确也被打包进了main.js

解决: 引入什么打包什么, 使用Tree Shaking。 (注意: tree Shaking 只支持ES module方式的引入, 不支持commonJs引入)

Webpack.config.js 中:

```
optimization:{
  usedExports : true // 哪些导出的模块被使用了, 再做打包
},
```

Package.json 设置sideEffects

```
{
  "name": "lesson2",
  "sideEffects":["@babel/polyfill", "*.css"], //像@babel/polyfill 绑定window.promise 但没有export东西的, 运用tree shaking
  就需要sideEffects 进行设置,
```

```
  "version": "1.0.0",
  "description": "",
  "private": true,
```

开发环境下打包结果:

```
/*! exports provided: add, minus */
/*! exports used: add */
```

生成环境下设置:

```
mode:"production",
```

```
// optimization:{
//   usedExports : true // 哪些导出的模块被使用了, 再做打包
// },
```

打包结果main.js 中, 减法minus 方法就不会放在main.js中, 减小了文件的大小。

- 4-2 Development 和 Production 模式的区分打包

安装: `npm i -D webpack-merge`

Webpack.dev.js:

```
const path = require('path');
const webpack = require('webpack');
const merge = require('webpack-merge');
const commonConfig = require('./webpack.common');
// plugin 可以在webpack运行到某个时刻的时候, 帮你做一些事情
const devConfig = {
  mode: "development", // development 开发模式, 未被压缩, production 线上模式, 被压缩了
  // 最佳实践
  devtool: 'cheap-module-eval-source-map', // mode:"development" 模式下推荐使用
  devServer: {
    port: 8090, // 端口号
    contentBase: './dist',
    open: true, // 直接打开浏览器,
    hot: true, // 热模块更新
    // hotOnly: true // 不让浏览器重新刷新
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin() //开启webpack HMR 功能
  ],
  optimization: {
    usedExports: true // 哪些导出的模块被使用了, 再做打包
  }
}
module.exports = merge(commonConfig, devConfig);
```

Webpack.prod.js:

```
const path = require('path');
const merge = require('webpack-merge');
const commonConfig = require('./webpack.common');
const prodConfig = {
  // mode: "development", // development 开发模式, 未被压缩, production 线上模式, 被压缩了
  mode: "production",
  // 最佳实践
  // devtool: 'cheap-module-eval-source-map', // mode:"development" 模式下推荐使用
  devtool: 'cheap-module-source-map', // mode:"production" 模式下推荐使用
};
module.exports = merge(commonConfig, prodConfig);
```

Webpack.common.js:

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const {CleanWebpackPlugin} = require('clean-webpack-plugin');
module.exports = {
  entry: {
    main: './src/index.js',
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "babel-loader", // 只是桥梁作用, 还需要 @babel/preset-env
        /* options:{
          // presets:[["@babel/preset-env",{
            //   targets:{
            //     chrome:"67" // chrome>67版本的, 就不要再进行babel的es6->es6转换了
            //   },
            //   useBuiltIns:'usage' //babel-polyfill 填充的时候, 根据业务代码来决定加什么, 从而减小打包文件的大小
            // }]],
        */
      }
    ]
  }
}
```

```

        "plugins": [
          ["@babel/plugin-transform-runtime",
            {
              // "absoluteRuntime": false,
              "corejs": 2,
              "helpers": true,
              "regenerator": true,
              "useESModules": false
            }
          ]
        ]
      }
    ],
    test: /\. (jpg|png|gif)$/, //正则，以. jpg结尾的文件
    use: {
      // loader: 'file-loader',
      loader: 'url-loader', //把图片打包成base64的字符串
      options: {
        // placeholder 占位符
        // name: '[name].[ext]', //name 指原来的文件名, ext 文件后缀
        name: '[name]_[hash].[ext]', //name 指原来的文件名, ext 文件后缀
        outputPath: 'images/', // 将图片打包至images目录下
        limit: 2048 // >2kb打包成文件, 否则打包成base64的字符串
      }
    }
  }, {
    test: /\.css$/, //正则，以. jpg结尾的文件
    use: [
      'style-loader', 'css-loader', 'postcss-loader'
    ] //有执行顺序，从下到上，从右到左
  }, {
    test: /\. (eot|ttf|svg|woff2|woff)$/, //正则，以. jpg结尾的文件
    use: {
      loader: 'file-loader',
    }
  }
]
},
plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlwebpackPlugin 会在打包结束后，自动生成一个html文件，并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin(), //打包前先清除dist目录
],
output: {
  // publicPath : '/', // 设置根路径
  filename: '[name].js', //输出的文件名
  path: path.resolve(__dirname, 'dist') //配置路径
}
}
}

```

4-3 Webpack 和 Code Splitting (1) p8 00:16:18

webpack配置文件在build目录下，把dist变成与build同级。修改webpack.common.js文件：

```

output: {
  // publicPath : '/', // 设置根路径
  filename: '[name].js', //输出的文件名
  path: path.resolve(__dirname, '../dist') //配置路径
}

plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  })
]

```

```

    })), //htmlWebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
    new CleanWebpackPlugin(), //打包前先清除dist目录
  ],
  // 新版本中 newCleanWebpackPlugin() 不用在配置即可删除lesson4-3/dist 目录

```

- CodeSplitting

Index.js 代码:

```

// 第一种方式
// 首次访问页面时, 加载main.js 2mb
// 当页面业务逻辑发生变化时, 又要加载2mb的内容
console.log(_.join(['a', 'b', '1c'], '***'));
console.log(_.join(['a', 'b', '2c'], '***'));
// 第二种方式
// main.js 被拆成lodash.js(1Mb), main.js(1Mb)
// 当页面业务逻辑发生变化时, 只要加载main.js即可(1Mb)
// Code Splitting

```

添加loadsh.js:

```

import _ from 'lodash';
window._ = _;

```

Webpack.common.js:

```

entry: {
  // 注意先后顺序
  lodash: './src/lodash.js',
  main: './src/index.js'
},

```

4-4 Webpack 和 Code Splitting (2)

- 同步引入代码:

Index.js

```

import _ from 'lodash';
console.log(_.join(['a', 'b', '1c'], '***'));
console.log(_.join(['a', 'b', '2c'], '***'));

```

Webpack.common.js:

```

optimization:{
  // code splitting 代码风格
  splitChunks:{
    chunks:'all' //async只对异步代码生效, initial只对同步代码生效, all同步异步等进行代码分割 (vendors配置按照官网配置可生效)
  }
},

```

打包生成一个vendors~main.js文件

- 异步引入代码:

Index.js

// 异步加载

```

function getComment() {
  return import('lodash').then(({default:_})=>{
    var element = document.createElement('div');
    element.innerHTML = _.join(['Dell', 'Lee'], '-');
    return element;
  })
}
getComment().then(element=>{
  document.body.appendChild(element);
});

```

未安装 babel-plugin-dynamic-import-webpack
打包生成一个0.js文件

- 总结:

- // 代码分割, 和webpack无关
- // webpack中实现代码分割, 两种方式
- // 1. 同步方式: 只需要在webpack.common.js中做optimization的配置
- // 2. 异步代码(import): 异步代码, 无需做如何配置, 会自动进行代码分割, 放置到新的文件中

☑️ • 4-5 SplitChunksPlugin 配置参数详解 (1) p8 1:01:23

- 0.js改个名称

// 异步加载

/** webpackChunkName: "lodash" */ (魔法注释)

```
function getComment() {  
  return import(/*webpackChunkName: "lodash"*/ 'lodash').then(({default: _}) => {  
    var element = document.createElement('div');  
    element.innerHTML = _.join(['Dell', 'Lee'], '-');  
    return element;  
  })  
}  
getComment().then(element => {  
  document.body.appendChild(element);  
});
```

添加魔法注释后, 直接打包文件名称已经变成vendors~lodash.js, 下面步骤可以省略

1. 安装@babel/plugin-syntax-dynamic-import插件 <https://babeljs.io/docs/en/babel-plugin-syntax-dynamic-import>
2. .babelrc

```
{  
  "presets": [  
    [  
      "@babel/preset-env",  
      {  
        targets: {  
          chrome: "67"  
        },  
        useBuiltIns: 'usage'  
      }  
    ],  
    "@babel/preset-react"  
  ],  
  "plugins": ["@babel/plugin-syntax-dynamic-import"]  
}
```

☑️ • 4-6 SplitChunksPlugin 配置参数详解 (2)

官网地址: <https://webpack.js.org/plugins/split-chunks-plugin/> 里面有默认配置

```
module.exports = {  
  //...  
  optimization: {  
    splitChunks: {  
      chunks: 'all', //async只对异步代码生效, initial只对同步代码生效, all同步异步等进行代码分割 (vendors配置按照官网配置可生效)  
  
      minSize: 30000, //引入的库文件大小>30000才做代码分割  
      maxSize: 0,  
      minChunks: 1, //引入的库被引入几次后, 才进行代码分割  
      maxAsyncRequests: 5, //同时加载的模块数最多是5个, 前5个做代码分割, 超过的不做分割处理。一般按照默认配置  
      maxInitialRequests: 3, //入口文件做代码分割, 最多3个, 超过3个不做分割。一般按照默认配置
```


来自 <<https://webpack.js.org/plugins/split-chunks-plugin/#optimizationsplitchunks>>

- ## 一. Lazy Loading 懒加载

- ```
Index.js
// 异步函数
async function getComment() {
 const {default : _} =await import(/*webpackChunkName:"lodash"*/ 'lodash');
 const element = document.createElement('div');
 element.innerHTML = _.join(['Dell', 'Lee'], '-');
 return element;
}
document.addEventListener("click", ()=>{
 getComment().then(element=>{
 document.body.appendChild(element);
 });
})
```

### 1. 每个.js都叫做一个chunk

- ## 参考SplitChunksPlugin 中的配置

- ```
optimization: {
  splitChunks: {
    chunks: 'all', //async只对异步代码生效, initial只对同步代码生效, all同步异步等进行代码分割 (vendors配置按照官网可生效)
  }
}
```

- ## 一. Webpack analyse 打包分析工具

- 分区 webpack 的第 17 页

2. webpack --profile --json > stats.json

Package.json配置script

