学习 Webpack5 之路（优化篇）

上传日期：2021.08.16 842

本篇将从优化开发体验、加快编译速度、减小打包体积、加快加载速度 4 个角度出发，介绍如何对 webpack 项目进行优化。

一、前言

从 0 到 1 学习的朋友可参考前置学习文章：

- 学习 Webpack5 之路（基础篇）
- 学习 Webpack5 之路（实践篇）

前置文章 学习 Webpack5 之路（基础篇） 对 webpack 的概念做了简单介绍， 学习 Webpack5 之路（实践篇） 则从配置着手，用 webpack 搭建了一个 SASS + TS + React 的项目。

本篇将从优化开发体验、加快编译速度、减小打包体积、加快加载速度 4 个角度出发，介绍如何对 webpack 项目进行优化。

本文依赖的 webpack 版本信息如下：

- webpack-cli@4.7.2
- webpack@5.46.0

二、优化效率工具

在优化开始之前，需要做一些准备工作。

安装以下 webpack 插件，帮助我们分析优化效率：

- progress-bar-webpack-plugin ：查看编译进度；
- speed-measure-webpack-plugin ：查看编译速度；
- webpack-bundle-analyzer ：打包体积分析。

1. 编译进度条

一般来说，中型项目的首次编译时间为 5-20s，没个进度条等得多着急，通过 progress-bar-webpack-plugin 插件查看编译进度，方便我们掌握编译情况。

安装：

```
npm i -D progress-bar-webpack-plugin
```

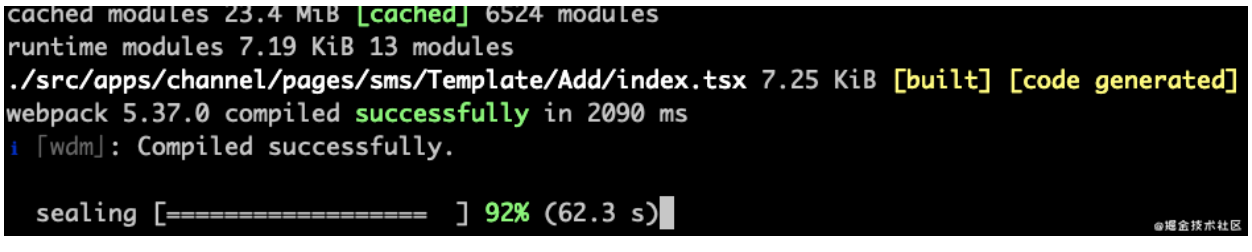
webpack.common.js 配置方式如下：

```
const chalk = require('chalk')
const ProgressBarPlugin = require('progress-bar-webpack-plugin')
module.exports = {
  plugins: [
    // 进度条
    new ProgressBarPlugin({
        format: `  :msg [:bar] ${chalk.green.bold(':percent')} (:elapsed s)`
      })
```

```
    ],
  }
```

> 贴心的为进度百分比添加了加粗和绿色高亮态样式。

包含内容、进度条、进度百分比、消耗时间，进度条效果如下：



2. 编译速度分析

优化 webpack 构建速度，首先需要知道是哪些插件、哪些 loader 耗时长，方便我们针对性的优化。

通过 [speed-measure-webpack-plugin](#)插件进行构建速度分析，可以看到各个 loader、plugin 的构建时长，后续可针对耗时 loader、plugin 进行优化。

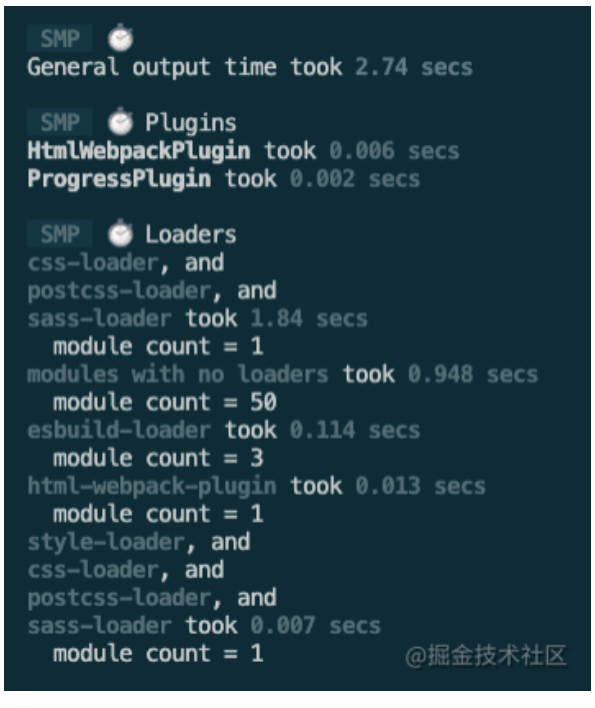
安装：

```
npm i -D speed-measure-webpack-plugin
```

`webpack.dev.js` 配置方式如下：

```
const SpeedMeasurePlugin = require("speed-measure-webpack-plugin");
const smp = new SpeedMeasurePlugin();
module.exports = smp.wrap({
  // ...webpack config...
})
```

包含各工具的构建耗时，效果如下：



> 注意：这些灰色文字的样式，是因为我在 *vscode* 终端运行的，导致有颜色的字体都显示为灰色，换个终端就好了，如 *iTerm2* 。

3. 打包体积分析

3. 打包体积分析

同样，优化打包体积，也需要先分析各个 bundle 文件的占比大小，来进行针对优化。

使用 webpack-bundle-analyzer 查看打包后生成的 bundle 体积分析，将 bundle 内容展示为一个便捷的、交互式、可缩放的树状图形式。帮助我们分析输出结果来检查模块在何处结束。

