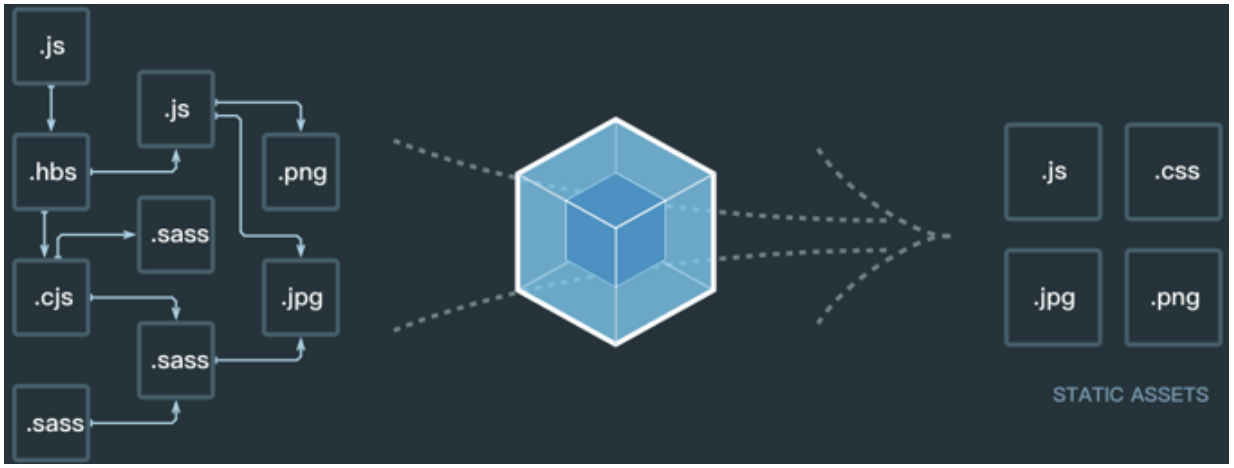
# 1. 什么是Webpack?

- 什么是webpack?
  - 这个webpack还真不是一两句话可以说清楚的。
- 我们先看看官方的解释：
  - At its core, **webpack** is a *static module bundler* for modern JavaScript applications.
  - 从本质上来讲，webpack是一个现代的JavaScript应用的静态**模块打包**工具。
- 但是它是什么呢？用概念解释概念，还是不清晰。
  - 我们从两个点来解释上面这句话：**模块** 和 **打包**




# 2. 前端模块化

> 前端模块化：

- 在前面学习中，我已经用了大量的篇幅解释了为什么前端需要模块化。
- 而且我也提到了目前使用前端模块化的一些方案：AMD、CMD、CommonJS、ES6。
- 在ES6之前，我们要想进行模块化开发，就必须借助于其他的工具，让我们可以进行模块化开发。
- 并且在通过模块化开发完成了项目后，还需要处理模块间的各种依赖，并且将其进行整合打包。

- 而webpack其中一个核心就是让我们可能进行模块化开发，并且会帮助我们处理模块间的依赖关系。
- 而且不仅仅是JavaScript文件，我们的CSS、图片、json文件等等在webpack中都可以被当做模块来使用（在后续我们会看到）。
- 这就是webpack中模块化的概念。

# 3. 打包如何理解呢?

- 理解了webpack可以帮助我们进行模块化，并且处理模块间的各种复杂关系后，打包的概念就非常好理解了
- 就是将webpack中的各种资源模块进行打包合并成一个或多个包(Bundle)。
- 并且在打包的过程中，还可以对资源进行处理，比如压缩图片，将scss转成css，将ES6语法转成ES5语法，将TypeScript转成JavaScript等等操作。
- 但是打包的操作似乎grunt/gulp也可以帮助我们完成，它们有什么不同呢?

# 4. 和grunt/gulp的对比

> grunt/gulp的核心是Task

- 我们可以配置一系列的task，并且定义task要处理的事务（例如ES6、ts转化，图片压缩，scss转成css）
- 之后让grunt/gulp来依次执行这些task，而且让整个流程自动化。
- 所以grunt/gulp也被称为前端自动化任务管理工具。

> 我们来看一个gulp的task

- 下面的task就是将src下面的所有js文件转成ES5的语法。
- 并且最终输出到dist文件夹中。

```
const gulp = require('gulp');
const babel = require('gulp-babel');

gulp.task('js', () =>
    gulp.src('src/*.js')
        .pipe(babel({
            presets: ['es2015']
        }))
        .pipe(gulp.dest('dist'))
);
```

> 什么时候用grunt/gulp呢?

- 如果你的工程模块依赖非常简单，甚至是没有用到模块化的概念。
- 只需要进行简单的合并、压缩，就使用grunt/gulp即可。
- 但是如果整个项目使用了模块化管理，而且相互依赖非常强，我们就可以使用更加强大的webpack了。

> 所以，grunt/gulp和webpack有什么不同呢?

- grunt/gulp更加强调的是前端流程的自动化，模块化不是它的核心。
- webpack更加强调模块化开发管理，而文件压缩合并、预处理等功能，是他附带的功能。

# 5. webpack安装

## 5.1. webpack和node和npm关系

- webpack模块打包工具为了能正常运行，必须依赖node环境
- node环境为了可以正常的执行很多代码，其中必须包含各种依赖的包
- npm工具(node packages manager)用于包的管理

## 5.2. 安装步骤

- 安装webpack首先需要安装Node.js，Node.js自带了软件包管理工具npm

- 查看自己的node版本：一定要大于8.0，因为后期使用Vue Cli脚手架对node版本有要求

```
node -v
```

- 全局安装webpack(这里我先指定版本号3.6.0，因为vue cli2依赖该版本)
  - -g 全局安装

```
npm install webpack@3.6.0 -g
```

- 局部安装webpack（后续才需要）
  - `--save-dev`是开发时依赖，项目打包后不需要继续使用的。

```
npm install webpack@3.6.0 --save-dev
```

- 为什么全局安装后，还需要局部安装呢?
  - 在终端直接执行webpack命令，使用的全局安装的webpack
  - 当在package.json中定义了scripts时，其中包含了webpack命令，那么使用的是局部webpack

# 6. webpack基本使用过程

## 6.1. 准备工作

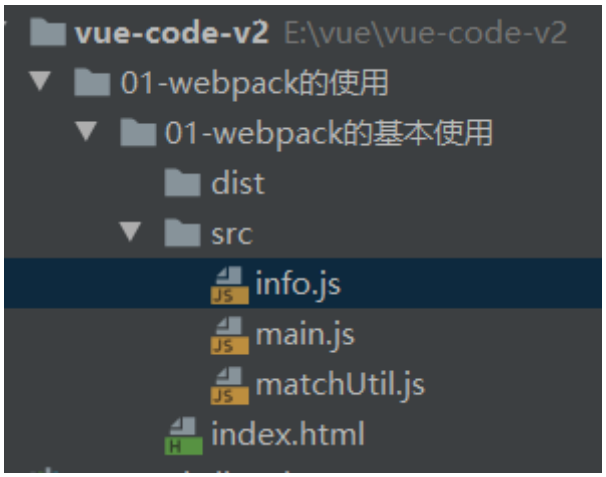
- 我们创建如下文件和文件夹：

- **文件和文件夹解析：**
  - dist文件夹：用于存放之后打包的文件
  - src文件夹：用于存放我们写的源文件
    - main.js：项目的入口文件。具体内容查看下面详情。
    - mathUtils.js：定义了一些数学工具函数，可以在其他地方引用，并且使用。具体内容查看下面的详情。
  - index.html：浏览器打开展示的首页html