```
"dev-build": "webpack --profile --json > stats.json --config ./build/webpack.dev.js",
```

3. 把stats.json 上传至 <http://webpack.github.com/analyse>, 生成分析结果

4. 其它工具: 官网 <https://webpack.js.org/guides/code-splitting/#bundle-analysis>

- <https://alexkuz.github.io/webpack-chart/>

- <https://github.com/webpack-contrib/webpack-bundle-analyzer> (老师推荐用这个)

二. Preloading, Prefetching

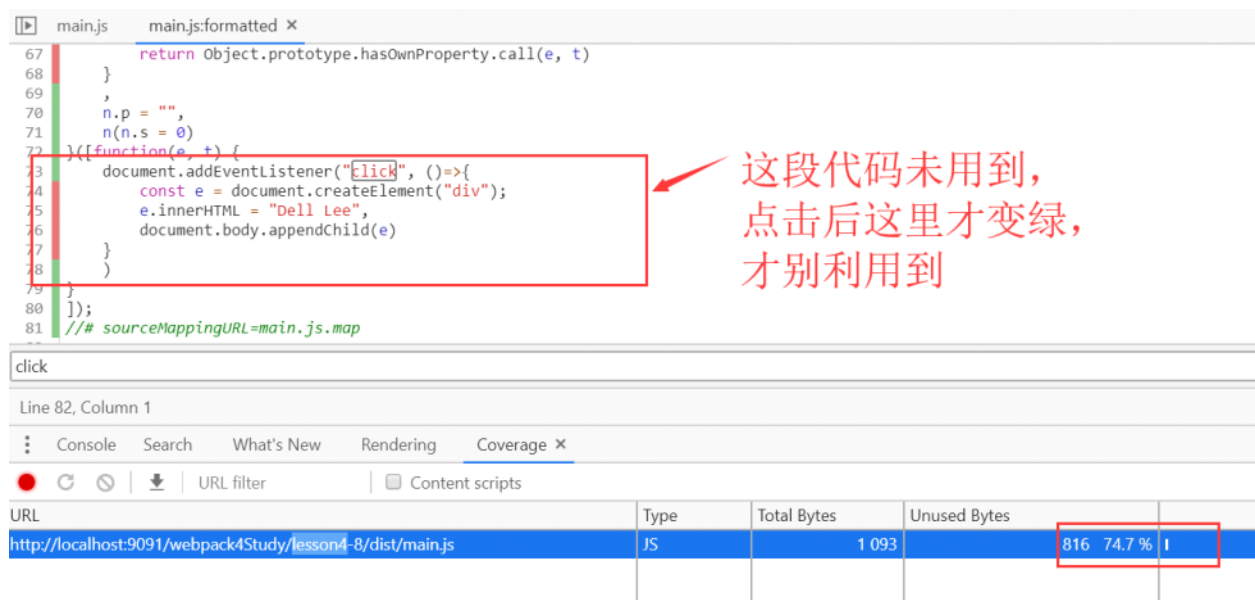
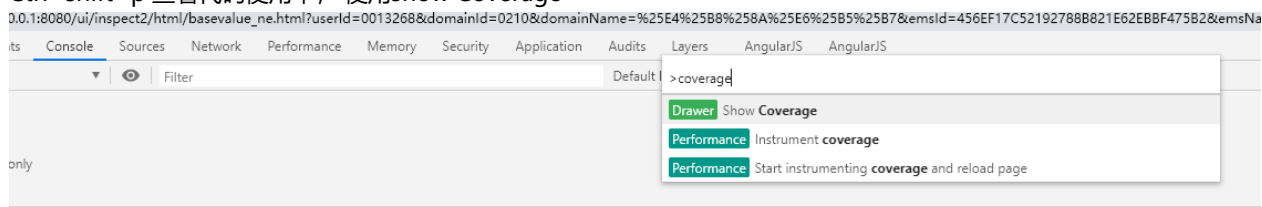
1. 异步加载模式的好处, splitChunks.chunks: 'async' 例: lesson4-8a

Index.js 代码:

```
document.addEventListener("click", ()=>{
  const element = document.createElement('div');
  element.innerHTML = "Dell Lee";
  document.body.appendChild(element);
})
```

注意: 老师打包用的是build及线上模式 npm run build

Ctrl+shift+p 查看代码使用率, 使用show Coverage



优化:

- 建立一个click.js文件

```
function handleClick() {
  const element = document.createElement('div');
  element.innerHTML = "Dell Lee";
  document.body.appendChild(element);
}
export default handleClick;
```

Index.js文件

```
document.addEventListener("click", ()=>{
  import('./click.js').then(({default:func})=>{
    func();
  })
})
```

```
});
```

(感觉没从数据上体现出来，数据利用率感觉老师说错了)

The screenshot shows the DevTools interface with the 'Sources' tab selected. The file 'main.js' is open, and a red box highlights a function definition. Below the code, the 'Coverage' tab is active, showing a table with the following data:

| URL | Type | Total Bytes | Unused Bytes |
|--|------|-------------|--------------|
| http://localhost:9091/webpack4Study/lesson4-8/dist/main.js | JS | 2 205 | 1 784 80.9 % |

2. 一般异步加载的使用场景，登录框的加载 例： lesson4-8b

Prefetch 主的流程加载完成后，有空闲时再预加载click.js文件，注意浏览器的兼容性问题。

Index.js

```
document.addEventListener("click", ()=>{
  import(/* webpackPrefetch: true */ './click.js').then(() =>{
    func();
  })
});
```

The screenshot shows the DevTools Network tab. A red arrow points to the 'Disable cache' checkbox, which is unchecked. The table below shows the network requests:

| Name | Method | Status | Remote Address | Type | Initiator | Size | Time | P... | Waterfall |
|---|--------|--------|----------------|-----------------|-----------------------|--------------|-------|------|-----------|
| index.html?_ijt=onvit9e58f58jgvo1famlr7jk | GET | 200 | 127.0.0.1:9091 | document | Other | 544 B | 10 ms | H... | |
| main.js | GET | 200 | 127.0.0.1:9091 | script | index.html?_ijt=on... | 2.7 KB | 15 ms | H... | |
| 1.js | GET | 200 | 127.0.0.1:9091 | javascript | main.js:1 | 586 B | 5 ms | ... | |
| favicon.ico | GET | 200 | 127.0.0.1:9091 | vnd.microsof... | Other | 2.1 KB | 5 ms | H... | |
| 1.js | GET | 200 | 127.0.0.1:9091 | script | main.js:1 | (disk cac... | 1 ms | L... | |

Preload是和主的一起加载的，与主的是平行的

prefetch和preload区别：

Preload directive has a bunch of differences compared to prefetch:

- A preloaded chunk starts loading in parallel to the parent chunk. A prefetched chunk starts after the parent chunk finishes loading.
- A preloaded chunk has medium priority and is instantly downloaded. A prefetched chunk is downloaded while the browser is idle.
- A preloaded chunk should be instantly requested by the parent chunk. A prefetched chunk can be used anytime in the future.
- Browser support is different.

来自 <<https://webpack.js.org/guides/code-splitting/#prefetchingpreloading-modules>>

页面js代码利用率提高，优先使用懒加载，而不是缓存

4-9 CSS 文件的代码分割 p8 02:21:30

```
output: {
  // publicPath : '/', // 设置根路径
  filename: '[name].js', // 入口文件输出的文件名
  chunkFilename: '[name].chunk.js', // 其他引用的包的输出文件名
  path: path.resolve(__dirname, '../dist') //配置路径
}
```

1. `npm install --save-dev mini-css-extract-plugin` 来自 <https://webpack.js.org/plugins/mini-css-extract-plugin/>

注意: 该插件现在不支持 HMR 模块热更新, 一般用于线上代码, 不用于开发环境中, 会降低开发效率 (新版本支持了)

Webpack.common.js:(css和scss移出去了)

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const {CleanWebpackPlugin} = require('clean-webpack-plugin');
module.exports = {
  entry: {
    main: './src/index.js'
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "babel-loader", // 只是桥梁作用, 还需要 @babel/preset-env
        /* options:{
          // presets:[["@babel/preset-env",{
            //   targets:{
            //     chrome:"67" // chrome>67版本的,就不要再进行babel的es6->es6转换了
            //   },
            //   useBuiltIns:'usage' //babel-polyfill 填充的时候, 根据业务代码来决定加什么, 从而减小打包文件的大
          }
        ]],
        "plugins":[
          ["@babel/plugin-transform-runtime",
            {
              // "absoluteRuntime": false,
              "corejs": 2,
              "helpers": true,
              "regenerator": true,
              "useESModules": false
            }
          ]
        ]
      }
    ]
  },
  {
    test: /\. (jpg|png|gif) $/, //正则, 以. jpg结尾的文件
    use: {
      loader: 'file-loader',
      loader: 'url-loader', //把图片打包成base64的字符串
      options: {
        // placehoder 占位符
        // name: '[name].[ext]', //name 指原来的文件名, ext 文件后缀
        name: '[name]_[hash].[ext]', //name 指原来的文件名, ext 文件后缀
        outputPath: 'images/', // 将图片打包至images目录下
        limit: 2048 // >2kb打包成文件, 否则打包成base64的字符串
      }
    }
  },
  {
    test: /\. (eot|ttf|svg|woff2|woff) $/, //正则, 以. jpg结尾的文件
    use: {
```

```

        loader: 'file-loader',
      }
    }
  ],
},
plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlwebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin(), //打包前先清除dist目录
],
optimization: {
  usedExports : true , // 哪些导出的模块被使用了, 再做打包, tree shaking 相关
  splitChunks: {
    chunks: 'all', //async只对异步代码生效, initial只对同步代码生效, all同步异步等进行代码分割 (vendors配置按照官网配置可生效)
  }
},
output: {
  // publicPath : '/', // 设置根路径
  filename: '[name].js', // 入口文件输出的文件名
  chunkFilename: '[name].chunk.js', // 其他引用的包的输出文件名
  path: path.resolve(__dirname, '../dist') //配置路径
}
}
}

```

```

Webpack.dev.js:
const path = require('path');
const webpack = require('webpack');
const merge = require('webpack-merge');
const commonConfig = require('./webpack.common');
// plugin 可以在webpack运行到某个时刻的时候, 帮你做一些事情
const devConfig = {
  mode: "development", // development 开发模式, 未被压缩, production 线上模式, 被压缩了
  // 最佳实践
  devtool : 'cheap-module-eval-source-map', // mode:"development" 模式下推荐使用
  devServer : {
    port: 8090, // 端口号
    contentBase: '../dist',
    open:true, // 直接打开浏览器,
    hot:true, // 热模块更新
    // hotOnly:true // 不让浏览器重新刷新
  },
  module: {
    rules: [
      {
        test:/\.scss$/, //正则, 以 .jpg结尾的文件
        use:[
          'style-loader',
          {
            loader: 'css-loader',
            options : {
              importLoaders : 2, // 例如scss文件, 嵌套引入scss文件, 配置importLoaders以后, 会再走postcss-loader和sass-loader这两个loader
            }
          },
          'sass-loader',
          'postcss-loader'
        ] //有执行顺序, 从下到上, 从右到左
      },
      {
        test:/\.css$/, //正则, 以 .jpg结尾的文件

```

```

        use:[
            'style-loader',
            'css-loader',
            'postcss-loader'
        ] //有执行顺序, 从下到上, 从右到左
    },
]
},
plugins: [
    new webpack.HotModuleReplacementPlugin() //开启webpack HMR 功能
],
}
}
module.exports = merge(commonConfig, devConfig);

Webpack.prod.js:
// const path = require('path');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const merge = require('webpack-merge');
const commonConfig = require('./webpack.common');
const prodConfig = {
    // mode: "development", // development 开发模式, 未被压缩, production 线上模式, 被压缩了
    mode: "production",
    // 最佳实践
    // devtool : 'cheap-module-eval-source-map', // mode:"development" 模式下推荐使用
    devtool : 'cheap-module-source-map', // mode:"production" 模式下推荐使用
    module:{
        rules:[
            {
                test:/\.scss$/, //正则, 以 .jpg结尾的文件
                use:[
                    // 'style-loader',
                    MiniCssExtractPlugin.loader, // 替换掉 style-loader
                    {
                        loader: 'css-loader',
                        options : {
                            importLoaders : 2, // 例如scss文件, 嵌套引入scss文件, 配置importLoaders以后, 会再走postcss-
                            loader和sass-loader这两个loader
                        }
                    },
                    // modules :true // 开启css模块化打包
                ],
            },
            {
                test:/\.css$/, //正则, 以 .jpg结尾的文件
                use:[
                    MiniCssExtractPlugin.loader, // 替换掉 style-loader
                    'css-loader',
                    'postcss-loader'
                ] //有执行顺序, 从下到上, 从右到左
            },
        ],
    },
    plugins: [
        new MiniCssExtractPlugin({})
    ]
};
module.exports = merge(commonConfig, prodConfig);

```

注意: package.json中tree shaking的设置

```

{
    "name": "lesson2",
    "sideEffects": ["*.css"],
    "version": "1.0.0",
}

```

```
    "description": "",
    .....

