

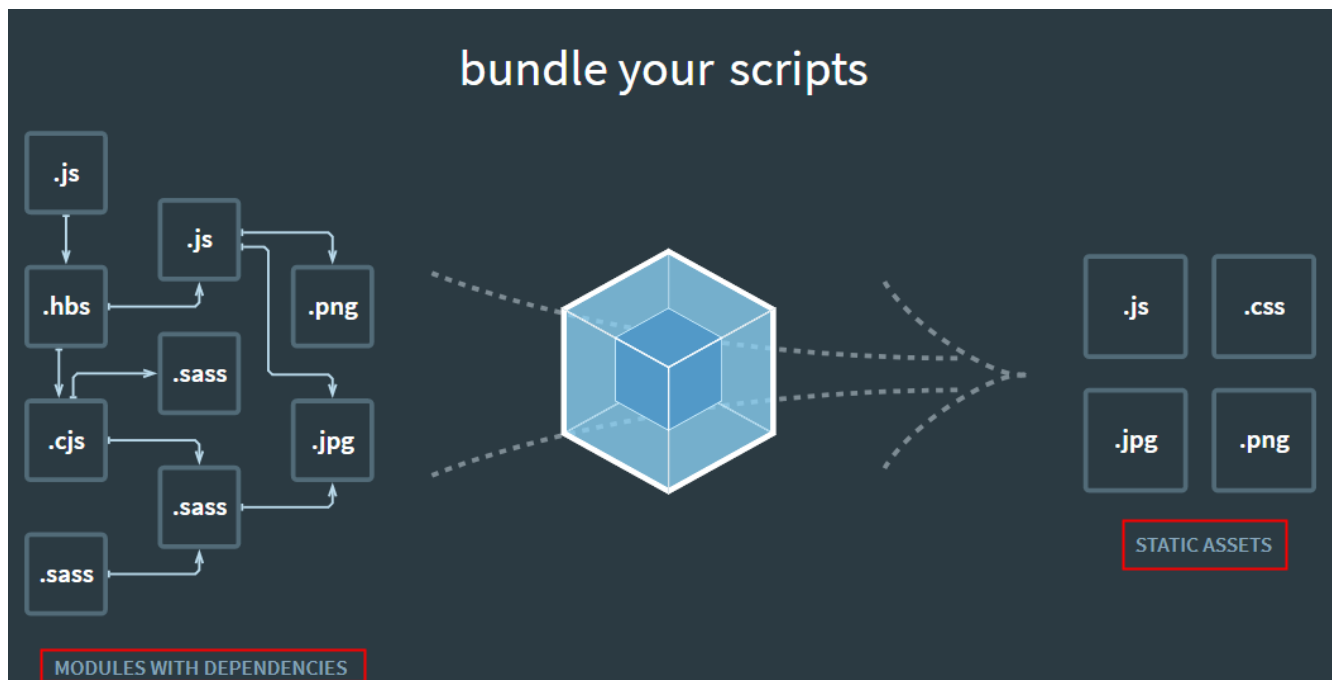
第一章 Webpack 介绍

1.1 Webpack 是什么

Webpack 是一个前端的**静态模块资源打包工具**，能让浏览器也支持模块化。它将根据模块的依赖关系进行静态分析，然后将这些模块按照指定的规则生成对应的静态资源。

1.2 Webpack 作用

- Webpack 核心主要进行 JavaScript 资源打包
- 如下图，它可以结合其他插件工具，将多种静态资源css、png、sass 分类转换成一个个静态文件，这样可以减少页面的请求。
- 可集成 babel 工具实现 EcmaScript 6 转 EcmaScript 5，解决兼容性问题
- 可集成 http 服务器
- 可集成模块热加载，当代码改变后自动刷新浏览器 等等功能



1.3 参考资料

webpack1 和 webpack2+ 版本变化很大，基本上推倒重来，webpack1 目前已经基本不用了。

- webpack1 官网 <https://webpack.github.io/>
- webpack2.x 英文官网 <https://webpack.js.org/>
- webpack2.x 中文官网 <https://webpack.docschina.org/>
- webpack2.x 指南文档：<https://webpack.docschina.org/guides/>

大家目前所使用的不管 3 还是 4 版本, 都是称为 webpack2.x

第二章 Webpack 安装和案例

2.1 全局安装

1. 安装 webpack

```
1 安装最新版本
2  npm install --global webpack
3 或者 安装特定版本
4  npm install --global webpack@<version>
```

2. 如果上面安装的是 webpack v4+ 版本, 还需要安装 CLI, 才能使用 webpack 命令行

```
1  npm install --global webpack-cli
```

可通过 `npm root -g` 查看全局安装目录

3. 如果安装后, 命令行窗口 `webpack` 命令不可用, 则手动配置 全局目录 的环境变量, 具体见 2.2.1

2.2 快速入门

VSCode 中安装插件 Node Snippets ,有代码快捷提示

2.2.1 打包 JS 模块

默认情况下, 模块化 JS 浏览器不能识别, 可通过 webpack 打包后让浏览器识别模块化 JS

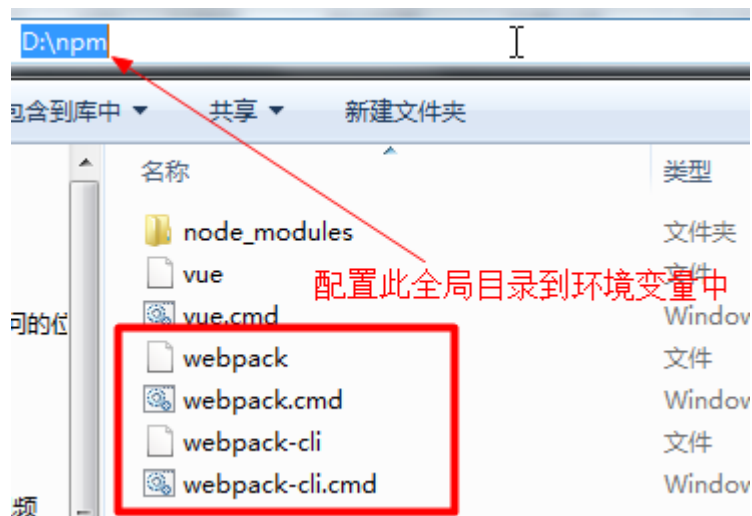
1. 全局安装 webpack@v4.35.2 与 webpack-cli@3.3.6

```
1  npm i -g webpack@v4.35.2
2  npm i -g webpack-cli@3.3.6
```

