Webpack-Day2~3

- loader: file-loader：处理静态资源模块

  loader: file-loader

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

  所以我们什么时候用file-loader呢?

  场景：就是当我们需要模块，仅仅是从源代码挪移到打包目录，就可以使用file-loader来处理，txt，svg，csv，excel，图片资源啦等等

  ```
  npm install file-loader -D
  ```

  案例：

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

```
            //打包后的存放位置
            outputPath: "images/"
          }
        }
      }
    ]
  },
```

```
import pic from "./logo.png";

var img = new Image();
img.src = pic;
img.classList.add("logo");

var root = document.getElementById("root");
root.append(img);
```

- 处理字体 https://www.iconfont.cn/?spm=a313x.7781069.1998910419.d4d0a486a

```
//css
@font-face {
  font-family: "webfont";
  font-display: swap;
  src: url("webfont.woff2") format("woff2");
}

body {
  background: blue;
  font-family: "webfont" !important;
}


//webpack.config.js
{
  test: /\.(eot|ttf|woff|woff2|svg)$/,
  use: "file-loader"
}
```

- url-loader file-loader加强版本

url-loader内部使用了file-loader,所以可以处理file-loader所有的事情，但是遇到jpg格式的模块，会把该图片转换成base64格式字符串，并打包到js里。对小体积的图片比较合适，大图片不合适。

```
npm install url-loader -D
```

案例；

```
module: {
    rules: [
      {
        test: /\.(png|jpe?g|gif)$/,
        use: {
          loader: "url-loader",
          options: {
            name: "[name]_[hash].[ext]",
            outputPath: "images/",
            //小于2048，才转换成base64
            limit: 2048
          }
        }
      }
    ]
  },
```

样式处理：

Css-loader 分析css模块之间的关系，并合成一个css

Style-loader 会把css-loader生成的内容，以style挂载到页面的heade部分

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

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


{
```

```
  test:/\.css$/,
    use:[{
      loader:"style-loader",
      options: {
              injectType: "singletonStyleTag" // 将所有的style标签合并成一个
            }
    },"css-loader"]
}
```

Less样式处理

less-load 把less语法转换成css

```
$ npm install less less-loader --save-dev
```

案例：

loader有顺序，从右到左，从下到上

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

样式自动添加前缀：

https://caniuse.com/

Postcss-loader

```
npm i  postcss-loader autoprefixer -D
```

新建postcss.config.js

```
//webpack.config.js
{
  test: /\.css$/,
  use: ["style-loader", "css-loader", "postcss-loader"]
},

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

```
      require("autoprefixer")({
        overrideBrowserslist: ["last 2 versions", ">1%"]
      })
    ]
};
```

loader 处理webpack不支持的格式文件，模块

一个loader只处理一件事情

loader有执行顺序

5.Plugins

- 作用于webpack打包整个过程
- webpack的打包过程是有（生命周期概念）钩子

plugin 可以在webpack运行到某个阶段的时候，帮你做一些事情，类似于生命周期的概念

扩展插件，在 Webpack 构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要的事情。

作用于整个构建过程

HtmlWebpackPlugin

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模块引入到该html中。

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

配置：

```
title: 用来生成页面的 title 元素
filename: 输出的 HTML 文件名，默认是 index.html, 也可以直接配置带有子目录。
template: 模板文件路径，支持加载器，比如 html!./index.html
inject: true | 'head' | 'body' | false  ,注入所有的资源到特定的 template 或者 templateContent 中，如果设置为 true 或者 body，所有的 javascript 资源将被放置到 body 元素的底部，'head' 将放置到 head 元素中。
favicon: 添加特定的 favicon 路径到输出的 HTML 文件中。
minify: {} | false , 传递 html-minifier 选项给 minify 输出
hash: true | false, 如果为 true, 将添加一个唯一的 webpack 编译 hash 到所有包含的脚本和 CSS 文件，对于解除 cache 很有用。
cache: true | false，如果为 true, 这是默认值，仅仅在文件修改之后才会发布文件。
showErrors: true | false, 如果为 true, 这是默认值，错误信息会写入到 HTML 页面中
chunks: 允许只添加某些块 (比如，仅仅 unit test 块)
chunksSortMode: 允许控制块在添加到页面之前的排序方式，支持的值：'none' | 'default' | {function}-default:'auto'
excludeChunks: 允许跳过某些块，(比如，跳过单元测试的块)
```

案例：

```
const path = require("path");
const htmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
 ...
  plugins: [
    new htmlWebpackPlugin({
      title: "My App",
      filename: "app.html",
      template: "./src/index.html"
    })
  ]
};


//index.html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <div id="root"></div>
```

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

clean-webpack-plugin

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

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

...