- package.json：通过npm init生成的，npm包管理的文件（暂时没有用上，后面才会用上）

- mathUtils.js文件中的代码：

- main.js文件中的代码：




## 6.2. 模块化开发

> matchUtil.js

```
let tag = "webpack标签";
function add(number1,number2) {
  return number1 + number2;
}
function mul(number1,number2) {
  return number2*number1;
}

// 使用CommonJs模块化导出
module.exports = {
  tag, add, mul
}
```

> info.js

```
// 使用ES6模块化进行开发
export let name = "guangyang";
export let age = 24;
```

main.js

```
// 使用CommonJs模块化导入
const match = require("./matchUtil");
console.log(match.tag)
console.log(match.add(100, 200));
console.log(match.mul(100, 200));

// 使用ES6进行导入,最后由webpack进行打包，这里的info可以不加后缀
import {name,age} from "./info";
console.log(age);
console.log(name);
```

index.html

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

# 6.3. js打包

- 现在的js文件中使用了模块化的方式进行开发，他们可以直接使用吗？不可以。
  - 因为如果直接在index.html引入这两个js文件，浏览器并不识别其中的模块化代码。
  - 另外，在真实项目中当有许多这样的js文件时，我们一个个引用非常麻烦，并且后期非常不方便对它们进行管理。
- 我们应该怎么做呢？使用webpack工具，对多个js文件进行打包。
  - 我们知道，webpack就是一个模块化的打包工具，所以它支持我们代码中写模块化，可以对模块化的代码进行处理。（如何处理的，待会儿在原理中，我会讲解）

    - 另外，如果在处理完所有模块之间的关系后，将多个js打包到一个js文件中，引入时就变得非常方便了。

- 打包后会在dist文件下，生成一个bundle.js文件
    - 文件内容有些复杂，这里暂时先不看，后续再进行分析。
    - bundle.js文件，是webpack处理了项目直接文件依赖后生成的一个js文件，我们只需要将这个js文件在index.html中引入即可

> 如何打包呢

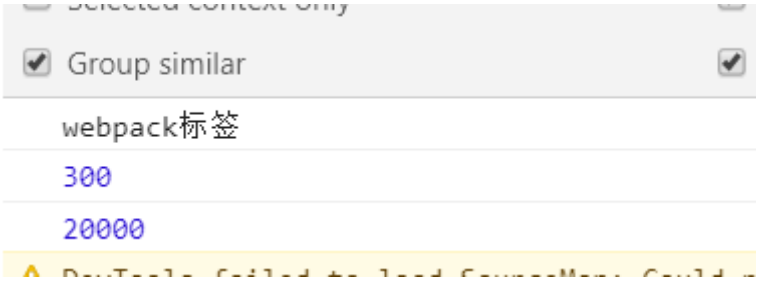
- 这里你可能会有疑问，为什么只对main.js打包呢?
    - 因为这是一个入口，在这个入口中我导入了matchUtil.js和info.js模块，webpack我自动找到这两个文件进行打包

```
webpack ./src/main.js ./dist/bundle.js
```

```
Version: webpack 3.6.0
Time: 40ms
    Asset     Size  Chunks             Chunk Names
bundle.js  2.95 kB       0  [emitted]  main
   [0] ./src/main.js 169 bytes {0} [built]
   [1] ./src/matchUtil.js 230 bytes {0} [built]
```

# 6.4. 使用打包后的文件

- 打包后会在dist文件下，生成一个bundle.js文件
    - 文件内容有些复杂，这里暂时先不看，后续再进行分析。
    - bundle.js文件，是webpack处理了项目直接文件依赖后生成的一个js文件，我们只需要将这个js文件在index.html中引入即可




# 7. webpack.config.js和package.json配置

## 7.1. 入口和出口

- 我们考虑一下，如果每次使用webpack的命令都需要写上入口和出口作为参数，就非常麻烦，有没有一种方法可以将这两个参数写到配置中，在运行时，直接读取呢?
- 当然可以，就是创建一个webpack.config.js文件

## 7.2. webpack.config.js

```
// path模块是node里的包不需要我们写
const path = require("path");

module.exports = {
  // 入口: 可以是字符串/数组/对象,这里我们的入口只有一个，所以写一个字符串即可
  entry: "./src/main.js",
  // 出口: 通常是一个对象，里面至少包含两个重要属性,path和filename
  output: {
    // __dirname是node中的命令，他会获取webpack.config.js的绝对路径
    path: path.resolve(__dirname,'dist'),// path: 通常是一个绝对路径
    filename: "bundle.js"
  }
}
```

这回就可以使用webpack进行打包了

```
E:\vue\vue-code-v2\01-webpack的使用\02-webpack的配置>webpack
Hash: 698cb90b2637f4841e6c
Version: webpack 3.6.0
Time: 45ms
    Asset     Size  Chunks             Chunk Names
bundle.js  3.75 kB       0  [emitted]  main
   [0] ./src/main.js 329 bytes {0} [built]
   [1] ./src/matchUtil.js 230 bytes {0} [built]
   [2] ./src/info.js 87 bytes {0} [built]
```

# 7.3. 局部安装webpack

> 只要在命令行中输入的webpack都是使用全局的webpack版本

> 目前，我们使用的webpack是全局的webpack，如果我们想使用局部来打包呢？

- 因为一个项目往往依赖特定的webpack版本，全局的版本可能很这个项目的webpack版本不一致，导出打包出现问题。
- 所以通常一个项目，都有自己局部的webpack。

> 第一步，项目中需要安装自己局部的webpack

- 这里我们让局部安装webpack3.6.0
- Vue CLI3中已经升级到webpack4，但是它将配置文件隐藏了起来，所以查看起来不是很方便。
- `--save-dev`就是局部安装，开发时使用，真正打包后就不需要webpack了

```
npm install webpack@3.6.0 --save-dev
```

> 第二步，通过node_modules/.bin/webpack启动webpack打包

- 如果直接在终端中输入webpack进行打包还是使用全局的webpack版本

```
bogon:02-webpac配置 xmg$ node_modules/.bin/webpack

Webpack is watching the files…

Hash: d5221bf06be28062ca8a
Version: webpack 3.6.0
Time: 83ms
    Asset     Size  Chunks             Chunk Names
bundle.js  2.82 kB       0  [emitted]  main
   [0] ./src/main.js 131 bytes {0} [built]
   [1] ./src/mathUtils.js 138 bytes {0} [built]
```

