一、webpack

> 前端项目工程化的具体解决方案，打包工具

1. webpack 基本使用

1. **项目中安装：**

```
npm install webpack@5.5.1 webpack-cli@4.2.0 -D
```

```
#使用最新版本,可以不指定版本
npm i webpack webpack-cli -D
```

2. **配置webpack**

① 在项目根目录中，创建名为 webpack.config.js 的 webpack 配置文件，并初始化如下的基本配置：

```
module.exports = {
  mode: 'development' // mode 用来指定构建模式。可选值有 development 和 production
}
```

② 在 package.json 的 scripts 节点下，新增 dev 脚本如下：

```
"scripts": {
  "dev": "webpack" // script 节点下的脚本，可以通过 npm run 执行。例如 npm run dev
}
```

③ 在终端中运行 npm run dev 命令，启动 webpack 进行项目的打包构建

3. **webpack.config.js 的作用**

webpack.config.js 是 webpack 的配置文件。webpack 在真正开始打包构建之前，会先读取这个配置文件，从而基于给定的配置，对项目进行打包。

注意：由于 webpack 是基于 node.js 开发出来的打包工具，因此在它的配置文件中，支持使用 node.js 相关的语法和模块进行 webpack 的个性化配置。

4. **自定义打包的入口和出口**

在 webpack 中有如下的默认约定：

① 默认的打包入口文件为 src -> index.js

② 默认的输出文件路径为 dist -> main.js

注意：可以在 webpack.config.js 中修改打包的默认约定

在 webpack.config.js 配置文件中，通过 entry 节点指定打包的入口。通过 output 节点指定打包的出口。示例代码如下：

```
const path = require('path') // 导入 node.js 中专门操作路径的模块

module.exports = {
  entry: path.join(__dirname, './src/index.js'), // 打包入口文件的路径
  output: {
    path: path.join(__dirname, './dist'), // 输出文件的存放路径
    filename: 'bundle.js' // 输出文件的名称
  }
}
```

2. webpack 的插件

通过安装和配置第三方的插件，可以拓展webpack的能力，从而让webpack用起来更方便。最常用的webpack插件有如下两个:

① webpack-dev-server

- 类似于 node.js 阶段用到的 nodemon 工具
- 每当修改了源代码，webpack 会自动进行项目的打包和构建

② html-webpack-plugin

- webpack 中的 HTML 插件（类似于一个模板引擎插件）
- 可以通过此插件自定制 index.html 页面的内容

1. webpack-dev-server

> js代码保存后，自动打包

- 1.1 安装

运行如下的命令，即可在项目中安装此插件：

```
npm install webpack-dev-server@3.11.0 -D
```

```
#使用最新版本：
npm i webpack-dev-server  -D
```

- 1.2 配置

① 修改 package.json -> scripts 中的 dev 命令如下：

```
"scripts": {
    "dev": "webpack serve", // script 节点下的脚本，可以通过 npm run 执行
}
```

② 再次运行 npm run dev 命令，重新进行项目的打包

- 1.3 打包生成的文件

> `ctrl` + `s` 后，就可自动打包

① 不配置 webpack-dev-server 的情况下，webpack 打包生成的文件，会存放到实际的物理磁盘上

- 严格遵守开发者在 webpack.config.js 中指定配置
- 根据 output 节点指定路径进行存放

② 配置了 webpack-dev-server 之后，打包生成的文件存放到了内存中

- 不再根据 output 节点指定的路径，存放到实际的物理磁盘上
- 提高了实时打包输出的性能，因为内存比物理磁盘速度快很多

- 1.4 使用

> 实时打包后的js文件放置在内存中，在项目根路径上，因此修改index.html中js的引入路径，`/` 表示根目录

```
<!-- 使用webpack-cli-server -->
<script src="/build.js"></script>
```

devServer 节点

> 实时处理的打包插件 `webpack-dev-server` 的配置项。
>
> 包括：打包完成是否自动打开浏览器；打包所使用的主机地址；打包所使用的端口号

在 webpack.config.js 配置文件中，可以通过 devServer 节点对 webpack-dev-server 插件进行更多的配置，示例代码如下：

```
devServer: {
    open: true, // 初次打包完成后，自动打开浏览器
    host: '127.0.0.1', // 实时打包所使用的主机地址
    port: 80, // 实时打包所使用的端口号
}
```

