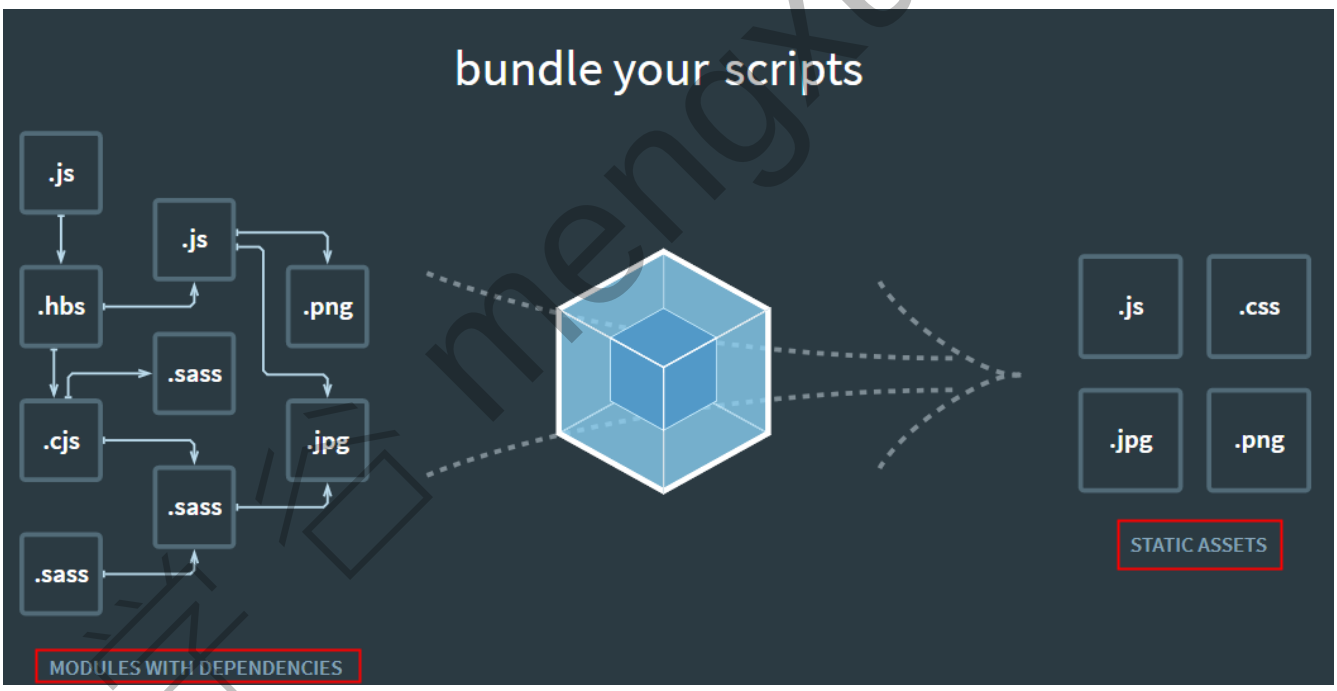
第一章 Webpack 介绍

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

```
安装最新版本
npm install --global webpack
或者 安装特定版本
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

```
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

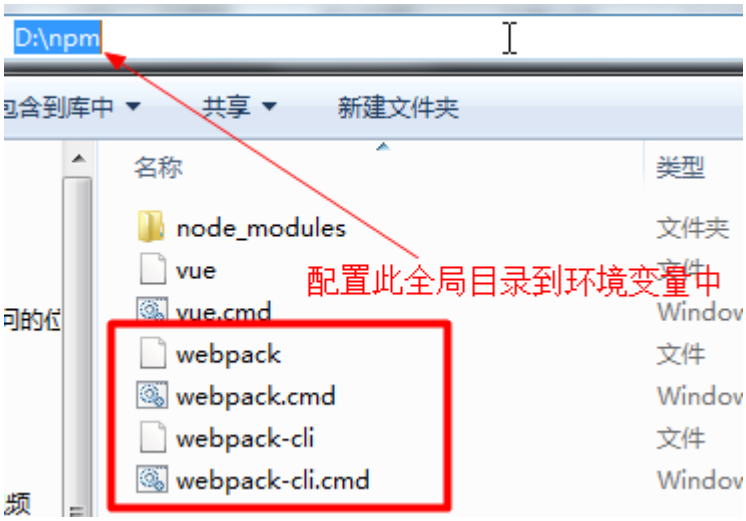
1. 全局安装 `webpack@v4.35.2` 与 `webpack-cli@3.3.6`