# 7.4. package.json中定义启动

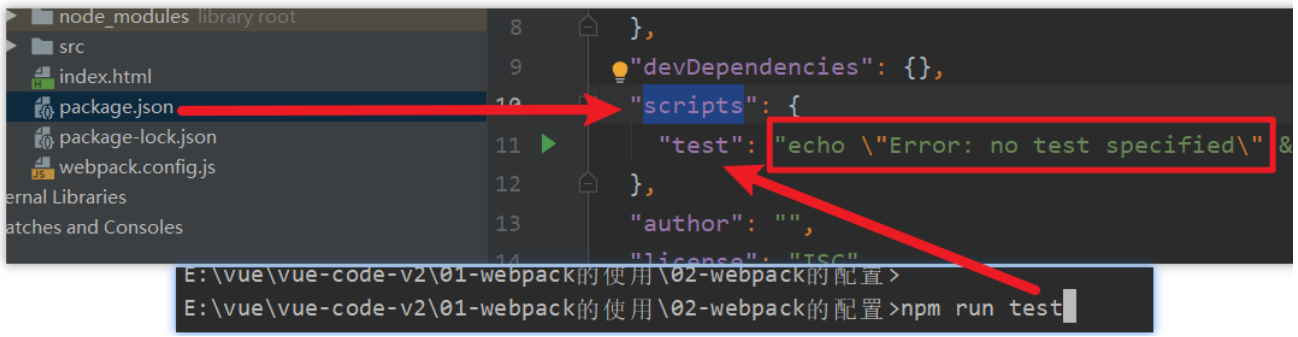
> 但是，每次执行都敲这么一长串有没有觉得不方便呢？

- OK，我们可以在package.json的scripts中定义自己的执行脚本。

> package.json中的scripts的脚本在执行时，会按照一定的顺序寻找命令对应的位置。

- 首先，会寻找本地的node_modules/.bin路径中对应的命令。
- 如果没有找到，会去全局的环境变量中寻找。
- 如何执行我们的build指令呢?
  - 在终端中执行`npm init`生成package.json文件
  - 在package.json文件中有一个scripts属性，在这个属性中定义脚本，使用`npm run 脚本名称`
  - 在scripts属性中添加`"build": "webpack"`
  - 执行脚本，这个会优先执行局部的webpack，也就是本地的webpack

```
npm run build
```



```
E:\vue\vue-code-v2\01-webpack的使用\02-webpack的配置>npm run build

> meetwebpack@1.0.0 build E:\vue\vue-code-v2\01-webpack的使用\02-webpack的配置
> webpack

Hash: 698cb90b2637f4841e6c
Version: webpack 3.6.0
Time: 42ms
    Asset     Size  Chunks             Chunk Names
bundle.js  3.75 kB       0  [emitted]  main
   [0] ./src/main.js 329 bytes {0} [built]
   [1] ./src/matchUtil.js 230 bytes {0} [built]
   [2] ./src/info.js 87 bytes {0} [built]
```

# 8. webpack中使用css文件的配置

## 8.1. 什么是loader?

> loader是webpack中一个非常核心的概念。

> nwebpack用来做什么呢?

- 在我们之前的实例中，我们主要是用webpack来处理我们写的js代码，并且webpack会自动处理js之间相关的依赖。
- 但是，在开发中我们不仅仅有基本的js代码处理，我们也需要加载css、图片，也包括一些高级的将ES6转成ES5代码，将TypeScript转成ES5代码，将scss、less转成css，将.jsx、.vue文件转成js文件等等。
- 对于webpack本身的能力来说，对于这些转化是不支持的。
- 那怎么办呢？给webpack扩展对应的loader就可以啦。

> loader使用过程:

- 步骤一：通过npm安装需要使用的loader
- 步骤二：在webpack.config.js中的modules关键字下进行配置

> 大部分loader我们都可以在webpack的官网中找到，并且学习对应的用法。

## 8.2. css文件处理 - 准备工作

> 项目开发过程中，我们必然需要添加很多的样式，而样式我们往往写到一个单独的文件中。

- 在src目录中，创建一个css文件，其中创建一个normal.css文件。
- 我们也可以重新组织文件的目录结构，将零散的js文件放在一个js文件夹中。

> normal.css中的代码非常简单，就是将body设置为red，并且设置h2标识是蓝色

```
body {
    background-color: red;
}

.font {
    color: aqua;
}
```

> index.html代码如下

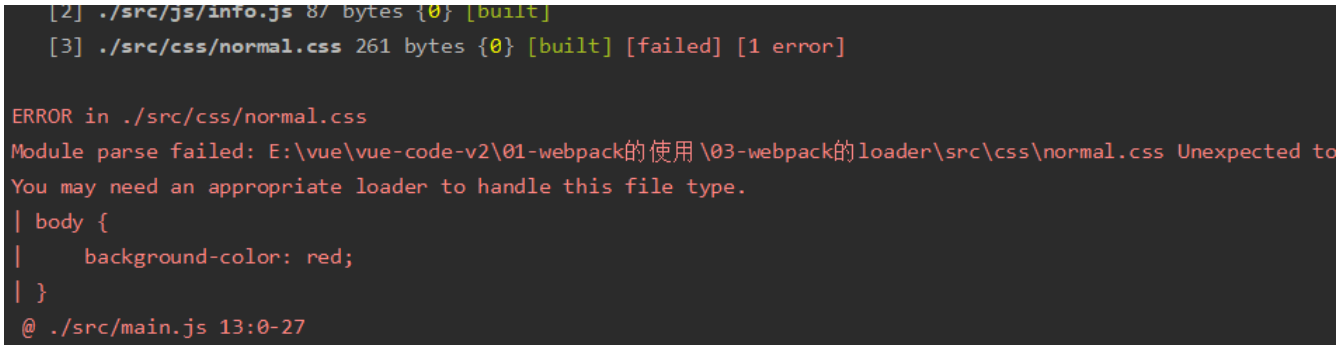
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  <h2 class="font">css</h2>
  <script src="./dist/bundle.js"></script>
</body>
</html>
```

> 但是，这个时候normal.css中的样式会生效吗？

- 当然不会，因为我们压根就没有引用它。
- webpack也不可能找到它，因为我们只有一个入口，webpack会从入口开始查找其他依赖的文件。


## 8.3. css文件处理 – 打包报错信息

> 重新打包，会出现如下错误：npm run build




> 这个错误告诉我们：加载normal.css文件必须有对应的loader。

## 8.4. css文件处理 – css-loader

> 在webpack的官方中，我们可以找到如下关于样式的loader使用方法：

- 安装loader

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

- 在main.js文件中引入css文件

