

# 前端工程化与 webpack



# 录 Contents

## ◆ 前端工程化

◆ webpack 的基本使用

◆ webpack 中的插件

◆ webpack 中的 loader

◆ 打包发布

◆ Source Map

## 1. 小白眼中的前端开发 vs 实际的前端开发

小白眼中的前端开发：

- 会写 HTML + CSS + JavaScript 就会前端开发
- 需要美化页面样式，就拽一个 bootstrap 过来
- 需要操作 DOM 或发起 Ajax 请求，再拽一个 jQuery 过来
- 需要快速实现网页布局效果，就拽一个 Layui 过来

实际的前端开发：

- 模块化（js 的模块化、css 的模块化、资源的模块化）
- **组件化**（复用现有的 UI 结构、样式、行为）
- **规范化**（目录结构的划分、编码规范化、接口规范化、文档规范化、Git 分支管理）
- **自动化**（自动化构建、自动部署、自动化测试）

## 2. 什么是前端工程化

前端工程化指的是：在**企业级的前端项目开发**中，把前端开发所需的**工具、技术、流程、经验**等进行规范化、标准化。

企业中的 Vue 项目和 React 项目，都是基于**工程化的方式**进行开发的。

好处：前端开发**自成体系**，有一套**标准的开发方案和流程**。

## 3. 前端工程化的解决方案

早期的前端工程化解决方案：

- **grunt** ( <https://www.gruntjs.net/> )
- **gulp** ( <https://www.gulpjs.com.cn/> )

目前主流的前端工程化解决方案：

- **webpack** ( <https://www.webpackjs.com/> )
- **parcel** ( <https://zh.parceljs.org/> )

# 录 Contents

- ◆ 前端工程化
- ◆ webpack 的基本使用
- ◆ webpack 中的插件
- ◆ webpack 中的 loader
- ◆ 打包发布
- ◆ Source Map

## 1. 什么是 webpack

概念：webpack 是前端项目工程化的具体解决方案。

主要功能：它提供了友好的前端模块化开发支持，以及代码压缩混淆、处理浏览器端 JavaScript 的兼容性、性能优化等强大的功能。

好处：让程序员把工作的重心放到具体功能的实现上，提高了前端开发效率和项目的可维护性。

注意：目前 Vue, React 等前端项目，基本上都是基于 webpack 进行工程化开发的。



## 2. 创建列表隔行变色项目

- ① 新建项目空白目录，并运行 `npm init -y` 命令，初始化包管理配置文件 `package.json`
- ② 新建 `src` 源代码目录
- ③ 新建 `src -> index.html` 首页和 `src -> index.js` 脚本文件
- ④ 初始化首页基本的结构
- ⑤ 运行 `npm install jquery -S` 命令，安装 jQuery
- ⑥ 通过 ES6 模块化的方式导入 jQuery，实现列表隔行变色效果



## 3. 在项目中安装 webpack

在终端运行如下的命令，安装 webpack 相关的两个包：

```
npm install webpack@5.42.1 webpack-cli@4.7.2 -D
```



## 4. 在项目中配置 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：

```
1 module.exports = {  
2   mode: 'development' // mode 用来指定构建模式。可选值有 development 和 production  
3 }
```

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

```
1 "scripts": {  
2   "dev": "webpack" // script 节点下的脚本，可以通过 npm run 执行。例如 npm run dev  
3 }
```

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

## 4.1 mode 的可选值

mode 节点的可选值有两个，分别是：

### ① development

- 开发环境
- 不会对打包生成的文件进行代码压缩和性能优化
- 打包速度快，适合在开发阶段使用

### ② production

- 生产环境
- 会对打包生成的文件进行代码压缩和性能优化
- 打包速度很慢，仅适合在项目发布阶段使用

## 4.2 webpack.config.js 文件的作用

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。

## 4.3 webpack 中的默认约定

在 webpack 4.x 和 5.x 的版本中，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



## 4.4 自定义打包的入口与出口

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：

```
1 const path = require('path') // 导入 node.js 中专门操作路径的模块
2
3 module.exports = {
4   entry: path.join(__dirname, './src/index.js'), // 打包入口文件的路径
5   output: {
6     path: path.join(__dirname, './dist'), // 输出文件的存放路径
7     filename: 'bundle.js' // 输出文件的名称
8   }
9 }
```