2. `html-webpack-plugin`

html-webpack-plugin 是 webpack 中的 HTML 插件，可以通过此插件自定制 index.html 页面的内容。

需求：通过 html-webpack-plugin 插件，将 src 目录下的 index.html 首页，复制到项目根目录中一份！

- 2.1 安装

运行如下的命令，即可在项目中安装此插件：

```
npm install html-webpack-plugin@4.5.0 -D
```

- 2.2 配置，在webpack.config.js

```
// 1. 导入 HTML 插件，得到一个构造函数
const HtmlPlugin = require('html-webpack-plugin')

// 2. 创建 HTML 插件的实例对象
const htmlPlugin = new HtmlPlugin({
  template: './src/index.html', // 指定原文件的存放路径
  filename: './index.html', // 指定生成的文件的存放路径
})

module.exports = {
    mode: 'development',
    plugins: [htmlPlugin], // 3. 通过 plugins 节点，使 htmlPlugin 插件生效
}
```

- 2.3 解惑

① 通过 HTML 插件复制到项目根目录中的 index.html 页面，也被放到了内存中

② HTML 插件在生成的 index.html 页面的底部，自动注入了打包的 bundle.js 文件

最新版 `html-webpack-plugin` 插件是在注入在 `head` 标签里，并添加 `defer` 属性，最后加载

> **注意：** 开启实时打包后，删除 `dist` 文件夹， `npm run dev` ，依然可以运行项目

3. `clean-webpack-plugin`

自动清理dist目录下的旧文件

为了在每次打包发布时自动清理掉 dist 目录中的旧文件，可以安装并配置 clean-webpack-plugin 插件：

```
// 1. 安装清理 dist 目录的 webpack 插件
npm install clean-webpack-plugin@3.0.0 -D

// 2. 按需导入插件，得到插件的构造函数之后，创建插件的实例对象
const { CleanWebpackPlugin } = require('clean-webpack-plugin')
const cleanPlugin = new CleanWebpackPlugin()

// 3. 把创建的 cleanPlugin 插件实例对象，挂载到 plugins 节点中
plugins: [htmlPlugin, cleanPlugin], // 挂载插件
```

3. webpack中的loader

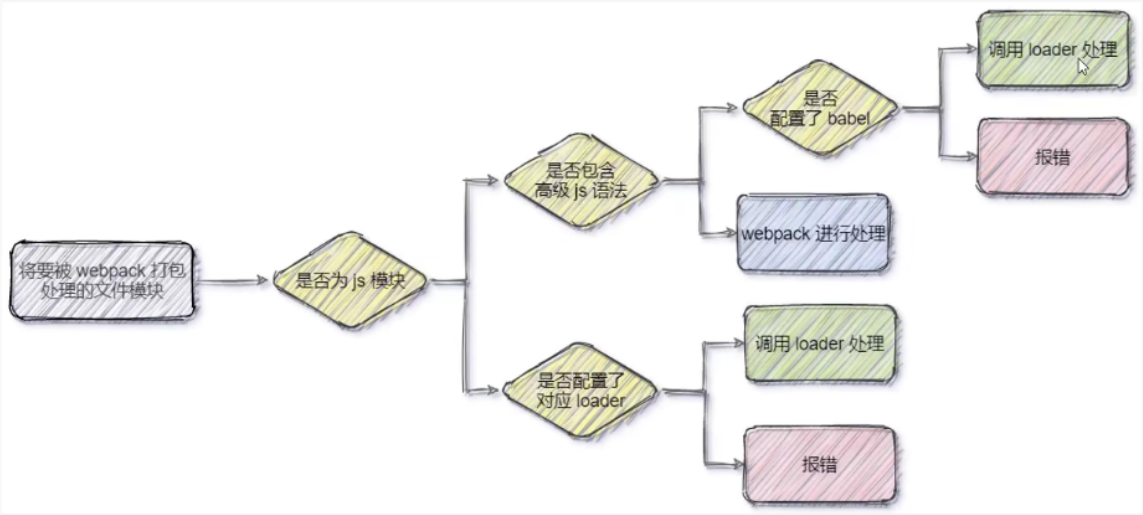
3.1 loader概述

在实际开发过程中，webpack默认只能打包处理以.js后缀名结尾的模块。其他非.js后缀名结尾的模块，webpack 默认处理不了，需要调用loader加载器才可以正常打包，否则会报错!

loader 加载器的作用：协助webpack打包处理特定的文件模块。比如：

- `css-loader` 可以打包处理.css相关的文件
- `less-loader` 可以打包处理.less相关的文件
- `babel-loader` 可以打包处理webpack无法处理的高级JS语法