```
// 在这里必须引入css样式模块
require("./css/normal.css")
```

> 按照官方配置webpack.config.js文件

- 注意：配置中有一个style-loader，我们并不知道它是什么，所以可以暂时不进行配置。

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [ 'css-loader' ]
      }
    ]
  }
}
```

> 重新打包项目：

- 但是，运行index.html，你会发现样式并没有生效。
- 原因是css-loader只负责加载css文件，但是并不负责将css具体样式嵌入到文档中。
- 这个时候，我们还需要一个style-loader帮助我们处理。

# 8.5. css文件处理 – style-loader

> 我们来安装style-loader

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

- 注意：style-loader需要放在css-loader的前面。

- 疑惑：不对吧？按照我们的逻辑，在处理css文件过程中，应该是css-loader先加载css文件，再由style-loader来进行进一步的处理，为什么会将style-loader放在前面呢？
- 答案：这次因为webpack在读取使用的loader的过程中，是按照从右向左的顺序读取的。
- 目前，webpack.config.js的配置如下

```
// path模块是node里的包不需要我们写
const path = require("path");

module.exports = {
  // 入口：可以是字符串/数组/对象,这里我们的入口只有一个，所以写一个字符串即可
  entry: "./src/main.js",
  // 出口：通常是一个对象，里面至少包含两个重要属性,path和filename
  output: {
    // __dirname是node中的命令，他会获取webpack.config.js的绝对路径
    path: path.resolve(__dirname,'dist'),// path：通常是一个绝对路径
    filename: "bundle.js"
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [ 'style-loader', 'css-loader' ]
      }
    ]
  }
}
```


# 9. webpack中less文件的处理

## 9.1. less文件处理 – 准备工作

- 如果我们希望在项目中使用less、scss、stylus来写样式，webpack是否可以帮助我们处理呢?
  - 我们这里以less为例，其他也是一样的。
- 我们还是先创建一个less文件，依然放在css文件夹中

```
@fontSize:50px;
@fontColor: #ff7979;

.font-v2{
  font-size: @fontSize;
  color: @fontColor;
}
```

- main.js文件中引入less并且创建一个div加上样式

```
// 使用CommonJs模块化导入
const match = require("./js/matchUtil");
console.log(match.tag)
console.log(match.add(100, 200));
console.log(match.mul(100, 200));

// 使用ES6进行导入,最后由webpack进行打包，这里的info可以不加后缀
import {name,age} from "./js/info";
console.log(age);
console.log(name);

// 在这里必须引入css样式模块
require("./css/normal.css")
// 引入less样式模块
require("./css/special.less")
// 为了查看less的最终效果，创建一个div并且加上font-v2样式
document.writeln(`
  <div class="font-v2">
    你好啊，阳哥
  </div>
`)
```

# 9.2. less文件处理 – less-loader

- 继续在官方中查找，我们会找到less-loader相关的使用说明

- 首先，还是需要安装对应的loader
- 注意：我们这里还安装了less，因为webpack会使用less对less文件进行编译

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

- 其次，修改对应的配置文件
- 添加一个rules选项，用于处理.less文件

## 脚手架2 vuecli2

### webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.less$/,
        loader: 'less-loader', // compiles Less to CSS
      },
    ],
  },
};
```

```
{
        test: /\.less$/,
        use: [{
                loader: "style-loader" // creates style nodes from JS strings
        }, {
                loader: "css-loader" // translates CSS into CommonJS
        }, {
                loader: "less-loader" // compiles Less to CSS
        }]
}
```

- 开始打包

```
npm run build
```

# 10. webpack中图片文件处理

## 10.1. 资源准备阶段

> 首先，我们在项目中加入两张图片：

- 一张较小的图片test01.jpg(小于8kb)，一张较大的图片test02.jpeg(大于8kb)
- 待会儿我们会针对这两张图片进行不同的处理

> 我们先考虑在css样式中引用图片的情况，所以我更改了normal.css中的样式：

```
body {
    background-color: red;
    background: url("../img/test01.jpg");
}
```

> 如果我们现在直接打包，会出现如下问题

```
ERROR in ./src/img/test01.jpg
Module parse failed: E:\vue\vue-code-v2\01-webpack的使用\03-webpack的loader\src\img\test01.jpg Unexpected character '' (1:0)
You may need an appropriate loader to handle this file type.
(Source code omitted for this binary file)
 @ ./node_modules/css-loader/dist/cjs.js!./src/css/normal.css 4:36-64
 @ ./src/css/normal.css
 @ ./src/main.js
```

## 10.2. url-loader

> 图片处理，我们使用url-loader来处理，依然先安装url-loader

```
npm install --save-dev url-loader
```

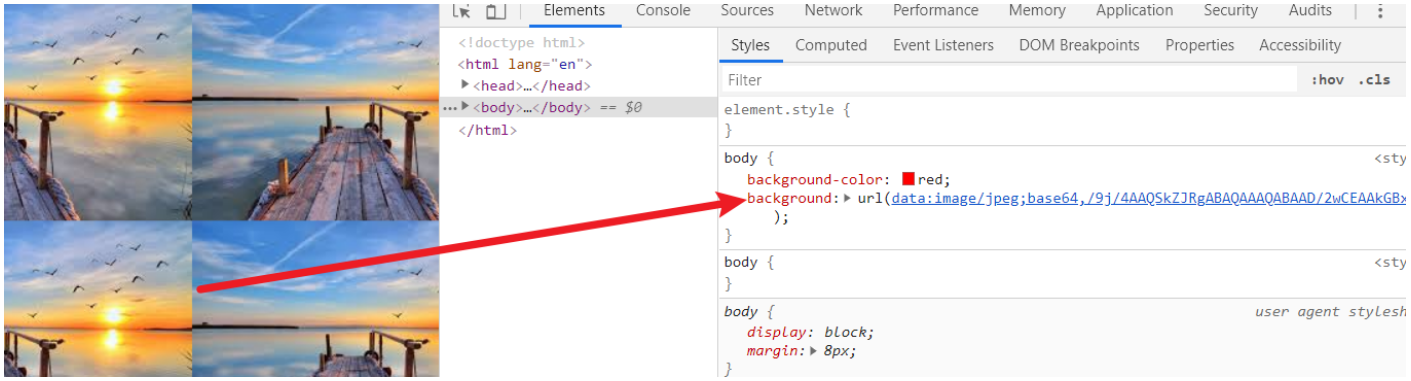
> 修改webpack.config.js配置文件：