```

2. 配置自己的文件名称

```
plugins: [
  new MiniCssExtractPlugin({
    filename: '[name].css', // 直接引用的, 从入口文件过来的
    chunkFilename: '[name].chunk.css' // 间接的引用的,
  })
]
```

Main.css 直接被html页面引用的, 走filename, 间接的被引用的走chunkFilename

3. Css压缩 [optimize-css-assets-webpack-plugin](https://webpack.js.org/plugins/mini-css-extract-plugin/) 来自 <<https://webpack.js.org/plugins/mini-css-extract-plugin/>>

安装 npm install --save-dev optimize-css-assets-webpack-plugin 来自 <<https://github.com/NMFR/optimize-css-assets-webpack-plugin>>

用法参考: <https://webpack.js.org/plugins/mini-css-extract-plugin>

Webpack.prod.js:

```
// const path = require('path');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const OptimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');
const merge = require('webpack-merge');
const commonConfig = require('./webpack.common');
const prodConfig = {
  // mode: "development", // development 开发模式, 未被压缩, production 线上模式, 被压缩了
  mode: "production",
  // 最佳实践
  // devtool : 'cheap-module-eval-source-map', // mode:"development" 模式下推荐使用
  devtool : 'cheap-module-source-map', // mode:"production" 模式下推荐使用
  module: {
    rules: [
      {
        test: /\.scss$/, //正则, 以 .jpg结尾的文件
        use: [
          // 'style-loader',
          MiniCssExtractPlugin.loader, // 替换掉 style-loader
          {
            loader: 'css-loader',
            options: {
              importLoaders: 2, // 例如scss文件, 嵌套引入scss文件, 配置importLoaders以后, 会再走postcss-
              // loader和sass-loader这两个loader
              // modules :true // 开启css模块化打包
            }
          },
          'sass-loader',
          'postcss-loader'
        ] //有执行顺序, 从下到上, 从右到左
      },
    ],
    {
      test: /\.css$/, //正则, 以 .jpg结尾的文件
      use: [
        MiniCssExtractPlugin.loader, // 替换掉 style-loader
        'css-loader',
        'postcss-loader'
      ] //有执行顺序, 从下到上, 从右到左
    },
  ],
  optimization: {

```

```

    minimizer: [ new OptimizeCSSAssetsPlugin({}) ],
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: '[name].css', // 直接引用的, 从入口文件过来的
      chunkFilename: '[name].chunk.css' // 间接的引用的,
    })
  ]
};
module.exports = merge(commonConfig, prodConfig);

```

//将css文件压缩

4. 多个入口文件配置在entry中, 希望打包在一个css文件中

例如:

```

entry: {
  main: './src/index.js',
  main1: './src/index1.js'
},

```

则配置:

```

optimization: {
  splitChunks: {
    cacheGroups: {
      styles: {
        name: 'styles', //
        test: /\.css$/,
        chunks: 'all',
        enforce: true, //true表示不管minsize等参数, 都打包到styles文件中
      },
    },
  },
},

```

5. 根据入口的entry入口的不同, 把css文件打包到不同的文件中

例如:

```

entry: {
  main: './src/index.js',
  main1: './src/index1.js'
},

```

打包到不同文件中, 即main.css和main1.css中

Extracting CSS based on entry 来自 <<https://webpack.js.org/plugins/mini-css-extract-plugin/>>

```

const path = require('path');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
function recursiveIssuer(m) {
  if (m.issuer) {
    return recursiveIssuer(m.issuer);
  } else if (m.name) {
    return m.name;
  } else {
    return false;
  }
}
.....

optimization: {
  splitChunks: {
    cacheGroups: {
      fooStyles: {
        name: 'foo', // entry.foo
        test: (m, c, entry = 'foo') =>
          m.constructor.name === 'CssModule' && recursiveIssuer(m) === entry,
        chunks: 'all',
        enforce: true,
      },
    },
  },
}

```



```
barStyles: {  
  name: 'bar',  
  test: (m, c, entry = 'bar') =>  
    m.constructor.name === 'CssModule' && recursiveIssuer(m) === entry,  
  chunks: 'all',  
  enforce: true,  
},  
,  
,  
,
```

- 4-10 Webpack 与浏览器缓存 (Caching) p8 02:47:37

WARNING in entryptoint size limit: The following entryptoint(s) combined asset size exceeds the recommended limit (244 KiB). This can impact web performance.

- 去除这个警告:

Webpack.common.js中添加:

```
performance:false, // 去除文件大于244kb的警告
```

去掉source map

```
// devtool : 'cheap-module-source-map', // mode:"production" 模式下推荐使用
```

- Webpack.common.js: 使得打包文件变成vendors.js

```
optimization: {
  usedExports : true ,    // 哪些导出的模块被使用了, 再做打包, tree shaking 相关
  splitChunks: {
    chunks: 'all', // async 只对异步代码生效, initial 只对同步代码生效, all 同步异步等进行代码分割 (vendors 配置按照官网配置可生效)
    cacheGroups: {
      vendors: {
        test: /[\\/]node_modules[\\/]/, // 引入的库是否在 node_modules 目录下的
        priority: -10, // 值越大, 优先级越高
        name: 'vendors' //
      }
    }
  }
},
```

- 使用contenthash, 根据内容产生不同的hash值

Webpack.dev.js开发环境中:

```
output: {
  filename: '[name].js', // 入口文件输出的文件名
  chunkFilename: '[name].chunk.js', // 其他引用的包的输出文件名
}
```

Webpack.prod.js 生成环境中:

```
output: {
  filename: '[name].[contenthash].js', // 入口文件输出的文件名
  chunkFilename: '[name].[contenthash].chunk.js', // 其他引用的包的输出文件名
}
```

打包结果，文件名后会跟上hash值

```

Time: 1238ms
Built at: 2018-12-18 00:45:07

    Asset      Size  Chunks             Chunk Names
    index.html 294 bytes          [emitted]
main.7b43412215c5d5a84c59.js 6.95 KiB          [emitted]  main
vendors.81e4f3a1c1a490fd2b19.js 794 KiB          [emitted]  vendors
Entrypoint main = vendors.81e4f3a1c1a490fd2b19.js main.7b43412215c5d5a84c59.js

[2] ./src/index.js 146 bytes {0} [built]
[3] (webpack)/buildin/global.js 489 bytes {1} [built]
[4] (webpack)/buildin/module.js 497 bytes {1} [built]
+ 2 hidden modules
Child html-webpack-plugin for "index.html":
  1 asset
  Entrypoint undefined = index.html
  [0] ./node_modules/html-webpack-plugin/lib/loader.js!./src/index.html 34
6 bytes {0} [built]
  [2] (webpack)/buildin/global.js 489 bytes {0} [built]
  [3] (webpack)/buildin/module.js 497 bytes {0} [built]
+ 1 hidden module
dells-mac:lesson dell$

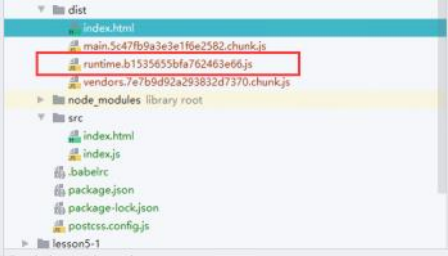
```

针对webpac4老版本, 有可能两次打包的值不同, 还需配置: (//manifest的关系)

```

optimization: {
  runtimeChunk: {
    name: 'runtime'
  },
  usedExports: true, // 哪些导出的模块被使用了, 再做打包, tree shaking 相关
  splitChunks: {
    chunks: 'all', // async只对异步代码生效, initial只对同步代码生效, all同步异步等进行代码分割 (vendors配置按照官网配置可生效)
    cacheGroups: {
      vendors: {
        test: /[\\/]node_modules[\\/]/, //引入的库是否在node_modules目录下的
        priority: -10, //值越大, 优先级越高
        name: 'vendors' //
      }
    }
  },
},

```



```

Asset      Size  Chunks             Chunk Names
    index.html 400 bytes          [emitted]
main.5c47fb9a3e1f6e2582.chunk.js 913 bytes          [emitted] [immutable]  main
runtime.b1535655bfa762463e06.js 5.11 KiB          [emitted] [immutable]  runtime
vendors.7e7b9d92a293832d7370.chunk.js 803 KiB          [emitted] [immutable]  vendors
Entrypoint main = runtime.b1535655bfa762463e06.js vendors.7e7b9d92a293832d7370.chunk.js main.5c47fb9a3e1f6e2582.chunk.js

[2] ./src/index.js 134 bytes {0} [built]
[3] (webpack)/buildin/global.js 472 bytes {1} [built]
[4] (webpack)/buildin/module.js 497 bytes {1} [built]
+ 2 hidden modules
Child html-webpack-plugin for "index.html":
  1 asset
  Entrypoint undefined = index.html
  [0] ./node_modules/html-webpack-plugin/lib/loader.js!./src/index.html 373 bytes {0} [built]

```

4-11 Shimming 的作用 p8 03:07:17 (shim 垫片) lesson4-11a

- 例1, 模块里的变量想使用上一文件模块中的变量, 不可能, 模块间不会耦合

jQuery.ui.js:

```

export function ui() {
  $('body').css('background', 'red');
}

```

```
Index.js
import _ from 'lodash';
import $ from 'jquery';
import {ui} from './jquery.ui';
ui();
const dom = $('<div>');
dom.html(_.join(['Dell', 'Lee'], '-'));
$('body').append(dom);
```

打包运行后报错: (jquery.ui.js中\$没有定义)

```
Uncaught ReferenceError: $ is not defined
    at ui (main.914aff2...chunk.js:19)
    at Module.4 (main.914aff2...chunk.js:25)
    at __webpack_require__ (runtime.b153565...js:80)
    at checkDeferredModules (runtime.b153565...js:46)
    at Array.webpackJsonpCallback [as push] (runtime.b153565...js:33)
    at main.914aff2...chunk.js:1
```

解决方法:

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

.....
plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlWebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin(), //打包前先清除dist目录,
  new webpack.ProvidePlugin({
    $: 'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量
  })
],
```

注意: jquery.ui.js 中并没有引入node_modules中的jquery, 而是在webpack中配置实现重新打包, 不报错, 功能实现。

2. 例2

将例1中jquery.ui.js修改成:

```
export function ui() {
  // $('body').css('background', 'red');
  $('body').css('background', _.join(['red'], ''));
}
```

Webpack.common.js配置:

```
plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlWebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin(), //打包前先清除dist目录,
  new webpack.ProvidePlugin({
    $: 'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量
    _: 'lodash'
  })
],
```

3. 使用lodash, 直接使用_join

jQuery.ui.js

```
export function ui() {
  // $('body').css('background', 'red');
  // $('body').css('background', _.join(['blue'], ''));
  $('body').css('background', _join(['green'], ''));
}
```

Webpack.common.js配置:

```
plugins: [  
  new HtmlWebpackPlugin({  
    template: 'src/index.html' // 使用的模板  
  }), //htmlwebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中  
  new CleanWebpackPlugin(), //打包前先清除dist目录,  
  new webpack.ProvidePlugin({  
    $: 'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量  
    // _: 'lodash'  
    _join: ['lodash', 'join']  
  })  
],
```

4. 使每一个模块中的this 都指向window, 可使用插件 imports-loader, 命令npm install imports-loader --save-dev
Index.js中

```
// this 指向模块自身, 加载imports-loader后显示true  
console.log(this === window);
```

Webpack.common.js

```
module: {  
  rules: [  
    {  
      test: /\.js$/,  
      exclude: /node_modules/,  
      use: [  
        {  
          loader: "babel-loader", // 只是桥梁作用, 还需要 @babel/preset-env  
        }, {  
          loader: "imports-loader?this=>window" // 注意写法, 把this改成window  
        }  
      ]  
    },  
  ],  
}
```

打包后控制台打印true了

imports-loader 改变了模块中this,指向了window

官网参考文档: <https://webpack.js.org/guides/shimming/>

- 4-12 环境变量的使用方法 p8 03:24:05

Package.json:

```
"scripts": {  
  "dev-build": "webpack --config ./build/webpack.common.js",  
  "dev": "webpack-dev-server --config ./build/webpack.common.js",  
  "build": "webpack --env.production --config ./build/webpack.common.js",  
  "build2": "webpack --env.production --config ./build/webpack.common.js"  
},
```

Webpack.dev.js:

```
const webpack = require('webpack');  
const devConfig = {  
  mode: "development", // development 开发模式, 未被压缩, production 线上模式, 被压缩了  
  // 最佳实践  
  devtool: 'cheap-module-eval-source-map', // mode:"development" 模式下推荐使用  
  devServer: {  
    port: 8090, // 端口号  
    contentBase: './dist',  
    open: true, // 直接打开浏览器,  
    hot: true, // 热模块更新  
    // hotOnly: true // 不让浏览器重新刷新  
  },  
}
```

```

module: {
  rules: [
    {
      test: /\.scss$/, //正则, 以 .jpg结尾的文件
      use: [
        'style-loader',
        {
          loader: 'css-loader',
          options: {
            importLoaders: 2, // 例如scss文件, 嵌套引入scss文件, 配置importLoaders以后, 会再走postcss-loader和
            sass-loader这两个loader
          },
          // modules :true // 开启css模块化打包
        }
      ],
      // 有执行顺序, 从下到上, 从右到左
    },
    {
      test: /\.css$/, //正则, 以 .jpg结尾的文件
      use: [
        'style-loader',
        'css-loader',
        'postcss-loader'
      ],
      // 有执行顺序, 从下到上, 从右到左
    }
  ],
  plugins: [
    new webpack.HotModuleReplacementPlugin() //开启webpack HMR 功能
  ],
  output: {
    filename: '[name].js', // 入口文件输出的文件名
    chunkFilename: '[name].chunk.js', // 其他引用的包的输出文件名
  }
}
module.exports = devConfig;

```

Webpack.prod.js:

```

const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const OptimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');
const prodConfig = {
  // mode: "development", // development 开发模式, 未被压缩, production 线上模式, 被压缩了
  mode: "production",
  // 最佳实践
  // devtool : 'cheap-module-eval-source-map', // mode:"development" 模式下推荐使用
  devtool : 'cheap-module-source-map', // mode:"production" 模式下推荐使用
  module: {
    rules: [
      {
        test: /\.scss$/, //正则, 以 .jpg结尾的文件
        use: [
          // 'style-loader',
          MiniCssExtractPlugin.loader, // 替换掉 style-loader
          {
            loader: 'css-loader',
            options: {
              importLoaders: 2, // 例如scss文件, 嵌套引入scss文件, 配置importLoaders以后, 会再走postcss-loader和
              sass-loader这两个loader
            },
            // modules :true // 开启css模块化打包
          }
        ],
        // 有执行顺序, 从下到上, 从右到左
      },
      {
        loader: 'css-loader',
        options: {
          importLoaders: 2, // 例如scss文件, 嵌套引入scss文件, 配置importLoaders以后, 会再走postcss-loader和
          sass-loader这两个loader
        },
        // modules :true // 开启css模块化打包
      },
      'sass-loader',
      'postcss-loader'
    ]
  }
}

```

```

    ] //有执行顺序, 从下到上, 从右到左
  },
  {
    test: /\.css$/, //正则, 以 .jpg结尾的文件
    use: [
      MiniCssExtractPlugin.loader, // 替换掉 style-loader
      'css-loader',
      'postcss-loader'
    ] //有执行顺序, 从下到上, 从右到左
  },
]
},
optimization: {
  minimizer: [ new OptimizeCSSAssetsPlugin({}) ],
},
plugins: [
  new MiniCssExtractPlugin({
    filename: '[name].css', // 直接引用的, 从入口文件过来的
    chunkFilename: '[name].chunk.css' // 间接的引用的,
  })
],
output: {
  filename: '[name].[contenthash].js', // 入口文件输出的文件名
  chunkFilename: '[name].[contenthash].chunk.js', // 其他引用的包的输出文件名
}
};
module.exports = prodConfig;

```

Webpack.common.js:

```

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const {CleanWebpackPlugin } = require('clean-webpack-plugin');
const webpack = require('webpack');
const merge = require('webpack-merge');
const devConfig = require('./webpack.dev');
const prodConfig = require('./webpack.prod');
const commonConfig = {
  entry: {
    main: './src/index.js'
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: [
          {
            loader: "babel-loader", // 只是桥梁作用, 还需要 @babel/preset-env
          }, {
            loader: "imports-loader?this=>window" // 注意写法, 把this改成window
          }
        ]
      },
      {
        test: /\. (jpg|png|gif) $/, //正则, 以 .jpg结尾的文件
        use: {
          // loader: 'file-loader',
          loader: 'url-loader', //把图片打包成base64的字符串
          options: {
            // placeholder 占位符
            // name: '[name].[ext]', //name 指原来的文件名, ext 文件后缀
            name: '[name]_[hash].[ext]', //name 指原来的文件名, ext 文件后缀
            outputPath: 'images/', // 将图片打包至images目录下
            limit: 2048 // >2kb打包成文件, 否则打包成base64的字符串
          }
        }
      }
    ]
  }
};

```

```

    }
  }, {
    test: /\. (eot|ttf|svg|woff2|woff)$/, //正则, 以. jpg结尾的文件
    use: {
      loader: 'file-loader',
    }
  }
]
},
plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlWebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin(), //打包前先清除dist目录,
  new webpack.ProvidePlugin({
    $: 'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量
    // _: 'lodash'
    __join: ['lodash', 'join']
  })
],
optimization: {
  runtimeChunk: {
    name: 'runtime'
  },
  usedExports: true, // 哪些导出的模块被使用了, 再做打包, tree shaking 相关
  splitChunks: {
    chunks: 'all', //async只对异步代码生效, initial只对同步代码生效, all同步异步等进行代码分割 (vendors配置按照官网配置可生效)
    cacheGroups: {
      vendors: {
        test: /[\\/]node_modules[\\/]/, //引入的库是否在node_modules目录下的
        priority: -10, //值越大, 优先级越高
        name: 'vendors' //
      }
    }
  },
  performance: false, // 去除文件大于244kb的警告
}
module.exports = (env) => {
  if (env && env.production) {
    return merge(commonConfig, prodConfig);
  } else {
    return merge(commonConfig, devConfig);
  }
}
}

```

其它写法:

Package.json

```

"scripts": {
  "dev-build": "webpack --config ./build/webpack.common.js",
  "dev": "webpack-dev-server --config ./build/webpack.common.js",
  "build": "webpack --env.production --config ./build/webpack.common.js",
  "build2": "webpack --env.production=abc --config ./build/webpack.common.js"
},

```

Webpack.common.js

```

module.exports = (env) => {
  // if (env && env.production) {
  if (env && env.production === "abc") {
    return merge(commonConfig, prodConfig);
  } else {
    return merge(commonConfig, devConfig);
  }
}

```

```
}  
}
```

一般还是采用之前dev和prod分开的模式模式

- development和production模式的区分打包 **lesson4-2b(其实是该视频开通部分)**

1. Development 中Sourcemap 比较全，快速定位代码问题。Production Sourcemap 更加简洁。
2. Development 中代码不压缩，Production 一般是被压缩过的代码。

代码清单：p8 03:36:33 （参考代码lesson4-2b）

- **第5章 Webpack 实战配置案例讲解**

本章通过库文件打包，PWA项目打包，TypeScript打包支持等实战常见 Webpack 配置案例，带大家了解最新前端工程化常识，并在实例实现的过程中，巩固前三章节的基础知识点。同时章节末尾进行了 Webpack 打包性能优化的内容，帮助同学们了解如何在打包速度过慢时进行合理的打包过程优化。...

- 5-1 Library 的打包

npm install webpack webpack-cli lodash --save

Webpack.config.js配置

```
const path = require('path');  
module.exports = {  
  mode: "production",  
  entry: './src/index.js',  
  // externals:["lodash"],// 自己的库里引用了lodash, 如果用户的文件也引用了lodash, 就多余了, 所以要配置这个设置  
  externals:{  
    lodash: { // 这个lodash 和 const lodash 名称需一致  
      commonjs: 'lodash', // const lodash = require('lodash');const library = require("library")  
      commonjs2: 'lodash', //  
      amd: 'lodash',  
      root: '_', // <script src ="/library.js"></script> 然后 页面里必须使用 "_" 这个变量  
    },  
  },  
  output : {  
    path: path.resolve(__dirname, 'dist'),  
    filename: 'library.js',  
    libraryTarget: 'umd', // 通用 import require 等形式引用  
    // libraryTarget: 'this', // 会挂载到全局的this上, this.library  
    library: 'library' // <script src="library.js"> 然后library.math 这样用(全局变量里增加一个library的变量)  
  }  
}
```

Example.html

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <title>Title</title>  
  <script src="../node_modules/lodash/lodash.js"></script>  
  <script src="../dist/library.js"></script>  
</head>  
<body>  
</body>  
<script>  
  console.log(library);  
  console.log(library.default.math.add(1,2));  
  console.log(library.default.string.join(1,2));  
</script>  
</html>
```

Externals:打包库的时候，lodash不打包库里去，而让业务代码即用户外部自己去加载

libraryTarget:"umd", 库在外部可以以何种方式被引入，“umd”指commonjs，AMD都可以引入

Library:"root",在script标签引入的时候会注入一个变量root
可以参考官方文档

Package.json修改配置:

```
{
  "name": "library-dell-lee-2019", //名称特殊一点, 避免冲突
  "version": "1.0.0",
  "description": "",
  "main": "./dist/library.js",
  "scripts": {
    "build": "webpack"
  },
}
```

然后通过npm发布

- 5-2 PWA 的打包配置 p9 00:29:22

Progressive Web Application 在服务器挂掉后, 本地浏览器中的页面还能通过缓存显示

- 安装插件 npm install workbox-webpack-plugin --save-dev
- 修改webpack.prod.js (pwa 生成环境下体验更好, 调试模式下不需要)

```
const WorkboxPlugin = require('workbox-webpack-plugin');

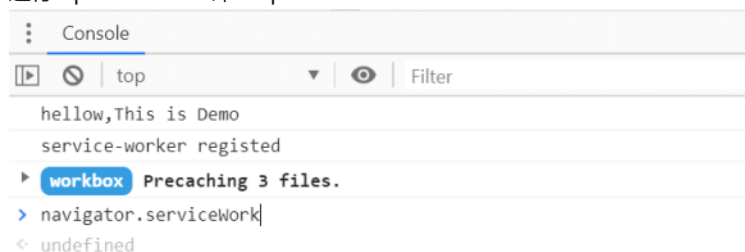
.....
plugins: [
  new MiniCssExtractPlugin({
    filename: '[name].css', // 直接引用的, 从入口文件过来的
    chunkFilename: '[name].chunk.css' // 间接的引用的,
  }),
  new WorkboxPlugin.GenerateSW({
    clientsClaim: true,
    skipWaiting: true
  })
],
```

重新打包会生成service-workder.js 和precache-manifest.ccd51bbf05a0801108a2e6c905d71ac3.js文件