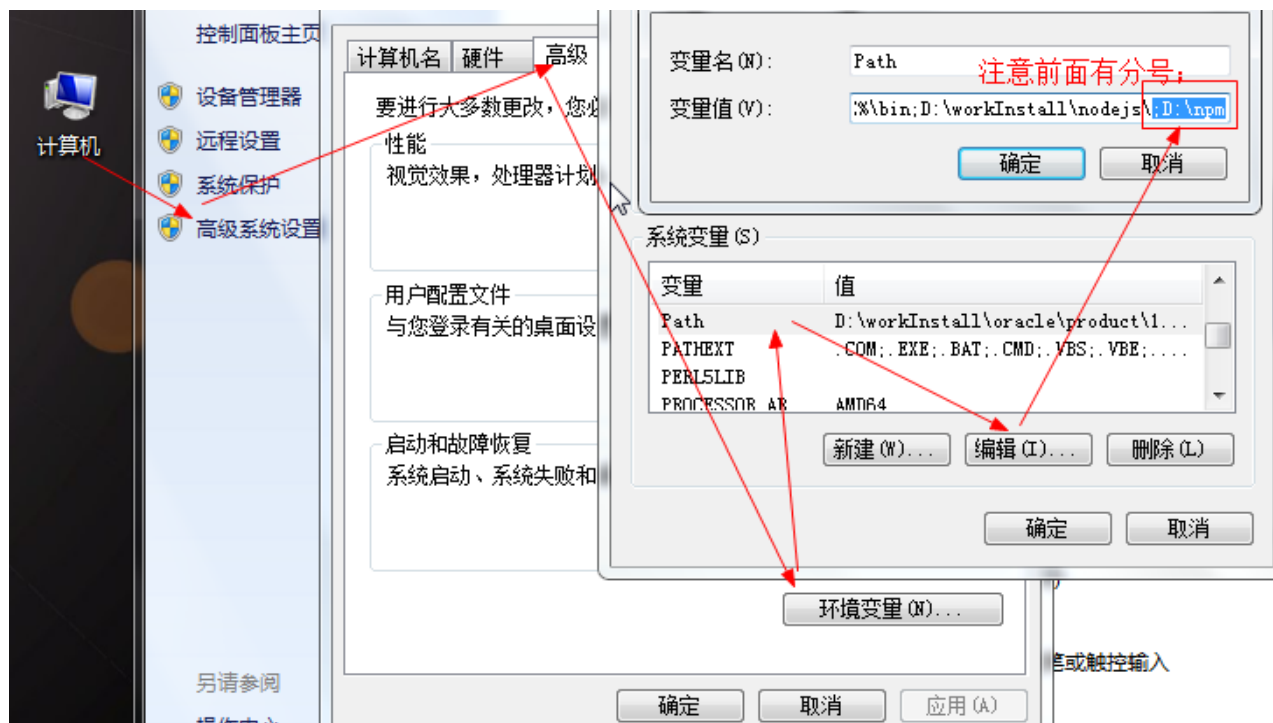
2. 安装后查看版本号。如果有红色字提醒, 没关系忽略它。

```
1  webpack -v
```

3. 如果安装后, 命令行窗口 `webpack` 命令不可用, 则配置环境变量:



我的电脑 -> 右键属性 -> 高级系统设置 -> 高级 -> 环境变量 -> 系统变量 -> path ->
在末尾添加上面的路径，记得前面用 ; 分号隔开
然后再重新打开 cmd 命令行窗口，输入 `webpack -v` 即可使用。



4. 创建以下目录结构和文件：

```
1 webpack demo1
2   |- index.html
3   |- js
4   |- bar.js
5   |- main.js
```

5. `bar.js` 文件内容如下：

```
1 // node 模块化编程，导出函数
2 module.exports = function () {
3     console.log('我是 bar 模块')
4 }
```

6. main.js 文件内容如下：

```
1 var bar = require('./bar') // 可省略 .js 后缀名
2 bar() // 调用 bar.js 中的函数
```

