

Webpack打包图片-JS-Vue

王红元 coderwhy

目录

content



- 1 Webpack打包图片
- 2 Webpack打包JS代码
- 3 Babel和babel-loader
- 4 Webpack打包Vue
- 5 resolve模块解析

加载图片案例准备

■ 为了演示我们项目中可以加载图片，我们需要在项目中使用图片，比较常见的使用图片的方式是两种：

- **img元素**，设置**src**属性；
- **其他元素**（比如div），设置**background-image**的css属性；

```
// 2. image元素
const zznImage = new Image();
zznImage.src = zznImg;
// zznImage.src = require("../img/zzn.png");
element.appendChild(zznImage);

// 3. 增加一个div, 用于存放图片
const bgDiv = document.createElement('div');
bgDiv.style.width = 200 + 'px';
bgDiv.style.height = 200 + 'px';
bgDiv.style.display = 'inline-block';
bgDiv.className = 'bg-image';
bgDiv.style.backgroundColor = 'red';
element.appendChild(bgDiv);
```

```
.bg-image {
  background-image: url("../img/nhlt.jpg");
  background-size: contain;
}
```

这个时候，打包会报错

认识asset module type

■ 我们当前使用的webpack版本是webpack5:

- 在webpack5之前, 加载这些资源我们需要使用一些loader, 比如raw-loader、url-loader、file-loader;
- 在webpack5开始, 我们可以直接使用资源模块类型 (asset module type), 来替代上面的这些loader;

■ 资源模块类型(asset module type), 通过添加 4 种新的模块类型, 来替换所有这些 loader:

□ asset/resource 发送一个单独的文件并导出 URL。

- ✓ 之前通过使用 file-loader 实现;

□ asset/inline 导出一个资源的 data URI。

- ✓ 之前通过使用 url-loader 实现;

□ asset/source 导出资源的源代码

- ✓ 之前通过使用 raw-loader 实现;

□ asset 在导出一个 data URI 和发送一个单独的文件之间自动选择。

- ✓ 之前通过使用 url-loader, 并且配置资源体积限制实现;

asset module type的使用

- 比如加载图片，我们可以使用下面的方式：

```
{  
  test: /\. (png|svg|jpg|jpeg|gif)$/i,  
  type: "asset/resource"  
},
```

- 但是，如何可以自定义文件的输出路径和文件名呢？

- 方式一：修改output，添加assetModuleFilename属性；
- 方式二：在Rule中，添加一个generator属性，并且设置filename；

- 我们这里介绍几个最常用的placeholder：

- [ext]：处理文件的扩展名；
- [name]：处理文件的名称；
- [hash]：文件的内容，使用MD4的散列函数处理，生成的一个128位的hash值（32个十六进制）；

```
output: {  
  filename: "js/bundle.js",  
  path: path.resolve(__dirname, "./dist"),  
  assetModuleFilename: "img/[name].[hash:6][ext]"  
},
```

```
{  
  test: /\. (png|svg|jpg|jpeg|gif)$/i,  
  type: "asset/resource",  
  generator: {  
    filename: "img/[name].[hash:6][ext]"  
  }  
},
```

url-loader的limit效果

■ 开发中我们往往是小的图片需要转换，但是大的图片直接使用图片即可

- 这是因为小的图片转换base64之后可以和页面一起被请求，减少不必要的请求过程；
- 而大的图片也进行转换，反而会影响页面的请求速度；

■ 我们需要两个步骤来实现：

- 步骤一：将type修改为asset；
- 步骤二：添加一个parser属性，并且制定dataUrl的条件，添加maxSize属性；

```
rules: [  
  {  
    test: /\. (png|svg|jpg|jpeg|gif)$/i,  
    type: "asset",  
    generator: {  
      filename: "img/[name].[hash:6].[ext]"  
    },  
    parser: {  
      dataUrlCondition: {  
        maxSize: 100 * 1024  
      }  
    }  
  },  
]
```

为什么需要babel?

■ 事实上，在开发中我们很少直接去接触babel，但是babel对于前端开发来说，目前是不可缺少的一部分：

- 开发中，我们想要使用ES6+的语法，想要使用TypeScript，开发React项目，它们都是离不开Babel的；
- 所以，学习Babel对于我们理解代码从编写到线上的转变过程至关重要；

■ 那么，Babel到底是什么呢？

- Babel是一个工具链，主要用于旧浏览器或者环境中将ECMAScript 2015+代码转换为向后兼容版本的JavaScript；
- 包括：语法转换、源代码转换等；

```
[1, 2, 3].map((n) => n + 1);

[1, 2, 3].map(function(n) {
  return n + 1;
});
```



Babel命令行使用



■ babel本身可以作为一个**独立的工具**（和postcss一样），不和webpack等构建工具配置来单独使用。

■ 如果我們希望在命令行尝试使用babel，需要安装如下库：

□ @babel/core: babel的核心代码，必须安装；

□ @babel/cli: 可以让我们在命令行使用babel；

```
npm install @babel/cli @babel/core -D
```

■ 使用babel来处理我们的源代码：

□ src: 是源文件的目录；

□ --out-dir: 指定要输出的文件夹dist；

```
npx babel src --out-dir dist
```


插件的使用

- 比如我们需要转换箭头函数，那么我们就可以使用**箭头函数转换相关的插件**：

```
npm install @babel/plugin-transform-arrow-functions -D
```

```
npx babel src --out-dir dist --plugins=@babel/plugin-transform-arrow-functions
```