# 录 Contents

- ◆ 前端工程化
- ◆ webpack 的基本使用
- ◆ webpack 中的插件
- ◆ webpack 中的 loader
- ◆ 打包发布
- ◆ Source Map

## 1. webpack 插件的作用

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义 index.html 页面的内容



## 2. webpack-dev-server

**webpack-dev-server** 可以让 webpack 监听项目源代码的变化，从而进行自动打包构建。

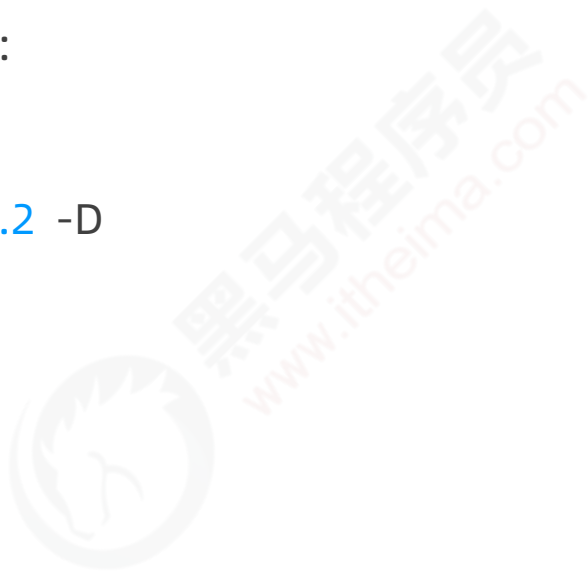


黑马程序员  
www.itheima.com

## 2.1 安装 webpack-dev-server

运行如下的命令，即可在项目中安装此插件：

```
npm install webpack-dev-server@3.11.2 -D
```



## 2.2 配置 webpack-dev-server

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

```
1 "scripts": {  
2   "dev": "webpack serve", // script 节点下的脚本，可以通过 npm run 执行  
3 }
```

② 再次运行 `npm run dev` 命令，重新进行项目的打包

③ 在浏览器中访问 `http://localhost:8080` 地址，查看自动打包效果

注意：webpack-dev-server 会启动一个实时打包的 http 服务器



## 2.3 打包生成的文件哪儿去了？

- 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



## 2.4 生成到内存中的文件该如何访问？

webpack-dev-server 生成到内存中的文件，默认放到了项目的根目录中，而且是虚拟的、不可见的。

- 可以直接用 / 表示项目根目录，后面跟上要访问的文件名称，即可访问内存中的文件
- 例如 `/bundle.js` 就表示要访问 webpack-dev-server 生成到内存中的 bundle.js 文件

## 3. html-webpack-plugin

html-webpack-plugin 是 **webpack** 中的 **HTML 插件**，可以通过此插件**自定义** 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**！



## 3.1 安装 html-webpack-plugin

运行如下的命令，即可在项目中安装此插件：

```
npm install html-webpack-plugin@5.3.2 -D
```

## 3.2 配置 html-webpack-plugin

```
1 // 1. 导入 HTML 插件，得到一个构造函数
2 const HtmlWebpackPlugin = require('html-webpack-plugin')
3
4 // 2. 创建 HTML 插件的实例对象
5 const htmlPlugin = new HtmlWebpackPlugin({
6   template: './src/index.html', // 指定原文件的存放路径
7   filename: './index.html', // 指定生成的文件的存放路径
8 })
9
10 module.exports = {
11   mode: 'development',
12   plugins: [htmlPlugin], // 3. 通过 plugins 节点，使 htmlPlugin 插件生效
13 }
```



## 3.3 解惑 html-webpack-plugin

- 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，自动注入了打包的 bundle.js 文件

## 4. devServer 节点

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：

```
1 devServer: {  
2   open: true, // 初次打包完成后，自动打开浏览器  
3   host: '127.0.0.1', // 实时打包所使用的主机地址  
4   port: 80, // 实时打包所使用的端口号  
5 }
```

