写在前面

在第五、六章中，我们完成了原生 js 项目到 webpack5 的模块化框架升级和 ZBestPC 项目进阶升级的学习。在这两章内容中，我们接触到了许多 webpack 工程化应用的高级技巧。接下来，是我们的加餐环节：了解 webpack 的常用优化手段。在开始阅读之前，我希望大家能先思考一个问题：我们进行优化打包的目的是什么？

webpack打包优化方向

- **打包速度**：优化打包速度，主要是提升了我们的开发效率，更快的打包构建过程，将让你保持一颗愉悦的心
- **打包体积**：优化打包体积，主要是提升产品的使用体验，降低服务器资源成本，更快的页面加载，将让产品显得更加“丝滑”，同时也可以让打包更快

webpack打包速度优化

webpack 进行打包速度优化有七种常用手段

1. 优化 loader 搜索范围

对于 loader 来说，影响打包效率首当其冲必属 Babel 了。因为 Babel 会将代码转为字符串生成 AST，然后对 AST 继续进行转变最后再生成新的代码，项目越大，转换代码越多，效率就越低。优化正则匹配、使用 include 和 exclude 指定需要处理的文件,忽略不需要处理的文件

```
rules: [{
   // 优化正则匹配
   test: /\.js$/,
   // 指定需要处理的目录
   include: path.resolve(__dirname, 'src')
   // 理论上只有include就够了，但是某些情况需要排除文件的时候可以用这个，排除不需要处理文件
   // exclude: []
 }]
```

2. 多进程/多线程

受限于 node 是单线程运行的，所以 webpack 在打包的过程中也是单线程的，特别是在执行 loader 的时候，长时间编译的任务很多，这样就会导致等待的情况。我们可以使用一些方法将 loader 的同步执行转换为并行，这样就能充分利用系统资源来提高打包速度了

```
{
    test: /\.js?$/,
    exclude: /node_modules/,
    use: [
    {
      loader: "thread-loader",
      options: {
        workers: 3 // 进程 3 个
      }
    },
      {
        loader: "babel-loader",
        options: {
          presets: ["@babel/preset-env"],
          plugins: ["@babel/plugin-transform-runtime"]
        }
      }
    ]
 },
```

3. 分包

在使用 webpack 进行打包时候，对于依赖的第三方库，比如 vue，vuex 等这些不会修改的依赖，我们可以让它和我们自己编写的代码分开打包，这样做的好处是每次更改我本地代码的文件的时候，webpack 只需要打包我项目本身的文件代码，而不会再去编译第三方库，那么第三方库在第一次打包的时候只打包一次，以后只要我们不升级第三方包的时候，那么 webpack 就不会对这些库去打包，这样可以快速提高打包的速度。因此为了解决这个问题，DllPlugin 和 DllReferencePlugin 插件就产生了。这种方式可以极大的减少打包类库的次数，只有当类库更新版本才需要重新打包，并且也实现了将公共代码抽离成单独文件的优化方案

```
// webpack.dll.conf.js
const path = require('path')
const webpack = require('webpack')
const UglifyJsPlugin = require('uglifyjs-webpack-plugin')
module.exports = {
```

```
    mode: 'production',
    devtool: false,
    entry: {
        vue: [
            'vue',
            'vue-router',
            'iscroll',
            'vuex'
        ],
    },
    output: {
        path: path.join(__dirname, '../dist'),
        filename: 'lib/[name]_[hash:4].dll.js',
        library: '[name]_[hash:4]'
    },
    performance: {
        hints: false,
        maxAssetSize: 300000, //单文件超过300k，命令行告警
        maxEntrypointSize: 300000, //首次加载文件总和超过300k，命令行告警
    },
    optimization: {
        minimizer: [
            new UglifyJsPlugin({
                parallel: true  // 开启多线程并行
            })
        ]
    },
    plugins: [
        new webpack.DllPlugin({
            context: __dirname,
            path: path.join(__dirname, '../dist/lib', '[name]-manifest.json'),
            name: '[name]_[hash:4]'
        })
    ]
}

// webpack.prod.cong.js
plugins: [
  new webpack.DllReferencePlugin({
        context: __dirname,
        manifest: require('../dist/lib/vue-manifest.json')
    }),
]
```