安装：

```
npm i -D webpack-bundle-analyzer
```

`webpack.prod.js` 配置方式如下：

```
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;
module.exports = {
  plugins: [
    // 打包体积分析
    new BundleAnalyzerPlugin()
  ],
}
```

包含各个 bundle 的体积分析，效果如下：

三、优化开发体验

1. 自动更新

自动更新 指的是，在开发过程中，修改代码后，无需手动再次编译，可以自动编译代码更新编译后代码的功能。

webpack 提供了以下几种可选方式，实现自动更新功能：

1. webpack's Watch Mode
2. webpack-dev-server
3. webpack-dev-middleware

webpack 官方推荐的方式是 `webpack-dev-server`，在 学习 Webpack5 之路（实践篇） - DevServer 章节 已经介绍了 webpack-dev-server 帮助我们在代码发生变化后自动编译代码实现**自动更新**的用法，在这里不重复赘述。

> 这是针对开发环境的优化，修改 *`webpack.dev.js`* 配置。

2. 热更新

2. 热更新

热更新 指的是，在开发过程中，修改代码后，仅更新修改部分的内容，无需刷新整个页面。

2.1 修改 webpack-dev-server 配置

使用 webpack 内置的 HMR 插件，更新 webpack-dev-server 配置。

`webpack.dev.js` 配置方式如下：

```
module.export = {
    devServer: {
        contentBase: './dist',
        hot: true, // 热更新
      },
}
```

2.2 引入 react-refresh-webpack-plugin

使用 react-refresh-webpack-plugin 热更新 react 组件。

安装：

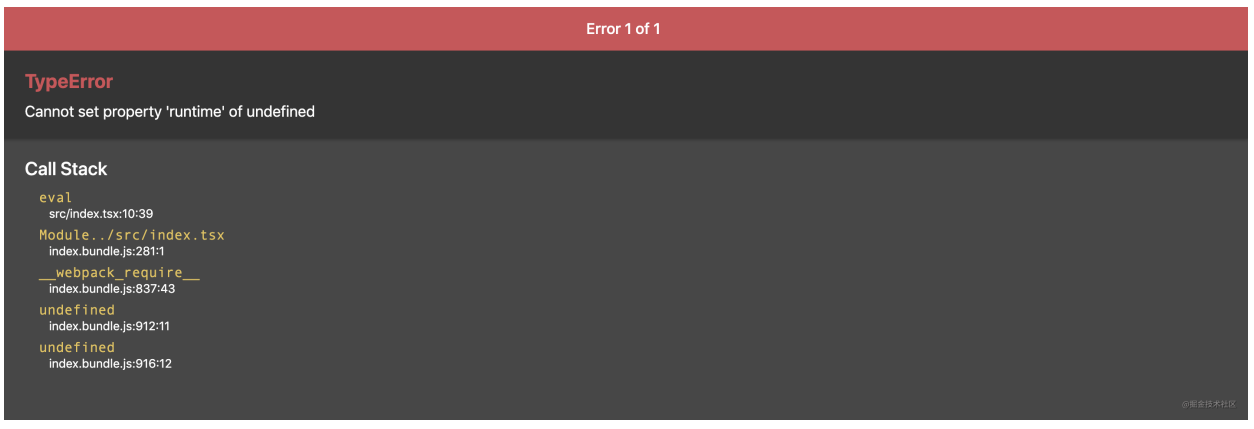
```
npm install -D @pmmmwh/react-refresh-webpack-plugin react-refresh
```

`webpack.dev.js` 配置方式如下：

```
const ReactRefreshWebpackPlugin = require('@pmmmwh/react-refresh-webpack-plugin');

module.exports = {
    plugins: [
        new webpack.HotModuleReplacementPlugin(),
        new ReactRefreshWebpackPlugin(),
    ]
}
```

遇到的问题：

配置了 SpeedMeasurePlugin 后，热更新就无效了，会提示 `runtime is undefined` 。



解决方案：

仅在分析构建速度时打开 SpeedMeasurePlugin 插件，这里我们先关闭 SpeedMeasurePlugin 的使用，来查看热更新效果。

最终效果：

更新 react 组件代码时，无需刷新页面，仅更新组件部分。

四、加快构建速度

四、加快构建速度

1. 更新版本

1.1 webpack 版本

使用最新的 webpack 版本，通过 webpack 自身的迭代优化，来加快构建速度。

这一点是非常有效的，如 webpack5 较于 webpack4，新增了持久化缓存、改进缓存算法等优化，webpack5 新特性可查看 参考资料 。

1.2 包管理工具版本

将 **Node.js** 、package 管理工具（例如 `npm` 或者 `yarn` ）更新到最新版本，也有助于提高性能。较新的版本能够建立更高效的模块树以及提高解析速度。

本文依赖的版本信息如下：

- `webpack@5.46.0`
- `node@14.15.0`
- `npm@6.14.8`

2. 缓存

2.1 cache

通过配置 webpack 持久化缓存 `cache: filesystem` ，来缓存生成的 webpack 模块和 chunk，改善构建速度。

简单来说，通过 `cache: filesystem` 可以将构建过程的 webpack 模板进行缓存，大幅提升二次构建速度、打包速度，当构建突然中断，二次进行构建时，可以直接从缓存中拉取，可提速 **90%** 左右。

