

Webpack 性能优化

在这一章节中，我不会浪费篇幅给大家讲如何写配置文件。如果你想学习这方面的内容，那么完全可以去官网学习。在这部分的内容中，我们会聚焦于以下两个知识点，并且每一个知识点都属于高频考点：

- 有哪些方式可以减少 Webpack 的打包时间
- 有哪些方式可以让 Webpack 打出来的包更小

减少 Webpack 打包时间

优化 Loader

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。当然了，我们是有办法优化的。

首先我们可以优化 Loader 的文件搜索范围

```
module.exports = {
  module: {
    rules: [
      {
        // js 文件才使用 babel
        test: /\.js$/,
        loader: 'babel-loader',
        // 只在 src 文件夹下查找
        include: [resolve('src')],
        // 不会去查找的路径
        exclude: /node_modules/
      }
    ]
  }
}
```

对于 Babel 来说，我们肯定是希望只作用在 JS 代码上的，然后 `node_modules` 中使用的代码都是编译过的，所以我们也完全没有必要再去处理一遍。

当然这样做还不够，我们还可以将 Babel 编译过的文件缓存起来，下次只需要编译更改过的代码文件即可，这样可以大幅度加快打包时间

```
loader: 'babel-loader?cacheDirectory=true'
```

HappyPack

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。

HappyPack 可以将 Loader 的同步执行转换为并行的，这样就能充分利用系统资源来加快打包效率了

```
module: {
  loaders: [
    {
      test: /\.js$/,
      include: [resolve('src')],
      exclude: /node_modules/,
      // id 后面的内容对应下面
      loader: 'happypack/loader?id=happybabel'
    }
  ]
},
plugins: [
  new HappyPack({
    id: 'happybabel',
    loaders: ['babel-loader?cacheDirectory'],
    // 开启 4 个线程
    threads: 4
  })
]
```

DllPlugin

DllPlugin 可以将特定的类库提前打包然后引入。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有需要重新打包，并且也实现了将公共代码抽离成单独文件的优化方案。

接下来我们就来学习如何使用 DllPlugin

```
// 单独配置在一个文件中
// webpack.dll.conf.js
const path = require('path')
const webpack = require('webpack')
module.exports = {
  entry: {
    // 想统一打包的类库
    vendor: ['react']
  },
  output: {
    path: path.join(__dirname, 'dist'),
    filename: '[name].dll.js',
    library: '[name]-[hash]'
  },
  plugins: [
    new webpack.DllPlugin({
      // name 必须和 output.library 一致
      name: '[name]-[hash]',
      // 该属性需要与 DllReferencePlugin 中一致
      context: __dirname,
      path: path.join(__dirname, 'dist', '[name]-manifest.json')
    })
  ]
}
```

然后我们需要执行这个配置文件生成依赖文件，接下来我们需要使用 DllReferencePlugin 将依赖文件引入项目中

```
// webpack.conf.js
module.exports = {
  // ...省略其他配置
  plugins: [
    new webpack.DllReferencePlugin({
      context: __dirname,
      // manifest 就是之前打包出来的 json 文件
      manifest: require('./dist/vendor-
manifest.json'),
    })
  ]
}
```

代码压缩

在 Webpack3 中，我们一般使用 UglifyJS 来压缩代码，但是这个单线程运行的，为了加快效率，我们可以使用 webpack-parallel-uglify-plugin 来并行运行 UglifyJS，从而提高效率。

在 Webpack4 中，我们就不需要以上这些操作了，只需要将 mode 设置为 production 就可以默认开启以上功能。代码压缩也是我们必做的性能优化方案，当然我们不止可以压缩 JS 代码，还可以压缩 HTML、CSS 代码，并且在压缩 JS 代码的过程中，我们还可以通过配置实现比如删除 console.log 这类代码的功能。

一些小的优化点

我们还可以通过一些小的优化点来加快打包速度

- resolve.extensions: 用来表明文件后缀列表，默认查找顺序是 ['.js', '.json']，如果你的导入文件没有添加后缀就会按照这个顺序查找文件。我们应该尽可能减少后缀列表

长度，然后将出现频率高的后缀排在前面

- `resolve.alias`: 可以通过别名的方式来映射一个路径，能让 Webpack 更快找到路径
- `module.noParse`: 如果你确定一个文件下没有其他依赖，就可以使用该属性让 Webpack 不扫描该文件，这种方式对于大型类库很有帮助

减少 Webpack 打包后的文件体积

注意：该内容也属于性能优化领域。

按需加载

想必大家在开发 SPA 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 JS 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。那么为了首页能更快地呈现给用户，我们肯定是希望首页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**。当然不仅仅路由可以按需加载，对于 `lodash` 这种大型类库同样可以使用这个功能。

按需加载的代码实现这里就不详细展开了，因为鉴于用的框架不同，实现起来都是不一样的。当然了，虽然他们的用法可能不同，但是底层的机制都是一样的。都是当使用的时候再去下载对应文件，返回一个 `Promise`，当 `Promise` 成功以后去执行回调。

Scope Hoisting

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。

比如我们希望打包两个文件

```
// test.js
export const a = 1
// index.js
import { a } from './test.js'
```

对于这种情况，我们打包出来的代码会类似这样

```
[
  /* 0 */
  function (module, exports, require) {
    //...
  },
  /* 1 */
  function (module, exports, require) {
    //...
  }
]
```

但是如果我们使用 Scope Hoisting 的话，代码就会尽可能的合并到一个函数中去，也就变成了这样的类似代码

```
[
  /* 0 */
  function (module, exports, require) {
    //...
  }
]
```

这样的打包方式生成的代码明显比之前的少多了。如果在 Webpack4 中你希望开启这个功能，只需要启用 `optimization.concatenateModules` 就可以了。

```
module.exports = {  
  optimization: {  
    concatenateModules: true  
  }  
}
```

Tree Shaking

Tree Shaking 可以实现删除项目中未被引用的代码，比如

```
// test.js  
export const a = 1  
export const b = 2  
// index.js  
import { a } from './test.js'
```

对于以上情况，test 文件中的变量 b 如果没有在项目中使用到的话，就不会被打包到文件中。

如果你使用 Webpack 4 的话，开启生产环境就会自动启动这个优化功能。

小结

在这一章节中，我们学习了如何使用 Webpack 去进行性能优化以及如何减少打包时间。

Webpack 的版本更新很快，各个版本之间实现优化的方式可能都会有区别，所以我没有使用过多的代码去展示如何实现一个功能。这一章节的重点是学习到我们可以通过什么方式去优化，具体的代码实现可以查找具体版本对应的代码即可。