3.2 loader的调用过程



3.3 打包处理 `css` 文件

① 运行 npm i style-loader@2.0.0 css-loader@5.0.1 -D 命令，安装处理 css 文件的 loader

② 在 webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
module: { // 所有第三方文件模块的匹配规则
    rules: [ // 文件后缀名的匹配规则
        { test: /\.css$/, use: ['style-loader', 'css-loader'] }
    ]
}
```

其中，test 表示匹配的文件类型， use 表示对应要调用的 loader

注意：

- use 数组中指定的 loader 顺序是固定的
- 多个 loader 的调用顺序是：从后往前调用

3.4 打包处理 `less` 文件

① 运行 npm i less-loader@7.1.0 less@3.12.2 -D 命令

② 在 webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
module: { // 所有第三方文件模块的匹配规则
    rules: [ // 文件后缀名的匹配规则
        { test: /\.less$/, use: ['style-loader', 'css-loader', 'less-loader'] },
    ]
}
```

3.5 打包处理样式表中与 url路径相关 的文件

① 运行 npm i url-loader@4.1.1 file-loader@6.2.0 -D 命令

② 在 webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
module: { // 所有第三方文件模块的匹配规则
    rules: [ // 文件后缀名的匹配规则
        { test: /\.jpg|png|gif$/, use: 'url-loader?limit=22229' },
    ]
}
```

limit参数： 判断是否转换成base64格式的图片

其中？之后的是 loader 的参数项：

- limit 用来指定图片的大小，单位是字节（byte）
- 只有 ≤ limit 大小的图片，才会被转为 base64 格式的图片

3.6 带参数的loader的另一种配置方式

带参数项的 loader 还可以通过对象的方式进行配置：

```
module: {    // 用来处理所有的第三方模块
    rules: [ // 第三方模块的匹配规则
      {
          test: /\.jpg|png|gif$/, // 匹配图片文件
          use: {
              loader: 'url-loader', // 通过 loader 属性指定要调用的 loader
              options: { // 通过 options 属性指定参数项
                  limit: 22229
              }
          }
      }
    ]
}
```

3.7 打包处理js文件中的高级语法

webpack 只能打包处理一部分高级的 JavaScript 语法。对于那些 webpack 无法处理的高级 js 语法，需要借助于 babel-loader 进行打包处理。例如 webpack 无法处理下面的 JavaScript 代码：

```
class Person {
  // 通过 static 关键字，为 Person 类定义了一个静态属性 info
  //webpack 无法打包处理“静态属性”这个高级语法
  static info = 'person info'
}

console.log(Person.info)
```

记：2021年8月7日，webpack最新版已可以正常打包class

运行如下的命令安装对应的依赖包：

```
npm i babel-loader@8.2.1  @babel/core@7.12.3  @babel/plugin-proposal-class-properties@7.12.1  -D
```

包的名称及版本号列表如下（红色是包的名称、黑色是包的版本号）：

- babel-loader@8.2.1
- @babel/core@7.12.3
- @babel/plugin-proposal-class-properties@7.12.1

配置：

在 webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
{
    test: /\.js$/,
    // exclude 为排除项，
    // 表示 babel-loader 只需处理开发者编写的 js 文件，不需要处理 node_modules 下的 js 文件
    exclude: /node_modules/,
    use: {
      loader: 'babel-loader',
      options: { // 参数项
        // 声明一个 babel 插件，此插件用来转化 class 中的高级语法
        plugins: ['@babel/plugin-proposal-class-properties'],
      },
    },
}
```

4.打包发布

4.1 为什么要打包发布?

项目开发完成之后，使用 webpack 对项目进行打包发布的主要原因有以下两点：

① 开发环境下，打包生成的文件存放于内存中，无法获取到最终打包生成的文件

② 开发环境下，打包生成的文件不会进行代码压缩和性能优化

4.2 配置 webpack 的打包发布

在 package.json 文件的 scripts 节点下，新增 build 命令如下：

```
"scripts": {
    "dev": "webpack serve", // 开发环境中，运行 dev 命令
    "build": "webpack --mode production" // 项目发布时，运行 build 命令
}
```

--model 是一个参数项，用来指定 webpack 的运行模式。production 代表生产环境，会对打包生成的文件进行代码压缩和性能优化。

注意：通过 --model 指定的参数项，会覆盖 webpack.config.js 中的 model 选项。

4.3 整理dist文件夹下的文件

- JavaScript文件统一放到 js 目录中

在 webpack.config.js 配置文件的 output 节点中，进行如下的配置：

```
output: {
    path: path.join(__dirname, 'dist'),
    // 明确告诉 webpack 把生成的 bundle.js 文件存放到 dist 目录下的 js 子目录中
    filename: 'js/bundle.js',
}
```

- 图片文件统一放在image目录中

修改 webpack.config.js 中的 url-loader 配置项，新增 outputPath 选项即可指定图片文件的输出路径：

```
{
    test: /\.jpg|png|gif$/,
    use: {
        loader: 'url-loader',
        options: {
            limit: 22228,
            // 明确指定把打包生成的图片文件，存储到 dist 目录下的 image 文件夹中
            outputPath: 'image',
        },
    },
}
```

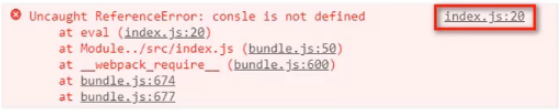
5. Source Map

Source Map就是一个信息文件，里面储存着位置信息 。也就是说，Source Map文件中存储着代码压缩混淆 前后 的 对应关系 。

有了它，出错的时候，除错工具将 直接显示原始代码 ， 而不是转换后的代码 ，能够极大的方便后期的调试.

