目录

8 Webpack 性能优化两三事

更新时间：2019-08-01 11:35:40

> 没有引发任何行动的思想都不是思想，而是梦想。
>
> —— 马丁

相信前端开发的同学都对Webpack并不陌生，它就是一个打包工具。其实在Webpack之前也有很多打包工具，比如：Gulp、Grunt、Rollup等，而Webpack能够在诸多的打包工具当中能够脱颖而出，成为现在打包工具中的老大，必然有它厉害的地方。

很多人都觉得Webpack的配置复杂多变，非常不利于新手入门。其实很多时候这不是Webpack能够决定的，由于前端业务越来越复杂，自然需要更多复杂的配置项来支撑。对于Webpack具体一些详细的配置，不是我们这部分的重点，这里我推荐大家直接阅读这个专栏即可，我们这里主要探讨Webpack性能优化。

引入

我们都知道WebPack在大型项目当中是有2套配置代码，一套是针对开发时候的配置，另外一套是针对生成环境打包的配置，开发的时候我们主要的优化点是提高构建速度，生产环境打包我们主要的优化点是希望输出打包文件尽量小。所以整个WebPack优化章节，我们会从这2个方面进行讲解。

构建速度优化

随着前端项目应用规模和复杂度的不断攀升，前端项目的构建速度也在不断增加，那么下面我们就介绍一些常见的与构建速度优化相关的方法。

npm install 过程中的优化

现在所有的前端项目都需要安装npm包，npm安装的时候都会经历如下阶段：

一步分析，如果有依赖文件这里也会一并下载。

2. 然后npm会获取包的下载地址，在版本描述文件中一般都有resolved字段来记录包的地址，方便npm寻找，然后进行下载。
3. 到这一步，我们就可以下载相关包了。npm当中也有缓存机制，在这一步会检查本地是否有缓存信息，如果有，就直接使用缓存下载，这也是为什么我们第二次安装相关依赖的时候，速度会大大加快的原因。
4. 最后下载后的压缩包拷贝至本地的node_moudules文件夹当中。

分析上面的流程，最耗费时间的部分就是递归分析包的依赖关系和版本。现在一个普通的前端项目中就有几百个包，加上每个包都有相关的依赖，可想而知工作量有多大。所以这部分是我们优化的重点，优化的办法也很简单，就是增加版本描述文件。如果你的npm版本是5以下，那么可以使用shrinkwrap.json，如果是npm5以上，则可以使用npm自带的package-lock.json，还有使用Facebook官方出品的yarn工具，可以使用yarn.lock。上面介绍的这些文件都是固定npm版本号以及具体的下载地址的，如下图：

有了这些信息之后，整个npm install的时间就会大大缩短。

具体仓库地址的选择

npm官方的服务器地址，在安装一些包的时候常常会失败，因此，这里我们推荐大家使用淘宝提供的npm仓库，我们输入如下命令即可：

```
npm config set registry https://registry.npm.taobao.org
```

如果公司有独立服务器的话，我们也可以自己去同步一些常用的包过来放到自己公司服务器来加快安装速度。

提升Webpack构建速度

Webpack是我们每天都会用到的打包工具，想必Webpack构建速度一直是一件让人头疼的事情，如果你的项目很大，那么Webpack构建速度可能需要几分钟，这个时候我们很容易注意力不集中去做一些别的事情，造成工作效率的下降。

那么如何提升构件速度呢？首先，我们简单介绍一下Webpack的打包流程，整个打包流程我们可以将其理解为一个函数，配置文件则是其参数，传入合理的参数后，运行函数就能得到我们想要的结果。打包的内部机制我们无需过多关注，我们只需要思考如何将配置文件当中的各项配置合理运用，最终加快构建速度，下面我们就来一一介绍这些方法：

1. 区分开发环境和生产环境，上面提到我们需要区分开发环境和生产环境，在WebPack4.x中非常贴心的为我们引入了mode配置项，通过指定production 或 development来区分是开

增量编译，而不是覆盖更新，这样我们可以节省大量编译时间。当我们把mode配置为production，则会默认开启运行时的性能优化以及构建结果优化。

当然两种运行环境下还有其他很多配置上的区别，但是WebPack4.x为我们提供的mode配置则是一定要配置的一个选项，配置之后WebPack4.x针对这两种环境的一些默认优化也就使用在项目当中了，对于两种环境配置的其他一些区别，我们下面会逐步介绍。

2. 减少不必要的编译，我们在使用loader处理文件的时候，应该尽量把文件范围缩小，对于一些不需要处理的文件直接忽略。这里我们以babel-loader为例，看一下如何处理：

```
module: {
  rules: [
    {
      //处理后缀名为js的文件
      test: /\.js$/,
      //exclude去掉不需要转译的第三方包 && 或者这里使用include去声明哪些文件需要被
      exclude: /(node_modules|bower_components)/,
      //babel的常用配置项
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env'],
          //缓存设置为开启
          cacheDirectory: true
        }
      }
    }
  ]
}
```