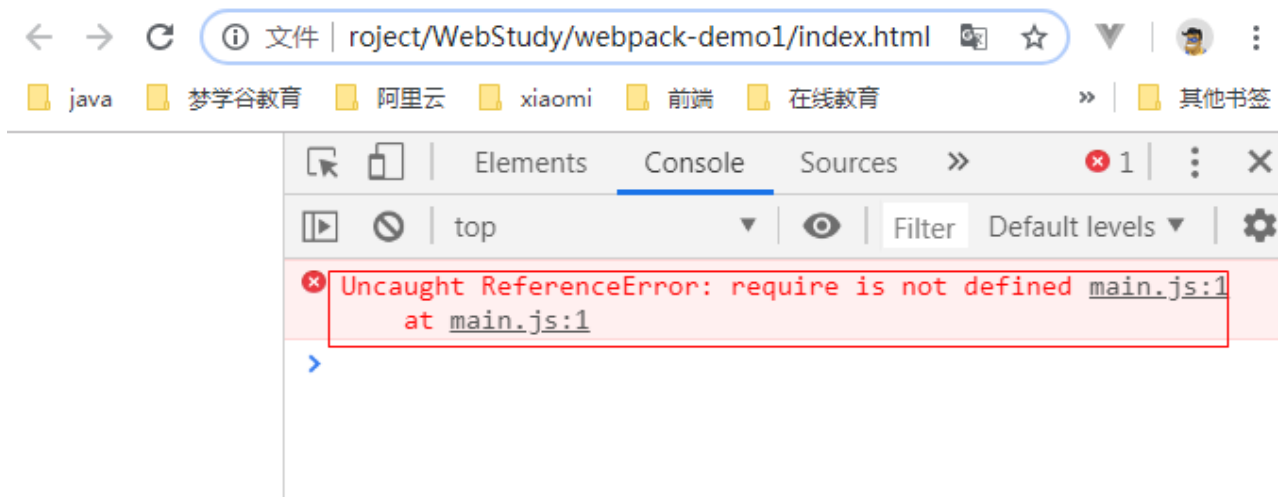
7. node 运行 js 模块，注意命令执行所在目录：**WebStudy\webpack-demo1**

```
1 d:\StudentProject\WebStudy\webpack-demo1>node ./js/main.js
2 我是 bar 模块
```

8. index.html 文件引入 main.js，如下：

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6 </head>
7 <body>
8     <script src="./js/main.js"></script>
9 </body>
10 </html>
```

9. 访问 index.html，浏览器无法识别 JS 模块化文件



10. 打包 JS，注意命令执行所在目录：**webStudy\webpack-demo1**，不要少了 -o

命令: webpack 模块入口文件路径 -o 模块出口文件路径

```
1 d:\StudentProject\WebStudy\webpack-demo1>webpack ./js/main.js -o ./js/bundle.js
```

打包时，出现黄色警告，先忽略，后面会进行解决

查看 bundle.js 会发现里面包含了上面 bar.js 文件的内容。

11. 将 index.html 引入的 JS 文件改为打包之后，浏览器可以识别的 JS 目标文件

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <!-- <script src="./js/main.js"></script> -->
9   <!-- 将 index.html 引入的 JS 文件改为打包之后，浏览器可以识别的 JS 目标文件 -->
10  <script src="./js/bundle.js"></script>
11 </body>
```

12. 浏览器访问 index.html 后，按 F12 控制台正常输出。

2.2.2 改造目录结构

1. 改造目录结构和文件的划分，划分为 src 和 dist 目录

- 把源码存储到 src 目录中
- 把打包后的结果存储到 dist 目录中

```
1 webpack demo2
2 |- index.html
3 |- src
4   |- bar.js
5   |- main.js
6 |- dist // 在打包时，指定 dist 目录后会自动创建
7 |- bundle.js
```

2. 打包 JS

```
1 d:\StudentProject\WebStudy\webpack-demo2>webpack ./src/main.js -o ./dist/bundle.js
```

3. 修改 index.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <!-- <script src="./js/main.js"></script> -->
9   <!-- 将 index.html 引入的 js 文件改为打包之后，浏览器可以识别的 js 目标文件 -->
10  <!-- <script src="./js/bundle.js"></script> -->
11  <script src="./dist/bundle.js"></script>
12 </body>
```

2.2.3 打包配置文件 webpack.config.js

每当修改js文件内容后，都要 webpack 重新打包，打包时要指定入口和出口比较麻烦，可通过配置解决。

1. 在 webpack-demo2 目录下创建 webpack.config.js 配置文件，该文件与 src 处于同级目录

```
1 // 引用 Node.js 中的 path 模块，处理文件路径的小工具
2 const path = require("path");
3
4 // 1. 导出一个webpack具有特殊属性配置的对象
5 module.exports = {
6   // 入口
7   entry: './src/main.js', // 入口模块文件路径
8   // 出口是对象
9   output: {
10     // path 必须是一个绝对路径，__dirname 是当前js的绝对路径：
11     // D:\StudentProject\WebStudy\webpack-demo2
12     path: path.join(__dirname, './dist/'), // 打包的结果文件存储目录
13     filename: 'bundle.js' // 打包的结果文件名
14   }
15 }
```

总结：读取当前目录下 src 文件夹中的 main.js（入口文件）内容，把对应的 js 文件打包，打包后的 bundle.js 文件放入当前目录的 dist 文件夹下

2. 执行打包命令

```
1 webpack
```

3. 解决打包时出现黄色警告：

通过 mode 选项指定模式配置，告诉webpack使用对应环境的预设插件，

参考：<https://webpack.js.org/configuration/mode/>

```
1  const path = require("path");
2
3  module.exports = {
4      // 指定模式配置,取值: none (什么也没有), development or production(默认的)
5      // 如, production 模式打包后 bundle.js是压缩版本的, development则不是压缩的
6      + mode: 'none',
7      entry: './src/main.js', // 入口模块文件路径
8      output: {
9          path: path.join(__dirname, './dist/'), // 打包的结果文件存储目录
10         filename: 'bundle.js' // 打包的结果文件名
11     }
12 }
```

重新打包，发现没有黄色警告了。

4. 测试访问 index.html，按 F12 查看控制台输出的信息

2.3 总结全局安装

不推荐全局安装 webpack。全局安装的 webpack，在打包项目的时候，使用的是你安装在自己电脑上的 webpack，如果项目到了另一个人的电脑上，他可能安装的是旧版本 webpack。那么就可能涉及兼容性的问题。而且如果他没有在全局安装 webpack 则就无法打包。

所以，为了解决以上的问题，官方推荐本地安装 webpack，就是将 webpack 安装到对应项目中。这样项目到哪里，webpack 就跟到哪里（webpack 打包工具随着项目走）。

2.4 本地安装（推荐）

2.4.1 说明

本地安装的时候，建议把 webpack 安装到 `devDependencies` 开发依赖（`--save-dev`）中，因为 webpack 只是一个打包工具，项目如果需要上线，上线的是打包的结果，而不是这个工具。

所以我们为了区分生产环境和开发环境依赖，通过 `--save`（生产环境）和 `--save-dev`（开发环境）来区分。

2.4.2 本地安装命令

1. 安装 webpack