```
{
        test: /\.(png|jpg|gif)$/,
        use: [
```

```
        {
              loader: 'url-loader',
              options: {
                limit: 8192
              }
        }
      ]
}
```

> 再次打包，运行index.html，就会发现我们的背景图片选出了出来。

- 而仔细观察，你会发现背景图是通过base64显示出来的
- OK，这也是limit属性的作用，当图片小于8kb时，对图片进行base64编码

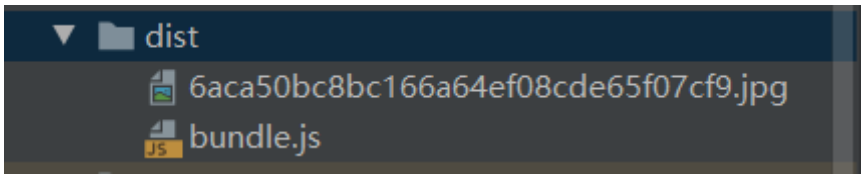



# 10.3. file-loader

- 那么问题来了，如果大于8kb呢？我们将background的图片改成test02.jpg
  - 这次因为大于8kb的图片，会通过file-loader进行处理，但是我们的项目中并没有file-loader
- 所以，我们需要安装file-loader

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

- 再次打包，就会发现dist文件夹下多了一个图片文件



# 10.4. 修改文件名称

> 我们发现webpack自动帮助我们生成一个非常长的名字

- 这是一个32位hash值，目的是防止名字重复
- 但是，真实开发中，我们可能对打包的图片名字有一定的要求
- 比如，将所有的图片放在一个文件夹中，跟上图片原来的名称，同时也要防止重复

> 所以，我们可以在options中添加上如下选项：

- img：文件要打包到的文件夹
- name：获取图片原来的名字，放在该位置
- hash:8：为了防止图片名称冲突，依然使用hash，但是我们只保留8位
- ext：使用图片原来的扩展名

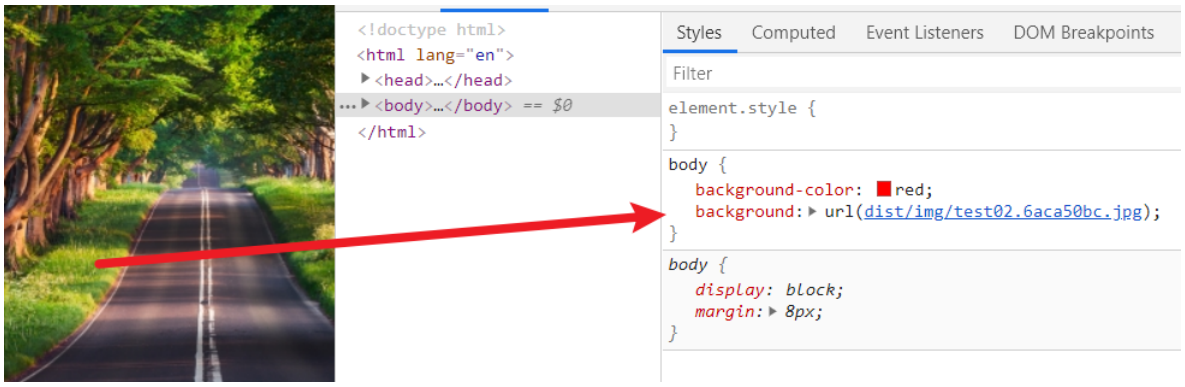
```
{
  test: /\.(png|jpg|gif)$/,
  use: [
    {
      loader: 'url-loader',
      options: {
        limit: 8192,
        name: 'img/[name].[hash:8].[ext]'
      }
    }
  ]
}
```

> 但是，我们发现图片并没有显示出来，这是因为图片使用的路径不正确

- 默认情况下，webpack会将生成的路径直接返回给使用者,在使用bundle.js的html文件目录下
- 但是，我们整个程序是打包在dist文件夹下的，所以这里我们需要在路径下再添加一个dist/

```
output: {
  // __dirname是node中的命令，他会获取webpack.config.js的绝对路径
  path: path.resolve(__dirname,'dist'),// path: 通常是一个绝对路径
  filename: "bundle.js",
```

```
  publicPath: "dist/"
},
```




# 11. ES6转ES5的babel

- 如果你仔细阅读webpack打包的js文件，发现写的ES6语法并没有转成ES5，那么就意味着可能一些对ES6还不支持的浏览器没有办法很好的运行我们的代码。
    - 在前面我们说过，如果希望将ES6的语法转成ES5，那么就需要使用babel。
- 而在webpack中，我们直接使用babel对应的loader就可以了。

```
npm install --save-dev babel-loader@7 babel-core babel-preset-es2015
```

- 配置webpack.config.js文件

```
{
  test: /\.js$/,
  exclude: /(node_modules|bower_components)/,
  use: {
        loader: 'babel-loader',
        options: {
          presets: ['es2015']
        }
  }
}
```

- 重新打包，查看bundle.js文件，发现其中的内容变成了ES5的语法

# 12. 使用Vue的配置过程

## 12.1. 前期准备

> 后续项目中，我们会使用Vuejs进行开发，而且会以特殊的文件来组织vue的组件。

- 所以，下面我们来学习一下如何在我们的webpack环境中集成Vuejs

> 现在，我们希望在项目中使用Vuejs，那么必然需要对其有依赖，所以需要先进行安装

- 注：因为我们后续是在实际项目中也会使用vue的，所以并不是开发时依赖，默认安装最新版本

```
npm install vue --save
```

> 那么，接下来就可以按照我们之前学习的方式来使用Vue了

- main.js代码

```
// 导入vue模块
import Vue from "vue";
new Vue({
  el: "#app",
  data: {
    message: "vue webpack"
  }
});
```

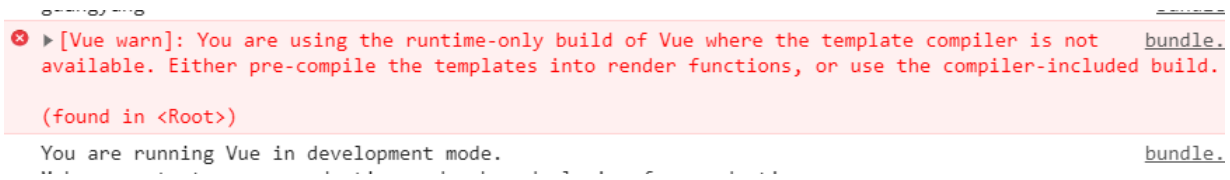

- index.html代码

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  <h2 class="font">css</h2>
  <div id="app">
    {{message}}}
  </div>
  <script src="./dist/bundle.js"></script>
</body>
</html>
```

# 12.2. 打包项目 – 错误信息

> 修改完成后，重新打包，运行程序：

- 打包过程没有任何错误(因为只是多打包了一个vue的js文件而已)
- 但是运行程序，没有出现想要的效果，而且浏览器中有报错




- 这个错误说的是我们打包时候使用的是runtime-only版本的Vue，什么意思呢?