plugins: [
    new CleanWebpackPlugin()
]
```

mini-css-extract-plugin

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


{
  test: /\.css$/,
  use: [MiniCssExtractPlugin.loader, "css-loader"]
}



 new MiniCssExtractPlugin({
   filename: "[name][chunkhash:8].css"
 })
```

chunk：一个入口肯定是一个chunk，但是一个chunk不一定只有一个依赖

webpack5.x 减少hash的概念

Contenthash

6.sourceMap

1.为了开发时，快速的定位问题

2.线上代码，我们有时候也会开启 前端错误监控，快速的定位问题

源代码与打包后的代码的映射关系，通过sourceMap定位到源代码。

在dev模式中，默认开启，关闭的话 可以在配置文件里

```
devtool:"none"
```

devtool的介绍：https://webpack.js.org/configuration/devtool#devtool

eval:速度最快,使用eval包裹模块代码,

source-map： 产生`.map`文件 外部产生 错误代码的准确信息和位置

cheap:较快，不包含列信息

Module：第三方模块，包含loader的sourcemap（比如jsx to js ， babel的sourcemap）

inline： 将`.map`作为DataURI嵌入，不单独生成`.map`文件

配置推荐：

```
devtool:"cheap-module-eval-source-map",// 开发环境配置

//线上不推荐开启
devtool:"cheap-module-source-map",   // 线上生成配置
```

WebpackDevServer

- 提升开发效率的利器

每次改完代码都需要重新打包一次，打开浏览器，刷新一次，很麻烦,我们可以安装使用webpackdevserver来改善这块的体验

- 安装

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

- 配置

修改下package.json

```
"scripts": {
    "server": "webpack-dev-server"
  },
```

在webpack.config.js配置：

```
devServer: {
    contentBase: "./dist",
    open: true,
    port: 8081
  },
```

- 启动

```
npm run server
```

启动服务后，会发现dist目录没有了，这是因为devServer把打包后的模块不会放在dist目录下，而是放到内存中，从而提升速度

- 本地mock,解决跨域：

前后端分离

前端和后端是可以并行开发的，

前端会依赖后端的接口

先给接口文档，和接口联调日期的

我们前端就可以本地mock数据，不打断自己的开发节奏

UI 下周二 视觉妹子 出稿

服务端聊：

接口文档，定义字段 日期定下 两周后=》联调接口时间=完成交互，进入测试

首先，从性能角度上分析，几个接口比较好

服务端： 这个优先级靠后吧，先紧现在的接口用着

安心mock数据

可以安心的把精力放在业务上，不被扯淡时间拉散精力

开发模式：前后端分离

项目会议

联调期间，前后端分离，直接获取数据会跨域，上线后我们使用nginx转发，开发期间，webpack就可以搞定这件事

启动一个服务器，mock一个接口：

```
// npm i express -D
// 创建一个server.js 修改scripts "server":"node server.js"

//server.js
const express = require('express')

const app = express()

app.get('/api/info', (req,res)=>{
    res.json({
        name:'开课吧',
        age:5,
        msg:'欢迎来到开课吧学习前端高级课程'
    })
})

app.listen('9092')

//node server.js

http://localhost:9092/api/info
```

项目中安装axios工具

```
//npm i axios -D

//index.js
import axios from 'axios'
axios.get('http://localhost:9092/api/info').then(res=>{
    console.log(res)
})

会有跨域问题
```

修改webpack.config.js 设置服务器代理

```
proxy: {
      "/api": {
        target: "http://localhost:9092"
      }
    }
```

修改index.js

```
axios.get("/api/info").then(res => {
  console.log(res);
});
```

Hot Module Replacement (HMR:热模块替换)

- 不支持抽离出的css 我们要使用css-loader
-

启动hmr

```
devServer: {
    contentBase: "./dist",
    open: true,
    hot:true,
    //即便HMR不生效，浏览器也不自动刷新，就开启hotOnly
    hotOnly:true
  },
```

配置文件头部引入webpack

```
//const path = require("path");
//const HtmlWebpackPlugin = require("html-webpack-plugin");
//const CleanWebpackPlugin = require("clean-webpack-plugin");

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

在插件配置处添加：

```
plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      template: "src/index.html"
    }),
    new webpack.HotModuleReplacementPlugin()
  ],
```

案例：

```
//index.js
import "./css/index.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);
};


//index.css
div:nth-of-type(odd) {
  background: yellow;
}
```

注意启动HMR后，css抽离会不生效，还有不支持contenthash，chunkhash

处理js模块HMR

需要使用module.hot.accept来观察模块更新 从而更新

案例：

```
//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 = 13000;
  document.body.appendChild(div);
}
export default number;




//index.js

import counter from "./counter";
import number from "./number";

counter();
number();

if (module.hot) {
  module.hot.accept("./b", function() {
    document.body.removeChild(document.getElementById("number"));
    number();
  });
}
```

作业：

实现多页面打包通用方案

实现setMPA

Babel处理ES6

官方网站：https://babeljs.io/

中文网站：https://www.babeljs.cn/

Babel是JavaScript编译器，能将ES6代码转换成ES5代码，让我们开发过程中放心使用JS新特性而不用担心兼容性问题。并且还可以通过插件机制根据需求灵活的扩展。

Babel在执行编译的过程中，会从项目根目录下的`.babelrc`JSON文件中读取配置。没有该文件会从loader的options地方读取配置。

测试代码

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

arr.map(item => {
  console.log(item);
});
```