```
1  安装最新版本
2  npm install --save-dev webpack
3  安装特定版本
4  npm install --save-dev webpack@<version>
```

2. 如果上面安装的是 `webpack v4+` 版本，还需要安装 CLI，才能使用 webpack 命令行

```
1 npm install --save-dev webpack-cli
```

2.5 本地安装案例

1. 为了测试本地安装，先把全局安装的 `webpack` 和 `webpack-cli` 卸载掉

```
1 npm uninstall -g webpack
2 npm uninstall -g webpack-cli
```

2. 安装 `webpack@v4.35.2` 与 `webpack-cli`

将 `webpack-demo2` 复制一份为 `webpack-demo3`

```
1 # 1. 进入到 webpack-demo3
2 cd d:\StudentProject\WebStudy\webpack-demo3
3 # 2. 初始化项目 -y 是采用默认配置
4 npm init -y
5 # 3. 安装 v4.35.2 , 不要少了 v
6 npm i -D webpack@v4.35.2
7 # 安装 CLI
8 npm i -D webpack-cli@3.3.6
```

3. 执行 `webpack` 命令会报错

在本地安装的 `webpack`，要通过在项目文件夹下 `package.json` 文件中的 `scripts` 配置命令映射

```
1 "scripts": {
2   "show": "webpack -v",
3   "start": "node ./src/main.js",
4   "build": "webpack"
5 },
```

4. 然后再通过 `npm run` 命令别名 执行对应命令，如：

- 查看 `webpack` 版本号：

```
1 npm run show
```

- 运行 `main.js` 模块：

```
1 npm run start
```

- 注意：如果命令映射的别名是 `start`，可省略 `run` 进行简写执行，即：

```
1 npm start
```

- 打包构建


```
1 npm run build
```

第三章 EcmaScript 6 模块规范

- 导出模块 `export`（等价于 `module.exports`）
- 导入模块 `import`（等价于 `require`）

创建以下目录结构和文件：

```
1 webpack demo4
2   |- index.html
3   |- src
4     |- bar.js
5     |- main.js
6   |- webpack.config.js
7   |- package.json
```

3.1 导出默认成员

1. 语法：默认成员只能有一个，否则会报错

```
1 export default 成员
```

2. 示例：

bar.js

```
1 // 导出函数
2 /* module.exports = function () {
3     console.log('我是 bar 模块---Node')
4 } */
5
6 // ES6，导出一个默认成员(任意类型)，一个js中只能有一个 default。可以默认导出任意类型成员
7 /* export default function () {
8     console.log('我是 bar 模块---ES6')
9 } */
10 // export default 'hello'
11 export default {
12     name: 'mxg'
13 }
```

3.2 导入默认成员

语法：

```
1 // 如果模块文件中没有 default 成员，则加载的是 undefined
2 import xxx from 模块文件
```

示例：

main.js

```
1 // Node 导入模块
2 // var bar = require('./bar')
3 // bar()
4
5 // ES6 导入
6 // 默认加载的是 export default 成员
7 import bar from './bar'
8 // bar()
9 // console.log( bar )
```

3.3 导出非默认成员

语法：非默认成员必须要有成员名称

```
1 export 成员
```

示例：

bar.js

```
1 // Node 导出非默认成员
2 // exports.x = 'xxx'
3 // exports.y = 'yyy'
4
5 // ES6 导出非默认成员
6 export const x = 'xxx'
7 export const y = 'yyy'
8 export function add (a, b) {
9     return a + b
10 }
```

错误示例：

```
1 // 没有变量名，错误的
2 export 'xxx'
3 // 没有函数名，错误的
4 export function (a, b) {
5     return a + b
6 }
```

3.4 导入非默认成员

语法：

```
1 // 方式一：按需导入指定成员，采用 解构赋值 的方式
2 import {成员名1, 成员名2, ..., 成员名n} from 模块文件
3 // 方式二：一次导入模块文件中的所有成员（包含 default 成员）
4 import * as 别名 from 模块文件
```

示例：

main.js

```
1 // 通过 export xxx 导出的非默认成员，可使用 解构赋值 的方式按需加载成员
2 // x 对应就是 bar.js 中的 x 成员，y 对应就是 bar.js 中的 y 成员，
3 import {x, y, add} from './bar'
4 console.log(x, y, add(10, 20))
5
6 // 一次性加载 export xxx 导出所有成员，不采用解构赋值
7 import * as bar2 from './bar'
8 console.log(bar2)
```

第四章 打包 CSS/Images 等资源

- Webpack 本身只能处理 JavaScript 模块，如果要处理其他类型的文件，就需要结合插件来使用，这些插件在 Webpack 中被称为 Loader（加载器）来进行转换。
- Loader 可以理解为是模块和资源的转换器，它本身是一个函数，参数接受的是源文件，返回值是转换后的结果。
- 这样，我们就可以通过 require 或 import 来加载任何类型的模块或文件，比如 CSS、图片。

4.1 打包 CSS 资源

创建 webpack-demo5

1. 安装 style-loader 和 css-loader 依赖

首先安装相关 Loader 插件：

css-loader 是将 css 装载到 javascript；

style-loader 是让 javascript 认识 css。

```
1 npm install --save-dev style-loader css-loader
```

2. 修改 webpack.config.js

