webpack性能优化&webpack打包原理分析

多页面打包通用方案

打包公共库

自定义plugin

Webpack:4.43.0

Webpack-cli:3.3.12

Webpack-dev-server:3.11.0

优化的目的

- 优化开发体验
- 优化输出质量

缩小文件搜索范围

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。接下来看看如何优化loader。

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。

优化loader查找范围

- test include exclude三个配置项来缩小loader的处理范围
- 推荐include

```
//string
include: path.resolve(__dirname, "./src"),


//array
include: [
  path.resolve(__dirname, 'app/styles'),
  path.resolve(__dirname, 'vendor/styles')
]
```

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。

优化resolve.modules配置

resolve.modules用于配置webpack去哪些目录下寻找第三方模块，默认是['node_modules']

寻找第三方模块，默认是在当前项目目录下的node_modules里面去找，如果没有找到，就会去上一级目录../node_modules找，再没有会去../../node_modules中找，以此类推，和Node.js的模块寻找机制很类似。

如果我们的第三方模块都安装在了项目根目录下，就可以直接指明这个路径。

```
module.exports={
  resolve:{
    modules: [path.resolve(__dirname, "./node_modules")]
  }
}
```

优化resolve.alias配置

resolve.alias配置通过别名来将原导入路径映射成一个新的导入路径

拿react为例，我们引入的react库，一般存在两套代码

- cjs
  - 采用commonJS规范的模块化代码
- umd
  - 已经打包好的完整代码，没有采用模块化，可以直接执行

> 默认情况下，webpack会从入口文件./node_modules/bin/react/index开始递归解析和处理依赖的文件。我们可以直接指定文件，避免这处的耗时。

```
alias: {
  "@": path.join(__dirname, "./pages"),
  react: path.resolve(
  __dirname,
  "./node_modules/react/umd/react.production.min.js"
  ),
  "react-dom": path.resolve(
  __dirname,
  "./node_modules/react-dom/umd/react-dom.production.min.js"
  )
}
```

```
resolve: {
    alias: {
      "@assets": path.resolve(__dirname, "../src/images/"),
    },
  },

  //html-css中使用
  .sprite3 {
  background: url("~@assets/s3.png");
}
```

优化resolve.extensions配置

resolve.extensions在导入语句没带文件后缀时，webpack会自动带上后缀后，去尝试查找文件是否存在。

默认值：

```
extensions:['.js','.json','.jsx','.ts']
```

- 后缀尝试列表尽量的小
- `导入语句尽量的带上后缀。

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。

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：

```
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)
                ]
            }
        ]
    }
};
```

缓存cache相关

提升构建速度的另外一个大杀器是使用缓存！

- babel-loader

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
        },
    }
];
```

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
            })
        ]
    }
};
```

使用externals优化cdn静态资源

//公司有cdn

//静态资源有部署到cdn 有链接了

// 我想使用cdn！！！！！！！！

我的bundle文件里，就不用打包进去这个依赖了，体积会小

我们可以将一些JS文件存储在 `CDN` 上(减少 `Webpack`打包出来的 `js` 体积)，在 `index.html` 中通过 标签引入，如:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <div id="root">root</div>
    <script src="http://libs.baidu.com/jquery/2.0.0/jquery.min.js"></script>
</body>
</html>
```

我们希望在使用时，仍然可以通过 `import` 的方式去引用(如 `import $ from 'jquery'`)，并且希望 `webpack` 不会对其进行打包，此时就可以配置 `externals`。

```
//webpack.config.js
module.exports = {
    //...
    externals: {
        //jquery通过script引入之后，全局中即有了 jQuery 变量
        'jquery': 'jQuery'
    }
}
```

使用静态资源路径publicPath(CDN)

CDN通过将资源部署到世界各地，使得用户可以就近访问资源，加快访问速度。要接入CDN，需要把网页的静态资源上传到CDN服务上，在访问这些资源时，使用CDN服务提供的URL。

```
##webpack.config.js
output:{
 publicPath: '//cdnURL.com', //指定存放JS文件的CDN地址
}
```

- 咱们公司得有cdn服务器地址
- 确保静态资源文件的上传与否

development vs Production模式区分打包

```
npm install webpack-merge -D
```

案例

```
const merge = require("webpack-merge")
const commonConfig =  require("./webpack.common.js")
const devConfig = {
  ...
}

module.exports = merge(commonConfig,devConfig)

//package.js
"scripts":{
  "dev":"webpack-dev-server --config ./build/webpack.dev.js",
  "build":"webpack --config ./build/webpack.prod.js"
}
```

基于环境变量区分

- 借助cross-env

```
npm i cross-env -D
```

package里面配置命令脚本，传入参数

```
##package.json

"test": "cross-env NODE_ENV=test webpack --config ./webpack.config.test.js",
```

在webpack.config.js里拿到参数

```
process.env.NODE_ENV
```

```
//外部传入的全局变量
module.exports = (env)=>{
  if(env && env.production){
    return merge(commonConfig,prodConfig)
  }else{
    return merge(commonConfig,devConfig)
  }
}