安装

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

`1.babel-loader是webpack 与 babel的通信桥梁，不会做把es6转成es5的工作，这部分工作需要用到@babel/preset-env来做`

`2.@babel/preset-env里包含了es，6，7，8转es5的转换规则`

Ecma 5 6 7 8... 草案（评审通过的，还有未通过的）

面向未来的

env是babel7之后推行的预设插件

env{

ecma 5

ecma 6

ecma 7

ecma 8

。。。

}

Webpack.config.js

```
{
  test: /\.js$/,
  exclude: /node_modules/,
```

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


  "browserslist": [
    "last 2 version",
    "> 1%",
    "not ie < 11",
    "cover 99.5%",
    "dead"
  ]
```

通过上面的几步 还不够，默认的Babel只支持let等一些基础的特性转换，Promise等一些还有转换过来，这时候需要借助@babel/polyfill，把es的新特性都装进来，来弥补低版本浏览器中缺失的特性

@babel/polyfill

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

```
npm install --save @babel/polyfill
```

```
//index.js 顶部
import "@babel/polyfill";
```

按需加载，减少冗余

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

当然可以

修改Webpack.config.js

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

```
            "@babel/preset-env",
            {
              targets: {
                edge: "17",
                firefox: "60",
                chrome: "67",
                safari: "11.1"
              },
              corejs: 2,//新版本需要指定核心库版本
              useBuiltIns: "entry"//按需注入
            }
          ]
        ]
      }
```

`useBuiltIns`选项是`babel 7`的新功能，这个选项告诉`babel`如何配置`@babel/polyfill`。 它有三个参数可以使用： ①entry: 需要在`webpack`的入口文件里`import "@babel/polyfill"`一次。`babel`会根据你的使用情况导入垫片，没有使用的功能不会被导入相应的垫片。 ②usage: 不需要`import`，全自动检测，但是要安装`@babel/polyfill`。（试验阶段） ③false: 如果你`import "@babel/polyfill"`，它不会排除掉没有使用的垫片，程序体积会庞大。(不推荐)

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

扩展：

babelrc文件：

新建.babelrc文件，把options部分移入到该文件中，就可以了

```
//.babelrc

{
  presets: [
    [
      "@babel/preset-env",
      {
        targets: {
          edge: "17",
          firefox: "60",
          chrome: "67",
          safari: "11.1"
        },
        corejs: 2, //新版本需要指定核心库版本
        useBuiltIns: "usage" //按需注入
```

```
      }
    ]
  ]
}


//webpack.config.js

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

暗号：做人嘛，最重要的是开心

作业：实现text-webpack-plugin 自定义插件

要求：提交代码截图 ，在emit阶段，往资源列表里插入一个新的txt文档，文档的内容和体积不限

配置React打包环境

安装

```
npm install react react-dom --save
```

解析插件

转换插件

编写react代码：

```
//index.js
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("app"));
```

安装babel与react转换的插件：

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

在babelrc文件里添加：

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "edge": "17",
          "firefox": "60",
          "chrome": "67",
          "safari": "11.1",
          "Android":"6.0"
        },
        "useBuiltIns": "usage", //按需注入
      }
    ],
    "@babel/preset-react"
  ]
}
```

如果是库的作者的话，提供模块的时候代码怎么打包的?

构建速度会越来越慢，怎么优化

扩展：

多页面打包通用方案

```
entry:{
  index:"./src/index",
  list:"./src/list",
  detail:"./src/detail"
}

new htmlWebpackPlugins({
  title: "index.html",
  template: path.join(__dirname, "./src/index/index.html"),
  filename:"index.html",
  chunks:[index]
})
```

1.目录结构调整

- src
  - index
    - index.js
    - index.html
  - list
    - index.js
    - index.html
  - detail
    - index.js
    - index.html
- 2.使用 glob.sync 第三方库来匹配路径

```
npm i glob -D

const glob = require("glob")
```

```
//MPA多页面打包通用方案

const setMPA = () => {
  const entry = {};
  const htmlWebpackPlugins = [];


  return {
    entry,
    htmlWebpackPlugins
```

```
  };
};

const { entry, htmlWebpackPlugins } = setMPA();
```

```
const setMPA = () => {
  const entry = {};
  const htmlWebpackPlugins = [];

  const entryFiles = glob.sync(path.join(__dirname, "./src/*/index.js"));

  entryFiles.map((item, index) => {
    const entryFile = entryFiles[index];
    const match = entryFile.match(/src\/(.*)\/index\.js$/);
    const pageName = match && match[1];
    entry[pageName] = entryFile;
    htmlWebpackPlugins.push(
      new htmlWebpackPlugin({
        title: pageName,
        template: path.join(__dirname, `src/${pageName}/index.html`),
        filename: `${pageName}.html`,
        chunks: [pageName],
        inject: true
      })
    );
  });
  return {
    entry,
    htmlWebpackPlugins
  };
};

const { entry, htmlWebpackPlugins } = setMPA();



module.exports = {
  entry,
  output:{
    path: path.resolve(__dirname, "./dist"),
    filename: "[name].js"
  }
  plugins: [
    // ...
```

```
    ...htmlWebpackPlugins//展开数组
  ]
}
```