```
1 const path = require("path");
```

```
2
3 module.exports = {
4   mode: 'none',
5   entry: './src/main.js', // 入口模块文件路径
6   output: {
7     // path 必须是一个绝对路径，__dirname 是当前js的绝对路径
8     path: path.join(__dirname, './dist/'), // 打包的结果文件存储目录
9     filename: 'bundle.js' // 打包的结果文件名
10  },
11  module: { // 模块
12    rules: [ // 规则
13      {
14        test: /\.css$/, // 正则表达式，匹配 .css 文件资源
15        use: [ // 使用的 Loader，注意顺序不能错
16          'style-loader',
17          'css-loader'
18        ]
19      }
20    ]
21  }
22 }
23
```

3. 在src文件夹创建 css 文件夹, css文件夹下创建 style.css

```
1 body {
2   background: red
3 }
```

4. 在 main.js 只引入 style.css

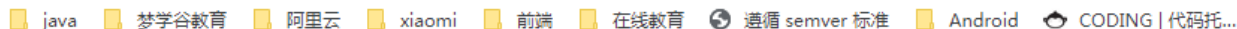
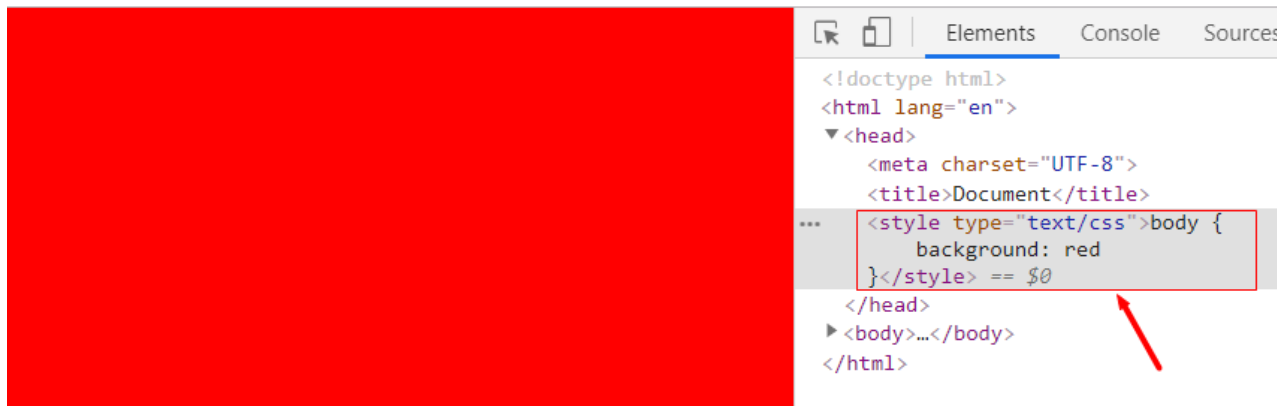
```
1 // 模块方式导入 css，最终会打包成js，打包在 bundle.js 中
2 import './css/style.css'
```

5. 重新打包编译

```
1 npm run build
```

打包后，查看 `bundle.js`，发现已经将 css 样式以 js 方式引入了

6. 访问 index.html，看看背景是不是变成红色

 127.0.0.1:5500/webpack-demo4/index.html java 梦学谷教育 阿里云 xiaomi 前端 在线教育 遵循 semver 标准 Android CODING | 代码托...

7. 原理：

F12查看 index.html 源码后，其实是将 CSS 文件内容转成一个 JavaScript 模块，然后在运行 JavaScript 时，会将样式动态使用 `<style>` 标签作用在页面 `<head>` 标签下

4.2 打包 Images 资源

4.2.1 打包 Images 步骤

1. 安装 file-loader 依赖

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

2. 修改 webpack.config.js

```
1 const path = require("path");
2
3 module.exports = {
4   mode: 'none',
5   entry: './src/main.js', // 入口模块文件路径
6   output: {
7     path: path.join(__dirname, './dist/'), // 打包的结果文件存储目录
8     filename: 'bundle.js' // 打包的结果文件名
9   },
10  module: { // 模块
11    rules: [ // 规则
12      {
13        test: /\.css$/, // 正则表达式，匹配 .css 文件资源
14        use: [ // 使用的 Loader，注意顺序不能错
15          'style-loader',
16          'css-loader'
17        ]
18      },
19    ],
20    test: /\..(png|svg|jpg|gif)$/,
21    use: [
22      'file-loader'
```

```
23         ]
24     }
25 ]
26 }
27 }
```

3. 修改 style.css

```
1 body{
2     background: red;
3     background-image: url(./1.jpg)
4 }
```

4. 打包编译

```
1 npm run build
```

5. 访问根目录下的 index.html，背景图并未显示出来

6. 问题：

如果直接访问根目录下的 index.html，那么图片资源路径就无法访问到。

解决方案：就是把 index.html 放到 dist 目录中。

但是 dist 是打包编译的结果，而非源码，所以把 index.html 放到 dist 就不合适。

而且如果我们一旦把打包的结果文件名 bundle.js 改了之后，则 index.html 也要手动修改。

综合以上遇到的问题，可以使用一个插件：html-webpack-plugin 来解决。

4.2.1 使用 HtmlWebpackPlugin 插件

作用：解决文件路径问题

- 将 index.html 打包到 bundle.js 所在目录中
- 同时也会在 index.html 中自动的 <script> 引入 bundle.js

1. 安装插件

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

2. 修改 webpack.config.js

```
1 const path = require("path");
2 // 引入插件
3 const HtmlWebpackPlugin = require('html-webpack-plugin');
4
5 module.exports = {
6     entry: './src/main.js', // 入口模块文件路径
7     output: {
8         path: path.join(__dirname, './dist/'), // 打包的结果文件存储目录
9         filename: 'bundle.js' // 打包的结果文件名
```