`webpack.common.js` 配置方式如下：

```
module.exports = {
    cache: {
      type: 'filesystem', // 使用文件缓存
    },
}
```

引入缓存后，首次构建时间将增加 15%，二次构建时间将减少 90%，效果如下：

引入缓存前构建

```
SMP  ⏱
General output time took 3.11 secs

SMP  ⏱  Plugins
HtmlWebpackPlugin took 0.009 secs
HotModuleReplacementPlugin took 0.008 secs
ProgressPlugin took 0.003 secs
ReactRefreshPlugin took 0.001 secs

SMP  ⏱  Loaders
css-loader, and
postcss-loader, and
sass-loader took 2.13 secs
  module count = 1
modules with no loaders took 1.095 secs
  module count = 83
@pmmmwh/react-refresh-webpack-plugin, and
esbuild-loader took 0.234 secs
  module count = 3
html-webpack-plugin took 0.017 secs
  module count = 1
style-loader, and
css-loader, and
postcss-loader, and
sass-loader took 0.008 secs
  module count = 1


Build completed in 3.221s
```

引入缓存后首次构建

```
SMP  ⏱
General output time took 3.56 secs

SMP  ⏱  Plugins
HotModuleReplacementPlugin took 0.009 secs
HtmlWebpackPlugin took 0.008 secs
ProgressPlugin took 0.004 secs
ReactRefreshPlugin took 0.001 secs

SMP  ⏱  Loaders
css-loader, and
postcss-loader, and
sass-loader took 2.24 secs
  module count = 1
modules with no loaders took 0.841 secs
  module count = 83
@pmmmwh/react-refresh-webpack-plugin, and
esbuild-loader took 0.197 secs
  module count = 3
html-webpack-plugin took 0.013 secs
  module count = 1
style-loader, and
css-loader, and
postcss-loader, and
sass-loader took 0.008 secs
  module count = 1


Build completed in 3.666s
```

引入缓存后二次构建

```
SMP  ⏱
General output time took 0.383 secs

SMP  ⏱  Plugins
HtmlWebpackPlugin took 0.008 secs
ProgressPlugin took 0.004 secs
HotModuleReplacementPlugin took 0.003 secs
ReactRefreshPlugin took 0.002 secs



Build completed in 0.482s
```


2.2 dll ✕

在 webpack 官网构建性能 中看到关于 dll 的介绍：

在 webpack 官网构建性能 中看到关于 dll 的介绍：

> *dll* 可以为更改不频繁的代码生成单独的编译结果。可以提高应用程序的编译速度。

我兴冲冲的开始寻找 dll 的相关配置说明，太复杂了，接着找到了一个辅助配置 dll 的插件 autodll-webpack-plugin ，结果上面直接写了 webpack5 开箱即用的持久缓存是比 dll 更优的解决方案。

所以，不用再配置 dll了，上面介绍的 cache 明显更香。

2.3 cache-loader ✕

没错， cache-loader 也不需要引入了，上面的 cache 已经帮助我们缓存了。

3. 减少 loader、plugins

每个的 loader、plugin 都有其启动时间。尽量少地使用工具，将非必须的 loader、plugins 删除。

3.1 指定 include

为 loader 指定 include，减少 loader 应用范围，仅应用于最少数量的必要模块，。

> *webpack 构建性能文档*

> *rule.exclude* 可以排除模块范围，也可用于减少 *loader* 应用范围.

`webpack.common.js` 配置方式如下：

```
module.exports = {
    rules: [
        {
            test: /\.(js|ts|jsx|tsx)$/,
            include: paths.appSrc,
            use: [
              {
                loader: 'esbuild-loader',
                options: {
                  loader: 'tsx',
                  target: 'es2015',
                },
              }
            ]
        }
    ]
}
```

定义 loader 的 include 后，构建时间将减少 12%，效果如下：

定义include前构建

```
 SMP  
General output time took 2.79 secs

 SMP  Plugins
HotModuleReplacementPlugin took 0.008 secs
HtmlWebpackPlugin took 0.006 secs
ProgressPlugin took 0.005 secs
ReactRefreshPlugin took 0.002 secs

 SMP  Loaders
esbuild-loader took 2.42 secs
  module count = 73
css-loader, and
postcss-loader, and
sass-loader took 1.66 secs
  module count = 1
modules with no loaders took 0.297 secs
  module count = 10
@pmmmwh/react-refresh-webpack-plugin, and
esbuild-loader took 0.178 secs
  module count = 3
html-webpack-plugin took 0.016 secs
  module count = 1
style-loader, and
css-loader, and
postcss-loader, and
sass-loader took 0.007 secs
  module count = 1



Build completed in 2.858s
```

定义include后首次构建

```
 SMP  
General output time took 2.48 secs

 SMP  Plugins
HotModuleReplacementPlugin took 0.008 secs
HtmlWebpackPlugin took 0.005 secs
ReactRefreshPlugin took 0.003 secs
ProgressPlugin took 0.002 secs

 SMP  Loaders
css-loader, and
postcss-loader, and
sass-loader took 1.54 secs
  module count = 1
modules with no loaders took 0.988 secs
  module count = 83
@pmmmwh/react-refresh-webpack-plugin, and
esbuild-loader took 0.209 secs
  module count = 3
html-webpack-plugin took 0.017 secs
  module count = 1
style-loader, and
css-loader, and
postcss-loader, and
sass-loader took 0.006 secs
  module count = 1



Build completed in 2.536s
```



```
        include: [
          paths.appSrc,
        ],
        type: 'asset/resource',
      },
    ]
}
```

