

Webpack打包其他资源

王红元 coderwhy

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

- 要处理jpg、png等格式的图片，我们也需要有对应的loader：**file-loader**

- file-loader的作用就是帮助我们处理**import/require()方式**引入的一个文件资源，并且会将它放到我们**输出的文件夹**中；

- 当然我们待会儿可以学习如何修改它的名字和所在文件夹；

- 安装file-loader：

```
npm install file-loader -D
```

- 配置处理图片的Rule：

```
{  
  test: /\. (png|jpe?g|gif|svg)$/i,  
  use: {  
    loader: "file-loader"  
  }  
},
```





文件的命名规则

- 有时候我们处理后的**文件名称**按照一定的规则进行显示：
 - 比如保留原来的**文件名、扩展名**，同时为了防止重复，包含一个**hash值**等；
- 这个时候我们可以使用**PlaceHolders**来完成，webpack给我们提供了大量的PlaceHolders来显示不同的内容：
 - <https://webpack.js.org/loaders/file-loader/#placeholders>
 - 我们可以在文档中查阅自己需要的placeholder；
- 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）；
 - **[contentHash]**：在file-loader中和[hash]结果是一致的（在webpack的一些其他地方不一样，后面会讲到）；
 - **[hash:<length>]**：截图hash的长度，默认32个字符太长了；
 - **[path]**：文件相对于webpack配置文件的路径；

设置文件的名称

■ 那么我们可以按照如下的格式编写：

□ 这个也是vue的写法；

```
{  
  test: /\. (png|jpe?g|gif|svg)$/i,  
  use: {  
    loader: "file-loader",  
    options: {  
      name: "img/[name].[hash:8].[ext]"  
    }  
  }  
},
```

设置文件的存放路径

- 当然，我们刚才通过 `img/` 已经设置了文件夹，这个也是vue、react脚手架中常见的设置方式：
 - 其实按照这种设置方式就可以了；
 - 当然我们也可以通过`outputPath`来设置输出的文件夹；

```
{  
  test: /\. (png|jpe?g|gif|svg)$/i,  
  use: {  
    loader: "file-loader",  
    options: {  
      name: "[name].[hash:8].[ext]",  
      outputPath: "img"  
    }  
  }  
},
```

■ **url-loader**和**file-loader**的工作方式是相似的，但是可以将较小的文件，转成**base64**的URI。

■ 安装url-loader：

```
npm install url-loader -D
```

```
{
  test: /\.?(png|jpe?g|gif|svg)$/i,
  use: {
    // loader: "file-loader",
    loader: "url-loader",
    options: {
      name: "[name].[hash:8].[ext]",
      outputPath: "img"
    }
  }
},
```

显示结果是一样的，并且图片可以正常显示；

■ 但是在dist文件夹中，我们会看不到图片文件：

- 这是因为我的两张图片的大小分别是38kb和295kb；
- 默认情况下url-loader会将所有的图片文件转成base64编码

url-loader的limit

- 但是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；
- 那么，我们如何可以限制哪些大小的图片转换和不转换呢？
 - url-loader有一个options属性limit，可以用于设置转换的限制；
 - 下面的代码38kb的图片会进行base64编码，而295kb的不会；

```
{
  test: /\. (png|jpe?g|gif|svg)$/i,
  use: [{
    loader: "url-loader",
    options: {
      limit: 100 * 1024,
      name: "[name].[hash:8].[ext]",
      outputPath: "img",
    }
  }]
},
```




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；

```
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

■ 我们需要两个步骤来实现：

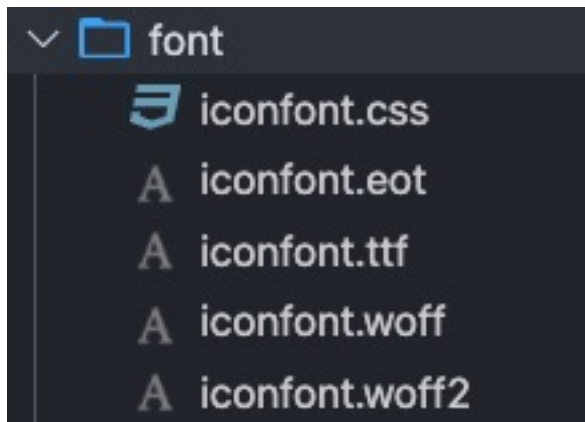
□ 步骤一：将type修改为asset；

□ 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  
      }  
    }  
  },  
]
```

加载字体文件

- 如果我们需要使用某些特殊的字体或者字体图标，那么我们会引入很多字体相关的文件，这些文件的处理也是一样的。
- 首先，我从阿里图标库中下载了几个字体图标：



- 在component中引入，并且添加一个i元素用于显示字体图标：

```
const iEl = document.createElement('i');  
iEl.className = 'iconfont icon-ashbin';  
element.appendChild(iEl);
```

字体的打包

- 这个时候打包会报错，因为无法正确的处理eot、ttf、woff等文件：
- 我们可以选择使用file-loader来处理，也可以选择直接使用webpack5的资源模块类型来处理；