4. 开启缓存

当设置 cache.type: “filesystem” 时，webpack 会在内部以分层方式启用文件系统缓存和内存缓存，将处理结果结存放到内存中，下次打包直接使用缓存结果而不需要重新打包

```
cache: {
    type: "filesystem"
    // cacheDirectory 默认路径是 node_modules/.cache/webpack
    // cacheDirectory: path.resolve(__dirname, '.temp_cache')
 },
```

5. 打包分析工具

显示测量打包过程中各个插件和 loader 每一步所消耗的时间,然后让我们可以有针对的分析项目中耗时的模块对其进行处理。

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

// webpack.prod.config.js

const SpeedMeatureWebpackPlugin = require("speed-measure-webpack-plugin");
const smp = new SpeedMeatureWebpackPlugin();

var webpackConfig = merge(baseWebpackConfig,{})
--> 修改为下面格式
var webpackConfig = {...}

module.exports = webpackConfig
--> 修改为下面格式
module.exports = smp.wrap(merge(baseWebpackConfig, webpackConfig));
```

6. ignorePlugin

这是 webpack 内置插件, 它的作用是忽略第三方包指定目录，让这些指定目录不要被打包进去，防止在 import 或 require 调用时，生成以下正则表达式匹配的模块

- requestRegExp 匹配（ test ）资源请求路径的正则表达式。
- contextRegExp（ 可选 ）匹配（ test ）资源上下文（ 目录 ）的正则表达式。

```
new webpack.IgnorePlugin({
  resourceRegExp: /^\.\/test$/,
  contextRegExp: /test$/,
})
```

7. 优化文件路径

alias：省下搜索文件的时间，让 webpack 更快找到路径
mainFiles：解析目录时要使用的文件名
extensions：指定需要检查的扩展名，配置之后可以不用在 require 或是 import 的时候加文件扩展名,会依次尝试添加扩展名进行匹配

```
resolve: {
    extensions: ['.js', '.vue'],
   mainFiles: ['index'],
    alias: {
      '@': resolve('src'),
    }
}
```

webpack打包体积优化

webpack打包体积优化有11种常用优化手段

1. 构建体积分析

npm run build 构建，会默认打开： http://127.0.0.1:8888/，可以看到各个包的体积,分析项目各模块的大小，可以按需优化。

```
npm install webpack-bundle-analyzer -D
const BundleAnalyzerPlugin = require("webpack-bundle-analyzer").BundleAnalyzerPlugin;

plugins:[
 new BundleAnalyzerPlugin()
]
```

2. 项目图片资源优化压缩处理

对打包后的图片进行压缩和优化，降低图片分辨率，压缩图片体积等

```
npm install image-webpack-loader -D

// webpack.base.conf.js

{
        test: /\.(gif|png|jpe?g|svg|webp)$/i,
        type: "asset/resource",
        parser: {
          dataUrlCondition: {
            maxSize: 8 * 1024
          }
        },
        generator: {
          filename: "images/[name].[hash:6][ext]"
        },
        use: [
          {
            loader: "image-webpack-loader",
            options: {
              mozjpeg: {
                progressive: true,
                quality: 65
              },
              optipng: {
                enabled: false
              },
              pngquant: {
                quality: [0.5, 0.65],
                speed: 4
              },
              gifsicle: {
                interlaced: false
              },
              webp: {
```

```
              quality: 75
            }
          }
        }
      ]
  }
```

3. 删除无用的 css 样式

有时候一些项目中可能会存在一些 css 样式被迭代废弃，需要将其删除，可以使用 purgecss-webpack-plugin 插件，该插件可以去除未使用的 css。