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

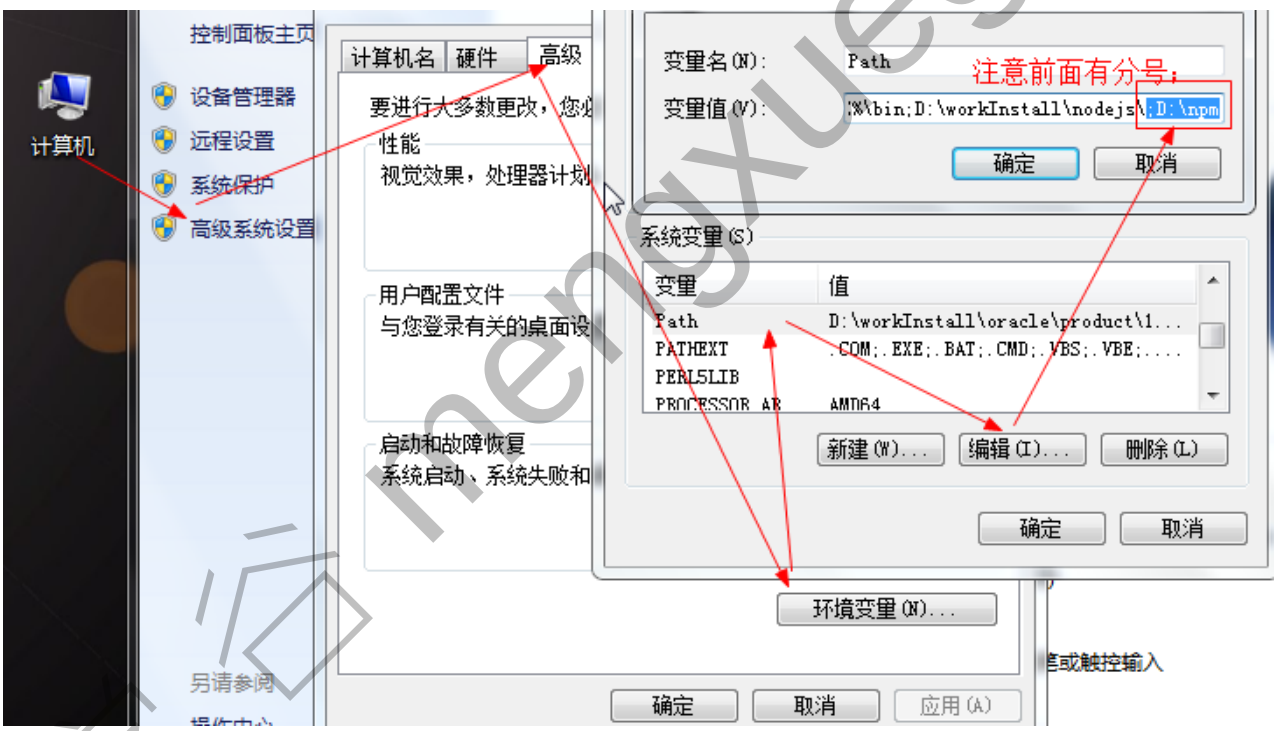
2. 安装后查看版本号。如果有红色字提醒，没关系忽略它。

```
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：