引入资源模块后，构建时间将减少 7%，效果如下：

4. 优化 resolve 配置

resolve 用来配置 webpack 如何解析模块，可通过优化 resolve 配置来覆盖默认配置项，减少解析范围。

4.1 alias

alias 可以创建 `import` 或 `require` 的别名，用来简化模块引入。

`webpack.common.js` 配置方式如下：

```
module.exports = {
    resolve: {
        alias: {
          '@': paths.appSrc, // @ 代表 src 路径
        },
    }
}
```

4.2 extensions

extensions 表示需要解析的文件类型列表。

根据项目中的文件类型，定义 extensions，以覆盖 webpack 默认的 extensions，加快解析速度。

由于 webpack 的解析顺序是从左到右，因此要将使用频率高的文件类型放在左侧，如下我将 `tsx` 放在最左侧。

`webpack.common.js` 配置方式如下：

```
module.exports = {
    resolve: {
        extensions: ['.tsx', '.js'], // 因为我的项目只有这两种类型的文件，如果有其他类型，需要添加进去。
    }
}
```

4.3 modules

modules 表示 webpack 解析模块时需要解析的目录。

指定目录可缩小 webpack 解析范围，加快构建速度。

`webpack.common.js` 配置方式如下：

```
module.exports = {
    modules: [
      'node_modules',
       paths.appSrc,
    ]
}
```

4.4 symlinks

如果项目不使用 symlinks（例如 `npm link` 或者 `yarn link`），可以设置 `resolve.symlinks: false`，减少解析工作量。

`webpack.common.js` 配置方式如下：

```
module.exports = {
    resolve: {
        symlinks: false,
    },
}
```

优化 resolve 配置后，构建时间将减少 1.5%，效果如下：

5. 多进程

上述可以看到 sass-loader 的构建时间有 1.56s，占据了整个构建过程的 60%，那么有没有方法来加快 sass-loader 的构建速度呢?

可以通过多进程来实现，试想将 sass-loader 放在一个独立的 worker 池中运行，就不会阻碍其他 loader 的

可以通过多进程来实现，例如将 sass-loader 放在一个独立的 worker 池中运行，就不会阻碍其他 loader 的构建了，可以大大加快构建速度。

5.1 thread-loader

通过 thread-loader 将耗时的 loader 放在一个独立的 worker 池中运行，加快 loader 构建速度。

安装：

```
npm i -D thread-loader
```

`webpack.common.js` 配置方式如下：

```
module.exports = {
    rules: [
        {
        test: /\.module\.(scss|sass)$/,
        include: paths.appSrc,
        use: [
          'style-loader',
          {
            loader: 'css-loader',
            options: {
              modules: true,
              importLoaders: 2,
            },
          },
          {
            loader: 'postcss-loader',
            options: {
              postcssOptions: {
                plugins: [
                  [
                    'postcss-preset-env',
                  ],
                ],
              },
            },
          },
          {
            loader: 'thread-loader',
            options: {
              workerParallelJobs: 2
            }
          },
          'sass-loader',
        ].filter(Boolean),
      },
    ]
}
```

> webpack 官网 提到 `node-sass` 中有个来自 *Node.js* 线程池的阻塞线程的 *bug*。 当使用 `thread-loader` 时，需要设置 `workerParallelJobs: 2` 。

由于 thread-loader 引入后，需要 0.6s 左右的时间开启新的 node 进程，本项目代码量小，可见引入 thread-loader 后，构建时间反而增加了0.19s。

因此，我们应该仅在非常耗时的 loader 前引入 thread-loader。

效果如下：

5.2 happypack ✕

happypack 同样是用来设置多线程，但是在 webpack5 就不要再使用 happypack 了，官方也已经不再维护了，推荐使用上文介绍的 thread-loader。

6. 区分环境

在 学习 Webpack5 之路（实践篇） - 模式（mode） 章节 已经介绍了 webpack 的不同模式的内置优化。

在开发过程中，切忌在开发环境使用生产环境才会用到的工具，如在开发环境下，应该排除 `[fullhash]` / `[chunkhash]` / `[contenthash]` 等工具。

同样，在生产环境，也应该避免使用开发环境才会用到的工具，如 webpack-dev-server 等插件。

7. 其他

7.1 devtool

不同的 `devtool` 设置，会导致性能差异。

在大多数情况下，最佳选择是 `eval-cheap-module-source-map` 。

详细区分可至 webpack devtool 查看。

`webpack.dev.js` 配置方式如下：

```
export.module = {
    devtool: 'eval-cheap-module-source-map',
}
```

7.2 输出结果不携带路径信息

默认 webpack 会在输出的 bundle 中生成路径信息，将路径信息删除可小幅提升构建速度。

```
module.exports = {
    output: {
        pathinfo: false,
    },
};
}
```

四、减小打包体积

1. 代码压缩

体积优化第一步是压缩代码，通过 webpack 插件，将 JS、CSS 等文件进行压缩。

1.1 JS 压缩

使用 TerserWebpackPlugin 来压缩 JavaScript。

webpack5 自带最新的 `terser-webpack-plugin` ，无需手动安装。

`terser-webpack-plugin` 默认开启了 `parallel: true` 配置，并发运行的默认数量： `os.cpus().length - 1` ，本文配置的 parallel 数量为 4，使用多进程并发运行压缩以提高构建速度。