注意：凡是修改了 webpack.config.js 配置文件，或修改了 package.json 配置文件，**必须重启实时打包的服务器**，否则最新的配置文件无法生效！

# 录 Contents

- ◆ 前端工程化
- ◆ webpack 的基本使用
- ◆ webpack 中的插件
- ◆ webpack 中的 loader
- ◆ 打包发布
- ◆ Source Map

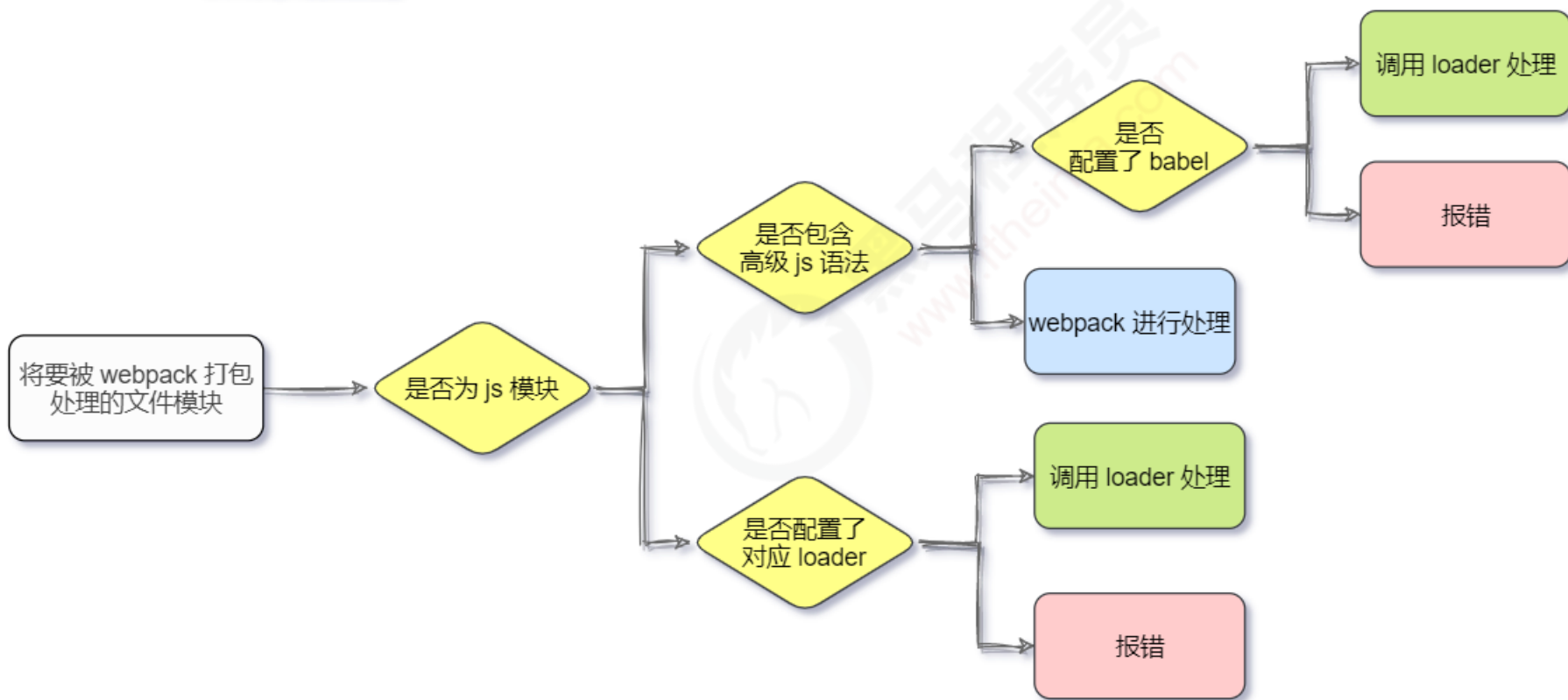
## 1. loader 概述

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

## 2. loader 的调用过程





## 3. 打包处理 css 文件

- ① 运行 `npm i style-loader@3.0.0 css-loader@5.2.6 -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：

```
1 module: { // 所有第三方文件模块的匹配规则
2     rules: [ // 文件后缀名的匹配规则
3         { test: /\.css$/, use: ['style-loader', 'css-loader'] }
4     ]
5 }
```