//外部传入变量
scripts:" --env.production"
```

css压缩

- 借助 optimize-css-assets-webpack-plugin
- 借助cssnano

  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：

```
##安装

npm install cssnano -D
npm i optimize-css-assets-webpack-plugin -D

const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");

new OptimizeCSSAssetsPlugin({
  cssProcessor: require("cssnano"), // 这里制定了引擎，不指定默认也是 cssnano
  cssProcessorOptions: {
    discardComments: { removeAll: 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。

压缩HTML

- 借助html-webpack-plugin

```
new htmlWebpackPlugin({
      title: "京东商城",
      template: "./index.html",
      filename: "index.html",
      minify: {
        // 压缩HTML文件
        removeComments: true, // 移除HTML中的注释
        collapseWhitespace: true, // 删除空白符与换行符
        minifyCSS: true // 压缩内联css
      }
    }),
```

压缩JS

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。

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

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

在实际开发中，我们可以通过移除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
    )],
  },
};
```

tree Shaking：擦除无用的JS,CSS

webpack2.x开始支持 tree shaking概念，顾名思义，"摇树"，清除无用 css,js(Dead Code)

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）
- Js tree shaking只支持ES module的引入方式！！！！，

Css tree shaking

```
npm install glob-all purify-css purifycss-webpack -D

const PurifyCSS = require('purifycss-webpack')
const glob = require('glob-all')
```

```
plugins:[
    // 清除无用 css
    new PurifyCSS({
      paths: glob.sync([
        // 要做 CSS Tree Shaking 的路径文件
        path.resolve(__dirname, './src/*.html'), // 请注意，我们同样需要对 html 文件进行 tree shaking
        path.resolve(__dirname, './src/*.js')
      ])
    })
]
```

JS tree shaking

只支持import方式引入，不支持commonjs的方式引入

案例：

```
//expo.js
export const add = (a, b) => {
  return a + b;
};

export const minus = (a, b) => {
  return a - b;
};


//index.js
import { add } from "./expo";
add(1, 2);
```

```
//webpack.config.js
optimization: {
   usedExports: true // 哪些导出的模块被使用了，再做打包
}
```

只要mode是production就会生效，develpoment的tree shaking是不生效的，因为webpack为了方便你的调试

可以查看打包后的代码注释以辨别是否生效。

生产模式不需要配置，默认开启

sideEffects 处理副作用

```
//package.json
"sideEffects":false  //正常对所有模块进行tree shaking  , 仅生产模式有效，需要配合usedExports

或者 在数组里面排除不需要tree shaking的模块

"sideEffects":['*.css','@babel/polyfill']
```

代码分割 code Splitting

单页面应用spa：

打包完后，所有页面只生成了一个bundle.js

- 代码体积变大，不利于下载
- 没有合理利用浏览器资源

多页面应用mpa:

如果多个页面引入了一些公共模块，那么可以把这些公共的模块抽离出来，单独打包。公共代码只需要下载一次就缓存起来了，避免了重复下载。

```
import _ from "lodash";

console.log(_.join(['a','b','c','****']))


假如我们引入一个第三方的工具库，体积为1mb，而我们的业务逻辑代码也有1mb，那么打包出来的体积大小会在2mb

导致问题：
  体积大，加载时间长
  业务逻辑会变化，第三方工具库不会，所以业务逻辑一变更，第三方工具库也要跟着变。
```

其实code Splitting概念 与 webpack并没有直接的关系，只不过webpack中提供了一种更加方便的方法供我们实现代码分割

基于https://webpack.js.org/plugins/split-chunks-plugin/

```
 optimization: {
    splitChunks: {
      chunks: "all", // 所有的 chunks 代码公共的部分分离出来成为一个单独的文件
    },
  },
```

```
optimization: {
    splitChunks: {
      chunks: 'async',//对同步 initial，异步 async，所有的模块有效 all
      minSize: 30000,//最小尺寸，当模块大于30kb
      maxSize: 0,//对模块进行二次分割时使用，不推荐使用
      minChunks: 1,//打包生成的chunk文件最少有几个chunk引用了这个模块
      maxAsyncRequests: 5,//最大异步请求数，默认5
      maxInitialRequests: 3,//最大初始化请求书，入口文件同步请求，默认3
      automaticNameDelimiter: '-',//打包分割符号
      name: true,//打包后的名称，除了布尔值，还可以接收一个函数function
      cacheGroups: {//缓存组
        vendors: {
          test: /[\\/]node_modules[\\/]/,
          name:"vendor", // 要缓存的 分隔出来的 chunk 名称
          priority: -10//缓存组优先级 数字越大，优先级越高
        },
        other:{
          chunks: "initial", // 必须三选一： "initial" | "all" | "async"(默认就是async)
          test: /react|lodash/, // 正则规则验证，如果符合就提取 chunk,
          name:"other",
          minSize: 30000,
          minChunks: 1,
        },
        default: {
          minChunks: 2,
          priority: -20,
          reuseExistingChunk: true//可设置是否重用该chunk
        }
      }
    }
  }
```

使用下面配置即可：

```
optimization:{
  //帮我们自动做代码分割
  splitChunks:{
    chunks:"all",//默认是支持异步，我们使用all
  }
}
```

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：