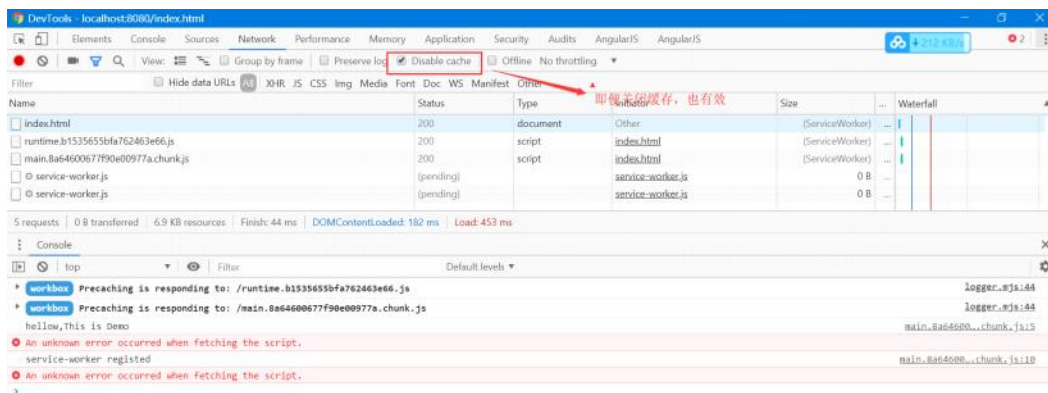
- 修改业务代码 index.js

```
console.log("hellow,This is Demo");
// 如果你的浏览器支持serviceWorker, 则自行一些代码
if ('serviceWorker' in navigator) {
  window.addEventListener('load', ()=>{
    navigator.serviceWorker.register('/service-worker.js')
      .then(registration=>{
        console.log("service-worker registered");
      }).catch(error=>{
        console.log("service-worker register error");
      })
  })
}
```

- 运行 npm run start 即 http-server dist



- 关闭http-server后



5-3 TypeScript 的打包配置 p9 00:47:00

- Package.json 安装的库

```
{
  "name": "lesson5-3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "webpack"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/lodash": "^4.14.144",
    "lodash": "^4.17.15",
    "ts-loader": "^6.2.1",
    "typescript": "^3.7.2",
    "webpack": "^4.41.2",
    "webpack-cli": "^3.3.10"
  }
}
```

- TypeScript 是javascript 的一个超集,可对属性、类型进行校验,可以避免一下错误。

Index.tsx

```
import * as _ from 'lodash';
class Greeter{
  greeting:string;
  constructor(message:string){
    this.greeting = message;
  }
  greet() {
    return _.join(['Hello,', ' ', this.greeting], ' ');
    // return "Hello,"+ this.greeting;
  }
}

let greeter = new Greeter("World");// new Greeter(123)会因为string类型而报错
alert(greeter.greet());
// let button = document.createElement('button');
// button.textContent = "Say Hello";
// button.onclick = function () {
//   alert(greeter.greet());
// }
// document.body.appendChild(button);
```

- Webpack.config.js

```
const path = require('path');
module.exports = {
  entry: './src/index.tsx',
  module: {
    rules: [{
      test: /\.tsx?$/,
      use: 'ts-loader', //需要tsconfig.json 配置文件
    }]
  }
}
```

```

        exclude: /node_modules/
      }]
    },
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  }
}

```

- Tsconfig.json

```

{
  "compilerOptions": {
    "outDir": "./dist", //生成文件放入dist目录
    "module": "es6", //也可以commonjs ,
    "target": "es5", //打包生成文件类型
    "allowJs": true //允许ts文件中再引入js的模块
  },
  "exclude": [
    "node_modules"
  ]
}

```

- 新版本typescript安装库 (lodash, jquery) 的推荐方式

参考github: <https://github.com/DefinitelyTyped/DefinitelyTyped> 中有typesearch 链接

注意:

```
import * as _ from 'lodash'; //引入外部库时
```

Npm install @types/lodash -D 安装库对应的类型文件, 可以报错提示

✔ 5-4 使用 WebpackDevServer 实现请求转发 p9 01:45:05

文档参考: <https://webpack.js.org/configuration/dev-server/#devserver-proxy>

Dev-server底层用到了 <http-proxy-middleware>

环境:

- Copy lesson3-13代码
- npm install axios -D

Index.js

```

// import '@babel/polyfill'; // 不引入, 英文.babelrc中已经配置了 "useBuiltIns": "usage", 会自动加载polyfill
import React, {Component} from 'react';
import ReactDOM from 'react-dom';
import axios from 'axios';

```

```

class App extends Component {
  componentDidMount() {
    // axios.get('http://www.dell-lee.com/react/api/header.json')// dell-lee 服务器允许跨域
    // 可以用charles fiddler 等工具进行代理
    // 现在用webpack devServer.proxy 进行配置
    axios.get('/react/api/header.json')// dell-lee 服务器允许跨域
      .then((res)=>{
        console.log(res);
      })
  }
  render() {
    return <div>Hello World</div>
  }
}
ReactDOM.render(<App />, document.getElementById('root'));

```

Webpack.config.js 中 部分配置代码:

```

devServer : {
  port: 8090, // 端口号
  contentBase: './dist',
  open:true, // 直接打开浏览器,
  hot:true, // 热模块更新
  hotOnly:true, // 不让浏览器重新刷新
  // 只能在webpack devserver 环境下生效, 即在开发环境下才生效, 线上代码无效
  proxy: {
    // '/react/api':'http://www.dell-lee.com'
    // '/react/api':{
    //   target:'http://www.dell-lee.com',
    //   pathRewrite:{
    //     'header.json':'demo.json' // 路径中请求header.json 返回 demo.json 的数据, 后台接口写完后, 可注释该行
    //   }
    // }
    // '/react/api':{
    //   target:'https://www.dell-lee.com', // 请求代理到哪个网址
    //   secure: false, // 如果是 https 方式, 开发模式下可以先将安全 关闭
    //   // bypass: function(req, res, proxyOptions) {
    //   //   // 如果这次请求是请求html文件, 就返回false, 跳过这次代理中的转发
    //   //   if (req.headers.accept.indexOf('html') !== -1) {
    //   //     console.log('Skipping proxy for browser request. ');
    //   //     return false;
    //   //   }
    //   // },
    //   pathRewrite:{
    //     'header.json':'demo.json' // 路径中请求header.json 返回 demo.json 的数据, 后台接口写完后, 可注释该行
    //   },
    //   // 变更请求头
    //   changeOrigin : true,
    //   headers: {
    //     host:"www.dell-lee.com",
    //     cookie : 'sdfsfa' //模拟登录或者鉴权的场景
    //   }
    // }
  },
},

```

proxy配置只能在webpack devserver 环境下生效, 即在开发环境下才生效, 线上代码无效

- 5-5 WebpackDevServer 解决单页面应用路由问题 p9 02:13:13

参考文档: <https://webpack.js.org/configuration/dev-server/#devserverhistoryapifallback>

参考文档2: <https://github.com/bripkens/connect-history-api-fallback>

测试代码安装: Npm install -save react-router-dom

Webpack.config.js 配置

这里只是开发模式下的配置, 上线后路由的跳转可能还需要后台nginx或者Apache等配置

Devserver 下:

```

// 这里只是开发模式下的配置, 上线后路由的跳转可能还需要后台nginx或者Apache等配置
historyApiFallback :true, // 对路径的请求转发到根路径的请求
// historyApiFallback : {
//   rewrites:[
//     {
//       from: /abc.html/, to: '/index.html'
//     },
//     { // 复杂一点的跳转逻辑
//       from: /\^\/libs\/.*$/,
//       to: function(context) {

```

```
//          return '/bower_components' + context.parsedUrl.pathname;
//      }
//  }
//  ]
// },
```

- Index.js

```
// import "@babel/polyfill"; // 不引入, 英文.babelrc中已经配置了 "useBuiltIns": "usage", 会自动加载polyfill
import React, {Component} from 'react';
import {BrowserRouter, Route } from 'react-router-dom'
import ReactDOM from 'react-dom';
import Home from './home'
import List from './list'

class App extends Component{
  render() {
    return(
      <BrowserRouter>
        <div>
          <Route path="/" exact component={Home}/>
          <Route path="/list" component={List}/>
        </div>
      </BrowserRouter>
    )
  }
}
ReactDOM.render(<App />, document.getElementById('root'));
```

- home.js

```
import React, {Component} from 'react';
import ReactDOM from 'react-dom';
class Home extends Component {
  render() {
    return <div>HomePage</div>
  }
}
export default Home;
```

- List.js

```
import React, {Component} from 'react';
import ReactDOM from 'react-dom';
class List extends Component {
  render() {
    return <div>ListPage</div>
  }
}
export default List;
```

本节主要配置: historyApiFallback :true

- 5-6 ESLint 在 Webpack 中的配置 (1) p9 02:28:05

Npm install eslint --save-dev

Npx eslint --init

版本和老师的不大一样

npx eslint src 检测代码

- 5-7 ESLint 在 Webpack 中的配置 (2)

视频不完整

- 5-8 webpack 性能优化 (1)

一、提升webpack打包速度的方法

1. 跟上技术的迭代 (Node, Npm, Yarn) (升级这些工具的版本)

2. 在尽可能少的模块上应用Loader

例如: `exclude:/node_modules/, //5-8`

3. Plugin 尽可能精简并确保可靠

```
optimization: {
  minimizer: [ new OptimizeCSSAssetsPlugin({}) ], // 代码压缩, 开发环境下 (dev) 则不使用, 从而提高了打包速度
},
```

5-9 webpack 性能优化(2)

4. resolve参数合理配置

```
resolve: {
  extensions: ['.js', '.jsx'], // 引入的时候, 先找js对应的文件, 然后再找jsx对应的文件
  // extensions: ['.css', '.jpg', '.js', '.jsx'] // 会执行多次查找, 性能损耗较大, 逻辑代码可配置在resolve中
  // mainFiles: ['index', 'child'], // 引入目录的时候, 先找index开头对应的文件, 然后再找child开头对应的文件, 同样会带来性能问题, 按需配置, 一般不配置
  alias: { // 别名
    delllee: path.resolve(__dirname, '../src/child/child') // 例如child在 ../src/a/b/c/child 目录下 这里配置就比较方便
  }
},
```

Index.js

```
import React, {Component} from 'react';
import ReactDOM from 'react-dom';
// import Child from './child/child'; // './child/child' webpack.common.js 中配置resolve.extensions
// import Child from './child/'; // webpack.common.js 中配置resolve.mainFiles
import Child from 'delllee'; // webpack.common.js 中配置resolve.alias 指向要引入的目录
class App extends Component {
  render() {
    return (
      <div>
        <div>This is App</div>
        <Child/>
      </div>
    )
  }
}
ReactDOM.render(<App/>, document.getElementById('root'));
```

5-10 Webpack 性能优化(3)

5. 使用DllPlugin提高打包速度

Npm run build 结果:

```
> lesson2@1.0.0 build D:\workspace\webpack4Study\lesson5-10
> webpack --env.production --config ./build/webpack.common.js
```

Hash: 34484500a12979579e85

Version: webpack 4.41.1

Time: 1351ms

Built at: 2019-11-15 7:44:49

| Asset | Size | Chunks | Chunk Names |
|------------------|-----------|-------------------|-------------|
| index.html | 337 bytes | [emitted] | |
| main.chunk.js | 2.25 KiB | main [emitted] | main |
| runtime.js | 31.1 KiB | runtime [emitted] | runtime |
| vendors.chunk.js | 2.73 MiB | vendors [emitted] | vendors |

Entrypoint main = runtime.js vendors.chunk.js main.chunk.js
[./src/index.js] 303 bytes {main} [built]
+ 11 hidden modules

Child html-webpack-plugin for "index.html":
1 asset
Entrypoint undefined = index.html
[./node_modules/html-webpack-plugin/lib/loader.js!./src/index.html] 373 bytes {0} [built]
[./node_modules/webpack/buildin/global.js] (webpack)/buildin/global.js 472 bytes {0} [built]
[./node_modules/webpack/buildin/module.js] (webpack)/buildin/module.js 497 bytes {0} [built]
+ 1 hidden module