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

## 4. 打包处理 less 文件

- ① 运行 `npm i less-loader@10.0.1 less@4.1.1 -D` 命令
- ② 在 `webpack.config.js` 的 `module` -> `rules` 数组中, 添加 loader 规则如下:

```
1 module: { // 所有第三方文件模块的匹配规则
2     rules: [ // 文件后缀名的匹配规则
3         { test: /\.less$/, use: ['style-loader', 'css-loader', 'less-loader'] },
4     ]
5 }
```



## 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：

```
1 module: { // 所有第三方文件模块的匹配规则
2     rules: [ // 文件后缀名的匹配规则
3         { test: /\.jpg|png|gif$/, use: 'url-loader?limit=22229' },
4     ]
5 }
```

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  $\leq$  limit 大小的图片，才会被转为 base64 格式的图片



## 6. 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：

```
1 // 1. 定义了名为 info 的装饰器
2 function info(target) {
3   // 2. 为目标添加静态属性 info
4   target.info = 'Person info'
5 }
6
7 // 3. 为 Person 类应用 info 装饰器
8 @info
9 class Person {}
10
11 // 4. 打印 Person 的静态属性 info
12 console.log(Person.info)
```

## 6.1 安装 babel-loader 相关的包

运行如下的命令安装对应的依赖包：

```
npm i babel-loader@8.2.2 @babel/core@7.14.6 @babel/plugin-proposal-decorators@7.14.5 -D
```

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

```
1 // 注意：必须使用 exclude 指定排除项；因为 node_modules 目录下的第三方包不需要被打包
2 { test: /\.js$/, use: 'babel-loader', exclude: /node_modules/ }
```

## 6.2 配置 babel-loader

在项目根目录下，创建名为 `babel.config.js` 的配置文件，定义 Babel 的配置项如下：

```
module.exports = {  
  // 声明 babel 可用的插件  
  plugins: [['@babel/plugin-proposal-decorators', { legacy: true }]]  
}
```

详情请参考 Babel 的官网 <https://babeljs.io/docs/en/babel-plugin-proposal-decorators>

# 录 Contents

- ◆ 前端工程化
- ◆ webpack 的基本使用
- ◆ webpack 中的插件
- ◆ webpack 中的 loader
- ◆ 打包发布
- ◆ Source Map

## 1. 为什么要打包发布

项目开发完成之后，需要使用 webpack 对项目进行打包发布，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

为了让项目能够在生产环境中高性能的运行，因此需要对项目进行打包发布。

## 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：

```
1 "scripts": {  
2   "dev": "webpack serve", // 开发环境中，运行 dev 命令  
3   "build": "webpack --mode production" // 项目发布时，运行 build 命令  
4 }
```

`--mode` 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 `--mode` 指定的参数项，会`覆盖` `webpack.config.js` 中的 `mode` 选项。

### 3. 把 JavaScript 文件统一生成到 js 目录中

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

```
1 output: {  
2   path: path.join(__dirname, 'dist'),  
3   // 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  
4   filename: 'js/bundle.js',  
5 }
```

## 4. 把图片文件统一生成到 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：

```
1 {
2   test: /\.jpg|png|gif$/,
3   use: {
4     loader: 'url-loader',
5     options: {
6       limit: 22228,
7       // 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
8       outputPath: 'image',
9     },
10  },
11 }
```