```
webpack-demo1
 |- index.html
 |- js
  |- bar.js
  |- main.js
```

5. `bar.js` 文件内容如下：

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

6. `main.js` 文件内容如下：

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

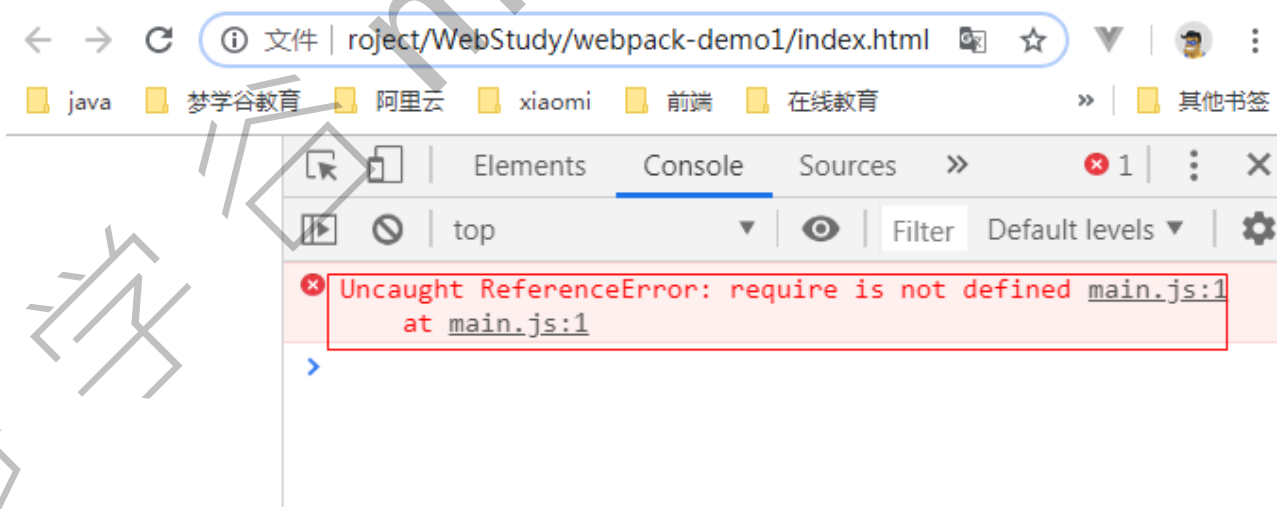
7. node 运行 js 模块，注意命令执行所在目录： WebStudy\webpack-demo1

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

8. `index.html` 文件引入 main.js , 如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <script src="./js/main.js"></script>
</body>
</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`

```
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

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <!-- <script src="./js/main.js"></script> -->
    <!-- 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 -->
    <script src="./js/bundle.js"></script>
</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

```
webpack-demo2
  |- index.html
  |- src
      |- bar.js
      |- main.js
  |- dist // 在打包时，指定 dist 目录后会自动创建
     |- bundle.js
```

2. 打包 JS

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

3. 修改 index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <!-- <script src="./js/main.js"></script> -->
    <!-- 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 -->
    <!-- <script src="./js/bundle.js"></script> -->
    <script src="./dist/bundle.js"></script>
</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

```js
// 引用 Node.js 中的 path 模块，处理文件路径的小工具
const path = require("path");

// 1. 导出一个webpack具有特殊属性配置的对象
module.exports = {
    // 入口
    entry: './src/main.js', // 入口模块文件路径
    // 出口是对象
    output: {
        // path 必须是一个绝对路径 ， __dirname 是当前js的绝对路径：D:\StudentProject\WebStudy\webpack-demo2
        path: path.join(__dirname, './dist/'),  // 打包的结果文件存储目录
        filename: 'bundle.js' // 打包的结果文件名
    }
}
```

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

```
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/

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

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

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

```
安装最新版本
npm install --save-dev webpack
安装特定版本
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

```
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

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

```
  "scripts": {
    "show": "webpack -v",
    "start": "node ./src/main.js",
    "build": "webpack"
  },
```

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：