`webpack.prod.js` 配置方式如下：

webpack.prod.js 配置方式如下：

```
const TerserPlugin = require('terser-webpack-plugin');
module.exports = {
    optimization: {
        minimizer: [
            new TerserPlugin({
              parallel: 4,
              terserOptions: {
                parse: {
                  ecma: 8,
                },
                compress: {
                  ecma: 5,
                  warnings: false,
                  comparisons: false,
                  inline: 2,
                },
                mangle: {
                  safari10: true,
                },
                output: {
                  ecma: 5,
                  comments: false,
                  ascii_only: true,
                },
              },
            }),
        ]
    }
}
```

体积减小 10%，效果如下：

1.1 ParallelUglifyPlugin ✕

你可能有听过 ParallelUglifyPlugin 插件，它可以帮助我们多进程压缩 JS，webpack5 的 TerserWebpackPlugin 默认就开启了多进程和缓存，无需再引入 ParallelUglifyPlugin。

1.2 CSS 压缩

使用 CssMinimizerWebpackPlugin 压缩 CSS 文件。

> 和 *optimize-css-assets-webpack-plugin* 相比， *css-minimizer-webpack-plugin* 在 *source maps* 和 *assets* 中使用查询字符串会更加准确，而且支持缓存和并发模式下运行。

`CssMinimizerWebpackPlugin` 将在 Webpack 构建期间搜索 CSS 文件，优化、压缩 CSS。

安装：

```
npm install -D css-minimizer-webpack-plugin
```

`webpack.prod.js` 配置方式如下：

```
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");

module.exports = {
  optimization: {
    minimizer: [
      new CssMinimizerPlugin({
          parallel: 4,
        }),
    ],
  }
}
```

由于 CSS 默认是放在 JS 文件中，因此本示例是基于下章节将 CSS 代码分离后的效果。

2. 代码分离

代码分离能够把代码分离到不同的 bundle 中，然后可以按需加载或并行加载这些文件。代码分离可以用于获取更小的 bundle，以及控制资源加载优先级，可以缩短页面加载时间。

2.1 抽离重复代码

SplitChunksPlugin 插件开箱即用，可以将公共的依赖模块提取到已有的入口 chunk 中，或者提取到一个新生成的 chunk。

webpack 将根据以下条件自动拆分 chunks：

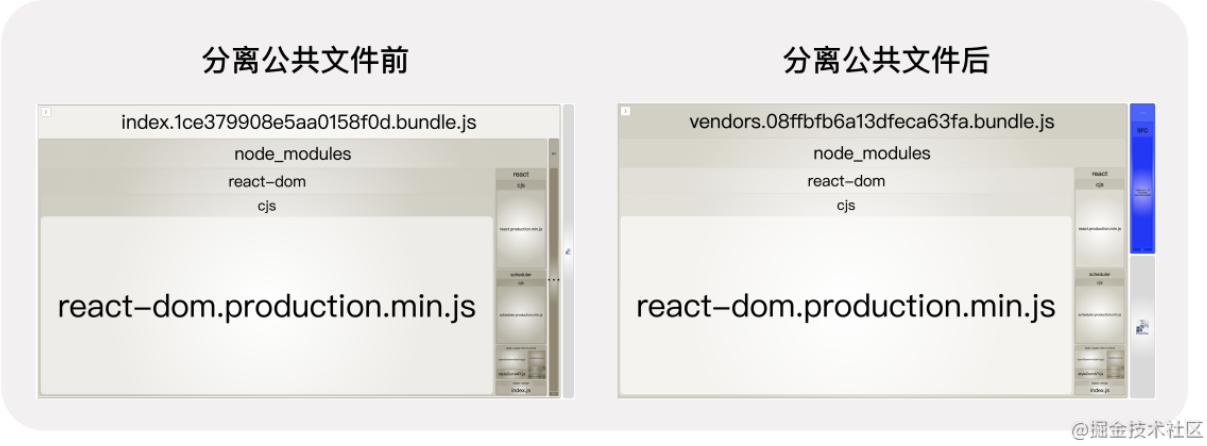
- 新的 chunk 可以被共享，或者模块来自于 `node_modules` 文件夹；
- 新的 chunk 体积大于 20kb（在进行 min+gz 之前的体积）；
- 当按需加载 chunks 时，并行请求的最大数量小于或等于 30；
- 当加载初始化页面时，并发请求的最大数量小于或等于 30； 通过 splitChunks 把 react 等公共库抽离出来，不重复引入占用体积。

> 注意：切记不要为 *cacheGroups* 定义固定的 *name*，因为 *cacheGroups.name* 指定字符串或始终返回相同字符串的函数时，会将所有常见模块和 *vendor* 合并为一个 *chunk*。这会导致更大的初始下载量并减慢页面加载速度。

`webpack.prod.js` 配置方式如下：

```
module.exports = {
    splitChunks: {
      // include all types of chunks
      chunks: 'all',
      // 重复打包问题
      cacheGroups:{
        vendors:{ // node_modules里的代码
          test: /[\\/]node_modules[\\/]/,
          chunks: "all",
          // name: 'vendors', 一定不要定义固定的name
          priority: 10, // 优先级
          enforce: true
        }
      }
    },
}
```

将公共的模块单独打包，不再重复引入，效果如下：



MiniCssExtractPlugin 插件将 CSS 提取到单独的文件中，为每个包含 CSS 的 JS 文件创建一个 CSS 文件，并且支持 CSS 和 SourceMaps 的按需加载。