- 查看转换后的结果：我们会发现 `const` 并没有转成 `var`

- 这是因为 `plugin-transform-arrow-functions`，并没有提供这样的功能；

- 我们需要使用 `plugin-transform-block-scoping` 来完成这样的功能；

```
npm install @babel/plugin-transform-block-scoping -D
```

```
npx babel src --out-dir dist --plugins=@babel/plugin-transform-block-scoping  
,@babel/plugin-transform-arrow-functions
```



Babel的预设preset



- 但是如果转换的内容过多，一个个设置是比较麻烦的，我们可以使用预设（preset）：

- 后面我们再具体来讲预设代表的含义；

- 安装@babel/preset-env预设：

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

- 执行如下命令：

```
npx babel src --out-dir dist --presets=@babel/preset-env
```

babel-loader

- 在实际开发中，我们通常会在构建工具中通过配置babel来对其进行使用的，比如在webpack中。
- 那么我们就需要去安装相关的依赖：
 - 如果之前已经安装了@babel/core，那么这里不需要再次安装；

```
npm install babel-loader -D
```

- 我们可以设置一个规则，在加载js文件时，使用我们的babel：

```
module: {  
  rules: [  
    {  
      test: /\.m?js$/,  
      use: {  
        loader: "babel-loader"  
      }  
    }  
  ]  
},
```



babel-preset

■ 如果我们一个个去安装使用插件，那么需要手动来管理大量的babel插件，我们可以直接给webpack提供一个preset，webpack会根据我们的预设来加载对应的插件列表，并且将其传递给babel。

■ 比如常见的预设有三个：

- env
- react
- TypeScript

■ 安装preset-env：

```
npm install @babel/preset-env
```

```
{
  test: /\.m?js$/,
  use: {
    loader: "babel-loader",
    options: {
      presets: [
        "@babel/preset-env"
      ]
    }
  }
}
```

编写App.vue代码

■ 在开发中我们会编写Vue相关的代码，webpack可以对Vue代码进行解析：

□ 接下来我们编写自己的App.vue代码；

```
ackjs / src / vue / App.vue
<template>
  <h2>{{title}}</h2>
  <p>{{content}}</p>
</template>

<script>
export default {
  data() {
    return {
      title: "我是App标题",
      content: "我是App的内容，哈哈哈"
    }
  }
}
</script>

<style>
h2 {
  color: red;
}
p {
  color: blue;
}
</style>
```

```
import { createApp } from "vue/dist/vue.esm-bundler";
import App from './vue/App.vue';

// Vue代码
createApp(App).mount("#app");
```

App.vue的打包过程

- 我们对代码打包会报错：我们需要合适的Loader来处理文件。

```
ERROR in ./src/vue/App.vue 1:0
Module parse failed: Unexpected token (1:0)
You may need an appropriate loader to handle this file type,
https://webpack.js.org/concepts/loaders
```

- 这个时候我们需要使用vue-loader：

```
npm install vue-loader -D
```

- 在webpack的模板规则中进行配置：

```
{
  test: /\.vue$/,
  loader: "vue-loader"
}
```



@vue/compiler-sfc



- 打包依然会报错，这是因为我们必须添加@vue/compiler-sfc来对template进行解析：

```
npm install @vue/compiler-sfc -D
```

- 另外我们需要配置对应的Vue插件：

```
const { VueLoaderPlugin } = require('vue-loader/dist/index');
```

```
new VueLoaderPlugin()
```

- 重新打包即可支持App.vue的写法
- 另外，我们也可以编写其他的.vue文件来编写自己的组件；



resolve模块解析

■ resolve用于设置模块如何被解析：

- ❑ 在开发中我们会有各种各样的模块依赖，这些模块可能来自于自己编写的代码，也可能来自第三方库；
- ❑ resolve可以帮助webpack从每个 `require/import` 语句中，找到需要引入到合适的模块代码；
- ❑ webpack 使用 [enhanced-resolve](#) 来解析文件路径；

■ webpack能解析三种文件路径：

■ 绝对路径

- ❑ 由于已经获得文件的绝对路径，因此不需要再做进一步解析。

■ 相对路径

- ❑ 在这种情况下，使用 `import` 或 `require` 的资源文件所处的目录，被认为是上下文目录；
- ❑ 在 `import/require` 中给定的相对路径，会拼接此上下文路径，来生成模块的绝对路径；

■ 模块路径

- ❑ 在 `resolve.modules`中指定的所有目录检索模块；
 - ✓ 默认值是 `['node_modules']`，所以默认会从`node_modules`中查找文件；
- ❑ 我们可以通过设置别名的方式来替换初始模块路径，具体后面讲解`alias`的配置；

确实文件还是文件夹

- 如果是一个文件：
 - 如果文件具有扩展名，则直接打包文件；
 - 否则，将使用 `resolve.extensions` 选项作为文件扩展名解析；
- 如果是一个文件夹：
 - 会在文件夹中根据 `resolve.mainFiles` 配置选项中指定的文件顺序查找；
 - ✓ `resolve.mainFiles` 的默认值是 `['index']`；
 - ✓ 再根据 `resolve.extensions` 来解析扩展名；

extensions和alias配置

■ extensions是解析到文件时自动添加扩展名：

□ 默认值是 ['.wasm', '.mjs', '.js', '.json'];

□ 所以如果我们代码中想要添加加载 .vue 或者 jsx 或者 ts 等文件时，我们必须自己写上扩展名；

■ 另一个非常好用的功能是配置别名alias：

□ 特别是当我们项目的目录结构比较深的时候，或者一个文件的路径可能需要 ../.././这种路径片段；

□ 我们可以给某些常见的路径起一个别名；

```
resolve: {  
  extensions: ['.wasm', '.mjs', '.js', '.json', '.jsx', '.ts', '.vue'],  
  alias: {  
    '@': resolveApp('./src'),  
    pages: resolveApp('./src/pages'),  
  },  
},
```