Index.js 中引入loadsh

```
import _ from 'lodash';  
<div>{_.join(['This', 'is', 'App'], '')}</div>
```

重新打包: npm run build

打包结果: Time: 1618ms 打包时间增加了

****优化思想:** 把引入的包打包成一个文件, 第一次的时候做分析, 后面重新打包的话就不再分析

Webpack.dll.js 把引入的文件打包成一个文件vendors.dll.js

Webpack.dll.js 配置

```
const path = require('path');  
module.exports = {  
  mode: "production",  
  entry: {  
    vendors: ['react', 'react-dom', 'lodash']  
  },  
  output: {  
    filename: '[name].dll.js',  
    path: path.resolve(__dirname, '../dll'),  
    library: "[name]" // 打包生成的vendors.dll.js 通过vendors这个全局变量暴露出来  
  }  
}
```

Package.json 中:

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

Webpack.common.js 中加入add-asset-html-webpack-plugin

npm install add-asset-html-webpack-plugin --save

```
const AddAssetHtmlWebpackPlugin = require('add-asset-html-webpack-plugin');  
  
plugins: [  
  new HtmlWebpackPlugin({  
    template: 'src/index.html' // 使用的模板  
  }), //htmlWebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中  
  new CleanWebpackPlugin(), //打包前先清除dist目录,  
  new webpack.ProvidePlugin({  
    $: 'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量  
    _: 'lodash'  
    __join: ['lodash', 'join']  
  }),  
  new AddAssetHtmlWebpackPlugin({  
    filepath: path.resolve(__dirname, '../dll/vendors.dll.js') // 往HtmlWebpackPlugin 生成的html加什么  
  })  
],
```

****完成目标:** 第三方模块只是打包一次

**** 引入第三方模块的时候, 要去使用dll文件引入**

更改webpack.dll.js

```
const path = require('path');  
const webpack = require('webpack');  
module.exports = {  
  mode: "production",  
  entry: {  
    vendors: ['react', 'react-dom', 'lodash']  
  },  
  output: {  
    filename: '[name].dll.js',  
    path: path.resolve(__dirname, '../dll'),  
    library: "[name]" // 打包生成的vendors.dll.js 通过vendors这个全局变量暴露出来  
  },  
  plugins: [  
    new webpack.DllPlugin({
```

```

// 对dll的库进行分析, 分析的结果放在path 下
name: '[name]',
path: path.resolve(__dirname, '../dll/[name].manifest.json')
})
]
}

```

更改webpack.common.js

```

plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlwebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin(), //打包前先清除dist目录,
  new webpack.ProvidePlugin({
    $: 'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量
    // _: 'lodash'
    __join: ['lodash', 'join']
  }),
  new AddAssetHtmlWebpackPlugin({
    filepath: path.resolve(__dirname, '../dll/vendors.dll.js') // 往HtmlWebpackPlugin 生成的html加什么
  }),
  new webpack.DllReferencePlugin({
    // 引入的第三方模块的时候, 到 ../dll/vendors.manifest.json 找寻映射关系, 找到映射关系, 就直接从vendors.dll.js (全局变量中拿) 中获取
    manifest: path.resolve(__dirname, '../dll/vendors.manifest.json')
  })
],

```

重新打包: npm run build

打包结果: Time: 932ms 打包时间减少了

- 5-11 Webpack 性能优化 (4)
对上一章节的复习

Dll 扩展成两个文件:

Webpack.dll.js

```

entry: {
  vendors: ['lodash'],
  react: ['react', 'react-dom'] // 拆解成两个dll文件
},

```

Webpack.common.js

```

plugins: [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlwebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin(), //打包前先清除dist目录,
  new webpack.ProvidePlugin({
    $: 'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量
    // _: 'lodash'
    __join: ['lodash', 'join']
  }),
  new AddAssetHtmlWebpackPlugin({
    filepath: path.resolve(__dirname, '../dll/vendors.dll.js') // 往HtmlWebpackPlugin 生成的html加什么
  }),
  new AddAssetHtmlWebpackPlugin({
    filepath: path.resolve(__dirname, '../dll/react.dll.js') // 往HtmlWebpackPlugin 生成的html加什么
  }),
  new webpack.DllReferencePlugin({
    // 引入的第三方模块的时候, 到 ../dll/vendors.manifest.json 找寻映射关系, 找到映射关系, 就直接从vendors.dll.js (全局变量中拿) 中获取
    manifest: path.resolve(__dirname, '../dll/vendors.manifest.json')
  })
],

```



```

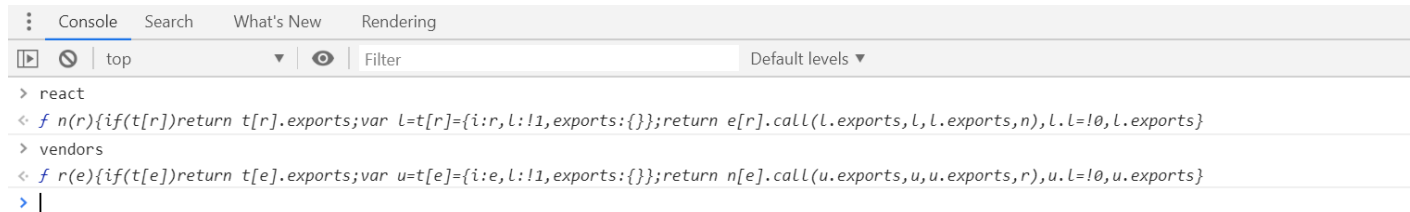
    }),
    new webpack.DllReferencePlugin({
      manifest: path.resolve(__dirname, '../dll/react.manifest.json')
    }),
  ],

```

Npm run build:dll 打包dll

Npm run build 打包

验证结果:



对上面拆分示例的优化:

Webpack.common.js

```

const path = require('path');
const fs = require('fs');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const {CleanWebpackPlugin} = require('clean-webpack-plugin');
const AddAssetHtmlWebpackPlugin = require('add-asset-html-webpack-plugin');
const webpack = require('webpack');
const merge = require('webpack-merge');
const devConfig = require('./webpack.dev');
const prodConfig = require('./webpack.prod');
const plugins = [
  new HtmlWebpackPlugin({
    template: 'src/index.html' // 使用的模板
  }), //htmlWebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new CleanWebpackPlugin(), //打包前先清除dist目录,
  new webpack.ProvidePlugin({
    $: 'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量
    // _: 'lodash'
    __join: ['lodash', 'join']
  })
];
const files = fs.readdirSync(path.resolve(__dirname, '../dll'));
// console.log(files);
files.forEach(file => {
  if(/.*\.dll\.js/.test(file)) {
    plugins.push(new AddAssetHtmlWebpackPlugin({
      filepath: path.resolve(__dirname, '../dll', file) // 往HtmlWebpackPlugin 生成的html加什么
    }));
  }
  if(/.*\.manifest\.json/.test(file)) {
    plugins.push(new webpack.DllReferencePlugin({
      manifest: path.resolve(__dirname, '../dll/', file)
    }));
  }
});
const commonConfig = {
  entry: {
    main: './src/index.js'
  },
  resolve: {
    extensions: ['.js', '.jsx'], // 引入的时候, 先找js对应的文件, 然后再找jsx对应的文件
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,

```

```

    exclude: /node_modules/, // 5-8
    // include: path.resolve(__dirname, '../src'),
    use: [
      {
        loader: "babel-loader", // 只是桥梁作用, 还需要 @babel/preset-env
      }
    ]
  },
  {
    test: /\. (jpg|png|gif)$/ , //正则, 以. jpg结尾的文件
    use: {
      // loader: 'file-loader',
      loader: 'url-loader', //把图片打包成base64的字符串
      options: {
        // placeholder 占位符
        // name: '[name].[ext]', //name 指原来的文件名, ext 文件后缀
        name: '[name]_[hash].[ext]', //name 指原来的文件名, ext 文件后缀
        outputPath: 'images/', // 将图片打包至images目录下
        limit: 2048 // >2kb打包成文件, 否则打包成base64的字符串
      }
    }
  }, {
    test: /\. (eot|ttf|svg|woff2|woff)$/ , //正则, 以. jpg结尾的文件
    use: {
      loader: 'file-loader',
    }
  }
]
},
plugins, //键和值一样可以简略, 等价 plugins: plugins
optimization: {
  runtimeChunk: {
    name: 'runtime'
  },
  usedExports : true , // 哪些导出的模块被使用了, 再做打包, tree shaking 相关
  splitChunks: {
    chunks: 'all', //async只对异步代码生效, initial只对同步代码生效, all同步异步等进行代码分割 (vendors配置按照官网配置可生效)

    cacheGroups: {
      vendors: {
        test: /[\\/]node_modules[\\/]/, //引入的库是否在node_modules目录下的
        priority: -10, //值越大, 优先级越高
        name: 'vendors' //
      }
    }
  }
},
performance: false, // 去除文件大于244kb的警告
}
module.exports = (env) => {
  // if(env && env.production) {
  if(env && env.production === "abc") {
    return merge(commonConfig, prodConfig);
  } else {
    return merge(commonConfig, devConfig);
  }
}
}

```

这样在webpack.dll.js添加任何模块就不用再修改webpack.common.js 代码了

```

entry: {
  vendors: ['lodash'],
  react: ['react', 'react-dom'], // 拆解成两个dll文件
  jquery: ['jquery']
},

```

- 5-12 webpack性能优化(5)
 6. 控制包文件大小
 7. Thread-loader,parallel0webpack,happypack多进程打包
 8. 合理使用sourceMap
 9. 结合stats分析打包结果
 10. 开发环境内存编译 (例如webpack-dev-server)
 11. 开发环境无用插件剔除 (--save-dev)