- Vue不同版本构建，

> runtime-only和runtime-compiler的区别。

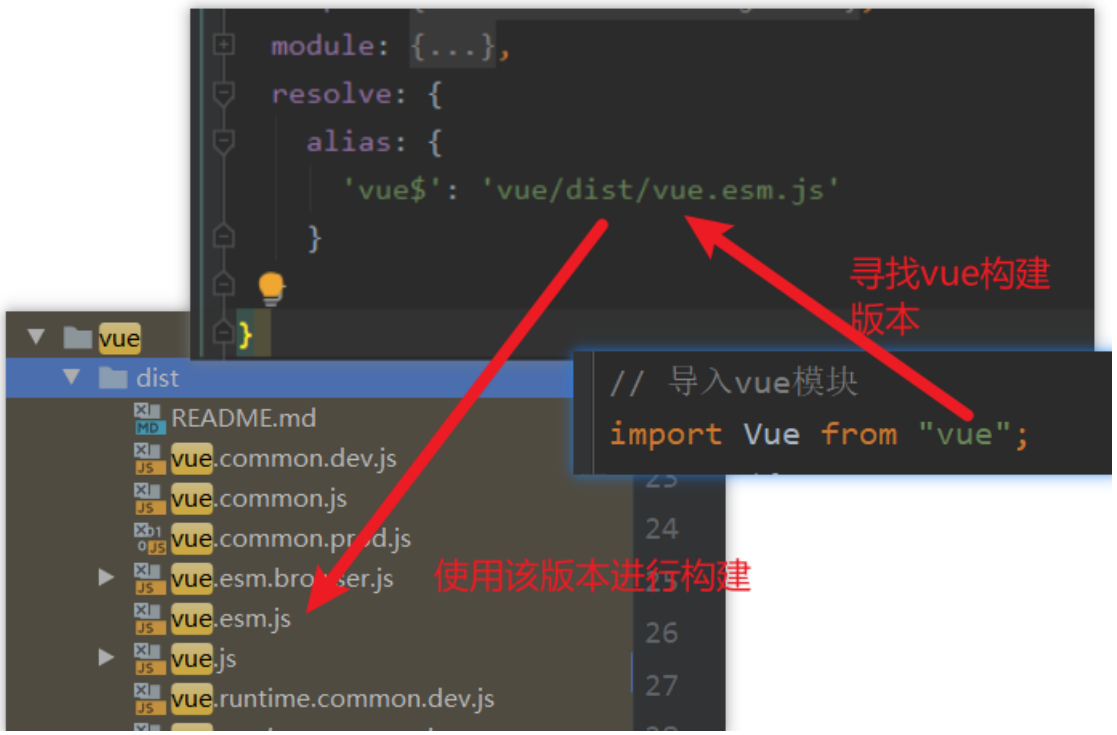
- runtime-only: 代码中，不可以有任何的template
- runtime-compiler：代码中，可以有template，因为有compiler可以编译template

> 这时候你可能会疑问?我的index.html和main.js代码中也没有任何的template


> 所以我们修改webpack的配置，添加如下内容即可

```
resolve: {
  alias: {
    'vue$': 'vue/dist/vue.esm.js'
  }
}
```

- 当导入vue模块时候，寻找的vue.esm.js版本的vue进行打包




# 13. el和template区别

> 正常运行之后，我们来考虑另外一个问题：

- 如果我们希望将data中的数据显示在界面中，就必须是修改index.html
- 如果我们后面自定义了组件，也必须修改index.html来使用组件
- 但是html模板在之后的开发中，我并不希望手动的来频繁修改，是否可以做到呢？

> 定义template属性：

- 在前面的Vue实例中，我们定义了el属性，用于和index.html中的#app进行绑定，让Vue实例之后可以管理它其中的内容
- 这里，我们可以将div元素中的{{message}}内容删掉，只保留一个基本的id为div的元素
- 但是如果我依然希望在其中显示{{message}}的内容，应该怎么处理呢？
- 我们可以再定义一个template属性，代码如下：

```
// 导入vue模块
import Vue from "vue";
new Vue({
  el: "#app",
  template: `<div id="app">{{message}}</div>`,
  data: {
    message: "vue webpack"
  }
});
```

+ index.html代码如下,一般不需要频繁修改

```
<body>
  <div class="font" id="app">
  </div>
  <script src="./dist/bundle.js"></script>
</body>
```

> 重新打包，运行程序，显示一样的结果和HTML代码结构

> 那么，el和template模板的关系是什么呢？

- 在我们之前的学习中，我们知道el用于指定Vue要管理的DOM，可以帮助解析其中的指令、事件监听等等。
- 而如果Vue实例中同时指定了template，那么template模板的内容会替换掉挂载的对应el的模板。

> 这样做有什么好处呢？

- 这样做之后我们就不需要在以后的开发中再次操作index.html，只需要在template中写入对应的标签即可

> 但是，书写template模块非常麻烦怎么办呢？

- 没有关系，稍后我们会将template模板中的内容进行抽离。

- 会分成三部分书写：template、script、style，结构变得非常清晰。

# 14. Vue的终极使用方案

## 14.1. Vue组件化开发引入

- 在学习组件化开发的时候，我说过以后的Vue开发过程中，我们都会采用组件化开发的思想。
    - 那么，在当前项目中，如果我也想采用组件化的形式进行开发，应该怎么做呢?
- 查看下面的代码：
    - 当然，我们也可以将下面的代码抽取到一个js文件中，并且导出。

```
// 导入vue模块
import Vue from "vue";

const App = {
  template: `
    <div>
          <h2>{{message}}</h2>
    </div>
  `,
  data() {
    return {
      message: "vue webpack"
    }
  }
}

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

- 使用js的方法，在js文件夹中创建一个App.js文件

```
export default  {
  template: `
    <div>
      <h2>{{message}}</h2>
    </div>
  `,
  data() {
    return {
      message: "vue webpack"
    }
  }
}
```

- 在main.js中引入，把App.js删除把！

```
// 导入vue模块
import Vue from "vue";
import App from "./vue/App.vue"

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

# 14.2. .vue文件封装处理

- 但是一个组件以一个js对象的形式进行组织和使用的时候是非常不方便的
  - 一方面编写template模块非常的麻烦
  - 另外一方面如果有样式的话，我们写在哪里比较合适呢?

> 创建vue文件，将组件进行分离

- 创建vue文件

```
<!--模板-->
<template>
  <div>
    <h2 class="title">{{message}}</h2>
  </div>
</template>

<!--组件-->
<script>
  export default {
    name: "App",
    data() {
      return {
        message: "vue webpack"
      }
    }
  }
</script>

<!--定义样式-->
<style scoped>
  .title {
    color: #ff7979;
    font-size: 60px;
  }
</style>
```

- 在main.js中导入App.vue模块

```
// 导入vue模块
import Vue from "vue";
import App from "./vue/App"
new Vue({
  el: "#app",
  template: `<App/>`,
  data: {
  },
  components: {
    App
  }
});
```

> 但是，这个时候这个文件可以被正确的加载吗？

- 必然不可以，这种特殊的文件以及特殊的格式，必须有人帮助我们处理。
- 谁来处理呢？vue-loader以及vue-template-compiler

```
npm install vue-loader vue-template-compiler --save-dev
```

