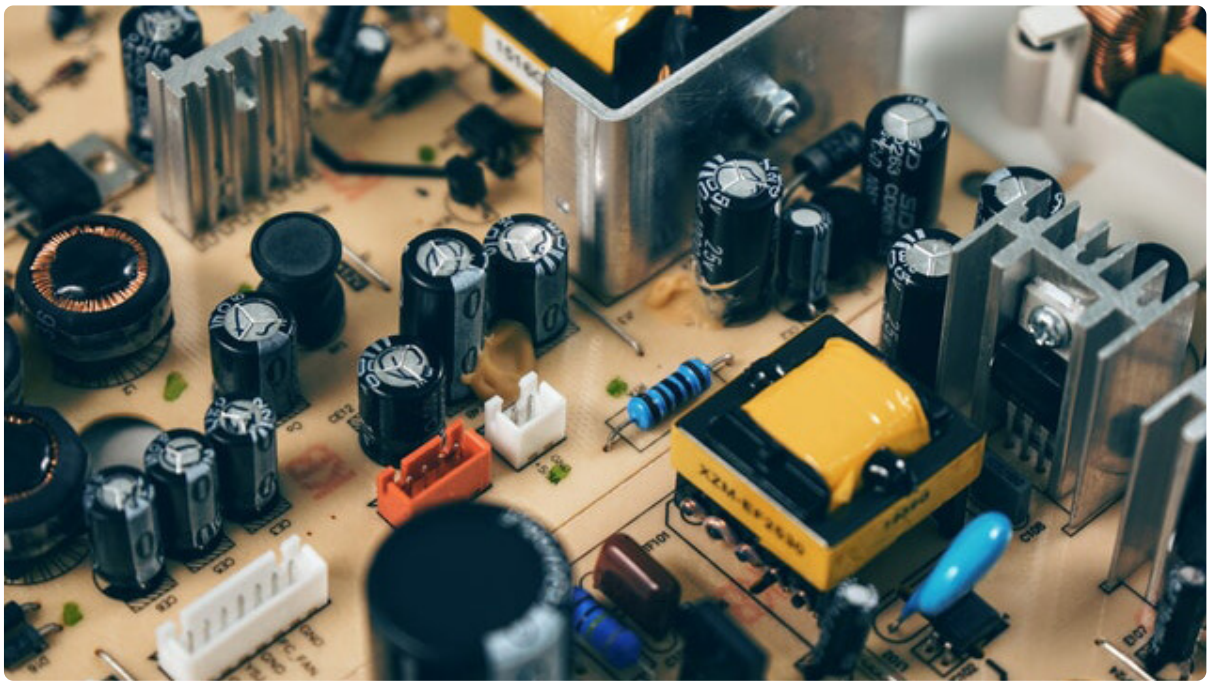
17 Webpack 优化之体积优化

更新时间：2019-06-24 09:30:27



> 自信和希望是青年的特权。
>
> ——大仲马

Webpack 毕竟是个项目打包工具，一般 web 项目，打完包之后，需要发布到服务器上供用户使用，受带宽的限制，我们的项目体积需要越小越好，所以 Webpack 中打包的体积是 Webpack 中重要的一环，本小节内容将从 JavaScript、CSS 和其他静态资源体积优化入手，介绍项目的体积优化方案。

JavaScript 压缩

在 mode=production 下，Webpack 会自动压缩代码，我们可以自定义自己的压缩工具，这里推荐 terser-webpack-plugin，它是 Webpack 官方维护的插件，使用terser来压缩 JavaScript 代码。UglifyJS 在压缩 ES5 方面做的很优秀，但是随着 ES6 语法的普及，UglifyJS 在 ES6 代码压缩上做的不够好，所以有了 uglify-es 项目，但是之后 uglify-es 项目不在维护了，terser 是从 uglify-es 项目拉的一个分支，来继续维护。terser-webpack-plugin 具有跟 Uglifyjs-webpack-plugin 相同的参数，我们在 Webpack 中可以通过配置文件直接调用：

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  optimization: {
    minimizer: [new TerserPlugin()]
  }
};
```

在实际开发中，我们可以通过移出一些不用的代码从而达到优化代码体积的作用，Tree-Shaking 也是依赖这个插件的：