## 5. 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 // 1. 安装清理 dist 目录的 webpack 插件
2 npm install clean-webpack-plugin@3.0.0 -D
3
4 // 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
5 const { CleanWebpackPlugin } = require('clean-webpack-plugin')
6 const cleanPlugin = new CleanWebpackPlugin()
7
8 // 3. 把创建的 cleanPlugin 插件实例对象，挂载到 plugins 节点中
9 plugins: [htmlPlugin, cleanPlugin], // 挂载插件
```

# 录 Contents

- ◆ 前端工程化
- ◆ webpack 的基本使用
- ◆ webpack 中的插件
- ◆ webpack 中的 loader
- ◆ 打包发布
- ◆ Source Map

## 1. 生产环境遇到的问题

前端项目在投入生产环境之前，都需要对 JavaScript 源代码进行**压缩混淆**，从而减小文件的体积，提高文件的加载效率。此时就不可避免的产生了另一个问题：

**对压缩混淆之后的代码除错（debug）是一件极其困难的事情**

```
20tAAAAAASUVORK5CYII="}}},t={};function n(r){if(t[r])return t[r].exports;var o=t[r]={id:r,exports:{}};return e[r].call(o.exports,o,o.exports,n),o.exports}n.n=e=>{var t=e&&e.__esModule?()=>e.default:()=>e;return n.d(t,{a:t}),t},n.d=(e,t)=>{for(var r in t)n.o(t,r)&&!n.o(e,r)&&Object.defineProperty(e,r,{enumerable:!0,get:t[r]})},n.o=(e,t)=>Object.prototype.hasOwnProperty.call(e,t),(()=>{"use strict";var e=n(755),t=n.n(e),r=n(379),o=n.n(r),i=n(340);o()(i.Z,{insert:"head",singleton:!1}),i.Z.locals;var a,s,u,l=n(543);o()(l.Z,{insert:"head",singleton:!1}),l.Z.locals,t()((function(){t>("li:odd").css("backgroundColor","pink"),t("li:even").css("backgroundColor","red")}));class c{u="person info",(s="info")in(a=c)?Object.defineProperty(a,s,{value:u,enumerable:!0,configurable:!0,writable:!0}):a.info=u,console.log(c.info)}})})();
```

- 变量被替换成**没有任何语义**的名称
- 空行和注释被剔除

## 2. 什么是 Source Map

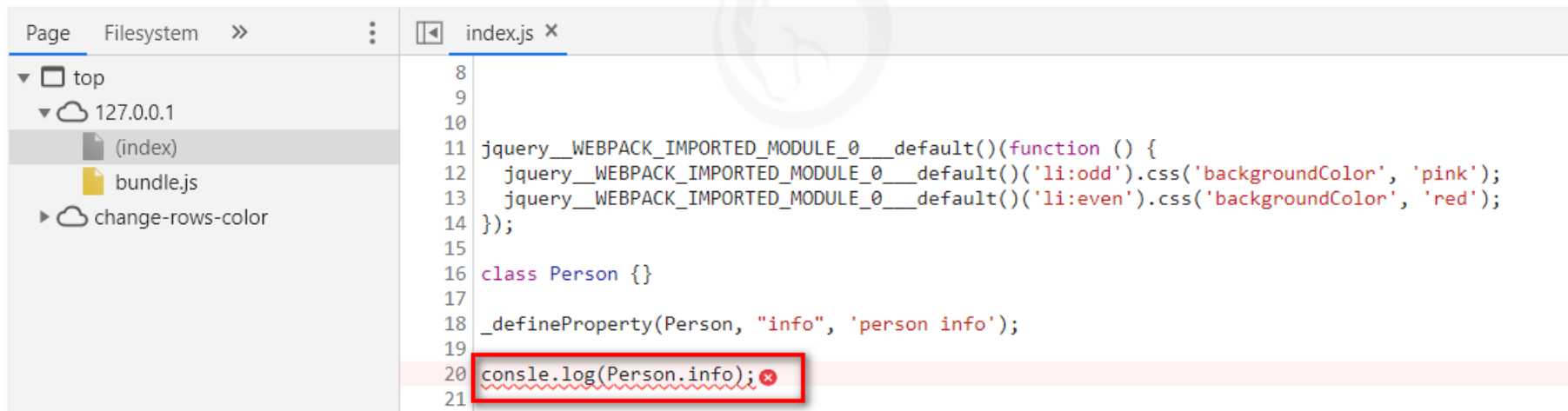
Source Map 就是一个信息文件，里面储存着位置信息。也就是说，Source Map 文件中存储着压缩混淆后的代码，所对应的转换前的位置。

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。

## 3. 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：

