打包优化

打包速度更快，输出资源更小

HappyPack

多线程提升webpack打包速度

工作原理

loader 预处理很耗时

- 配置中获取打包入口
- 匹配loader规则，并对入口模块进行转译
- 对转译后的模块进行依赖查找
- 对新找到的模块重复 2，3，知道没有新的依赖

2-4是递归，但是Webpack是单线程，只能串行执行
如果多个依赖很影响速度
HappyPack则在此开启多个线程，去转译，充分利用本地计算资源提升打包速度

单个loader优化

使用Webpack提供的loader替换原有loader，把原loader通过HappyPack插件传进去

```
// 初始Webpack配置（使用HappyPack前）
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        options: {
          presets: ['react'],
        },
      }
    ],
  },
};
```

```
// 使用HappyPack的配置
const HappyPack = require('happypack');
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'happypack/loader',
      }
    ],
  },
  plugins: [
    new HappyPack({
      loaders: [
        {
          loader: 'babel-loader',
          options: {
            presets: ['react'],
          },
        }
      ],
    })
  ],
};
```

多个loader的优化

需要为每个loader配置一个id

```
const HappyPack = require('happypack');
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'happypack/loader?id=js',
      },
      {
        test: /\.ts$/,
        exclude: /node_modules/,
        loader: 'happypack/loader?id=ts',
      }
    ],
  },
  plugins: [
    new HappyPack({
      id: 'js',
      loaders: [{
        loader: 'babel-loader',
        options: {}, // babel options
      }],
    }),
    new HappyPack({
      id: 'ts',
      loaders: [{
        loader: 'ts-loader',
        options: {}, // ts options


      }],
    })
  ]
};
```

可以配置线程数，是否debug等

减少打包作用域

exclude和include

配置loader时把node_modules目录排除
exclude 优先级更高

noParse

有些库不希望Webpack去解析。可以使用noParse进行忽略

仍会被打包进去，但是不会做解析

IgnorePlugin

完全排除这些模块，也不会被打包进资源

Cache

有些loader有cache配置项，打包完后会保存一份缓存，下次编译检测是否变化，没变化就直接用缓存

暂时没法检测缓存是否过期，未来会加

动态链接库与DllPlugin

动态链接库 早期 windows系统内存空间小出现的内存优化方法

一段相同的子程序被多个程序调用，为了减小内存消耗，将子程序存储为可执行文件

DllPlugin 对第三方模块不常变化模块预编译和打包，项目实际构建则之间取用

与 代码分片类似，但是又不同
代码分配 按照一定规则提取模块
DllPlugin 完全将vendor拆出来，并独立打包；更胜一筹

配置

webpack.vendor.config.js

```
// webpack.vendor.config.js
const path = require('path');
const webpack = require('webpack');
const dllAssetPath = path.join(__dirname, 'dll');
const dllLibraryName = 'dllExample';
module.exports = {
  entry: ['react'],
  output: {
    path: dllAssetPath,
    filename: 'vendor.js',
    library: dllLibraryName,
  },
  plugins: [
    new webpack.DllPlugin({
      name: dllLibraryName,
      path: path.join(dllAssetPath, 'manifest.json'),
    })


  ],
};
```

name: dll libary名字，对应output.libary的值
path: 资源清单的绝对路径，业务代码打包会使用这个清单进行模块索引

vendor打包

package.json 单独设置一条vendor打包语句

```
// package.json
{
  ...
  "scripts": {
    "dll": "webpack --config webpack.vendor.config.js"
  },
}
```

dll目录又vendor.js与manifest.json文件
库的代码 资源清单

潜在问题

更改vendor是会修改模块id导致
其他模块引入vendor是因为id变化自己的hash也变了

- vendor变化导致用户必须重新下载所有资源
- vendor变了而模块id没变，会出现意料之外的错误，很难排查问题

webpack 4之后就没这个问题了

tree shaking

ES6 Module依赖关系构建实在代码编译时运行
基于 Webpack tree shaking功能
打包过程中检测工程未引用的模块，“死代码”
Webpack会对这些代码标记，最后压缩的时候从bundle中去掉

只对 ES6 Module 有效

tree shaking 只对 ES6 Module生效
如果是CommonJS的库 并不能 起到标记的作用

webpack进行依赖关系构建

如果用到babel-loader，一定要禁用依赖解析
babel-loader做以来解析则是CommonJS形式的模块，无法tree-shaking

```
module.exports = {
  // ...
  module: {
    rules: [{
      test: /\.js$/,
      exclude: /node_modules/,
      use: [{
        loader: 'babel-loader',
        options: {
          presets: [
            // 这里一定要加上 modules: false
            [@babel/preset-env, { modules: false }]
          ],
        },
      }],
    }],
  },
};
```

压缩工具去除死代码

tree shaking只是为死代码添加标记，最后去除是压缩工具进行的。
terser
mode：production也可以达到相同效果

总结

如何加快打包速度，减小资源体积