```
new TerserPlugin({
  // 使用 cache，加快二次构建速度
  cache: true,
  terserOptions: {
    comments: false,
    compress: {
      // 删除无用的代码
      unused: true,
      // 删掉 debugger
      drop_debugger: true, // eslint-disable-line
      // 移除 console
      drop_console: true, // eslint-disable-line
      // 移除无用的代码
      dead_code: true // eslint-disable-line
    }
  }
});
```

压缩是发布前处理最耗时间的一个步骤，在 Webpack 配置中可以通过开启 terser-webpack-plugin 的多线程压缩来加速我们的构建压缩速度：

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
 optimization: {
  minimizer: [new TerserPlugin(
   parallel: true   // 多线程
  )],
 },
};
```

其他代码级别优化技巧

1. 合理划分代码职责，适当使用按需加载方案；
2. 善用 webpack-bundle-analyzer 插件，帮助分析 Webpack 打包后的模块依赖关系；
3. 设置合理的 SplitChunks 分组；
4. 对于一些 UI 组件库，例如 AntDesign、ElementUI 等，可以使用bable-plugin-import这类工具进行优化；
5. 使用 lodash、momentjs 这类库，不要一股脑引入，要按需引入，momentjs 可以用 date-fns 库来代替；
6. 合理使用 hash 占位符，防止 hash 重复出现，导致文件名变化从而 HTTP 缓存过期；
7. 合理使用 polyfill，防止多余的代码；
8. 使用 ES6 语法，尽量不使用具有副作用的代码，以加强 Tree-Shaking 的效果；
9. 使用 Webpack 的 Scope Hoisting（作用域提升）功能。

> Tips：其实 webpack 4 中，在 production 模式下已经根据大多数项目的优化经验做了通用的配置，类似 Tree-Shaking、Scope Hoisting 都是默认开启的，而且最新版本的 Webpack 使用的压缩工具就是 terser-webpack-plugin。

什么是 Scope Hoisting

作用域提升（Scope Hoisting）是指 webpack 通过 ES6 语法的静态分析，分析出模块之间的依赖关系，尽可能地把模块放到同一个函数中。下面通过代码示例来理解：

```
// utils.js
export default 'Hello, Webpack';
// entry.js
import str from './util.js';
console.log(str);
```

普通打包后，`utils.js`的内容和`entry.js`会分开，例如下面代码：

```
(function(module, __webpack_exports__, __webpack_require__) {
  var __WEBPACK_IMPORTED_MODULE_0__util_js__ = __webpack_require__(1);
  console.log(__WEBPACK_IMPORTED_MODULE_0__util_js__['a']);
},
  function(module, __webpack_exports__, __webpack_require__) {
    __webpack_exports__['a'] = 'Hello, Webpack';
  });
```

通过配置 webpack 4 的`optimization.concatenateModules=true`：

```
// webpack.config.js
module.exports = {
  optimization: {
    concatenateModules: true
  }
};
```

这样就开启了 Scope Hoisting，这时候打包变成了：

```
(function(module, __webpack_exports__, __webpack_require__) {
  var util = 'Hello, Webpack';
  console.log(util);
});
```

我们发现`utils.js`内容和`entry.js`的内容合并在一起了！所以通过 Scope Hoisting 的功能可以让 Webpack 打包出来的代码文件更小、运行的更快。

CSS

除了 JavaScript 外，样式文件也是前端中重要的资源，Webpack 本身是 JavaScript 的打包器，在 CSS 方面通过强大的插件社区，可以实现 CSS 的优化。

CSS 导出

首先我们的 CSS 文件应该是导出到单独的 CSS 文件中，而不要直接打包到 JavaScript 文件中，然后通过`style-loader`的`addStyles`方法添加进去，导出 CSS 文件就需要使用mini-css-extract-plugin这个插件。

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
module.exports = {
  plugins: [
    new MiniCssExtractPlugin({
      filename: '[name].css',
      chunkFilename: '[name].[contenthash:8].css'
    })
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
            options: {
              publicPath: '../',
              hmr: process.env.NODE_ENV === 'development'
            }
          },
          'css-loader'
        ]
      }
    ]
  }
};
```

还记得 CSS 章节介绍的 contenthash 吗，在 CSS 中推荐使用 [contenthash] 这个占位符做文件的 hash 算法。

CSS 压缩：cssnano

cssnano是基于 postcss 的一款功能强大的插件包，它集成了 30 多个插件，只需要执行一个命令，就可以对我们的 CSS 做多方面不同类型的优化，比如：

- 删除空格和最后一个分号；
- 删除注释；
- 优化字体权重；
- 丢弃重复的样式规则；
- 压缩选择器；
- 减少手写属性；
- 合并规则；
- …

我们来看个 cssnano 处理之前和处理之后的 CSS 内容就能体会到 cssnano 的强大功能了：