- 修改webpack.config.js的配置文件：

```
{
  test: /\.vue$/,
  use: ['vue-loader']
}
```

- 打包

```
npm run build
```

# 14.3. 打包——错误信息

```
ERROR in ./src/vue/App.vue
vue-loader was used without the corresponding plugin. Make sure to include VueLoaderPlugin in your webpack config.
 @ ./src/main.js 9:11-35
```

> 错误原因以及解决办法

- 在vue-loader14.0.0版本以上需要安装corresponding plugin插件
- 选择降低vue-loader版本到13的方式解决
- 修改package.json


- 执行 `npm install` 命令

- 重新打包

## 14.4. 以后的vue开发模式

- 在src/vue文件夹中创建一个App.vue组件作为根组件
- 再创建一个Cpn.vue组件，并且导出组件，在App.vue中引用该组件

> Cpn.vue组件

```
<template>
  <div class="div-class">
    <h2>我是Cpn组件</h2>
    <p>我是Cpn组件内容</p>
    <h2>{{name}}</h2>
  </div>
</template>

<script>
  export default {
    name: "Cpn",
    data() {
      return {
        name: "我是Cpn组件的name"
      }
    }
  }
</script>

<style scoped>
  .div-class {
    color: aqua;
  }
</style>
```

> App.vue组件

```
<!--模板-->
<template>
  <div>
    <h2 class="title">{{message}}</h2>
    <Cpn></Cpn>
```

```
  </div>
</template>

<!--组件-->
<script>
  import Cpn from "./Cpn.vue"
  export default {
    name: "App",
    components: {
      Cpn
    },
    data() {
      return {
        message: "vue webpack"
      }
    }
  }
</script>

<!--定义样式-->
<style scoped>
  .title {
    color: #ff7979;
    font-size: 60px;
  }
</style>
```

# 14.5. 导入模块省掉后缀

- 在以前导入模块的时候，需要写上后缀，比如导入vue组件

```
import Cpn from "./Cpn.vue"
```

- 如果不写，webpack就会找不到模块

> 配置webpack.config.js文件，省去后缀

```
resolve: {
  extensions: ['.js','.less','.vue','.css'],
  alias: {
    'vue$': 'vue/dist/vue.esm.js'
  }
}
```

- 这回可以不写后缀进行打包，webpack也可以找到模块

```
import Cpn from "./Cpn"
```

# 15. webpack——plugin

- plugin是什么?
  - plugin是插件的意思，通常是用于对某个现有的架构进行扩展。
  - webpack中的插件，就是对webpack现有功能的各种扩展，比如打包优化，文件压缩等等。
- loader和plugin区别
  - loader主要用于转换某些类型的模块，它是一个转换器/加载器。
  - plugin是插件，它是对webpack本身的扩展，是一个扩展器。
- plugin的使用过程：
  - 步骤一：通过npm安装需要使用的plugins(某些webpack已经内置的插件不需要安装)
  - 步骤二：在webpack.config.js中的plugins中配置插件。
- 下面，我们就来看看可以通过哪些插件对现有的webpack打包过程进行扩容，让我们的webpack变得更加好用。

## 15.1. 添加版权的插件

- 我们先来使用一个最简单的插件，为打包的文件添加版权声明
  - 该插件名字叫BannerPlugin，属于webpack自带的插件。
- 按照下面的方式来修改webpack.config.js的文件：

```
const webpack = require("webpack");
module.exports = {
  ....
  plugins: [
    new webpack.BannerPlugin("最终版权归沈光阳所有")
  ]
}
```

- 重新打包程序：查看bundle.js文件的头部，看到如下信息

```
/*! 最终版权归沈光阳所有 */
/******/ (function(modules) { // webpackBootstrap
```

## 15.2. 打包html的插件

> 目前，我们的index.html文件是存放在项目的根目录下的。

- 我们知道，在真实发布项目时，发布的是dist文件夹中的内容，但是dist文件夹中如果没有index.html文件，那么打包的js等文件也就没有意义了。
- 所以，我们需要将index.html文件打包到dist文件夹中，这个时候就可以使用HtmlWebpackPlugin插件

> HtmlWebpackPlugin插件可以为我们做这些事情：

- 自动生成一个index.html文件(可以指定模板来生成)
- 将打包的js文件，自动通过script标签插入到body中

> 安装HtmlWebpackPlugin插件

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

> 使用插件，修改webpack.config.js文件中plugins部分的内容如下：

- 这里的template表示根据什么模板来生成index.html
- 另外，我们需要删除之前在output中添加的publicPath属性
- 否则插入的script标签中的src可能会有问题

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
module.exports = {
  ...
  // 出口：通常是一个对象，里面至少包含两个重要属性,path和filename
  output: {
    // __dirname是node中的命令，他会获取webpack.config.js的绝对路径
    path: path.resolve(__dirname,'dist'),// path：通常是一个绝对路径
    filename: "bundle.js"
  },
  ...
  plugins: [
    new webpack.BannerPlugin("最终版权归沈光阳所有"),
    new HtmlWebpackPlugin({
      template: "index.html"
    })
  ]
}
```

```
+ index.html模板内容，script部分会自动插入
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  <div class="font" id="app">
  </div>
</body>
</html>
```

- dist中生成的index.html代码

```

```

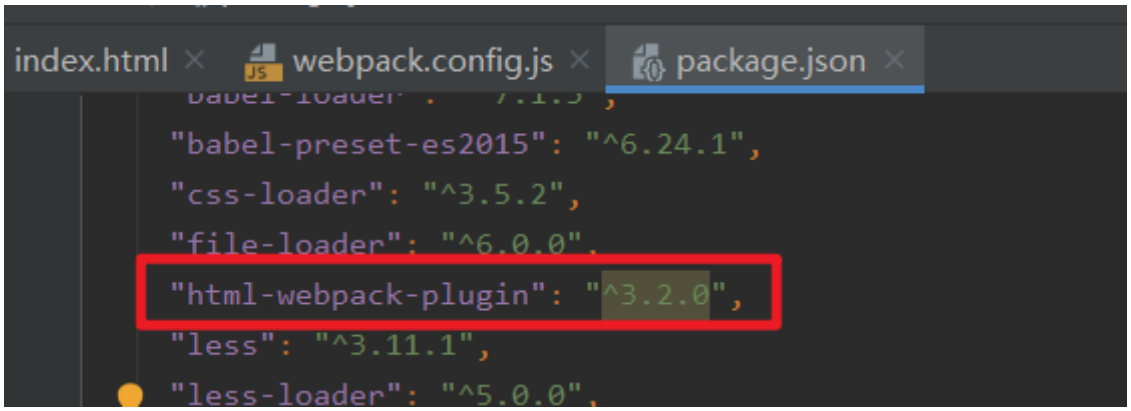
## 15.2.1. 打包报错

> 错误信息如下

```
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! meetwebpack@1.0.0 build: `webpack`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the meetwebpack@1.0.0 build script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.
```

> 安装的webpack版本是3.6.0，html-webpack-plugin插件版本是4.2.0

- 需要将html-webpack-plugin插件版本降到3.2.0




- 执行命令 `npm install`


## 15.3. 压缩js的插件

> 在项目发布之前，我们必然需要对js等文件进行压缩处理

- 这里，我们就对打包的js文件进行压缩
- 我们使用一个第三方的插件uglifyjs-webpack-plugin，并且版本号指定1.1.1，和CLI2保持一致