- 5-13 多页面打包配置

参考文档: <https://github.com/jantimon/html-webpack-plugin>

主要是看HtmlWebpackPlugin 配置

Webpack.common.js

```
const path = require('path');
const fs = require('fs');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const {CleanWebpackPlugin} = require('clean-webpack-plugin');
const AddAssetHtmlWebpackPlugin = require('add-asset-html-webpack-plugin');
const webpack = require('webpack');
const merge = require('webpack-merge');
const devConfig = require('./webpack.dev');
const prodConfig = require('./webpack.prod');
const plugins = [
  new HtmlWebpackPlugin({
    template: 'src/index.html', // 使用的模板
    filename: 'index.html',
    chunks: ['runtime', 'vendors', 'main']
  }), //htmlWebpackPlugin 会在打包结束后, 自动生成一个html文件, 并把打包生成的js自动引入到这个html文件中
  new HtmlWebpackPlugin({
    template: 'src/index.html', // 使用的模板
    filename: 'list.html',
    chunks: ['runtime', 'vendors', 'list']
  }),
  new CleanWebpackPlugin(), //打包前先清除dist目录,
  new webpack.ProvidePlugin({
    $: 'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量
    // _: 'lodash'
    _join: ['lodash', 'join']
  })
];
const files = fs.readdirSync(path.resolve(__dirname, '../dll'));
// console.log(files);
files.forEach(file=>{
  if(/.*\.dll\.js/.test(file)){
    plugins.push(new AddAssetHtmlWebpackPlugin({
      filepath: path.resolve(__dirname, '../dll', file) // 往HtmlWebpackPlugin 生成的html加什么
    }));
  }
  if(/.*\.manifest\.json/.test(file)){
    plugins.push(new webpack.DllReferencePlugin({
      manifest: path.resolve(__dirname, '../dll', file)
    }));
  }
});
const commonConfig = {
  entry: {
    main: './src/index.js',
    list: './src/list.js'
  },
  resolve: {
    extensions: ['.js', '.jsx'], // 引入的时候, 先找js对应的文件, 然后再找jsx对应的文件
  },
  module: {
```

```

rules: [
  {
    test: /\.jsx?$/,
    exclude: /node_modules/, // 5-8
    // include: path.resolve(__dirname, '../src'),
    use: [
      {
        loader: "babel-loader", // 只是桥梁作用, 还需要 @babel/preset-env
      }
    ]
  },
  {
    test: /\. (jpg|png|gif) $/, //正则, 以. jpg结尾的文件
    use: {
      // loader: 'file-loader',
      loader: 'url-loader', //把图片打包成base64的字符串
      options: {
        // placeholder 占位符
        // name: '[name].[ext]', //name 指原来的文件名, ext 文件后缀
        name: '[name].[hash].[ext]', //name 指原来的文件名, ext 文件后缀
        outputPath: 'images/', // 将图片打包至images目录下
        limit: 2048 // >2kb打包成文件, 否则打包成base64的字符串
      }
    }
  },
  {
    test: /\. (eot|ttf|svg|woff2|woff) $/, //正则, 以. jpg结尾的文件
    use: {
      loader: 'file-loader',
    }
  }
],
plugins, //键和值一样可以简略, 等价 plugins: plugins
optimization: {
  runtimeChunk: {
    name: 'runtime'
  },
  usedExports: true, // 哪些导出的模块被使用了, 再做打包, tree shaking 相关
  splitChunks: {
    chunks: 'all', //async只对异步代码生效, initial只对同步代码生效, all同步异步等进行代码分割 (vendors配置按照官网配置可生效)
  },
  cacheGroups: {
    vendors: {
      test: /[\\/]node_modules[\\/]$/, //引入的库是否在node_modules目录下的
      priority: -10, //值越大, 优先级越高
      name: 'vendors' //
    }
  }
},
performance: false, // 去除文件大于244kb的警告
}
module.exports = (env) => {
  // if(env && env.production) {
  if(env && env.production === "abc") {
    return merge(commonConfig, prodConfig);
  } else {
    return merge(commonConfig, devConfig);
  }
}
}

```

配置代码优化, 可动态增加detail页面:

```

const path = require('path');
const fs = require('fs');
const HtmlWebpackPlugin = require('html-webpack-plugin');

```

```

const {CleanWebpackPlugin }= require('clean-webpack-plugin');
const AddAssetHtmlWebpackPlugin = require('add-asset-html-webpack-plugin');
const webpack = require('webpack');
const merge = require('webpack-merge');
const devConfig = require('./webpack.dev');
const prodConfig = require('./webpack.prod');
const makePlugins =(configs)=>{
  const plugins=[
    new CleanWebpackPlugin(), //打包前先清除dist目录,
    new webpack.ProvidePlugin({
      $:'jquery', // 如果模块中使用了$字符串, 就在模块里自动引入jquery这个模块, 然后把jquery赋值给$这个变量
      // _: 'lodash'
      _join: ['lodash', 'join']
    })
  ];
  console.log(configs.entry);
  console.log(Object.keys(configs.entry));
  Object.keys(configs.entry).forEach(item=>{
    plugins.push(
      new HtmlWebpackPlugin({
        template: 'src/index.html', // 使用的模板
        filename: `${item}.html`,
        chunks: ['runtime', 'vendors', item]
      })
    )
  })
  const files = fs.readdirSync(path.resolve(__dirname, '../dll'));
  files.forEach(file=>{
    if(/.*\.dll\.js/.test(file)){
      plugins.push(new AddAssetHtmlWebpackPlugin({
        filepath: path.resolve(__dirname, '../dll', file) // 往HtmlWebpackPlugin 生成的html加什么
      }))
    }
    if(/.*\.manifest\.json/.test(file)){
      plugins.push(new webpack.DllReferencePlugin({
        manifest: path.resolve(__dirname, '../dll', file)
      }))
    }
  })
  return plugins;
}

// const commonConfig = {
const configs = {
  entry: {
    index: './src/index.js',
    list: './src/list.js',
    detail: './src/detail.js'
  },
  resolve: {
    extensions: ['.js', '.jsx'], // 引入的时候, 先找js对应的文件, 然后再找jsx对应的文件
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/, // 5-8
        // include: path.resolve(__dirname, '../src'),
        use: [
          {
            loader: "babel-loader", // 只是桥梁作用, 还需要 @babel/preset-env
          }
        ]
      },
      {
        test: /\. (jpg|png|gif) $/, //正则, 以 .jpg结尾的文件
        use: {
          // loader: 'file-loader',

```

```

    loader: 'url-loader', //把图片打包成base64的字符串
    options: {
      // placeholder 占位符
      // name: '[name].[ext]', //name 指原来的文件名, ext 文件后缀
      name: '[name].[hash].[ext]', //name 指原来的文件名, ext 文件后缀
      outputPath: 'images/', // 将图片打包至images目录下
      limit: 2048 // >2kb打包成文件, 否则打包成base64的字符串
    }
  }, {
    test: /\. (eot|ttf|svg|woff2|woff)$/, //正则, 以. jpg结尾的文件
    use: {
      loader: 'file-loader',
    }
  }
]
},
optimization: {
  runtimeChunk: {
    name: 'runtime'
  },
  usedExports: true, // 哪些导出的模块被使用了, 再做打包, tree shaking 相关
  splitChunks: {
    chunks: 'all', //async只对异步代码生效, initial只对同步代码生效, all同步异步等进行代码分割 (vendors配置按照官网配置可生效)
    cacheGroups: {
      vendors: {
        test: /[\\/]node_modules[\\/]$/, //引入的库是否在node_modules目录下的
        priority: -10, //值越大, 优先级越高
        name: 'vendors' //
      }
    }
  },
  performance: false, // 去除文件大于244kb的警告
}
configs.plugins = makePlugins(configs);
// const commonConfig =configs;
module.exports = (env) => {
  // if(env && env.production) {
  if(env && env.production === "abc") {
    return merge(configs, prodConfig);
  } else {
    return merge(configs, devConfig);
  }
}
}

```

• 第6章 Webpack 底层原理及脚手架工具分析

本章首先讲解如何自己实现 Webpack 中的 Loader 和 Plugin 的扩展。在了解 Webpack 扩展机制后, 进一步深入 Webpack 底层, 通过编码, 实现了类似 Webpack 的简单打包工具, 在此过程中, 让同学们能够真正理解打包过程中的各种复杂概念及底层原理。...

- 6-1 如何编写一个 Loader (1)
 - 简单loader

npm install webpack webpack-cli loader-utils --save-dev

Package.json

```

{
  "name": "lesson6-1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",

```

```

    "scripts": {
      "build": "webpack"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "loader-utils": "^1.2.3",
      "webpack": "^4.41.2",
      "webpack-cli": "^3.3.10"
    }
  }
}

```

Index.js

```
console.log("Hello dell");
```

Webpack.config.js

```

const path = require('path');
module.exports = {
  mode: 'development',
  entry: {
    main: './src/index.js'
  },
  module: {
    rules: [{
      test: /\.js$/,
      // use: [path.resolve(__dirname, './loaders/replaceloader.js')]
      use: [{
        loader: path.resolve(__dirname, './loaders/replaceloader.js'),
        options: {
          name: 'lee'
        }
      }]
    }]
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].js'
  }
}

```

Replaceloader.js 通过loader，更换hello dell中把dell 换成相应的词

```

const loaderUtils = require('loader-utils');
// 这里不能用箭头函数，一定用function
module.exports = function (source) {
  /* 示例1
  console.log(this.query);
  return source.replace('dell', this.query.name);
  */

  /*//示例2
  const options = loaderUtils.getOptions(this);
  return source.replace('dell', options.name);
  */

  // 示例3
  const options = loaderUtils.getOptions(this);
  const result = source.replace('dell', options.name);
  this.callback(null, result); // 传递信息出去
  // this.callback 参考https://webpack.js.org/api/loaders/#thiscallback
  // this.callback(null, result, source, meta);
  // this.callback(
  //   err: Error | null,
  //   content: string | Buffer,
  //   sourceMap?: SourceMap,
  //   meta?: any
  // );

```

```
}
```

参考文档: <https://webpack.js.org/api/loaders/#thisquery>

- 6-2 如何编写一个 Loader (2)

- 异步loader

Module build failed: Error: Final loader (./loaders/replaceloader.js) didn't return a Buffer or String
使用this.async();

replaceloaderAsync.js:

```
const loaderUtils = require('loader-utils');  
// 这里不能用箭头函数, 一定用function  
module.exports = function (source) {  
  const options = loaderUtils.getOptions(this);  
  const callback = this.async();  
  setTimeout(()=>{  
    const result = source.replace('dell', options.name);  
    callback(null, result);  
  }, 1000);  
}
```

- 多个loader, 注意使用顺序, 从下到上, 从右到左

例如: 把dell 替换成lee, 再把lee替换成world

replaceLoaderAsync.js 与上面相同

replaceLoader.js:

```
/*  
 * @Author: your name  
 * @Date: 2019-12-02 09:26:52  
 * @LastEditTime: 2019-12-02 10:36:19  
 * @LastEditors: Please set LastEditors  
 * @Description: In User Settings Edit  
 * @FilePath: \webpack4Study\lesson6-2\loaders\replaceloader.js  
 */  
const loaderUtils = require('loader-utils');  
// 这里不能用箭头函数, 一定用function  
module.exports = function (source) {  
  return source.replace('lee', 'world');  
}
```

Webpack.config.js:

```
const path = require('path');  
module.exports = {  
  mode: 'development',  
  entry: {  
    main: './src/index.js'  
  },  
  resolveLoader: {  
    modules: ['node_modules', './loaders'] // loader先从node_modules中找, 在从loaders目录中找  
  },  
  module: {  
    rules: [{  
      test: /\.js$/,  
      // use: [path.resolve(__dirname, './loaders/replaceLoader.js')]  
      use: [  
        {  
          // loader: path.resolve(__dirname, './loaders/replaceLoader.js')  
          loader: 'replaceLoader'  
        },  
        {  
          loader: 'replaceLoaderAsync',  
          options: {  
            name: 'lee'  
          }  
        }  
      ]  
    }]  
  }  
}
```



```

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

```

运用场景：异常捕获，国际化

- 6-3 如何编写一个 Plugin
 - 参考文档： <https://webpack.js.org/api/compiler-hooks/>

例子：实现在打包时，在dist目录下增加一个copyright.txt

Npm install webpac webpack-cli -D

Webpack.config.js

```

const path = require('path');
const CopyRightWebpackPlugin = require('./plugins/copyright-webpack-plugin')
module.exports = {
  mode: 'development',
  entry: {
    main: './src/index.js'
  },
  plugins: [
    new CopyRightWebpackPlugin({
      name: 'dell'
    })
  ],
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].js',
  }
}

```

Copyright-webpack-plugin.js

```

class CopyrightWebpackPlugin {
  constructor(options) {
    console.log("插件被使用了！");
    console.log(options);
  }
  apply(compiler) {
    // 参考文档： https://webpack.js.org/api/compiler-hooks/
    // compiler 存储了webpack的实例
    // console.log(compiler.hooks);
    // 同步时刻
    compiler.hooks.compile.tap('CopyrightWebpackPlugin', (compilation) => {
      console.log("同步时刻 compiler");
    })
    // 异步时刻
    compiler.hooks.emit.tapAsync('CopyrightWebpackPlugin', (compilation, cb) => {
      // 在代码增加到dist目录之前，又增加了一个文件copyright.txt
      // console.log(compilation.assets);

      // debugger; 用于node 调试
      compilation.assets["copyright.txt"] = {
        source: function() {
          return 'copyright by dell lee'
        },
        size: function() {
          return 21;
        }
      };
      cb();
    })
  }
}