-

```
// hello.js
export default 'Hello, Webpack';
// index.js
import str from './hello.js';
console.log(str);
```

打包后，`hello.js`的内容和`index.js`会分开

通过配置 optimization.concatenateModules=true`：开启 Scope Hoisting

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

我们发现hello.js内容和index.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。

使用工具量化

- speed-measure-webpack-plugin:可以测量各个插件和`loader`所花费的时间

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

//webpack.config.js
const SpeedMeasurePlugin = require("speed-measure-webpack-plugin");
const smp = new SpeedMeasurePlugin();

const config = {
    //...webpack配置
}

module.exports = smp.wrap(config);
```

- webpack-bundle-analyzer:分析webpack打包后的模块依赖关系：

```
npm install webpack-bundle-analyzer -D

const BundleAnalyzerPlugin = require('webpack-bundle-
analyzer').BundleAnalyzerPlugin;
module.exports = merge(baseWebpackConfig, {
    //....
    plugins: [
        //...
        new BundleAnalyzerPlugin(),
    ]
})
```

多入口打包配置通用方案

####

```
entry:{
  index:"./src/index",
  list:"./src/list",
  detail:"./src/detail"
}

new htmlWebpackPlugins({
  title: "index.html",
  template: path.join(__dirname, "./src/index/index.html"),
  filename:"index.html",
  chunks:[index]
})
```

1.目录结构调整

- src
  - index
    - index.js
    - index.html
  - list
    - index.js
    - index.html
  - detail
    - index.js
    - index.html
- 2.使用 glob.sync 第三方库来匹配路径

```
npm i glob -D

const glob = require("glob")
```

```
//MPA多页面打包通用方案

const setMPA = () => {
  const entry = {};
  const htmlWebpackPlugins = [];


  return {
    entry,
    htmlWebpackPlugins
  };
};

const { entry, htmlWebpackPlugins } = setMPA();
```

```
const setMPA = () => {
  const entry = {};
  const htmlWebpackPlugins = [];
```

```
  const entryFiles = glob.sync(path.join(__dirname, "./src/*/index.js"));

  entryFiles.map((item, index) => {
    const entryFile = entryFiles[index];
    const match = entryFile.match(/src\/(.*)\/index\.js$/);
    const pageName = match && match[1];
    entry[pageName] = entryFile;
    htmlWebpackPlugins.push(
      new htmlWebpackPlugin({
        title: pageName,
        template: path.join(__dirname, `src/${pageName}/index.html`),
        filename: `${pageName}.html`,
        chunks: [pageName],
        inject: true
      })
    );
  });
  return {
    entry,
    htmlWebpackPlugins
  };
};

const { entry, htmlWebpackPlugins } = setMPA();



module.exports = {
  entry,
  output:{
    path: path.resolve(__dirname, "./dist"),
    filename: "[name].js"
  }
  plugins: [
    // ...
    ...htmlWebpackPlugins//展开数组
  ]
}
```

webpack打包原理分析

接收webpack配置 进行读取

入口：从哪个文件开始分析

哪些是依赖模块

这些依赖模块的位置 也就是路径信息

内容

对内容处理，处理成浏览器正确解析的

递归处理其他依赖模块

生成chunk 代码片段

补齐函数 生成bundle文件的内容 -》 启动器函数

出口：生成资源文件的名称和位置

```
const fs = require('fs');
const path = require('path');
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const babel = require('@babel/core');

const moduleAnalyser = (filename) => {
  const content = fs.readFileSync(filename, 'utf-8');
  const ast = parser.parse(content, {
    sourceType: 'module'
  });
  const dependencies = {};
  traverse(ast, {
    ImportDeclaration({ node }) {
      const dirname = path.dirname(filename);
      const newFile = './' + path.join(dirname, node.source.value);
      dependencies[node.source.value] = newFile;
    }
  });
  const { code } = babel.transformFromAst(ast, null, {
    presets: ["@babel/preset-env"]
  });
  return {
    filename,
    dependencies,
    code
  }
}
```

```
const makeDependenciesGraph = (entry) => {
  const entryModule = moduleAnalyser(entry);
  const graphArray = [ entryModule ];
  for(let i = 0; i < graphArray.length; i++) {
    const item = graphArray[i];
    const { dependencies } = item;
    if(dependencies) {
      for(let j in dependencies) {
        graphArray.push(
          moduleAnalyser(dependencies[j])
        );
      }
    }
  }
  const graph = {};
  graphArray.forEach(item => {
    graph[item.filename] = {
      dependencies: item.dependencies,
      code: item.code
    }
  });
  return graph;
}

const generateCode = (entry) => {
  const graph = JSON.stringify(makeDependenciesGraph(entry));
  return `
    (function(graph){
      function require(module) {
        function localRequire(relativePath) {
          return require(graph[module].dependencies[relativePath]);
        }
        var exports = {};
        (function(require, exports, code){
          eval(code)
        })(localRequire, exports, graph[module].code);
        return exports;
      };
      require('${entry}')
    })(${graph});
  `;
}

const code = generateCode('./src/index.js');
console.log(code);
```