```
npm install uglifyjs-webpack-plugin@1.1.1 --save-dev
```

> 修改webpack.config.js文件，使用插件：

```
const uglifyJsPlugin = require("uglifyjs-webpack-plugin");
module.exports = {
  ...
  plugins: [
    new uglifyJsPlugin()
  ]
}
```

- 查看打包后的bunlde.js文件，是已经被压缩过了。

# 16. webpack-dev-server搭建本地服务器

- webpack提供了一个可选的本地开发服务器，这个本地服务器基于node.js搭建，内部使用express框架，可以实现我们想要的让浏览器自动刷新显示我们修改后的结果。
- 不过它是一个单独的模块，在webpack中使用之前需要先安装它

```
npm install --save-dev webpack-dev-server@2.9.3
```

> devserver也是作为webpack中的一个选项，选项本身可以设置如下属性：

- contentBase：为哪一个文件夹提供本地服务，默认是根文件夹，我们这里要填写./dist
- port：端口号
- inline：页面实时刷新

- historyApiFallback：在SPA页面中，依赖HTML5的history模式

> webpack.config.js文件配置修改如下：

```
devServer: {
  contentBase: "./dist",
  inline: true
}
```

> 启动本地服务器

- 不可以直接在命令行中输入`webpack-dev-server`
- 在命令行中，执行的是全局的webpack，我们是把webpack-dev-server安装到了本地
- 添加脚本到package.json文件中,`--open`表示执行脚本后自动打开浏览器的页面

```
"scripts": {
   "test": "echo \"Error: no test specified\" && exit 1",
   "build": "webpack",
   "dev": "webpack-dev-server --open"
},
```

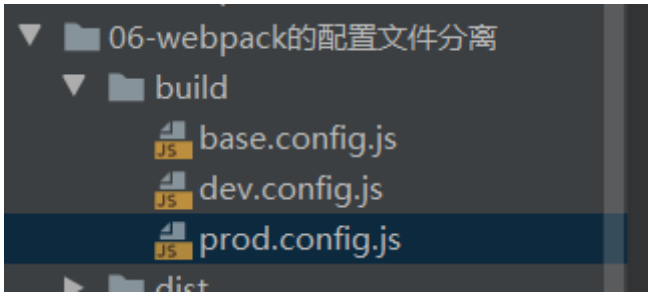
- 启动本地服务，`npm run dev`

# 17. 配置文件分离

## 17.1. 前期准备

- 在我们真实开发中，分开发配置文件和生产配置文件
- 目的
  - 在开发中，我们不建议使用压缩js代码插件，因为如果你进行了压缩，在浏览器中调试简直就是泯灭人性
  - 在开发中，我们需要搭建本地服务器，以便让页面自动刷新，而在生产环境中，不需要

- 在项目根目录下创建build文件夹
- 在build文件夹下创建base.config.js、dev.config.js、prod.config.js文件
  - base.config.js：无论开发还是生产环境都需要的公共配置
  - dev.config.js：开发时候需要的配置文件
  - prod.config.js生产时候需要的配置文件




- 将webpack.config.js中的配置分别在以上三个配置文件中各粘贴一份
  - 三个配置文件如下

> base.config.js

- 注意:要将path.resolve的路径 dist改为 ../dist，否则打包后的dist文件夹会在build中生成，而不是根目录下的dist

```
// path模块是node里的包不需要我们写
const path = require("path");
const webpack = require("webpack");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  // 入口：可以是字符串/数组/对象,这里我们的入口只有一个，所以写一个字符串即可
  entry: "./src/main.js",
  // 出口：通常是一个对象，里面至少包含两个重要属性,path和filename
  output: {
    // __dirname是node中的命令，他会获取webpack.config.js的绝对路径
    path: path.resolve(__dirname,'../dist'),// path：通常是一个绝对路径
    filename: "bundle.js"
  },
  module: {
    rules: [
      {
        test: /\.css$/,
```

```
        use: [ 'style-loader', 'css-loader' ]
      },
      {
        test: /\.less$/,
        use: [{
          loader: "style-loader" // creates style nodes from JS strings
        }, {
          loader: "css-loader" // translates CSS into CommonJS
        }, {
          loader: "less-loader" // compiles Less to CSS
        }]
      },
      {
        test: /\.(png|jpg|gif)$/,
        use: [
          {
            loader: 'url-loader',
            options: {
              limit: 8192,
              name: 'img/[name].[hash:8].[ext]'
            }
          }
        ]
      },
      {
        test: /\.js$/,
        exclude: /(node_modules|bower_components)/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['es2015']
          }
        }
      },
      {
        test: /\.vue$/,
        use: ['vue-loader']
      }
    ]
  },
  resolve: {
    extensions: ['.js','.less','.vue','.css'],
    alias: {
      'vue$': 'vue/dist/vue.esm.js'
    }
  },
  plugins: [
    new webpack.BannerPlugin("最终版权归沈光阳所有"),
```

```
    new HtmlWebpackPlugin({
      template: "index.html"
    })
  ]
}
```

> dev.config.js

```
module.exports = {
  devServer: {
    contentBase: "./dist",
    inline: true
  }
}
```

> prod.config.js

```
const uglifyJsPlugin = require("uglifyjs-webpack-plugin");
module.exports = {

  plugins: [
    new uglifyJsPlugin()
  ],
}
```

# 17.2. 合并配置文件

- 现在我们需要将base.config.js分别和dev.config.js以及prod.config.js文件进行合并
- 安装webpack-merge合并包

```
npm install webpack-merge --save-dev
```

> 修改dev.config.js和prod.config.js配置文件

- dev.config.js

```
const webpackMerge = require("webpack-merge");
const baseConfig = require("./base.config");
module.exports = webpackMerge(baseConfig,{
  devServer: {
    contentBase: "./dist",
    inline: true
  }
})
```

- prod.config.js

```
const uglifyJsPlugin = require("uglifyjs-webpack-plugin");
const webpackMerge = require("webpack-merge");
const baseConfig = require("./base.config");
module.exports = webpackMerge(baseConfig,{
  plugins: [
    new uglifyJsPlugin()
  ],
});
```

> 删除webpack.config.js配置文件

> 我们现在已经删除了webpack.config.js配置文件，如果执行运行npm run build命令肯定会报错

- 显示webpack.config.js文件找不到
- 需要在package.json中修改脚本，指定配置文件

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "webpack --config ./build/prod.config.js",
  "dev": "webpack-dev-server --open --config ./build/dev.config.js"
},
```