这里我们对于不需要处理的第三方包直接使用 exclude 属性排除在外，或者需要处理的文件使用 include 属性去包含，此外上面我们在 options 配置当中增加了 cacheDirectory: true，这样对于转译结果就可以直接缓存到文件系统当中，在我们下次需要的时候直接到缓存当中读取即可。

> Tips:bower是专门为前端设计的包管理器，npm与bower的区别是npm支持嵌套依赖管理，而bower只能支持扁平的依赖，目前bower已停止维护。

3.使用模块热替换(HMR)，传统的如果我们没有配置模块热替换，则需要每次刷新整个页面，效率很低。而使用模块热替换之后，我们只需要重新编译发生变化的模块，不需要编译所有模块，速度上面大大提高。具体配置方法如下：

```
module.exports = {
  ......
  plugins: [
    new webpack.HotModuleReplacementPlugin(), // 引入模块热替换插件
  ],
  devServer: {
    hot: true // 开启模块热替换模式
  }
}
```

CommonsChunkPlugin插件，后来Webpack升级到4.x之后使用的则是optimization.splitChunks。下面我们来分别介绍一下这2个插件的使用：

- CommonsChunkPlugin配置

```
new webpack.optimize.CommonsChunkPlugin({
    // 指定该代码块的名字
    name: "framework",
    // 指定输出代码的文件名
    filename: "framework.js",
    // 指定最小共享模块数
    minChunks: 4,
    // 指定作用于哪些入口
    chunks: ["pageA", "pageB", "pageC"]
})
```

> Tips：这里minChunks 需要单独解释一下，它代表一个最小值，当工程中至少有超过该值数量的入口引用了相同的一个模块时，这个模块才会被提取到 CommonsChunkPlugin 中。比如说工程中有 5 个入口文件，而 minChunks 是 4。那么当有 4 个或 4个以上的入口文件引用了 react，react 就会单独出现在 CommonsChunkPlugin 中。相反如果只有 2 个入口引用了 react，那么 react 就会分别被打包到这两个入口生成的 JS 文件中。

- optimization.splitChunks配置

```
optimization: {
        splitChunks: {
                //设置那些代码用于分割
            chunks: "all",
                        // 指定最小共享模块数(与CommonsChunkPlugin的minChunks类似)
            minChunks: 1,
                        // 形成一个新代码块最小的体积
            minSize: 0,
            cacheGroups: {
                framework: {
                    test: /react|lodash/,
                    name: "vendor",
                    enforce: true
                }
            }
        }
    }
```

> Tips：cacheGroups对象，定义了需要被抽离的模块，其中test属性是比较关键的一个值，他可以是一个字符串，也可以是正则表达式，还可以是函数。如果定义的是字符串，会匹配入口模块名称，会从其他模块中把包含这个模块的抽离出来。name属性是要缓存的分离出来的chunk名称。

复出现了多次，也会成倍地占用内存空间。

为了节省内存，可以将这段共享的子程序存储为一个可执行文件，被多个程序调用时只在内存中生成和使用同一个实例即可，这就是动态链接库思想。Webpack社区也依照这个思想开发了一个插件DLLPlugin，它用来分离我们在程序调用中反复会用到的代码。其实我们上面介绍的CommonsChunkPlugin和optimization.splitChunks和DLLPlugin是一类插件，但是DLLPlugin更加智能，因为配置过程非常复杂，我们单独来介绍。

- 配置动态链接库：首先需要为动态链接库单独创建一个 Webpack 配置文件，这里我们叫做 `webpack.vendor.config.js`。该配置对象需要引入 DLLPlugin，其中的 entry 指定了把哪些模块打包为 vendor。

```
const path    = require('path');
const webpack = require('webpack');
module.exports = {
  entry: {
        //提取的公共文件
      vendor: ['react','lodash',]
  },
  output: {
    path: path.resolve('./dist'),
    filename: 'vendor.js',
    library: '[name]_library'
  },
  plugins: [
    new webpack.DllPlugin({
      path: path.resolve('./dist', '[name]-manifest.json'),
      name: '[name]_library'
    })
  ]
};
```

- 打包动态链接库并生成 vendor 清单：使用该配置文件进行打包。会生成一个 vendor.js 以及一个资源的清单文件manifest.json，在内部每一个模块都会分配一个 ID。
- 将 vendor 连接到项目中：最后在工程的 `webpack.config.js` 中我们需要配置 DllReferencePlugin 来获取刚刚打包出来的模块清单。这相当于工程代码和 vendor 连接的过程。

```
module.exports = {
  plugins: [
    new webpack.DllReferencePlugin({
      // 指定需要用到的 manifest 文件
      manifest: path.resolve(__dirname, 'dist/manifest.json'),
    }),
  ],
}
```

- 最后不要忘了需要在HTML文件当中正确引入我们打包好的vendor.js文件。

```
<script src="vendor.js"></script>
```

这就是整个DLLPlugin打包一个流程，比较复杂，希望大家下来都能够自己亲自实践一下。另外社区最近又有一个新的基于DLLplugin开发的插件**autodll-webpack-plugin**，大家有兴趣的可以