```
npm install purgecss-webpack-plugin glod -D

// webpack.prod.conf.js

const PurgeCSSPlugin = require("purgecss-webpack-plugin");
const glob = require('glob')

const PATHS = {
  src: path.join(__dirname, 'src')
}

// plugins
new PurgeCSSPlugin({
      paths: glob.sync(`${PATHS.src}/**/*`, { nodir: true }),
      safelist: ["body"]
 }),
```

4. 代码压缩

对 js 文件进行压缩，从而减小 js 文件的体积，还可以压缩 html 、css 代码。

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

optimization: {
    minimize: true, //代码压缩
    usedExports: true, // treeshaking
    minimizer: [
      new TerserPlugin({
        terserOptions: {
          ecma: undefined,
          parse: {},
          compress: {},
          mangle: true, // Note `mangle.properties` is `false` by default.
          module: false,
          // Deprecated
          output: null,
          format: null,
          toplevel: false,
          nameCache: null,
          ie8: false,
          keep_classnames: undefined,
          keep_fnames: false,
          safari10: false
        }
      })
    ],
    splitChunks: {
      cacheGroups: {
        commons: {
          name: "commons",
          chunks: "initial",
          minChunks: 2
        }
      }
    }
  },
```

5. 开启 Scope Hoisting

Scope Hoisting 又译作“作用域提升”。只需在配置文件中添加一个新的插件，就可以让 webpack 打包出来的代码文件更小、运行的更快, Scope Hoisting 会分析出模块之间的依赖关系，尽可能的把打包出来的模块合并到一个函数中去，然后适当地重命名一些变量以防止命名冲突。

```
new webpack.optimize.ModuleConcatenationPlugin();
```

6. 提取公共代码

将项目中的公共模块提取出来，可以减少代码的冗余度，提高代码的运行效率和页面的加载速度。

```
new webpack.optimize.CommonsChunkPlugin(options);
```

7. 代码分离

代码分离能够将工程代码分离到各个文件中，然后按需加载或并行加载这些文件，也用于获取更小的 bundle，以及控制资源加载优先级,在配置文件中配置多入口，输出多个 chunk。

```
//多入口配置 最终输出两个chunk
module.exports = {
  entry: {
    index: 'index.js',
    login: 'login.js'
  },
  output: {
    //对于多入口配置需要指定[name]否则会出现重名问题
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

8. Tree-shaking

tree shaking 是一个术语，通常用于描述移除 JavaScript 上下文中的未引用代码（ dead-code ）。它依赖于 ES2015 模块语法的 静态结构 特性，例如 import 和 export。

9. CDN 加速

CDN 的全称是 Content Delivery Network，即内容分发网络。CDN 是构建在网络之上的内容分发网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。CDN 的关键技术主要有内容存储和分发技术。在项目中以 CDN 的方式加载资源，项目中不需要对资源进行打包，大大减少打包后的文件体积

10. 生产环境关闭 sourceMap

sourceMap 本质上是一种映射关系，打包出来的 js 文件中的代码可以映射到代码文件的具体位置,这种映射关系会帮助我们直接找到在源代码中的错误。但这样会使项目打包速度减慢，项目体积变大，可以在生产环境关闭 sourceMap

11. 按需加载

在开发项目的时候，项目中都会存在十几甚至更多的路由页面。如果我们将这些页面全部打包进一个文件的话，虽然将多个请求合并了，但是同样也加载了很多并不需要的代码，耗费了更长的时间。那么为了页面能更快地呈现给用户，我们肯定是希望页面能加载的文件体积越小越好，这时候我们就可以使用按需加载，将每个路由页面单独打包为一个文件。以下是常见的按需加载的场景

- 路由组件按需加载
- 按需加载需引入第三方组件
- 对于一些插件，如果只是在个别组件中用的到，也可以不要在 main.js 里面引入，而是在组件中按需引入