```
✖ Uncaught ReferenceError: console is not defined
    at eval (index.js:20)
    at Module../src/index.js (bundle.js:50)
    at __webpack_require__ (bundle.js:600)
    at bundle.js:674
    at bundle.js:677
```



```
Page  Filesystem  >>  index.js x
▼ top
  ▼ 127.0.0.1
    (index)
    bundle.js
    ► change-rows-color
8
9
10
11 jquery__WEBPACK_IMPORTED_MODULE_0___default()(function () {
12   jquery__WEBPACK_IMPORTED_MODULE_0___default()('li:odd').css('backgroundColor', 'pink');
13   jquery__WEBPACK_IMPORTED_MODULE_0___default()('li:even').css('backgroundColor', 'red');
14 });
15
16 class Person {}
17
18 _defineProperty(Person, "info", 'person info');
19
20 console.log(Person.info);
21
```

## 3.1 默认 Source Map 的问题

开发环境下默认生成的 Source Map，记录的是生成后的代码的位置。会导致运行时报错的行数与源代码的行数不一致的问题。示意图如下：

```
✖ Uncaught ReferenceError: consle is not defined
    at eval (index.js:20)
    at Module../src/index.js (bundle.js:50)
    at __webpack_require__ (bundle.js:600)
    at bundle.js:674
    at bundle.js:677
```

index.js:20

```
10  class Person {
11      static info = 'person info'
12  }
13
14  consle.log(Person.info)
15
```

实际出错的行数

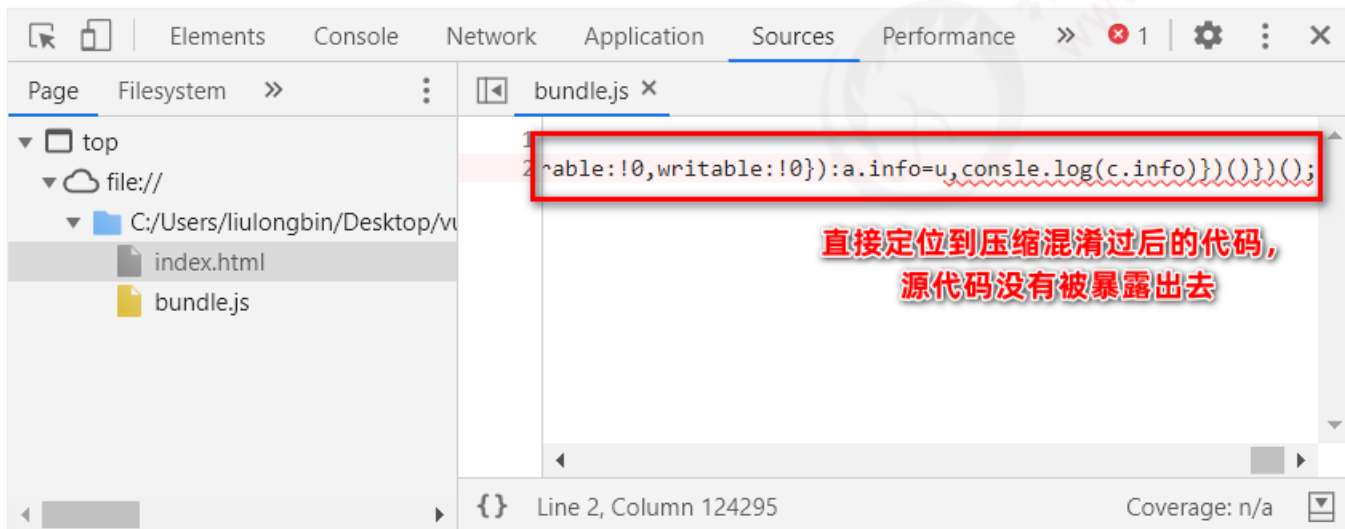
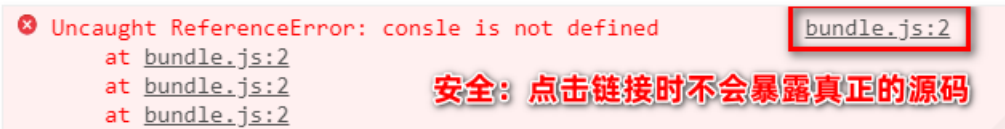
## 3.2 解决默认 Source Map 的问题

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：