打包文件质量优化

下面我们继续介绍如何优化打包后输出文件的质量，也就是如何使打包出的文件尽可能的小，这样我们在加载文件的时候才能更快。

1.压缩JavaScript代码，在压缩JavaScript代码的时候我们需要先将代码解析成AST语法树，这个过程计算量非常大，我们常用的插件是webpack-uglify-parallel 。通过 webpack-uglify-parallel 我们可以将每个资源的压缩过程交给单独的进程，以此来提升整体的压缩效率。这个插件并不在 Webpack 内部，需要我们单独安装。配置方法也比较简单，如下：

```
const os = require('os');
    const UglifyJsParallelPlugin = require('webpack-uglify-parallel');

    new UglifyJsParallelPlugin({
      //开启多进程
      workers: os.cpus().length,
      mangle: true,
      compressor: {
        //忽略警告
        warnings: false,
        //打开console
        drop_console: true,
        //打开debugger
        drop_debugger: true
       }
    })
```

2.压缩CSS代码，上面我们介绍了压缩JavaScript代码的方法， 同样CSS代码同样也可以被压缩。在WebPack3.x的时候推荐使用ExtractTextPlugin和cssnano，但是WebPack4.x已经淘汰了这2个配置项。WebPack4.x我们用MiniCssExtractPlugin和OptimizeCSSAssetsPlugin，具体配置如下：

```
const OptimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
module.exports = {
  module: {
    rules: [
      {
                test: /\.css$/,
                use: [
          MiniCssExtractPlugin.loader,   // 分离css代码
          'css-loader',
                ],
                        },
    ],
  },
  plugins:[
     new MiniCssExtractPlugin({
            filename: 'static/css/[name].[contenthash:8].css' //提取css存放目录
        }),
        new OptimizeCssAssetsPlugin()  // 使用OptimizeCssAssetsPlugin对CSS进行压缩
  ]
}
```

3.压缩图片，图片在一般项目当中都是最大的静态资源，所以图片的压缩就显得非常重要。图片压缩插件我们常用的是imagemin-webpack-plugin。配置较为简单，如下：

```
  plugins: [
      new ImageminPlugin({
        pngquant: {
          //指定压缩后的图片质量
          quality: '95-100'
        }
      })
        ]
}
```

4.使用Prepack，Prepack是Facebook开源的WebPack插件，Prepack是一个优化JavaScript源代码的工具：可以在编译时而不是运行时完成的计算被消除。Prepack使用等效代码替换JavaScript包的全局代码。我们先来看官网上面的一个例子：

可以看到Prepark处理后的代码非常简洁，这样我们的JavaScript引擎在处理代码的时候，速度也会大大增加，它的配置方法也非常简单，如下：

```
const PrepackWebpackPlugin = require('prepack-webpack-plugin').default;

module.exports = {
  plugins:[
     new PrepackWebpackPlugin()
  ]
}
```

5.最后请大家及时升级你的Webpack版本来获得更优的打包速度，Webpack4.x相比于Webpack3.x做了非常多的优化工作，主要就是性能优化方面的，因此大家下去要及时更新自己的Webpack版本。

关于Webpack具体的更新，大家可以查看这里进行查看，这里我们挑一些比较重要的做介绍：

- Webpack AST可以直接从loader传递给webpack，这样就可以避免额外的解析
- `CommonsChunkPlugin` 被删除 - > `optimization.splitChunks` ， `optimization.runtimeChunk`
- 队列不会两次将同一个作业排入队列
- 默认情况下，使用更快的md4哈希进行散列
- 使用 `for of` 替换 `forEach` ，使用 `Map` 和 `Set` 替换普通的对象字面量，使用includes替换indexOf，通过这些API的替换获得更快的速度。

这里是罗列了一些比较重要的更新，总的来说，Webpack4.x的性能提升还是巨大的，我更新到Webpack4.x之后，打包时间大概缩短了一半左右，因此，推荐使用Webpack3.x要及时更新。

这一节主要介绍了一些构建流程的优化方法，其实还有很多构建流程的优化方法，我们这里只挑了比较典型的案例介绍给大家，的确关于Webpack配置确实比较多，大家可能一下子记不住这么多，这是没有关系的，知识的学习就是不断重现的过程，只要下去多加实践，自然而然就会记住。当然这些方法也不是一成不变的，更多的还是**结合我们的业务**进行相关的优化工作，其实Webpack也是遵循"二八法则"的，我们只要找出我们项目当中关键的2个性能瓶颈，那么就可以提升80%的性能，因此，找到当前业务下的瓶颈是我们最需要去关心的地方。

← 7 聊聊 DNS Prefetch

9 图片加载优化（上） →

精选留言 1

欢迎在这里发表留言，作者筛选后可公开显示

NyanIT

我的react typescript 项目打包分析看到，vender.js这块代码很大，都有1.8Mib了。我是将react和lodash放在cacheGroup那里。这个结果是不是不很合理，该怎么进行下一步优化呢?

0 回复 2019-08-02

BinaryCoding 回复 **NyanIT**

结合可视化插件进行相关优化

回复 2019-08-30 20:38:33

千学不如一看，千看不如一练