5.1 webpack 开发环境 下的Source Map

在开发环境下，webpack 默认启用了 Source Map 功能。当程序运行出错时，可以直接在控制台提示错误行的位置，并定位到具体的源代码：



默认Source Map的问题：

报错行号不一致

解决：

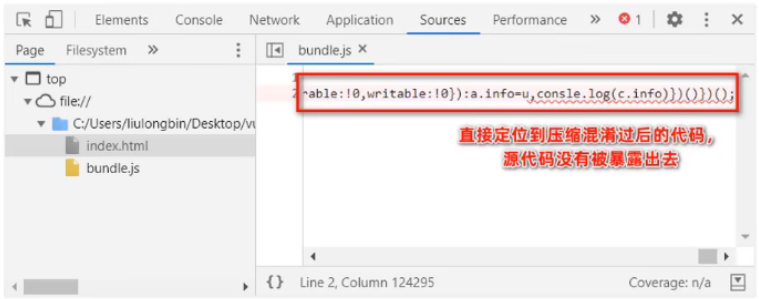
- 开发环境下：

开发环境下，推荐在 webpack.config.js 中添加如下的配置，即可保证运行时报错的行数与源代码的行数保持一致：

```
module.exports = {
    mode: 'development',
    // eval-source-map 仅限在"开发模式"下使用，不建议在“生产模式”下使用。
    // 此选项生成的 Source Map 能够保证"运行时报错的行数"与"源代码的行数"保持一致
    devtool: 'eval-source-map',
    // 省略其它配置项...
}
```

5.2 webpack 生产环境 下的Source Map

在生产环境下，如果省略了 devtool 选项，则最终生成的文件中不包含 Source Map。这能够防止原始代码通过 Source Map 的形式暴露给别有所图之人。

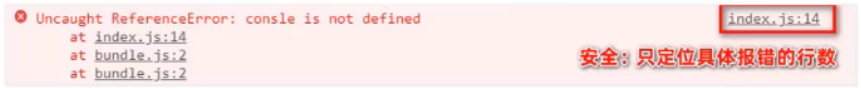


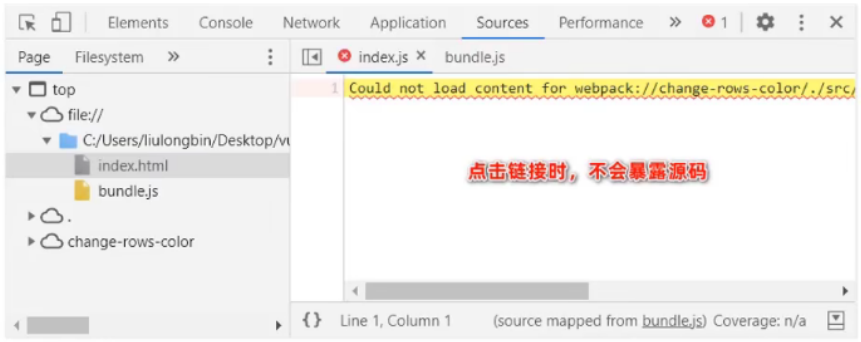
如何解决报错，调试问题?

- 1. 只定位行数不暴露源码

```
//配置webpack.config.js
devtool: "nosources-source-map"
```

在生产环境下，如果只想定位报错的具体行数，且不想暴露源码。此时可以将 devtool 的值设置为 nosources-source-map。实际效果如图所示：





- 2. 定位行数暴露源码

```
//配置webpack.config.js
devtool: "source-map"
```

在生产环境下，如果想在定位报错行数的同时，展示具体报错的源码。此时可以将 devtool 的值设置为 source-map。实际效果如图所示：

采用此选项后：你应该将你的服务器配置为，不允许普通用户访问 source map 文件！

5.3 Source Map 的最佳实践

① 开发环境下：

- 建议把 devtool 的值设置为 eval-source-map
- 好处：可以精准定位到具体的错误行

② 生产环境下：

- 建议关闭 Source Map 或将 devtool 的值设置为 nosources-source-map
- 好处：防止源码泄露，提高网站的安全性

6. 项目实例

E:\1workspace_for_Vscode\2021Code\08Webpack\webpack03\code01