```
npm run show
```

   - 运行 main.js 模块：

```
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：

```
npm start
```

   - 打包构建

```
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：

```
webpack-demo4
  |- index.html
  |- src
      |- bar.js
      |- main.js
  |- webpack.config.js
  |- package.json
```

3.1 导出默认成员

1. 语法：默认成员只能有一个，不然会报错

```
export default 成员
```

2. 示例：

`bar.js`

```
// 导出函数
/* module.exports = function () {
    console.log('我是 bar 模块---Node')
} */

// 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
/* export default function (){
    console.log('我是  bar 模块---ES6')
} */
// export default 'hello'
export default {
    name: 'mxg'
}
```

3.2 导入默认成员

语法：

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

示例：

`main.js`

```
// Node 导入模块
// var bar = require('./bar')
// bar()

// ES6 导入
// 默认加载的是 export default 成员
import bar from './bar'
// bar()
// 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

```
export 成员
```

示例：

`bar.js`

```
// Node 导出非默认成员
// exprots.x = 'xxx'
// exprots.y = 'yyy'

// ES6 导出非默认成员
export const x = 'xxx'
export const y = 'yyy'
export function add (a, b) {
    return a + b
}
```

错误示例：

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

3.4 导入非默认成员

语法：

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

示例：

`main.js`

```
// 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
// x 对应就是 bar.js 中的 x 成员，y 对应就是 bar.js 中的 y 成员，
import {x, y, add} from './bar'
console.log(x, y, add(10, 20))

// 一次性加载 export xxx 导出所有成员，不采用解构赋值
import * as bar2 from './bar'
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。

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

2. **修改 webpack.config.js**

   ```
   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

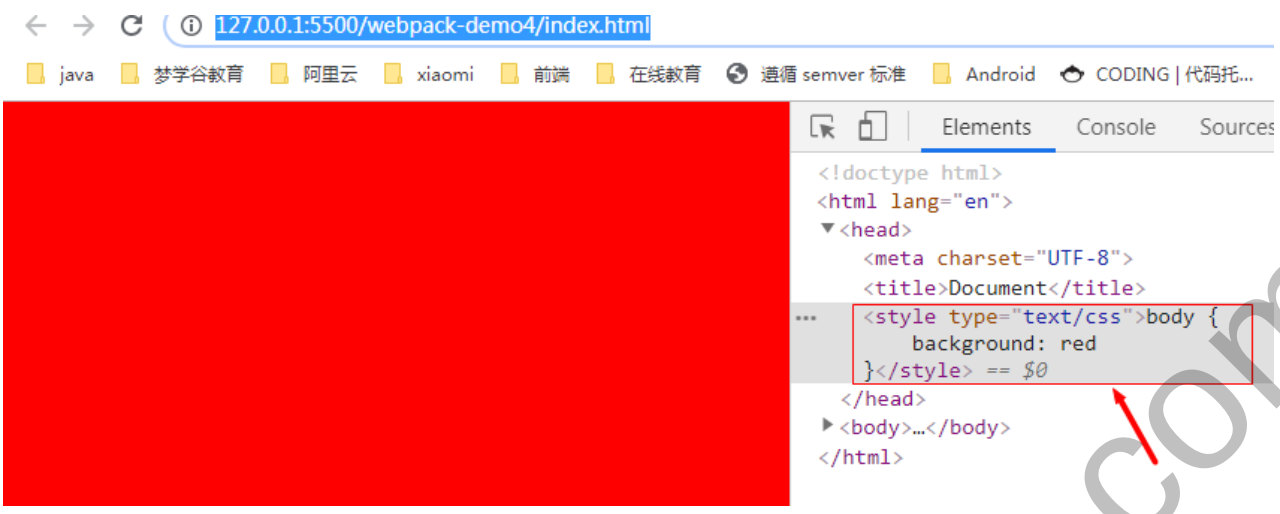
```
1  // 模块方式导入 css ， 最终会打包成js，打包在 bundle.js 中
2  import './css/style.css'
```

