01｜初级分析：使用webpack内置的stats

- stats:构建的统计信息
- package.json 中使用stats

```
{
    "script": {
        // ...
        "build:stats": "webpack --env production --json > stats.json"
    }
}
```

- Node.js中使用
    - 通过使用webpack接收回调
    - 的缺陷：颗粒度太粗，看不出问题所在 比如：某个js文件很大 只会显示最终的大小 那一个组件、loader无法分析出来

02｜速度分析：使用speed-measure-webpack-plugin

- 安装：npm install --save-dev speed-measure-webpack-plugin
- 代码示例

```
// 可以看到每个loader和插件执行耗时
const SpeedMeasurePlugin = require("speed-measure-webpack-plugin");
const smp = new SpeedMeasurePlugin();

const webpackConfig = smp.wrap({
  plugins: [new MyPlugin(), new MyOtherPlugin()],
});
```

- 速度分析插件的作用
  - 分析整个打包总耗时
  - 分析每个插件和loader的耗时情况 然后我可以针对性的进行处理 比如：某些插件很慢，看一下源代码，魔改一下

03｜体积分析：使用 webpack-bundle-analyzer

- 安装：npm install --save-dev webpack-bundle-analyzer
- 代码示例

```
const BundleAnalyzerPlugin = require('webpack-bundle-
analyzer').BundleAnalyzerPlugin;

module.exports = {
  plugins: [
    new BundleAnalyzerPlugin()
  ]
}
```

- 可以分析那些问题？
  - 依赖的第三方模块文件大小
  - 业务里面的组件代码大小

04｜使用高版本的webpack和node

- 例子： 同样一个项目 webpack4 比 webpack3 构建时间降低了60%～98%
- 使用 webpack5:优化原因

05｜多进程/多实例构建

- 可选方案
  - thread-loader
  - parallel-webpack
  - HappyPack（作者不怎么维护了）
- 使用 HappyPack解析资源
  - 原理：每次webpack 解析一个模块，HappyPack 会将它的依赖分配给worker线程中
- 使用 thread-loader 解析资源
  - 原理：每次webpack解析一个模块，thread-loader 会将它及它的依赖分配给 worker 线程中
  - 安装：npm install thread-loader -D
  - 示例代码

```
  module: {
      rules: [
          {
```

```
                test: /\.js$/,
                use: [
                    {
                        loader: 'thread-loader',
                        options: {
                            workers: 3
                        }
                    },
                    'babel-loader'
                ]
            }
        ]
    }
```

06 | 多进程 并行压缩代码

- 方法一：使用 webpack-parallet-uglify-plugin 插件
- 方法二：使用 uglifyjs-webpack-plugin 插件 开启 parallel（并列）-- 支持压缩ES6代码
- 方法三：terser-webpack-plugin 开启 parallel 参数（webpack4 开始推荐使用）
- 示例代码

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

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

07 | 进一步分包：预编译资源模块

- 分包：设置 Externals
  - 思路：将 react、react 等基础包通过cdn引入，不打入 bundle 中
  - 方法：使用 html-webpack-externals-plugin
  - 缺点：一个基础库必须制定一个cdn 不仅仅基础包还有很多其他的基础包 比如想 vuex、基础业务包等 这样的话会打出很多个script标签
- 进一步分包：预编译资源模块
  - 思路：将 react、react-dom、redux、react-redux基础包和业务基础包打包成一个文件
  - 方法：使用DLLPlugin(官方内置插件) 进行分包， DLLReferencePlugin 对 manifest.json 引用
- 使用DLLPlugin 进行分包 创建一个单独的配置文件 webpack.dll.js

```
const path = require('path')
const webpack = require('webpack')
```

```
module.exports = {
    context: process.cwd(),
    resolve: {
        extensions: ['.js', '.jsx', '.json', '.less', '.css'],
        modules: [__dirname, 'node_modules']
    },
    entry: {
        windowLibrary: [
            'react',
            'react-dom',
            'redux',
            'react-redux'
        ],
        // 如果需要 分为 基础库包 和 基础业务包 则新增一个key即可
        busLibrary: [

        ]
    },
    output: {
        filename: '[name].dll.js',
        path: path.resolve(__dirname, './build/library'),
        library: '[name]'
    },
    plugins: [
        new webpack.DLLPlugin({
            name: '[name]',
            path: './build/library/[name].json'
        })
    ]
}
```

