20 Webpack 优化之速度优化

更新时间：2019-06-24 09:32:18

> 什么是路？就是从没路的地方践踏出来的，从只有荆棘的地方开辟出来的。
>
> —— 鲁迅

当 Webpack 的项目文件多了之后，构建过程会越来越慢，这时候就需要做一些构建速度方面的优化手段了，本篇文章就是介绍了构建速度优化相关的配置。影响 Webpack 构建速度的有两个「大户」：一个是 loader 和 plugin 方面的构建过程，一个就是压缩，把这两个东西优化起来，可以减少很多发布的时间，所以本文将从构建过程和压缩两个方面来做讲解。

> Tips：
>
> 1. 使用最新版本的 Webpack 4 要比之前的 Webpack 3 和 2 版本快很多，所以单纯升级 Webpack 版本到最新版本就可以提升一大截构建速度；
> 2. 区分开发和生产环境两套配置，各司其职，例如在开发阶段，代码压缩、目录清理、计算 hash、提取 CSS 这些都没有必要做。

要优化构建过程，可以从减少查找过程、多线程、提前编译和 Cache 多个角度来优化。

减少查找过程

在这里我们要做的事情是聚焦 Webpack 要处理的代码目录，帮助缩小 Webpack 的查找过程。

使用 `resolve.alias` 减少查找过程

`resolve.alias` 配置项通过别名（alias）来把原导入路径映射成一个新的导入路径。比如我们经常使用的 react 库，其实 react 库中有两套代码，一套是基于 CommonJs 的模块化代码，一套是打包好的完整代码，react.js 用于开发环境，react.min.js 用生产环境。所以通过 resolve.alias 配置，可以让 Webpack 处理时，直接使用打包好的 react，从而跳过耗时的模块解析，还有我们项目中可能会有一些相对路径的写法，可以使用 alias 配置来减少查找过程，具体示例配置如下：

```
module.exports = {
    resolve: {
        // 使用 alias 把导入 react 的语句换成直接使用单独完整的 react.min.js 文件，
        // 减少耗时的递归解析操作
        alias: {
            react: path.resolve(__dirname, './node_modules/react/dist/react.min.js'),
            '@lib': path.resolve(__dirname, './src/lib/')
        }
    }
};
```

使用 `resolve.extensions` 优先查找

在导入语句没带文件后缀时，Webpack 会自动带上后缀后去尝试询问文件是否存在，查询的顺序是按照我们配置的`resolve.extensions`顺序从前到后查找，如果我们配置 `resolve.extensions= ['js', 'json']`，那么会先找`xxx.js`然后没有再查找`xxx.json`，所以我们应该把常用到的文件后缀写在前面，或者我们**导入模块时，尽量带上文件后缀名**。

排除不需要解析的模块

`module.noParse`配置项可以让 Webpack 忽略对部分没采用模块化的文件递归解析处理，例如：jQuery、ChartJS，它们体积庞大又没有采用模块化标准，让 Webpack 去解析这些文件耗时又没有意义，所以使用`module.noParse`排除它们。

```
module.exports = {
    module: {
        noParse: /node_modules\/(jquey\.js)/;
    }
}
```

> Tips：被忽略掉的文件里不应该包含 `import`、`require`、`define` 等模块化语句，不然会导致构建出的代码中包含无法在浏览器环境下执行的模块化语句。

合理配置 rule 的查找范围

在 rule 配置上，有`test`、`include`、`exclude`三个可以控制范围的配置，最佳实践是:

- 只在 `test` 和 文件名匹配中使用正则表达式；
- 在 `include` 和 `exclude` 中使用绝对路径数组；
- 尽量避免 `exclude`，更倾向于使用 `include`。

示例配置：

```
rules: [
    {
        // test 使用正则
        test: /\.js$/,
        loader: 'babel-loader',
        // 排除路径使用数组
        exclude: [path.resolve(__dirname, './node_modules')],
        // 查找路径使用数组
        include: [path.resolve(__dirname, './src')]
    }
];
```