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



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 `<sytle>` 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

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

2. 修改 webpack.config.js

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

module.exports = {
    mode: 'none',
    entry: './src/main.js', // 入口模块文件路径
    output: {
        path: path.join(__dirname, './dist/'),  // 打包的结果文件存储目录
        filename: 'bundle.js' // 打包的结果文件名
    },
    module: { // 模块
        rules: [ // 规则
            {
                test: /\.css$/, // 正则表达式，匹配 .css 文件资源
                use: [  // 使用的 Loader ,注意顺序不能错
                    'style-loader',
                    'css-loader'
                ]
            },
            {
                test: /\.(png|svg|jpg|gif)$/,
                use: [
                    'file-loader'
```

```
                ]
            }
        ]
    }
}
```

3. 修改 style.css

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

4. 打包编译

```
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**

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

2. 修改 **webpack.config.js**

```
const path = require("path");
// 引入插件
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
    entry: './src/main.js', // 入口模块文件路径
    output: {
        path: path.join(__dirname, './dist/'),  // 打包的结果文件存储目录
        filename: 'bundle.js' // 打包的结果文件名
```

```
    },
    // 配置插件
    plugins: [
        new HtmlWebpackPlugin({
            // 此插件作用是将 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
            // 注意：其中的文件名 bundle 取决于上面output.filename中指定的名称
            template: './index.html'
        })
    ],
```

3. 修改 index.html, 模拟下vue页面

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <!-- 使用了HtmlWebpackPlugin 插件会自动引入bundle.js中 -->
    <!-- <script src="./dist/bundle.js"></script> -->
    <div id="app"></div>
</body>
```

4. 重新打包

```
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`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <!-- 使用了HtmlWebpackPlugin 插件会自动引入bundle.js中 -->
    <!-- <script src="./dist/bundle.js"></script> -->
    <div id="#app"></div>

<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

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

2. 修改 `webpack.config.js` 配置

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

module.exports = {
    mode: 'none',
    entry: './src/main.js', // 入口模块文件路径
    output: {
        path: path.join(__dirname, './dist/'),  // 打包的结果文件存储目录
        filename: 'bundle.js' // 打包的结果文件名
    },
    // 实时重新加载
    devServer: {
        contentBase: './dist'
    },
```

3. 修改 `package.json` 的 `scripts`

   `--open` 选项打包成功，自动打开浏览器

```
  "scripts": {
    "show": "webpack -v",
    "build": "webpack",
    "watch-build": "webpack --watch",
+   "dev": "webpack-dev-server --open"
  },
```

4. 打包

```
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

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

第六章 Babel 浏览器兼容性

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

6.1 安装 Bable

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

6.2 配置 webapck.config.js

```
module: {
  rules: [
    {
      test: /\.m?js$/,
      exclude: /(node_modules|bower_components)/, // 排除的目录
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env'] // 内置好的转译工具
        }
      }
    }
  ]
}
```

6.3 main.js 代码

```
const a = 1
// a = 2 //直接编译报错

const arr = [1, 2, 3]
arr.forEach(item => {
    console.log(item)
})
```

`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

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

2. 修改 `webpack.config.js` 配置

```
const VueLoaderPlugin = require('vue-loader/lib/plugin')

module.exports = {
  module: {
    rules: [
      // ... 其它规则
      {
        test: /\.vue$/,
        loader: 'vue-loader'
      }
    ]
  },
  plugins: [
    // 请确保引入这个插件！
    new VueLoaderPlugin()
  ]
}
```