安装：

```
npm install -D mini-css-extract-plugin
```

webpack.common.js 配置方式如下：

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
  plugins: [new MiniCssExtractPlugin()],
  module: {
    rules: [
        {
        test: /\.module\.(scss|sass)$/,
        include: paths.appSrc,
        use: [
          'style-loader',
          isEnvProduction && MiniCssExtractPlugin.loader, // 仅生产环境
          {
            loader: 'css-loader',
            options: {
              modules: true,
              importLoaders: 2,
            },
          },
          {
            loader: 'postcss-loader',
            options: {
              postcssOptions: {
                plugins: [
                  [
                    'postcss-preset-env',
                  ],
                ],
              },
            },
          },
          {
            loader: 'thread-loader',
            options: {
              workerParallelJobs: 2
            }
          },
          'sass-loader',
        ].filter(Boolean),
      },
    ]
  },
};
```

> 注意：*MiniCssExtractPlugin.loader* 要放在 *style-loader* 后面。

效果如下：

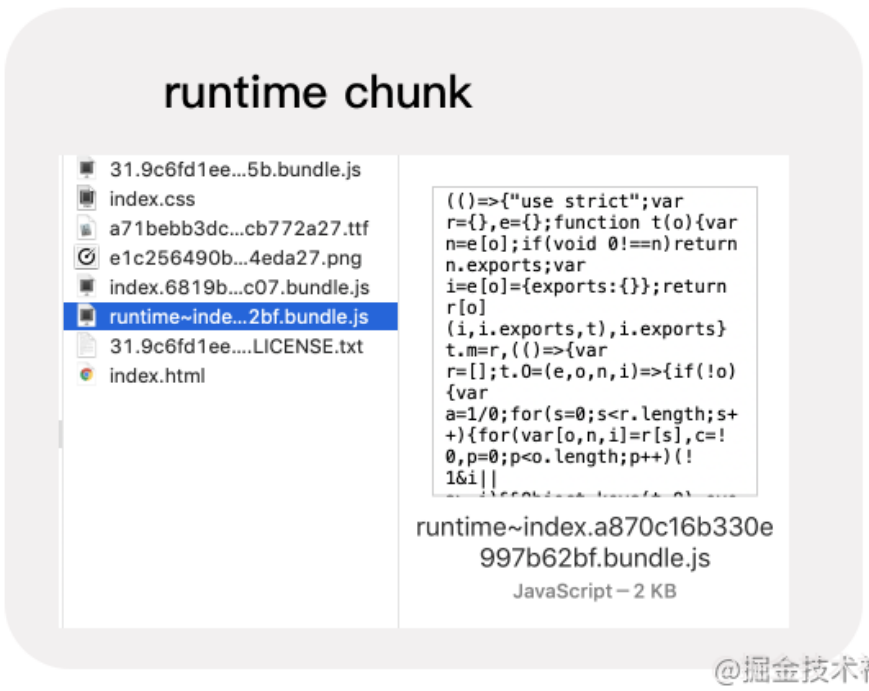
2.3 最小化 entry chunk

通过配置 `optimization.runtimeChunk = true`，为运行时代码创建一个额外的 chunk，减少 entry chunk 体积，提高性能。

`webpack.prod.js` 配置方式如下：

```
module.exports = {
    optimization: {
        runtimeChunk: true,
      },
    };
}
```

效果如下：



3. Tree Shaking（摇树）

摇树，顾名思义，就是将枯黄的落叶摇下来，只留下树上活的叶子。枯黄的落叶代表项目中未引用的无用代码，活的树叶代表项目中实际用到的源码。

3.1 JS

JS Tree Shaking 将 JavaScript 上下文中的未引用代码（Dead Code）移除，通过 `package.json` 的 `"sideEffects"` 属性作为标记，向 compiler 提供提示，表明项目中的哪些文件是 "pure(纯正 ES2015 模块)"，由此可以安全地删除文件中未使用的部分。

Dead Code 一般具有以下几个特征：

- 代码不会被执行，不可到达；
- 代码执行的结果不会被用到；
- 代码只会影响死变量（只写不读）。

3.1.1 webpack5 sideEffects

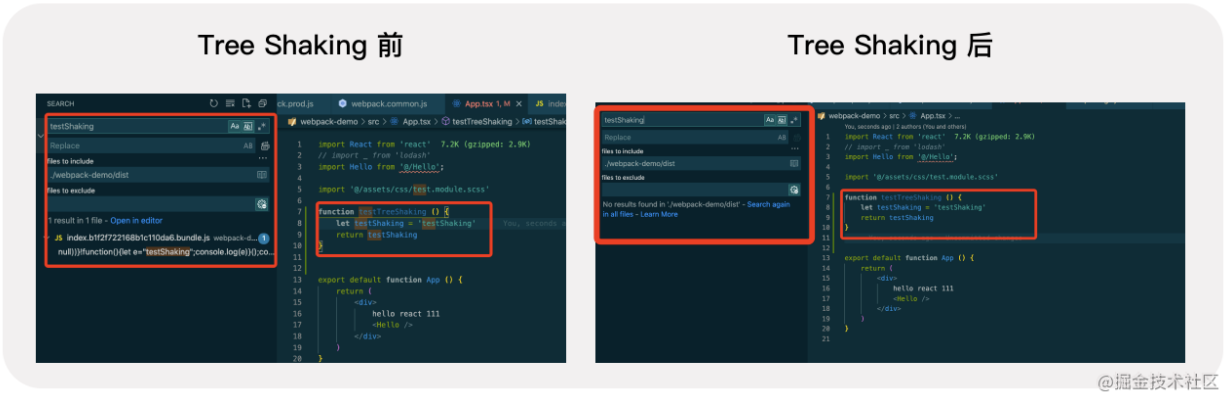
通过 package.json 的 `"sideEffects"` 属性，来实现这种方式。