```
/* input */
.a {
  background: red;
  color: yellow;
}
.b {
  font-size: bolder;
  background: red;
}
.c {
  color: yellow;
  text-align: center;
  font-size: bolder;
}
.d {
  display: flex;
  text-align: center;
}
/* output */
.a {
  color: #ff0;
}
.a,
.b {
  background: red;
}
.b,
.c {
  font-size: bolder;
}
.c {
  color: #ff0;
}
.c,
.d {
  text-align: center;
}
.d {
  display: flex;
}
```

通过观察上面输入和输出的内容差异，我们发现 cssnano 很智能，它能够将 CSS 规则相同的选择器进行合并，并且还能够将 color 进行任意的切换，这样的意义是为了缩短实际的字符串长度。

在 Webapck 中，css-loader 已经集成了 cssnano，我们还可以使用optimize-css-assets-webpack-plugin来自定义 cssnano 的规则。optimize-css-assets-webpack-plugin 是一个 CSS 的压缩插件，默认的压缩引擎就是 cssnano。我们来看下怎么在 Webpack 中使用这个插件：

```
// webpack.config.js
const OptimizeCssAssetsPlugin = require('optimize-css-assets-webpack-plugin');
module.exports = {
  plugins: [
    new OptimizeCssAssetsPlugin({
      assetNameRegExp: /\.optimize\.css$/g,
      cssProcessor: require('cssnano'), // 这里制定了引擎，不指定默认也是 cssnano
      cssProcessorPluginOptions: {
        preset: ['default', {discardComments: {removeAll: true}}]
      },
      canPrint: true
    })
  ]
};
```

optimize-css-assets-webpack-plugin 插件默认的 cssnano 配置已经做的很友好了，不需要额外的配置就可以达到最佳效果。

图片资源优化

通常我们的代码体积会比图片体积小很多，有的时候整个页面的代码都不如一张头图大。好在图片资源不会阻塞浏览器渲染，但是不合理的图片大小也会消耗一定的代码。在之前章节中也已经提到使用：url-loader、svg-url-loader 和 image-webpack-loader 来优化图片，还介绍了使用雪碧图来优化图片资源。

url-loader 可以按照配置将小于一定体积的静态文件内联进我们的应用。当我们指定了 limit 这个 options 选项，它会将文件编码成比无配置更小的 Base64 的数据 url 并将该 url 返回，这样可以将图片内联进 JavaScript 代码中，并节省一次 HTTP 请求。svg-url-loader 的工作原理类似于 url-loader，除了它利用 URL encoding 而不是 Base64 对文件编码，对于 SVG 图片来说，svg-url-loader 的这种方式这是有效的，因为 SVG 文件本质上是纯文本文件，这种 URL encoding 编码规模效应更加明显。

如果我们的项目中小图片特别多，例如有很多 icon 类的图标，这时候则推荐使用雪碧图（CSS Sprite）来合并这些小图到一张大图中，然后使用 background-position 来设置图片的位置，通过这样的方式可以节省多次小图片的请求。

对于大图片来说，可以使用image-webpack-loader来压缩图片，image-webpack-loader 它支持 JPG、PNG、GIF 和 SVG 格式的图片，因此我们在碰到所有这些类型的图片都会使用它。

总结

本小节是 Webpack 优化的第一篇文章，本文从体积优化方面入手，分别从前端项目中最常见的 JavaScript、CSS 和图片三部分入手，介绍了各自的优化方案，希望对大家项目的实际应用中有所帮助。

> 本小节 Webpack 相关面试题：
>
> 本章节一直在回答一个问题：webpack 怎么优化。本小节主要从减少代码体积方面来介绍 Webpack 优化方案。

16 Webpack 环境相关配置及配置文件拆分

18 Webpack 优化之增强缓存命中率