完整版本：

```
const path = require("path");
const HtmlWebpackPlugin = require('html-webpack-plugin');
// 1. 加载 Vue Loader 插件
const VueLoaderPlugin = require('vue-loader/lib/plugin')

module.exports = {
    mode: 'none',
    entry: './src/main.js', // 入口模块文件路径
    output: {
        path: path.join(__dirname, './dist/'),  // 打包的结果文件存储目录
        filename: 'bundle.js' // 打包的结果文件名
    },
    devServer: { // 实时重新加载
```

```
        contentBase: './dist'
    },
    plugins: [
        // 2. 引入Vue插件
        new VueLoaderPlugin(),

        new HtmlWebpackPlugin({
            template: './index.html'
        })
    ],
    module: { // 模块
        rules: [ // 规则
            {
                test: /\.css$/, // 正则表达式，匹配 .css 文件资源
                use: [  // 使用的 Loader ,注意顺序不能错
                    'style-loader',
                    'css-loader'
                ]
            },
            {
                test: /\.(png|svg|jpg|gif)$/,
                use: [
                    'file-loader'
                ]
            },
            {
                test: /\.m?js$/,
                exclude: /(node_modules)/, // 排除的目录
                use: {
                  loader: 'babel-loader',
                  options: {
                    presets: ['@babel/preset-env'] // babel中内容的转换规则工具
                  }
                }
            },
            // 3. 配置vue-loader
            {
                test: /\.vue$/,
                use: [
                    'vue-loader'
                ]
            }
        ]
    }
}
```

3. 在 src 目录下创建 `App.vue`

```vue
<template>
    <div >
        <h1>App</h1>
    </div>
</template>
<script>

</script>

<style >
</style>
```

4. 在 main.js 中导入 App.vue

```
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

```
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**

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

2. 安装 vue 模块

```
npm i vue
```

3. index.html 单页面入口

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <!-- Vue 入口 -->
    <div id="app"></div>
</body>
```

4. App.vue 根组件

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

main.js 打包入口文件

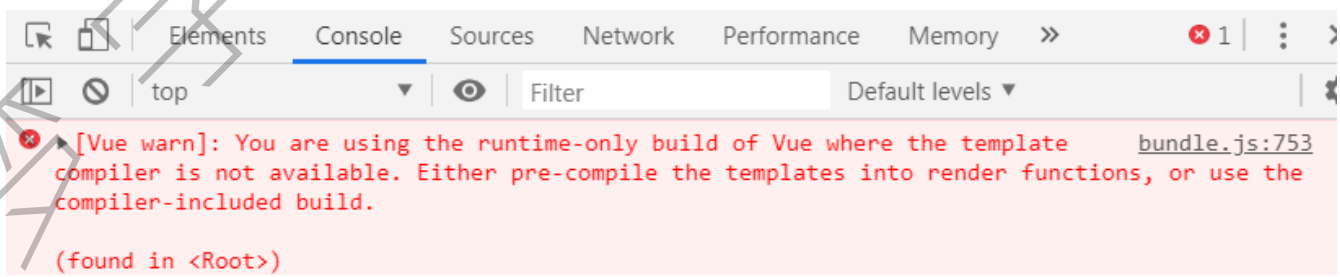
```
import Vue from 'vue'
import App from './App.vue'

new Vue({
    el: '#app',
    template: '<App />',
    components: {App}
})
```

打包构建

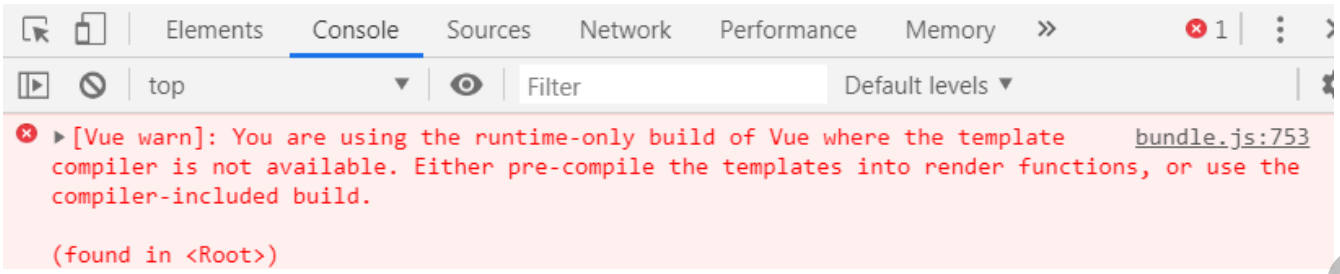
```
npm run build
```

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 查看控制台发现报警告：



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：