> Tips：`exclude`优先级要优于`include`和`test`，所以当三者配置有冲突时，`exclude`会优先于其他两个配置。

利用多线程提升构建速度

由于运行在 Node.js 之上的 Webpack 是单线程模型的，所以 Webpack 需要处理的事情需要一件一件的做，不能多件事一起做。

我们需要 Webpack 能同一时间处理多个任务，发挥多核 CPU 电脑的威力，HappyPack 就能让 Webpack 做到这点，它把任务分解给多个子进程去并发的执行，子进程处理完后再把结果发送给主进程。

我们知道 Node.js 是单线程模型的，Webpack 运行在 Node.js 上处理事情是一件件处理的，我们可以通过插件方式让 Webpack 支持多个线程进行同时打包，以便提高编译打包的速度。但是需要注意的是，如果项目比较简单，没有必要采用这种方式，简单的项目而使用多线程编译打包会因为多线程打包浪费更多的 CPU 资源，这样最终结果是不仅不能加快打包的速度，反而会降低打包的速度。

多线程打包有两种方案：thread-loader和HappyPack。

thread-loader

thread-loader 是针对 loader 进行优化的，它会将 loader 放置在一个 worker 池里面运行，以达到多线程构建。

thread-loader 在使用的时候，需要将其放置在其他 loader 之前，如下面实例：

```
// webpack.config.js

module.exports = {
    module: {
        rules: [
            {
                test: /\.js$/,
                include: path.resolve('src'),
                use: [
                    'thread-loader'
                    // 你的高开销的loader放置在此 (e.g babel-loader)
                ]
            }
        ]
    }
};
```

HappyPack

HappyPack 是通过多进程模型，来加速代码构建，具体的原理部分可以看它的介绍，里面有个详细的流程图，下面直接上它的示例代码：

```
// webpack.config.js
const os = require('os');
const HappyPack = require('happypack');
// 根据 cpu 数量创建线程池
const happyThreadPool = HappyPack.ThreadPool({size: os.cpus().length});
module.exports = {
    module: {
        rules: [
            {
                test: /\.js$/,
                use: 'happypack/loader?id=jsx'
            },

            {
                test: /\.less$/,
                use: 'happypack/loader?id=styles'
            }
        ]
    },
    plugins: [
        new HappyPack({
            id: 'jsx',
            // 多少个线程
            threads: happyThreadPool,
            loaders: ['babel-loader']
        }),

        new HappyPack({
            id: 'styles',
            // 自定义线程数量
            threads: 2,
            loaders: ['style-loader', 'css-loader', 'less-loader']
        })
    ]
};
```

> Tips：给 loader 配置使用 HappyPack 需要对应的 loader 支持才行，例如 url-loader 和 file-loader 就不支持 HappyPack，在 HappyPack 的 wiki 中有一份支持 loader 的列表。

使用 webpack.DllPlugin 来预先编译

预先编译和打包不会变动存在的文件，在业务代码中直接引入，加快 Webpack 编译打包的速度，但是并不能减少最后生成的代码体积。

例如下面代码：

```
import React, {Component} from 'react';
import ReactDOM, {render} from 'react-dom';
```

使用了 react 和 react-dom 两个库，这两个库一般我们日常业务代码开发的时候并不会升级版本或者修改内部代码，但是在每次构建的时候，这些代码都会重新编译和打包，这样就很浪费时间，`webpack.DllPlugin`就是来解决这个问题的插件，使用它可以在第一次编译打包后就生成一份不变的代码供其他模块引用，这样下一次构建的时候就可以节省开发时编译打包的时间。

要使用`DllPlugin`的功能，需要配合`webpack.DllReferencePlugin`来使用。

DLLPlugin 这个插件是在一个额外独立的 Webpack 设置中创建一个只有公共库的 dll 文件，这时候我们项目中应该单独为 dll 文件创建一个配置文件，例如`webpack.config.dll.js`，`webpack.config.dll.js` 作用是把所有的第三方库依赖打包到一个 bundle 的 `dll` 文件里面，还会生成一个名为 `manifest.json` 文件。生成的 `manifest.json` 会让 `DllReferencePlugin`在`webpack.config.js`配置中映射到相关的依赖上去的。