- 使用 DLLReferencePlugin 引用 manifest.json
  - 在webpack.prod.js 引入

```
module.exports = {
    // ...
    plugins: [
        // ...
        new webpack.DLLReferencePlugin({
            minifest: require('./build/library/manifest.json')
        })
    ]
}
```

- 小结：vue-cli 和 react 都移除了对dll的支持 因为webpack4的打包性能足够优秀 可以使用 hard-source-webpack-plugin

08 | 充分利用缓存提升二次构建速度

- 缓存
  - 目的：提升二次构建速度
  - 缓存思路：
    - babel-loader 开启缓存
    - terser-webpack-plugin 开启缓存
    - 使用 cache-loader 或 hard-source-webpack-plugin 模块转换(webpack5 持久化缓存解决)

```
module.exports = {
  // ...
  plugins: [
    new HappyPack({
      loaders: [
        'babel-loader?cacheDirectory=true' // 开启babel-loader 缓存
      ]
    })
  ]
}
```

09 | 缩小构建目标

- 缩小构建目标
  - 目的：尽可能少的构建模块
  - 比如 babel-loader 不解析 node_modules
    - exclude: 'node_modules'
- 减少文件搜索范围
  - 优化 resolve.modules 配置（减少模块搜索层级）
  - 优化 resolve.mainFieles 配置
  - 优化 resolve.extensions 配置
  - 合理使用 alias

```
module.exports = {
  resolve: {
    alias: {
      react: path.resolve(__dirname,
'./node_modules/react/dist/react.min.js')
    },
    modules: [
      path.resolve(__dirname, 'node_modules') // 先从当前项目去找 没找到的
话就去node_modules找 减少模块搜索层级
    ],
    extensions: ['.js'], // 先找后缀
    mainFields: ['main'] // 减少
  }
}
```

10｜使用Tree Shaking擦除无用的JS和CSS

- tree shaking（摇树优化）复习
  - 概念：一个模块可能有多个方法，只要其中某个用到了，则整个文件都会被达到bundle里面去
  - 使用：webpack 默认支持 mode 模式下默认开启
  - 要求：必须ES6语法 CJS不支持
- 无用的CSS如何删除掉?
  - PurifyCSS：遍历代码，识别已经用到的CSS class
  - uncss：HTML 需要通过 jsdom 加载，所有样式通过 PostCSS解析，通过 document.querySelector 来识别在HTML文件里不存在的选择器
- 使用 purgecss-webpack-plugin

11｜使用webpack进行图片压缩

- 图片压缩
  - 要求：基于 Node库的imagemin 或者 tinypng API
  - 使用：配置 image-webpack-loader
- Imagemin 的优点分析
  - 有很多定制选项
  - 可以引入更多第三方优化插件，比如：pngquant
  - 可以处理多种图片格式
- Imagemin的压缩原理
  - pngquant：是一款PNG压缩器，通过将图片转换为 具有alpha通道（通常比24/32位PNG文件小60%-80%）的更高效的8位PNG格式，可显著见效文件大小
  - pngcrush：主要目的是通过尝试不同的压缩级别和PNG过滤方式来降低PNG IDAT数据流的大小
  - optipong：设计灵感源自pngcrush，可以将图像文件重新压缩为更小尺寸，而不会丢失任何信息
  - tinypng：和pngquant类似原理，也是将24位png文件转化为更小有索引的8位图片，同时所有非必要的matadata也会被剥离掉
- 安装：npm i image-webpack-loader 还有一大堆插件

12｜使用动态Polyfill服务

- 动态 Polyfill
  - 方案：polyfill-service
  - 优点：只给用户返回需要的polyfill，社区维护
  - 缺点：部分国内奇葩浏览器UA可以无法识别（但可以降级返回所需全部polyfill）
- Polyfill Service原理
  - 识别 User Agent 下发不同的 Polyfill
- 如何使用动态 polyfill service
  - polyfill.io 官方提供的服务 script标签 CDN 引入
  - 基于官方自建polyfill服务
    - //huayang.qq.com/polyfill/v2/polyfill.min.js?unkown=polyfill&features=Promise,Map,Set

00 | 体积优化策略总结小结

- Scope Hoisting
- Tree Shaking
- 公共资源分离
- 图片压缩
- 动态 Polyfill