```
"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：

```
- 完整版：同时包含编译器和运行时的版本。
- 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。
- 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

```
import Vue from 'vue/dist/vue.js'

new Vue({
    el: '#app',
    template: '<App />',
    components: {App}
})
```

- 第2种方法：
    1. 依旧 `import Vue from 'vue'`

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

new Vue({
    el: '#app',
    template: '<App />',
    components: {App}
})
```

2. 然后 `webpack.config.js` 增加一个属性

```
module.exports = {
    // 解析完整版 vue.js
    resolve: {
        alias: {
            'vue$': 'vue/dist/vue.js'
        }
    }
}
```

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

```
// 方式3：采用 render 函数渲染组件
import Vue from 'vue'
import App from './App.vue'

new Vue({
    el: '#app',
    // 使用render后，当前可不使用 componntns
    // 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
    /* render: function (h) {
        return h(App)
    } */
    render: h => h(App) // ES6 箭头函数
})
```

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。

```vue
<template>
    <!-- 组件模板，.vue文件中可只出现 template 标签 -->
    <div>
        <h1>App 根组件</h1>
        <h2>{{ msg }}</h2>
        <!-- 引用子组件 -->
        <foo></foo>
    </div>
</template>

<script>
// 1. 先导入 Vue 组件，再使用
import Foo from './components/Foo.vue'

// 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，
// template 不用写，因为上面标签就是模板对象
export default {
    data () {
        return {
            msg: 'hello webpack'
        }
    },
    // 引用Foo子组件
    components: {
        Foo
    }
}
</script>

<style scoped>
    /*
      默认 CSS 样式会自动传递到父子组件中
      scoped 指定后，CSS样式只在当前组件中有效
     */
    h1 {
        color: red
    }
</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。

```vue
<template>
    <div>
        <h1>我是 Foo 子组件</h1>
        <h2>我是 Foo 子组件</h2>
    </div>
</template>

<script>
export default {
```

```
}
</script>

<style scoped>
    h2{
        color: blue
    }
</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：

```
const path = require("path");
const HtmlWebpackPlugin = require('html-webpack-plugin');
const VueLoaderPlugin = require('vue-loader/lib/plugin');
+++ const webpack = require('webpack');

module.exports = {
    mode: 'none',
    entry: './src/main.js', // 入口模块文件路径
    output: {
        path: path.join(__dirname, './dist/'),  // 打包的结果文件存储目录
        filename: 'bundle.js' // 打包的结果文件名
    },
    devServer: { // 实时重新加载
        contentBase: './dist',
+++         hot: true
    },
    plugins: [
        // 模块热替换
+++       new webpack.HotModuleReplacementPlugin(),

        // 引入Vue插件
```

```
        new VueLoaderPlugin(),
        new HtmlWebpackPlugin({
            template: './index.html'
        })
    ],
    module: { // 模块
        rules: [ // 规则
            {
                test: /\.css$/, // 正则表达式，匹配 .css 文件资源
                use: [  // 使用的 Loader ,注意顺序不能错
                    'style-loader',
                    'css-loader'
                ]
            },
            {
                test: /\.(png|svg|jpg|gif)$/,
                use: [
                    'file-loader'
                ]
            },
            {
                test: /\.vue$/,
                use: [
                    'vue-loader'
                ]
            }
        ]
    }
}
```

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

```
npm run dev
```

可能会报以下错:

```
internal/modules/cjs/loader.js:584
    throw err;
    ^

Error: Cannot find module 'uuid/v4'
```

执行命令，解决重新安装 webpack-dev-server：

```
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。