```
1 module.exports = {  
2   mode: 'development',  
3   // eval-source-map 仅限在"开发模式"下使用，不建议在"生产模式"下使用。  
4   // 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  
5   devtool: 'eval-source-map',  
6   // 省略其它配置项...  
7 }
```

## 4. webpack 生产环境下的 Source Map

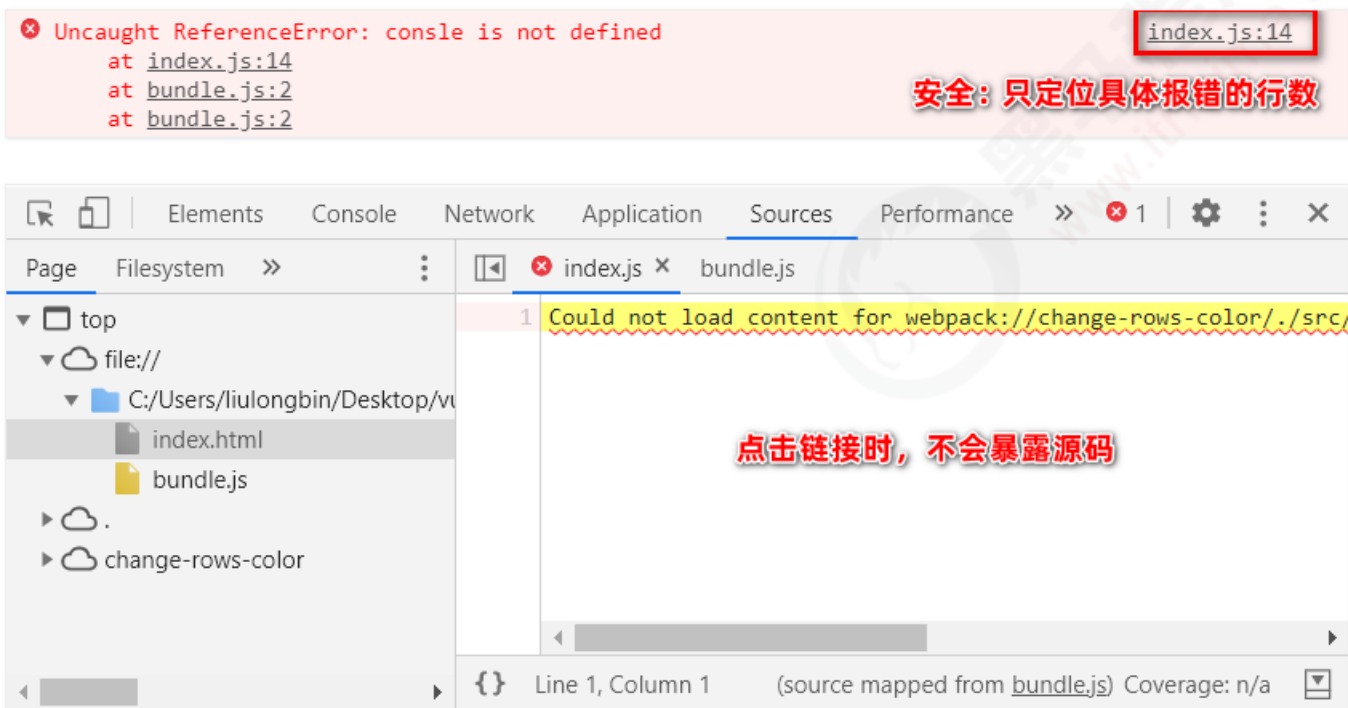
在生产环境下，如果省略了 devtool 选项，则最终生成的文件中不包含 Source Map。这能够防止原始代码通过 Source Map 的形式暴露给别有所图之人。





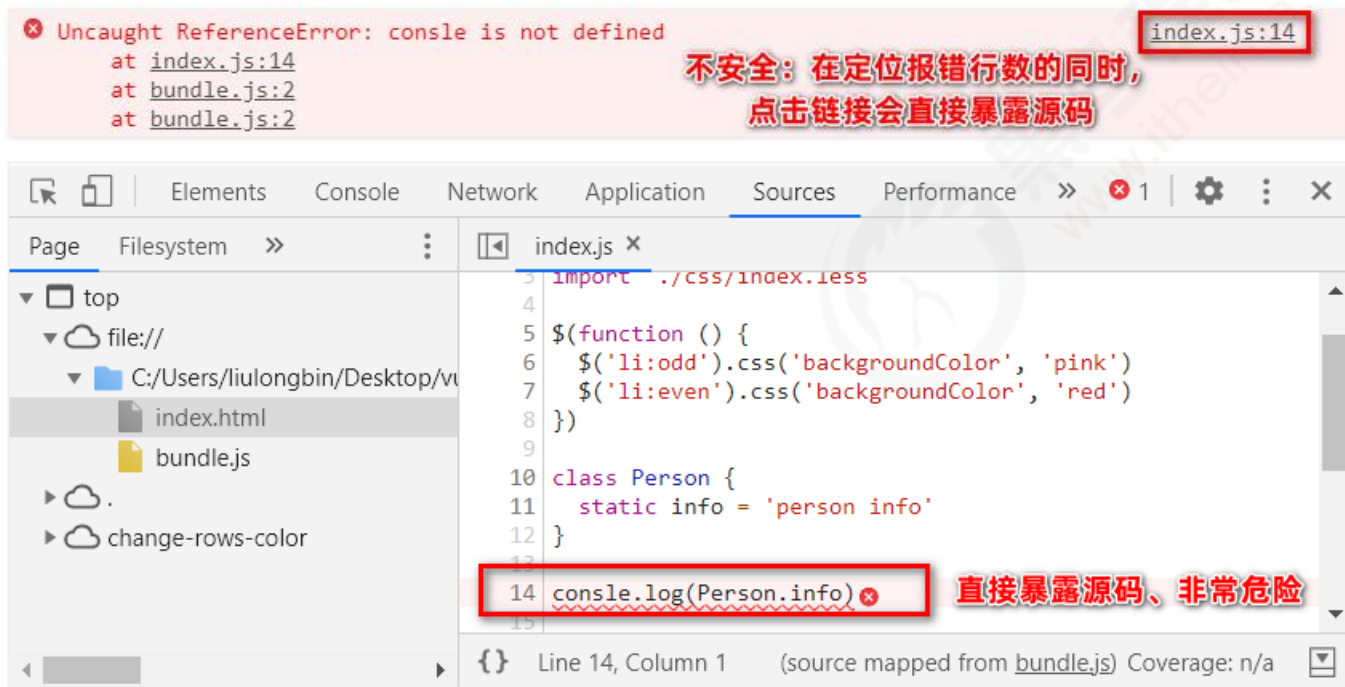
## 4.1 只定位行数不暴露源码

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：



## 4.2 定位行数且暴露源码

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. 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

## 实际开发中需要自己配置 webpack 吗？

思考

答案：不需要！

- 实际开发中会使命令行工具（俗称 CLI）一键生成带有 webpack 的项目
- 开箱即用，所有 webpack 配置项都是现成的！
- 我们只需要知道 webpack 中的基本概念即可！



# 总结

- ① 能够掌握 webpack 的基本使用
  - 安装、`webpack.config.js`、修改打包入口
- ② 了解常用的 plugin 的基本使用
  - `webpack-dev-server`、`html-webpack-plugin`
- ③ 了解常用的 loader 的基本使用
  - loader 的作用、`loader` 的调用过程
- ④ 能够说出 Source Map 的作用
  - 精准定位到错误行并显示对应的源码
  - 方便开发者调试源码中的错误



传智播客旗下高端IT教育品牌