```
{
  "name": "your-project",
  "sideEffects": false
}
```

需注意的是，当代码有副作用时，需要将 `sideEffects` 改为提供一个数组，添加有副作用代码的文件路径：

```
{
  "name": "your-project",
  "sideEffects": ["./src/some-side-effectful-file.js"]
}
```

添加 TreeShaking 后，未引用的代码，将不会被打包，效果如下：



3.1.2 对组件库引用的优化

webpack5 sideEffects 只能清除无副作用的引用，而有副作用的引用则只能通过优化引用方式来进行 `Tree Shaking`。

1. lodash

类似 `import { throttle } from 'lodash'` 就属于有副作用的引用，会将整个 lodash 文件进行打包。

优化方式是使用 `import { throttle } from 'lodash-es'` 代替 `import { throttle } from 'lodash'`，lodash-es 将 Lodash 库导出为 ES 模块，支持基于 ES modules 的 tree shaking，实现按需引入。

2. ant-design

ant-design 默认支持基于 ES modules 的 tree shaking，对于 js 部分，直接引入 `import { Button } from 'antd'` 就会有按需加载的效果。

假如项目中仅引入少部分组件，`import { Button } from 'antd'` 也属于有副作用，webpack不能把其他组件进行tree-shaking。这时可以**缩小引用范围**，将引入方式修改为 `import { Button } from 'antd/lib/button'` 来进一步优化。

3.2 CSS

上述对 JS 代码做了 Tree Shaking 操作，同样，CSS 代码也需要摇摇树，打包时把没有用的 CSS 代码摇走，可以大幅减少打包后的 CSS 文件大小。

使用 purgecss-webpack-plugin 对 CSS Tree Shaking。

安装：

```
npm i purgecss-webpack-plugin -D
```

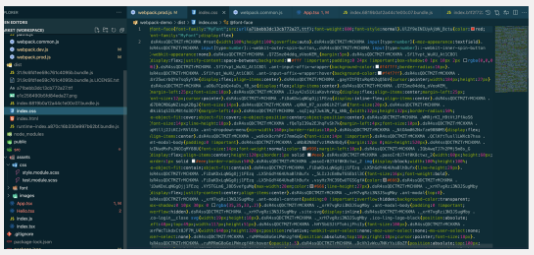
因为打包时 CSS 默认放在 JS 文件内，因此要结合 webpack 分离 CSS 文件插件 `mini-css-extract-plugin` 一起使用，先将 CSS 文件分离，再进行 CSS Tree Shaking。

`webpack.prod.js` 配置方式如下：

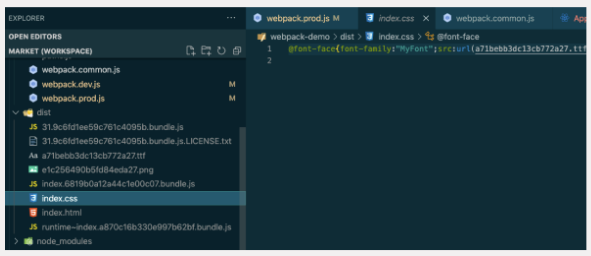
```
const glob = require('glob')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const PurgeCSSPlugin = require('purgecss-webpack-plugin')
const paths = require('paths')