```
10     },
11     // 配置插件
12     plugins: [
13       new HtmlWebpackPlugin({
14         // 此插件作用是将 index.html 打包到 bundle.js 所在目录中,
15         // 同时也会在 index.html 中自动的 <script> 引入 bundle.js
16         // 注意：其中的文件名 bundle 取决于上面output.filename中指定的名称
17         template: './index.html'
18       })
19     ],
```

3. 修改 index.html, 模拟下vue页面

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <!-- 使用了HtmlWebpackPlugin 插件会自动引入bundle.js中 -->
9   <!-- <script src="./dist/bundle.js"></script> -->
10  <div id="app"></div>
11 </body>
```

4. 重新打包

```
1 npm run build
```

运行后，你会发现 dist 目录下多有一个 index.html，并且文件中自动引入了 bundle.js

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <!-- 使用了HtmlWebpackPlugin 插件会自动引入bundle.js中 -->
9   <!-- <script src="./dist/bundle.js"></script> -->
10  <div id="#app"></div>
11
12  <script type="text/javascript" src="bundle.js"></script></body>
```

5. 运行 dist/index.html 文件，背景图正常显示了。不要运行了 根目录下的 index.html

第五章 实时重新加载

5.1 说明

- 问题：

每一次手动打包很麻烦，打包后还需要手动刷新浏览器。

- 解决：

采用 webpack 提供的工具：`webpack-dev-server`，它允许在运行时更新所有类型的模块后，而无需手动打包和刷新页面，会自动打包和刷新页面。可以很大程度提高开发效率。

参考：<https://webpack.docschina.org/guides/development/#使用-webpack-dev-server>

5.2 实操

1. 安装依赖

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

2. 修改 `webpack.config.js` 配置

```
1 const path = require("path");
2 const HtmlWebpackPlugin = require('html-webpack-plugin');
3
4 module.exports = {
5   mode: 'none',
6   entry: './src/main.js', // 入口模块文件路径
7   output: {
8     path: path.join(__dirname, './dist/'), // 打包的结果文件存储目录
9     filename: 'bundle.js' // 打包的结果文件名
10  },
11  // 实时重新加载
12  devServer: {
13    contentBase: './dist'
14  },
15 }
```

3. 修改 `package.json` 的 `scripts`

`--open` 选项打包成功，自动打开浏览器

```
1 "scripts": {
2   "show": "webpack -v",
3   "build": "webpack",
4   "watch-build": "webpack --watch",
5   + "dev": "webpack-dev-server --open"
6 },
```

4. 打包

```
1 npm run dev
```

5. 测试，修改 `style.css`，会自动打包且浏览器会自动刷新，如下注释掉图片，就只有背景色了


```
1 body{
2   background: red;
3   /*background-image: url(./1.jpg)*/
4 }
```

第六章 Babel 浏览器兼容性

参考: <https://webpack.docschina.org/loaders/babel-loader/>

6.1 安装 Babel

```
1 npm install -D babel-loader @babel/core @babel/preset-env
```

6.2 配置 webpack.config.js

```
1 module: {
2   rules: [
3     {
4       test: /\.m?js$/,
5       exclude: /(node_modules|bower_components)/, // 排除的目录
6       use: {
7         loader: 'babel-loader',
8         options: {
9           presets: ['@babel/preset-env'] // 内置好的转译工具
10        }
11      }
12    ]
13  }
14 }
```

6.3 main.js 代码

```
1 const a = 1
2 // a = 2 //直接编译报错
3
4 const arr = [1, 2, 3]
5 arr.forEach(item => {
6   console.log(item)
7 })
```

`npm run build` 打包, 然后查看 `bundle.js` 代码, 已经转换为了 ES5 语法。

第七章 Vue-Loader 打包Vue单文件组件

参考 <https://vue-loader.vuejs.org/zh/guide/#vue-cli>

7.1 打包 Vue 基本配置

基于 webpack-demo6

1. 安装 `vue-loader` 和 `vue-template-compiler` 依赖

```
1 npm install -D vue-loader vue-template-compiler
```

2. 修改 `webpack.config.js` 配置

```
1 const VueLoaderPlugin = require('vue-loader/lib/plugin')
2
3 module.exports = {
4   module: {
5     rules: [
6       // ... 其它规则
7       {
8         test: /\.vue$/,
9         loader: 'vue-loader'
10      }
11    ]
12  },
13   plugins: [
14     // 请确保引入这个插件！
15     new VueLoaderPlugin()
16   ]
17 }
```

完整版本：

```
1 const path = require("path");
2 const HtmlWebpackPlugin = require('html-webpack-plugin');
3 // 1. 加载 Vue Loader 插件
4 const VueLoaderPlugin = require('vue-loader/lib/plugin')
5
6 module.exports = {
7   mode: 'none',
8   entry: './src/main.js', // 入口模块文件路径
9   output: {
10     path: path.join(__dirname, './dist/'), // 打包的结果文件存储目录
11     filename: 'bundle.js' // 打包的结果文件名
12   },
13   devServer: { // 实时重新加载
```

```
14     contentBase: './dist'
15   },
16   plugins: [
17     // 2. 引入vue插件
18     new VueLoaderPlugin(),
19
20     new HtmlWebpackPlugin({
21       template: './index.html'
22     })
23   ],
24   module: { // 模块
25     rules: [ // 规则
26       {
27         test: /\.css$/, // 正则表达式，匹配 .css 文件资源
28         use: [ // 使用的 Loader，注意顺序不能错
29           'style-loader',
30           'css-loader'
31         ]
32       },
33       {
34         test: /\.(png|svg|jpg|gif)$/,
35         use: [
36           'file-loader'
37         ]
38       },
39       {
40         test: /\.m?js$/,
41         exclude: /(node_modules)/, // 排除的目录
42         use: {
43           loader: 'babel-loader',
44           options: {
45             presets: ['@babel/preset-env'] // babel中内容的转换规则工具
46           }
47         }
48       },
49       // 3. 配置vue-loader
50       {
51         test: /\.vue$/,
52         use: [
53           'vue-loader'
54         ]
55       }
56     ]
57   }
58 }
```

3. 在 src 目录下创建 App.vue