DllReferencePlugin 这个插件是在 `webpack.config.js` 中使用的，该插件的作用是把刚刚在 `webpack.config.dll.js` 中打包生成的 dll 文件引用到需要的预编译的依赖上来。什么意思呢？就是说，假设在 `webpack.config.dll.js` 中打包后会生成 `dll.js` 文件和 `manifest.json`两个文件，`dll.js` 文件包含所有的第三方库文件，`manifest.json` 文件会包含所有库代码的一个索引，当在使用 `webpack.config.js` 文件打包 `DllReferencePlugin` 插件的时候，会使用该 `DllReferencePlugin` 插件读取 `vendor-manifest.json` 文件，看看是否有该第三方库。`manifest.json` 文件就是有一个第三方库的一个映射而已。

当第一次使用 `webpack.config.dll.js` 文件会对第三方库打包，打包完成后就不会再打包它了，然后每次运行 `webpack.config.js` 文件的时候，都会打包项目中本身的文件代码，当需要使用第三方依赖的时候，会使用 `DllReferencePlugin` 插件去读取第三方依赖库，而只有我们修改第三方公共库的时候，才会执行`webpack.config.dll.js`。本质上来说 DLL 方案就是一种缓存机制。

> Tips：DLL 是动态链接库（Dynamic-link library）的英文缩写，最早是微软提出来的一种共享函数库概念，实际就是将一些经常会共享的代码制作成 DLL 文档，当其他代码需要使用这些 DLL 中的代码时，Windows 操作系统会将 DLL 文档加载到内存中。这里借用了 DLL 的概念，帮助 Webpack 使用者理解用途。

我们接下来看看具体怎么配置，首先我们的`src/index.js`按照正常代码写法即可，例如内容如下：

```
import React, {Component} from 'react';
import ReactDOM, {render} from 'react-dom';

console.log(React, Component, ReactDOM, render, 'hell dll');
```

然后我们创建一个`webpack.config.dll.js`，这里面添加了我们需要打包的第三方依赖库 react 和 react-dom：

```
// webpack.config.dll.js
const webpack = require('webpack');
// 这里是第三方依赖库
const vendors = ['react', 'react-dom'];

module.exports = {
    mode: 'production',
    entry: {
        // 定义程序中打包公共文件的入口文件vendor.js
        vendor: vendors
    },
    output: {
        filename: '[name].[chunkhash].js',
        // 这里是使用将 verdor 作为 library 导出，并且指定全局变量名字是[name]_[chunkhash]
        library: '[name]_[chunkhash]'
    },
    plugins: [
        new webpack.DllPlugin({
            // 这里是设置 mainifest.json 路径
            path: 'manifest.json',
            name: '[name]_[chunkhash]',
            context: __dirname
        })
    ]
};
```

这时候执行`webpack --config webpack.config.dll.js`，这显示打包成功：

```
Hash: 655d0e45ebeb59b0b718
Version: webpack 4.30.0
Time: 296ms
Built at: 04/27/2019 5:30:56 PM
                         Asset     Size  Chunks             Chunk Names
vendor.1c25daba4f58872736ee.js  117 KiB       0  [emitted]  vendor
Entrypoint vendor = vendor.1c25daba4f58872736ee.js
[2] dll vendor 12 bytes {0} [built]
[8] (webpack)/buildin/global.js 472 bytes {0} [built]
    + 7 hidden modules
```

查看一下目录结构如下：

```
├── dist
│   └── vendor.1c25daba4f58872736ee.js # 这个是刚刚打包出来的 dll 文件
├── manifest.json    # 这个是配置文件，后续要用
├── node_modules
├── package.json
├── src
│   └── index.js
├── webpack.config.dll.js # dll 配置
└── webpack.config.js # 普通配置
```