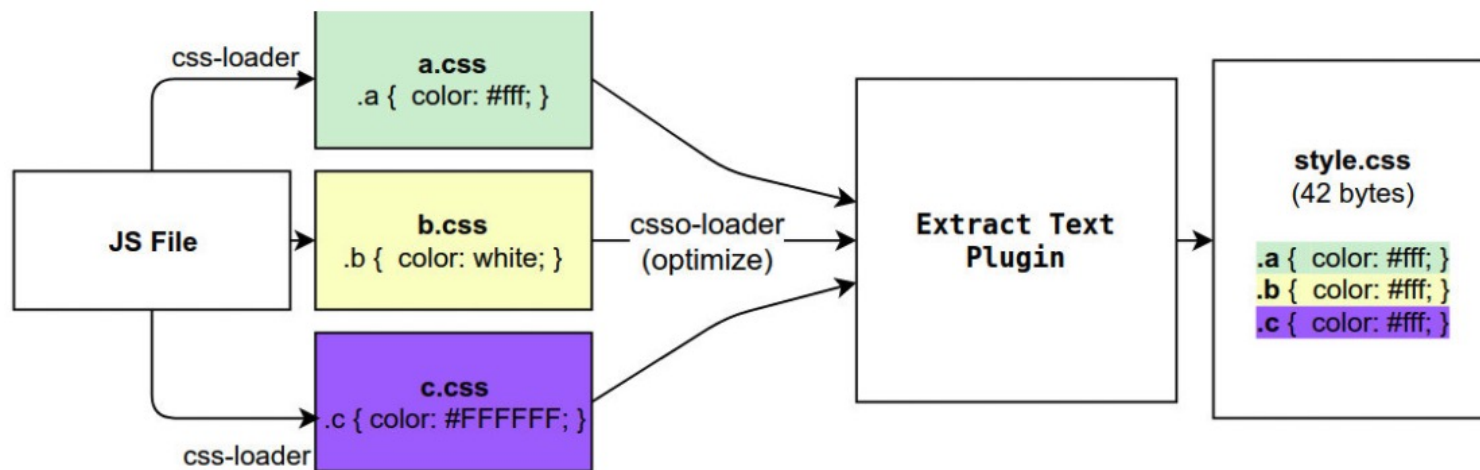
```
{  
  test: /\.woff2?|eot|ttf$/,  
  type: 'asset/resource',  
  generator: {  
    filename: "font/[name].[hash:6][ext]"  
  }  
}
```

■ Webpack的另一个核心是Plugin，官方有这样一段对Plugin的描述：

- While loaders are used to transform certain types of modules, plugins can be leveraged to perform a wider range of tasks like bundle optimization, asset management and injection of environment variables.

■ 上面表达的含义翻译过来就是：

- Loader是用于**特定的模块类型**进行转换；
- Plugin可以用于**执行更加广泛的任务**，比如打包优化、资源管理、环境变量注入等；



CleanWebpackPlugin

■ 前面我们演示的过程中，每次修改了一些配置，重新打包时，都需要手动删除dist文件夹：

□ 我们可以借助于一个插件来帮助我们完成，这个插件就是CleanWebpackPlugin；

■ 首先，我们先安装这个插件：

```
npm install clean-webpack-plugin -D
```

■ 之后在插件中配置：

```
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = {
  // 其他省略
  plugins: [
    new CleanWebpackPlugin()
  ]
}
```

HtmlWebpackPlugin

■ 另外还有一个不太规范的地方：

- 我们的HTML文件是编写在根目录下的，而最终打包的dist文件夹中是没有index.html文件的。
- 在进行项目部署的时，必然也是需要对应的入口文件index.html；
- 所以我們也需要对index.html进行打包处理；

■ 对HTML进行打包处理我们可以使用另外一个插件：HtmlWebpackPlugin；

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

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  // 其他省略
  plugins: [
    new HtmlWebpackPlugin({
      title: 'webpack案例'
    })
  ]
}
```


生成index.html分析

■ 我们会发现，现在自动在dist文件夹中，生成了一个index.html的文件：

□ 该文件中也自动添加了我们打包的bundle.js文件；

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>webpack案例</title>
    <meta name="viewport" content="width=device-width, initial-scale=1"></head>
    <body>
      <script src="bundle.js"></script></body>
</html>
```

■ 这个文件是如何生成的呢？

□ 默认情况下是根据ejs的一个模板来生成的；

□ 在html-webpack-plugin的源码中，有一个default_index.ejs模块；

自定义HTML模板

■ 如果我们想在自己的模块中加入一些比较特别的内容：

- 比如添加一个 **noscript** 标签，在用户的JavaScript被关闭时，给予响应的提示；
- 比如在 **开发vue或者react项目** 时，我们需要一个可以挂载后续组件的 **根标签** `<div id="app"> </div>`；

■ 这个我们需要一个属于自己的index.html模块：

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="%= BASE_URL %>favicon.ico">
    <title>%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work properly
        without JavaScript enabled. Please enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