```
1 <template>
2   <div >
3     <h1>App</h1>
4   </div>
5 </template>
6 <script>
7
8 </script>
9
10 <style >
11 </style>
```

4. 在 main.js 中导入 App.vue

```
1 import App from './App.vue'
```

导入之后，这个App 组件就可以作为子组件进行使用了

5. 打包

```
1 npm run build
```

控制台不报错，说明配置正确打包成功

7.2 webpack与 Vue 单文件组件案例

1. 创建 webpack-demo6

```
1 webpack demo6
2 |- index.html // 单页面入口
3 |- src        // 存放源文件目录
4   |- main.js  // 打包入口文件
5   |- App.vue  // 根组件，替换index.html中的 #app 处
6   |- router.js // 路由
7   |- components // 存放组件目录
8 |- webpack.config.js // webpack 配置
9 |- package.json // `npm init -y` 初始化项目
10 |- node_modules // 依赖目录
```

2. 安装 vue 模块

```
1 npm i vue
```

3. index.html 单页面入口

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <!-- vue 入口 -->
9   <div id="app"></div>
10 </body>
```

4. App.vue 根组件

```
1 <template>
2   <!-- 组件模板，.vue文件中可只出现 template 标签 -->
3   <div>
4     <h1>App 根组件</h1>
5   </div>
6 </template>
```

main.js 打包入口文件

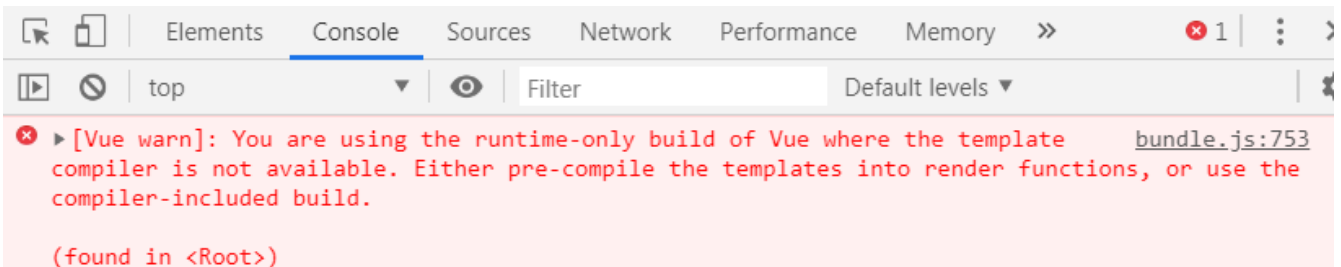
```
1 import Vue from 'vue'
2 import App from './App.vue'
3
4 new Vue({
5   el: '#app',
6   template: '<App />',
7   components: {App}
8 })
```

打包构建

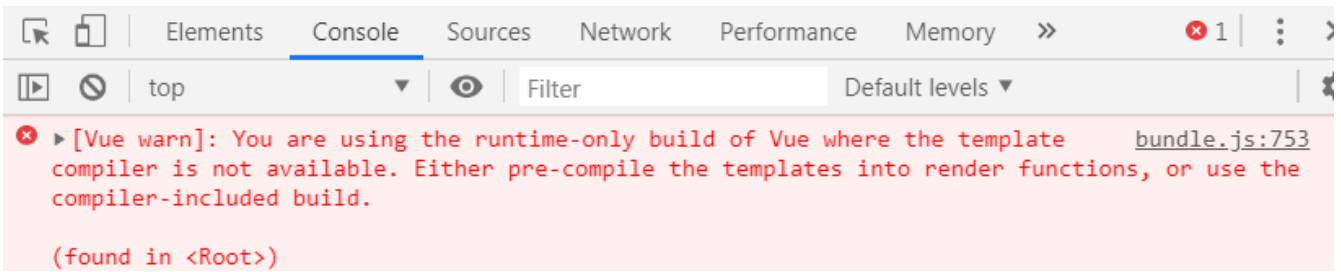
```
1 npm run build
```

打包成功，无报错。

访问 dist/index.html，发现App组件没有被渲染出来，按 F12 查看控制台发现警告：



7.3 解决警告问题



7.3.1 分析原因

1. 在node_modules目录，找到刚刚安装的vue目录，打开目录下的package.json文件，找到main属性：

```
1 "main": "dist/vue.runtime.common.js",
```

`import Vue from 'vue'` 导入的vue文件默认是 `package.json` 中的 `main` 属性指定的文件，可以发现它并不是我们熟悉的 `vue.js` 完整版文件，`import` 的是运行时版本，不是完整版，参考vue官方文档：

- 1 - 完整版：同时包含编译器和运行时的版本。
- 2 - 编译器：用来将 `template` 的模板字符串编译成为 `JavaScript` 渲染函数的代码。
- 3 - 运行时：用来创建 `vue` 实例、渲染并处理虚拟 `DOM` 等的代码。没有编译器功能，无法编译模板。

也就是说，`template` 渲染的字符串，运行时版本 `vue` 无法解析

7.3.2 两种解决方法

引用完整版 `vue.js`

- 第1种方法：`import` 导入完整版 `vue`

```
1 import Vue from 'vue/dist/vue.js'
2
3 new Vue({
4   el: '#app',
5   template: '<App />',
6   components: {App}
7 })
```

- 第2种方法：

1. 依旧 `import Vue from 'vue'`

```
1 import Vue from 'vue'
2 import App from './App.vue'
3
4 new Vue({
5   el: '#app',
6   template: '<App />',
7   components: {App}
8 })
```

2. 然后 `webpack.config.js` 增加一个属性

```
1 module.exports = {  
2   // 解析完整版 vue.js  
3   resolve: {  
4     alias: {  
5       'vue$': 'vue/dist/vue.js'  
6     }  
7   }  
8 }
```