下面我们来看下正常项目的配置文件`webpack.config.js`是怎么配置的，`webpack.config.js`中我们需要使用`DllReferencePlugin`指定`manifest.json`的内容即可：

```
// webpack.config.js
const webpack = require('webpack');

module.exports = {
    output: {
        filename: '[name].[chunkhash].js'
    },
    entry: {
        app: './src/index.js'
    },
    plugins: [
        new webpack.DllReferencePlugin({
            context: __dirname,
            // 这里导入 manifest配置内容
            manifest: require('./manifest.json')
        })
    ]
};
```

这时候执行`webpack`命令就可以生成`app.js`文件了，并且`app.js`并不会包含 dll 打包出来的`vendor.js`文件内容，打包速度也提升了不少！

```
Hash: c930652d2984598d0f9f
Version: webpack 4.30.0
Time: 103ms
Built at: 04/27/2019 5:48:44 PM
                        Asset      Size  Chunks             Chunk Names
app.8f79faabcf46570976b9.js  1.14 KiB       0  [emitted]  app
```

> Tips：在实际操作中，HTML 中不会主动引入 dll 的 vendor.js 文件，这时候需要我们想办法手动或者通过插件添加进去，比如使用add-asset-html-webpack-plugin，或者在 dll 打包的时候修改一下 html-webpack-plugin 的 template 文件，正常打包的时候直接使用这个 template 文件再打包一次即可。

缓存（Cache）相关

提升构建速度的另外一个大杀器是使用缓存！下面介绍两种缓存方式。

babel-loader 配置

Webpack 中打包的核心是 JavaScript 文件的打包，JavaScript 使用的是 babel-loader，其实打包时间长很多时候是 babel-loader 执行慢导致的。这时候我们不仅要使用`exclude`和`include`来尽可能准确的指定要转换内容的范畴，还需要关注 babel-loader 在执行的时候，可能会产生一些运行期间重复的公共文件，造成代码体积大冗余，同时也会减慢编译的速度。

babel-loader 提供了 `cacheDirectory` 配置给 Babel 编译时给定的目录，并且将用于缓存加载器的结果，但是这个设置默认是`false`关闭的状态，我们需要设置为`true`，这样 babel-loader 将使用默认的缓存目录 。

`node_modules/.cache/babel-loader`，如果在任何根目录下都没有找到 `node_modules` 目录，将会降级回退到操作系统默认的临时文件目录。

```
rules: [
    {
        test: /\.js$/,
        loader: 'babel-loader',
        options: {
            cacheDirectory: true
        },
        // 排除路径
        exclude: /node_modules/,
        // 查找路径
        include: [path.resolve('.src')]
    }
];
```

其他构建过程的优化点

1. sourceMap 生成耗时严重，根据之前 sourceMap[TODO](sourcemap 表格链接)表格选择合适的`devtool`值；
2. 切换一些 loader 或者插件，比如：fast-sass-loader可以并行处理 sass 文件，要比 sass-loader 快 5~10 倍；

压缩速度优化

相对于构建过程而言，压缩相对我们来说只有生产环境打包才会做，而且压缩我们除了添加 cache 和多线程支持之外，可以优化的空间较小。我们在使用terser-webpack-plugin的时候可以通过下面的配置开启多线程和缓存：

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

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

总结

本文主要从构建速度角度来介绍如何提升打包速度，分别从构建过程和压缩两个维度来介绍提升速度的方法，其中压缩角度可以优化的点比较少，而在构建过程中可以从减少查找过程、多线程、提前编译和缓存多个角度来优化，其中重点介绍了使用减少查找过程的几种配置方式，使用 thread-loader 和 HappyPack 来开启多线程，使用 DLLPlugin 来预先编译和使用 babel-loader 的缓存等方法。实际项目中并不是非要做构建速度的优化，如果项目简单完全没有必要，当做构建速度优化的时候也并不是本文介绍的所有方式都可以使用，要具体问题具体分析。不管怎样，保持 Webpack 版本最新是一个既简单又效果不错的方式！

> 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 Webpack 打包速度的方面介绍 Webpack 优化方案。

19 使用 Webpack 的 splitChunks 功能来拆分代码

21 使用 Webpack 的 Tree-Shaking