```

```

}
module.exports = CopyrightWebpackPlugin;

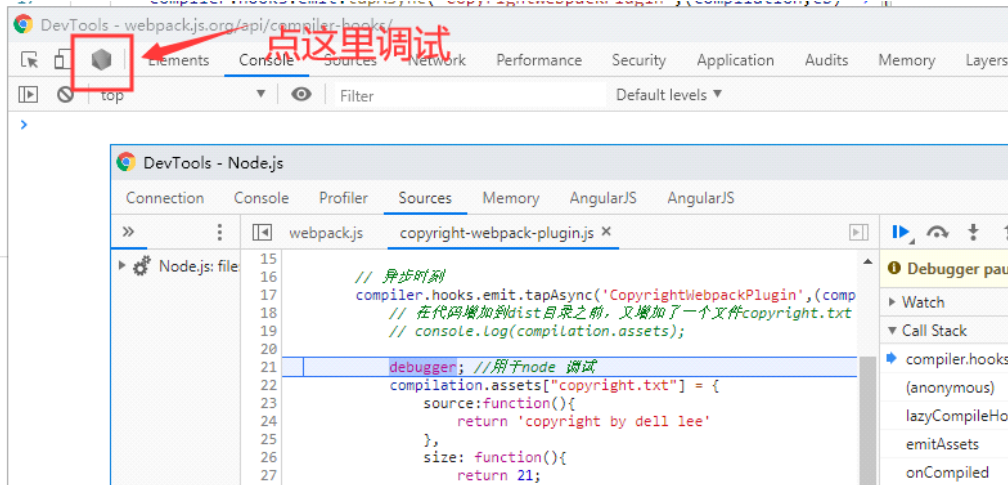
```

Node 调试方法:

Package.json中:

"debug": "node --inspect --inspect-brk node_modules/webpack/bin/webpack.js",

Node --inspect 调试模式 --inspect-brk 第一行打一个断点



- 6-4 Bundler 源码编写 (模块分析 1)

- 6-5 Bundler 源码编写 (模块分析 2)

例子: 对入口文件的代码进行分析

npm install cli-highlight -g npm 运行结果颜色显示

Node bundler.js | highlight 运行结果颜色显示

npm install @babel/parser --save 获取抽象语法树

npm install @babel/traverse --save

npm install @babel/core --save

npm install @babel/preset-env --save

Message.js:

```

import { word } from './word.js';
const message = `say ${word}`;
export default message;

```

Word.js

```

export const word = 'hello';

```

Index.js

```

import message from './message';
import message1 from './message1';
console.log(message);

```

Bundle.js

```

/*
 * @Author: Admin
 * @Date: 2019-12-21 16:43:56
 * @FilePath: \webpack4Study\lesson6-4\bundler.js
 * @Description: file content
 */
const fs = require('fs');
const path = require('path');
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;

```

```

const babel = require('@babel/core');
const moduleAnalyser = filename => {
  const content = fs.readFileSync(filename, 'utf-8');
  // 分析出抽象语法树
  const ast = parser.parse(content, {
    sourceType: 'module',
  });
  // 依赖的文件
  const dependencies = {};
  traverse(ast, {
    ImportDeclaration({ node }) {
      // 获取当前目录
      const dirname = path.dirname(filename);
      const newFile = './' + path.join(dirname, node.source.value); // ./相对于点前bundle.js文件目录
      dependencies[node.source.value] = newFile;
    },
  });
  // console.log(dependencies);
  // console.log(dependencies);
  // console.log(ast.program.body);
  // console.log(content);
  // 使代码能在浏览器上运行
  const { code } = babel.transformFromAst(ast, null, {
    presets: ['@babel/preset-env'],
  });
  // console.log(code);
  return {
    filename,
    dependencies,
    code,
  };
};
const moduleInfo = moduleAnalyser('./src/index.js');
console.log(moduleInfo);

```

- 6-6 Bundler 源码编写 (Dependencies Graph)

依赖图谱

node .\bundler.js | highlight

其它代码参照上例

Bundle.js

```

/*
 * @Author: Admin
 * @Date: 2019-12-21 16:43:56
 * @FilePath: \webpack4Study\lesson6-4\bundler.js
 * @Description: file content
 */
const fs = require('fs');
const path = require('path');
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const babel = require('@babel/core');
const moduleAnalyser = filename => {
  const content = fs.readFileSync(filename, 'utf-8');
  // 分析出抽象语法树
  const ast = parser.parse(content, {
    sourceType: 'module',
  });
  // 依赖的文件
  const dependencies = {};
  traverse(ast, {
    ImportDeclaration({ node }) {
      // 获取当前目录
      const dirname = path.dirname(filename);

```

```

    const newFile = './' + path.join(dirname, node.source.value); // 相对于点前bundle.js文件目录
    dependencies[node.source.value] = newFile;
  },
});
// console.log(dependencies);
// console.log(dependencies);
// console.log(ast.program.body);
// console.log(content);
// 使代码能在浏览器上运行
const { code } = babel.transformFromAst(ast, null, {
  presets: ['@babel/preset-env'],
});
// console.log(code);
return {
  filename,
  dependencies,
  code,
};
};
};
const makeDependenciesGraph = entry => {
  const entryModule = moduleAnalyser(entry);
  const graphArray = [entryModule];
  for (let i = 0; i < graphArray.length; i++) {
    const item = graphArray[i];
    const { dependencies } = item;
    if (dependencies) {
      for (let j in dependencies) {
        // 关键 有点类似递归, 获取当前文件import的文件, 再添加到数组中再进行分析
        graphArray.push(moduleAnalyser(dependencies[j]));
      }
    }
  }
  // console.log(graphArray);
  const graph = {};
  graphArray.forEach(item => {
    graph[item.filename] = {
      dependencies: item.dependencies,
      code: item.code,
    };
  });
  // console.log(graph);
  return graph;
};
const graphInfo = makeDependenciesGraph('./src/index.js');
console.log(graphInfo);

```

- 6-7 Bundler 源码编写（生成代码）

其它代码参照上例

注意：其它代码中不能有注释等信息，会任务是非法字符，影响执行

Bundle.js

```

/*
 * @Author: Admin
 * @Date: 2019-12-21 16:43:56
 * @FilePath: \webpack4Study\lesson6-4\bundler.js
 * @Description: file content
 */
const fs = require('fs');
const path = require('path');
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const babel = require('@babel/core');
const moduleAnalyser = filename => {
  const content = fs.readFileSync(filename, 'utf-8');
  // 分析出抽象语法树
  const ast = parser.parse(content, {

```

```

    sourceType: 'module',
  });
  // 依赖的文件
  const dependencies = {};
  traverse(ast, {
    ImportDeclaration({ node }) {
      // 获取当前目录
      const dirname = path.dirname(filename);
      const newFile = './' + path.join(dirname, node.source.value); // ./相对于当前bundle.js文件目录
      dependencies[node.source.value] = newFile;
    },
  });
  // console.log(dependencies);
  // console.log(dependencies);
  // console.log(ast.program.body);
  // console.log(content);
  // 使代码能在浏览器上运行
  const { code } = babel.transformFromAst(ast, null, {
    presets: ['@babel/preset-env'],
  });
  // console.log(code);
  return {
    filename,
    dependencies,
    code,
  };
};

const makeDependenciesGraph = entry => {
  const entryModule = moduleAnalyser(entry);
  const graphArray = [entryModule];
  for (let i = 0; i < graphArray.length; i++) {
    const item = graphArray[i];
    const { dependencies } = item;
    if (dependencies) {
      for (let j in dependencies) {
        // 关键 有点类似递归, 获取当前文件import的文件, 再添加到数组中再进行分析
        graphArray.push(moduleAnalyser(dependencies[j]));
      }
    }
  }
  // console.log(graphArray);
  const graph = {};
  graphArray.forEach(item => {
    graph[item.filename] = {
      dependencies: item.dependencies,
      code: item.code,
    };
  });
  // console.log(graph);
  return graph;
};

const generateCode = entry => {
  const graph = JSON.stringify(makeDependenciesGraph(entry));
  return `
(function(graph) {
  function require(module) {
    function localRequire(relativePath) {
      return require(graph[module].dependencies[relativePath])
    }
    var exports = {};
    (function(require, exports, code) {
      eval(code)
    })(localRequire, exports, graph[module].code);
    return exports;
  };
  require('${entry}')
})($graph);
`;
};

```

```
const code = generateCode('./src/index.js');
console.log(code);
```

// 把文件中的代码读取出来，再层层转意成可执行的代码

控制台打印出来的代码再贴到浏览器中执行，打印say hello

• 第7章 Create-React-App 和 Vue-Cli 3.0脚手架工具配置分析

最后一章增加了对 Create-React-App 和 Vue-Cli 3.0 两个前端脚手架工具中 Webpack 配置内容的分析，帮助同学了解不同脚手架工具设计的出发点，和配置的技巧。

- 7-1 通过 CreateReactApp 深入学习 Webpack 配置（1）

参考网址: <https://create-react-app.dev/>

npx create-react-app my-app

安装遇到错误:

```
error fork-ts-checker-webpack-plugin@3.1.0: The engine "yarn" is incompatible with this module. Expected version ">=1.0.0".
error Found incompatible module
```

原因是yarn版本本地1.0.0，重新卸载yarn安装yarn未成功，直接使用npm安装: npx create-react-app my-app --use-npm

- 文件准备:

Npm run eject 把隐藏的webpack，弹射出来，显示出来，会多出config和scripts两个文件夹

config/jest与scripts/test.js是自动化测试相关，做示例时暂时删除

Parkage.json 中删除 scripts.test 部分

- 重点看config目录下配置文件

分析build

```
"build": "node scripts/build.js"
```

Paths.js 配置项目路径

Env.js 初始化环境的文件

文件参考 lesson7-2/my-app/config 下配置

- 7-2 通过 CreateReactApp 深入学习 Webpack 配置（2）

Build.js 加载env.js path.js 文件

Webpack.config.js 核心配置

webpackDevServer.config.js webpackDevServer 配置

- 7-3 Vue CLI 3 的配置方法及课程总结（1）

Npm install -g @vue/cli

参考文档: <https://cli.vuejs.org/zh/config/#%E5%85%A8%E5%B1%80-cli-%E9%85%8D%E7%BD%AE>

Vue 本身有配置参数，再由vue框架配置成webpack的配置

- 7-4 Vue CLI 3 的配置方法及课程总结（2）

- Vue 与react 区别:

vue适合新手

react灵活

vue主要学习vue官网的配置

vue配置转webpack配置

lesson7-3\my-project\node_modules\@vue\cli-service\lib\Service.js vue配置转webpack配置

- 自己配置

建立vue.config.js文件中配置官网提供的配置，configureWebpack参数配置原生的配置

```
const path = require('path');
module.exports = {
  outputDir: 'dist',
```

```
// css: {  
//   modules:true  
// }  
// 原生配置, 定义到static里找内容  
// configureWebpack: {  
//   devServer: { contentBase: [path.resolve(__dirname, 'static')] },  
// },  
devServer: { contentBase: [path.resolve(__dirname, 'static')] },  
};
```

测试:

建立static/index.json

```
{ "abc": 123 }
```

<http://localhost:8080/index.json>

webpack官网 目录

Guides

- code Splitting 代码分割
- typescript

Concepts 概念

Configuration

- devServer hotonly 配置意思

Api

- loader plugins 自定义loader plugins api查询

Loaders

Plugins

本课程已完结

来自 <<https://coding.imooc.com/class/chapter/316.html#Anchor>>