总结：

1. 以上两种方法都可以解决。但是完整版比运行时 `vue` 大，性能不如运行时 `vue`。
2. 官方更推荐运行时 `vue`，因为 `vue-loader` 可以编译 `.vue` 文件，所以事实上是不需要 `vue` 的编译功能的，只需要渲染功能即可。
3. 而 `template` 自身没有渲染功能，最终渲染底层都是通过 `render` 函数够实现的。如果写 `template` 属性，则需要编译成 `render` 函数，这个编译过程对性能会有一定损耗。
4. 所以使用运行时 `vue` 通过 `render` 函数来渲染组件即可。

7.3.3 最优解决方法

Vue 实例中，不使用 `template`，而是直接使用 `render` 函数来渲染组件即可。

注意：上面方法2在 `webpack.config.js` 添加的配置记得取消掉

```
1 // 方式3：采用 render 函数渲染组件  
2 import Vue from 'vue'  
3 import App from './App.vue'  
4  
5 new Vue({  
6   el: '#app',  
7   // 使用render后，当前可不使用 componntns  
8   // h是函数用来生成 DOM 元素的，render得到完整Dom后，挂载到根节点上  
9   /* render: function (h) {  
10     return h(App)  
11   } */  
12   render: h => h(App) // ES6 箭头函数  
13 })
```

7.4 .vue 单文件组件完整版

7.4.1 App.vue

1. 在 `<script>` 导出一个默认成员对象，就是当前组件对象，Vue的 `data/methods`等选项直接定义在此对象中
2. 在 `<style>` 上使用 `scoped` 属性，CSS 样式只在当前组件有效，否则样式会自动作用到父子组件中。

```
1 <template>
2   <!-- 组件模板，.vue文件中可只出现 template 标签 -->
3   <div>
4     <h1>App 根组件</h1>
5     <h2>{{ msg }}</h2>
6     <!-- 引用子组件 -->
7     <foo></foo>
8   </div>
9 </template>
10
11 <script>
12 // 1. 先导入 vue 组件，再使用
13 import Foo from './components/Foo.vue'
14
15 // 导出一个默认成员对象，就是当前组件对象，vue的data/methods等选项直接写在这里，
16 // template 不用写，因为上面标签就是模板对象
17 export default {
18   data () {
19     return {
20       msg: 'hello webpack'
21     }
22   },
23   // 引用Foo子组件
24   components: {
25     Foo
26   }
27 }
28 </script>
29
30 <style scoped>
31   /*
32     默认 CSS 样式会自动传递到父子组件中
33     scoped 指定后，CSS样式只在当前组件中有效
34   */
35   h1 {
36     color: red
37   }
38 </style>
```

7.4.2 Foo.vue 子组件

- 在 components 目录下创建 Foo.vue
- 在 <style> 上不使用 scoped 属性，CSS 样式会自动作用到父子组件中。

```
1 <template>
2   <div>
3     <h1>我是 Foo 子组件</h1>
4     <h2>我是 Foo 子组件</h2>
5   </div>
6 </template>
7
8 <script>
9 export default {
```



```
10 }
11 </script>
12
13 <style scoped>
14   h2{
15     color: blue
16   }
17 </style>
```

7.4.3 测试

访问 `dist/index.html` 查看效果

7.5 模块热替换 (HMR)

7.5.1 介绍

- 模块热替换(hot module replacement 或 HMR)是 webpack 提供的最有用的功能之一。
模块热替换无需完全刷新页面，局部无刷新的情况下就可以更新。
- 参考：<https://webpack.docschina.org/guides/hot-module-replacement/>

7.5.2 配置

注意：要安装了 [webpack-dev-server](#) 模块，前面第五章已经安装过了。

配置以下3处 `+++` 的位置：

```
1  const path = require("path");
2  const HtmlWebpackPlugin = require('html-webpack-plugin');
3  const VueLoaderPlugin = require('vue-loader/lib/plugin');
4  +++ const webpack = require('webpack');
5
6  module.exports = {
7    mode: 'none',
8    entry: './src/main.js', // 入口模块文件路径
9    output: {
10      path: path.join(__dirname, './dist/'), // 打包的结果文件存储目录
11      filename: 'bundle.js' // 打包的结果文件名
12    },
13    devServer: { // 实时重新加载
14      contentBase: './dist',
15      +++      hot: true
16    },
17    plugins: [
18      // 模块热替换
19      +++      new webpack.HotModuleReplacementPlugin(),
20
21      // 引入Vue插件
```

```
22     new VueLoaderPlugin(),
23     new HtmlWebpackPlugin({
24       template: './index.html'
25     })
26   ],
27   module: { // 模块
28     rules: [ // 规则
29       {
30         test: /\.css$/, // 正则表达式，匹配 .css 文件资源
31         use: [ // 使用的 Loader，注意顺序不能错
32           'style-loader',
33           'css-loader'
34         ]
35       },
36       {
37         test: /\. (png|svg|jpg|gif) $/,
38         use: [
39           'file-loader'
40         ]
41       },
42       {
43         test: /\.vue$/,
44         use: [
45           'vue-loader'
46         ]
47       }
48     ]
49   }
50 }
```

7.5.3 测试

注意：只能更新组件，更新 js 是无法热替换的。

1. 打包构建：注意是 dev

```
1 npm run dev
```

可能会报以下错：

```
1 internal/modules/cjs/loader.js:584
2   throw err;
3   ^
4
5 Error: Cannot find module 'uuid/v4'
```

执行命令，解决重新安装 webpack-dev-server：

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

然后再进行打包 `npm run dev`

2. 访问 `dist/index.html`，然后对 `.vue` 单文件组件更新，会局部热替换。

注意：对 `js` 文件是无效果的，如修改 `main.js` 不会热替换的。