自定义模板数据填充

- 上面的代码中，会有一些类似这样的语法 `<% 变量 %>`，这个是EJS模块填充数据的方式。
- 在配置HtmlWebpackPlugin时，我们可以添加如下配置：
 - **template**：指定我们要使用的模块所在的路径；
 - **title**：在进行htmlWebpackPlugin.options.title读取时，就会读到该信息；

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  // 其他省略
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'webpack项目',
      template: './public/index.html'
    })
  ]
}
```



DefinePlugin的介绍

- 但是，这个时候编译还是会报错，因为在我们的模块中还使用到一个BASE_URL的常量：

```
ERROR in Template execution failed: ReferenceError: BASE_URL is not defined
ERROR in   ReferenceError: BASE_URL is not defined
```

- 这是因为在编译template模块时，有一个BASE_URL：
 - `<link rel="icon" href="<%= BASE_URL %>favicon.ico">`；
 - 但是我们并没有设置过这个常量值，所以会出现没有定义的错误；
- 这个时候我们可以使用DefinePlugin插件；

DefinePlugin的使用

- DefinePlugin允许在编译时创建配置的全局常量，是一个webpack内置的插件（不需要单独安装）：

```
const { DefinePlugin } = require('webpack');

module.exports = {
  // 其他省略
  plugins: [
    new DefinePlugin({
      BASE_URL: '"./"'
    })
  ]
}
```

- 这个时候，编译template就可以正确的编译了，会读取到BASE_URL的值；

CopyWebpackPlugin

■ 在vue的打包过程中，如果我们将一些文件放到public的目录下，那么这个目录会被复制到dist文件夹中。

□ 这个复制的功能，我们可以使用CopyWebpackPlugin来完成；

■ 安装CopyWebpackPlugin插件：

```
npm install copy-webpack-plugin -D
```

■ 接下来配置CopyWebpackPlugin即可：

□ 复制的规则在patterns中设置；

□ **from**：设置从哪一个源中开始复制；

□ **to**：复制到的位置，可以省略，会默认复制到打包的目录下；

□ **globOptions**：设置一些额外的选项，其中可以编写需要忽略的文件：

✓ .DS_Store：mac目录下回自动生成的一个文件；

✓ index.html：也不需要复制，因为我们已经通过HtmlWebpackPlugin完成了index.html的生成；

```
new CopyWebpackPlugin({
  patterns: [
    {
      from: "public",
      globOptions: {
        ignore: [
          '**/.DS_Store',
          '**/index.html'
        ]
      }
    }
  ]
})
```

- 前面我们一直没有讲mode。
- Mode配置选项，可以告知webpack使用响应模式的内置优化：
 - 默认值是production（什么都不设置的情况下）；
 - 可选值有：'none' | 'development' | 'production'；
- 这几个选项有什么样的区别呢？

选项	描述
development	会将 DefinePlugin 中 process.env.NODE_ENV 的值设置为 development。为模块和 chunk 启用有效的名。
production	会将 DefinePlugin 中 process.env.NODE_ENV 的值设置为 production。为模块和 chunk 启用确定性的混淆名称，FlagDependencyUsagePlugin，FlagIncludedChunksPlugin，ModuleConcatenationPlugin，NoEmitOnErrorsPlugin 和 TerserPlugin。
none	不使用任何默认优化选项

Mode配置代表更多

```
// webpack.development.config.js
module.exports = {
+ mode: 'development'
- devtool: 'eval',
- cache: true,
- performance: {
-   hints: false
- },
- output: {
-   pathinfo: true
- },
- optimization: {
-   moduleIds: 'named',
-   chunkIds: 'named',
-   mangleExports: false,
-   nodeEnv: 'development',
-   flagIncludedChunks: false,
-   occurrenceOrder: false,
-   concatenateModules: false,
-   splitChunks: {
-     hidePathInfo: false,
-     minSize: 10000,
-     maxAsyncRequests: Infinity,
-     maxInitialRequests: Infinity,
-   },
-   emitOnErrors: true,
-   checkWasmTypes: false,
-   minimize: false,
-   removeAvailableModules: false
- },
- plugins: [
-   new webpack.DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("development") })
- ]
}
```

```
// webpack.production.config.js
module.exports = {
+ mode: 'production',
- performance: {
-   hints: 'warning'
- },
- output: {
-   pathinfo: false
- },
- optimization: {
-   moduleIds: 'deterministic',
-   chunkIds: 'deterministic',
-   mangleExports: 'deterministic',
-   nodeEnv: 'production',
-   flagIncludedChunks: true,
-   occurrenceOrder: true,
-   concatenateModules: true,
-   splitChunks: {
-     hidePathInfo: true,
-     minSize: 30000,
-     maxAsyncRequests: 5,
-     maxInitialRequests: 3,
-   },
-   emitOnErrors: false,
-   checkWasmTypes: true,
-   minimize: true,
- },
- plugins: [
-   new TerserPlugin(/* ... */),
-   new webpack.DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("production") }),
-   new webpack.optimize.ModuleConcatenationPlugin(),
-   new webpack.NoEmitOnErrorsPlugin()
- ]
}
```