module.exports = {
  plugins: [
    // 打包体积分析
    new BundleAnalyzerPlugin(),
    // 提取 CSS
    new MiniCssExtractPlugin({
      filename: "[name].css",
    }),
    // CSS Tree Shaking
    new PurgeCSSPlugin({
      paths: glob.sync(`${paths.appSrc}/**/*`,  { nodir: true }),
    }),
  ]
}
```

上面为了测试 CSS 压缩效果，我引入了大量无效 CSS 代码，因此 Tree Shaking 效果也非常明显，效果如下：

Tree Shaking 前



Tree Shaking 后



3. CDN

上述是对 webpack 配置的优化，另一方面还可以通过 CDN 来减小打包体积。

> 这里引入 *CDN* 的首要目的为了减少打包体积，因此仅仅将一部分大的静态资源手动上传至 *CDN*，并修改本地引入路径。下文的加快加载速度，将介绍另一种 *CDN* 优化手段。

将大的静态资源上传至 CDN：

- 字体：压缩并上传至 CDN；
- 图片：压缩并上传至 CDN。

五、加快加载速度

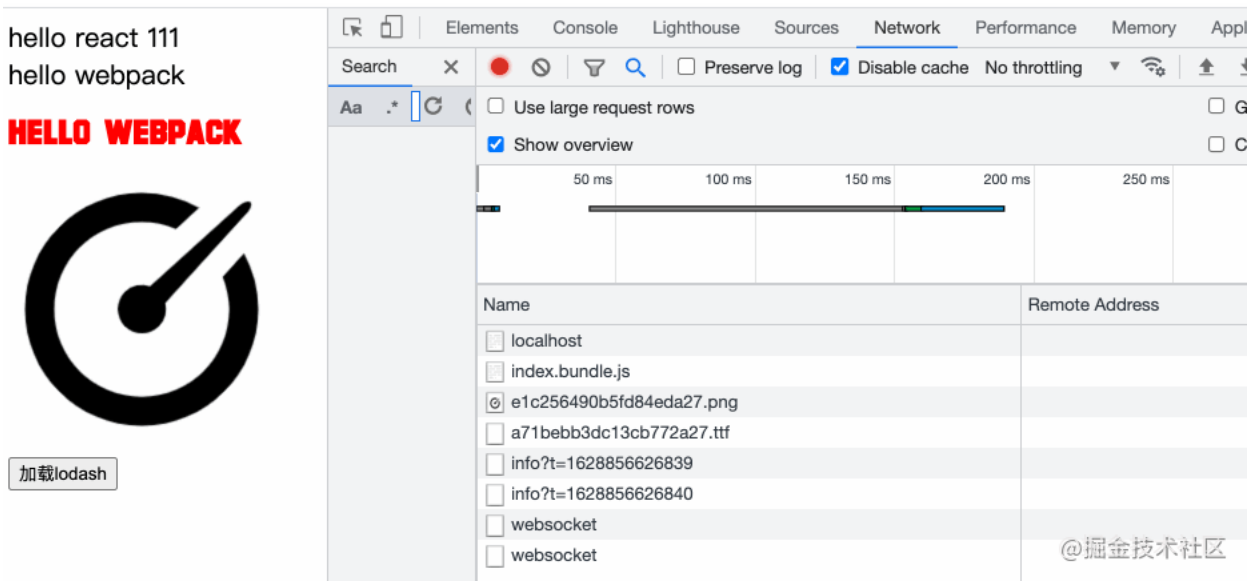
1. 按需加载

通过 webpack 提供的 import() 语法 动态导入 功能进行代码分离，通过按需加载，大大提升网页加载速度。

使用方式如下：

```
export default function App () {
    return (
        <div>
            hello react 111
            <Hello />
            <button onClick={() => import('lodash')}>加载lodash</button>
        </div>
    )
}
```

效果如下：



2. 浏览器缓存

浏览器缓存，就是进入某个网站后，加载的静态资源被浏览器缓存，再次进入该网站后，将直接拉取缓存资源，加快加载速度。

webpack 支持根据资源内容，创建 hash id，当资源内容发生变化时，将会创建新的 hash id。

配置 JS bundle hash， `webpack.common.js` 配置方式如下：

```
module.exports = {
  // 输出
  output: {
    // 仅在生产环境添加 hash
    filename: ctx.isEnvProduction ? '[name].[contenthash].bundle.js' : '[name].bundle.js',
  },
}
```

配置 CSS bundle hash， `webpack.prod.js` 配置方式如下：

```
module.exports = {
  plugins: [
    // 提取 CSS
    new MiniCssExtractPlugin({
      filename: "[hash].[name].css",
    }),
  ],
}
```

配置 optimization.moduleIds，让公共包 splitChunks 的 hash 不因为新的依赖而改变，减少非必要的 hash 变动， `webpack.prod.js` 配置方式如下：

```
module.exports = {
  optimization: {
    moduleIds: 'deterministic',
  }
}
```

通过配置 contenthash/hash，浏览器缓存了未改动的文件，仅重新加载有改动的文件，大大加快加载速度。

3. CDN

将所有的静态资源，上传至 CDN，通过 CDN 加速来提升加载速度。

`webpack.common.js` 配置方式如下：

```
export.modules = {
output: {
    publicPath: ctx.isEnvProduction ? 'https://xxx.com' : '', // CDN 域名
  },
}
```

六、优化前后对比

在仓库代码仅 webpack 配置不同的情况下，查看优化前后对比。

- 优化前 github 地址
- 优化后 github 地址

1. 构建速度

类型	首次构建	未修改内容二次构建	修改内容二次构建
优化前	2.7s	2.7s	2.7s
优化后	2.7s	0.5s	0.3s

JELLY
内容加载中...

2. 打包体积

类型	体积大小

类型 类型	体积大小 体积大小
优化前	250 kb
优化后	231 kb

七、总结

从上章节 **[优化前后对比]** 可知，在小型项目中，添加过多的优化配置，作用不大，反而会因为额外的 loader、plugin 增加构建时间。

在加快构建时间方面，作用最大的是配置 cache，可大大加快二次构建速度。

在减小打包体积方面，作用最大的是压缩代码、分离重复代码、Tree Shaking，可最大幅度减小打包体积。

在加快加载速度方面，按需加载、浏览器缓存、CDN 效果都很显著。

本篇就介绍到这儿啦，有更好的 webpack 优化方式欢迎评论区告诉我哦~

本文源码：

- webpack Demo2 优化前
- webpack Demo2 优化后

希望能对你有所帮助，感谢阅读 ~

别忘了点个赞鼓励一下我哦，笔芯♡

往期精彩

- 学习 Webpack5 之路（基础篇）
- 学习 Webpack5 之路（实践篇）

参考资料

- Tree-Shaking性能优化实践 - 原理篇
- Tree-Shaking性能优化实践 - 实践篇
- 三十分钟掌握Webpack性能优化
- 玩转 webpack，使你的打包速度提升 90%
- 带你深度解锁Webpack系列(优化篇)
- Webpack 5 中的新特性
- 辛辛苦苦学会的 webpack dll 配置，可能已经过时了