Webpack优化手段

webpack优化很有必要

使用**webpack**打包躲不开的就是 `webpack优化` 这个话题，无论是面试还是实际开发，优化都是非常重要的事情，毕竟**提升用户体验是我们前端工程师的职责**

构建时间优化

首先就是构建时间的优化了

thread-loader

多进程打包，可以大大提高构建的速度，使用方法是将 `thread-loader` 放在比较费时间的loader之前，比如 `babel-loader`

> 由于启动项目和打包项目都需要加速，所以配置在 `webpack.base.js`

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

```
// webpack.base.js

{
        test: /\.js$/,
        use: [
          'thread-loader',
          'babel-loader'
        ],
      }
}
```

cache-loader

缓存资源，提高二次构建的速度，使用方法是将 `cache-loader` 放在比较费时间的loader之前，比如 `babel-loader`

> 由于启动项目和打包项目都需要加速，所以配置在 `webpack.base.js`

```
npm i cache-loader -D
```

```
// webpack.base.js

{
        test: /\.js$/,
        use: [
          'cache-loader',
          'thread-loader',
          'babel-loader'
        ],
},
```

开启热更新

比如你修改了项目中某一个文件，会导致整个项目刷新，这非常耗时间。如果只刷新修改的这个模块，其他保持原状，那将大大提高修改代码的重新构建时间

> 只用于开发中，所以配置在 `webpack.dev.js`

```
// webpack.dev.js

//引入webpack
const webpack = require('webpack');
//使用webpack提供的热更新插件
   plugins: [
   new webpack.HotModuleReplacementPlugin()
    ],
    //最后需要在我们的devserver中配置
     devServer: {
+     hot: true
    },
```

exclude & include

- `exclude`：不需要处理的文件
- `include`：需要处理的文件

合理设置这两个属性，可以大大提高构建速度

> 在 `webpack.base.js` 中配置

```
// webpack.base.js
```

```
      {
        test: /\.js$/,
        //使用include来指定编译文件夹
        include: path.resolve(__dirname, '../src'),
        //使用exclude排除指定文件夹
        exclude: /node_modules/,
        use: [
          'babel-loader'
        ]
      },
```

构建区分环境

区分环境去构建是非常重要的，我们要明确知道，开发环境时我们需要哪些配置，不需要哪些配置；而最终打包生产环境时又需要哪些配置，不需要哪些配置：

- `开发环境`：去除代码压缩、gzip、体积分析等优化的配置，大大提高构建速度
- `生产环境`：需要代码压缩、gzip、体积分析等优化的配置，大大降低最终项目打包体积

> 上篇文章已经带大家进行了环境区分

提升webpack版本

webpack版本越新，打包的效果肯定更好

打包体积优化

> 主要是打包后项目整体体积的优化，有利于项目上线后的页面加载速度提升
>
> 本项目已经是webpack最新版本

CSS代码压缩

CSS代码压缩使用`css-minimizer-webpack-plugin`，效果包括压缩、去重

> 代码的压缩比较耗时间，所以只用在打包项目时，所以只需要在`webpack.prod.js`中配置

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

```
// webpack.prod.js

const CssMinimizerPlugin = require('css-minimizer-webpack-plugin')

  optimization: {
```

```
    minimizer: [
      new CssMinimizerPlugin(), // 去重压缩css
    ],
  }
```

JS代码压缩

JS代码压缩使用 `terser-webpack-plugin` ，实现打包后JS代码的压缩

> 代码的压缩比较耗时间，所以只用在打包项目时，所以只需要在 `webpack.prod.js` 中配置

```
npm i terser-webpack-plugin -D
```

```
// webpack.prod.js

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

  optimization: {
    minimizer: [
      new CssMinimizerPlugin(), // 去重压缩css
      new TerserPlugin({ // 压缩JS代码
        terserOptions: {
          compress: {
            drop_console: true, // 去除console
          },
        },
      }), // 压缩JavaScript
    ],
  }
```

tree-shaking

`tree-shaking` 简单说作用就是：只打包用到的代码，没用到的代码不打包，而 `webpack5` 默认开启 `tree-shaking` ，当打包的 `mode` 为 `production` 时，自动开启 `tree-shaking` 进行优化

```
module.exports = {
  mode: 'production'
}
```

source-map类型

`source-map` 的作用是：方便你报错的时候能定位到错误代码的位置。它的体积不容小觑，所以对于不同环境设置不同的类型是很有必要的。

- **开发环境**

开发环境的时候我们需要能精准定位错误代码的位置

```
// webpack.dev.js

module.exports = {
  mode: 'development',
  devtool: 'eval-cheap-module-source-map'
}
```

- **生产环境**

生产环境，我们想开启 `source-map` ，但是又不想体积太大，那么可以换一种类型

```
// webpack.prod.js

module.exports = {
  mode: 'production',
  devtool: 'nosources-source-map'
}
```

打包体积分析

使用 `webpack-bundle-analyzer` 可以审查打包后的体积分布，进而进行相应的体积优化

> 只需要打包时看体积，所以只需在 `webpack.prod.js` 中配置

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

```
// webpack.prod.js

const {
  BundleAnalyzerPlugin
} = require('webpack-bundle-analyzer')

  plugins: [
    new BundleAnalyzerPlugin(),
]
```

用户体验优化

模块懒加载

如果不进行 `模块懒加载` 的话，最后整个项目代码都会被打包到一个js文件里，单个js文件体积非常大，那么当用户网页请求的时候，首屏加载时间会比较长，使用 `模块懒加载` 之后，大js文件会分成多个小js文件，网页加载时会按需加载，大大提升首屏加载速度

```
// src/router/index.js

const routes = [
  {
    path: '/login',
    name: 'login',
    component: login
  },
  {
    path: '/home',
    name: 'home',
    // 懒加载
    component: () => import('../views/home/home.vue'),
  },
]
```

Gzip

开启Gzip后，大大提高用户的页面加载速度，因为gzip的体积比原文件小很多，当然需要后端的配合，使用 `compression-webpack-plugin`

> 只需要打包时优化体积，所以只需在 `webpack.prod.js` 中配置

```
npm i compression-webpack-plugin -D
```

```
// webpack.prod.js

const CompressionPlugin = require('compression-webpack-plugin')

  plugins: [
    // 之前的代码...

```

```
    // gzip
    new CompressionPlugin({
      algorithm: 'gzip',
      threshold: 10240,
      minRatio: 0.8
    })
  ]
```

小图片转base64

对于一些小图片，可以转base64，这样可以减少用户的http网络请求次数，提高用户的体验。
`webpack5` 中 `url-loader` 已被废弃，改用 `asset-module`

> 在 `webpack.base.js` 中配置

```
// webpack.base.js

{
   test: /\.(png|jpe?g|gif|svg|webp)$/,
   type: 'asset',
   parser: {
     // 转base64的条件
     dataUrlCondition: {
        maxSize: 25 * 1024, // 25kb
     }
   },
   generator: {
     // 打包到 image 文件下
    filename: 'images/[contenthash][ext][query]',
   },
},
```

合理配置hash

我们要保证，改过的文件需要更新hash值，而没改过的文件依然保持原本的hash值，这样才能保证在上线后，浏览器访问时没有改变的文件会命中缓存，从而达到性能优化的目的

> 在 `webpack.base.js` 中配置

```
// webpack.base.js

  output: {
    path: path.resolve(__dirname, '../dist'),
    // 给js文件加上 contenthash
```

```
    filename: 'js/chunk-[contenthash].js',
    clean: true,
  },
```