如何优化 webpack 构建性能

随着 webapp 的复杂程度不断地增加，同时 node 社区的崛起也让前端在除了浏览器之外各方面予以强力衍生与渗透，不得不承认目前的前端开发已然是一个庞大和复杂的体系，慢慢的前端工程化这个概念开始逐渐被强调和重视。但前端开发的灵活与便捷性却带来不少影响，代码不管是否本地经过验证，要发布必须重新全量构建整个项目，而有些同学的把预发调试当成和本地调试一样使用，先不提这种做法是否正确但衍生而来的问题是用户必须忍受每次全量构建的时长，另外也由于项目复杂的增大，通常一个中型的项目业务模块都会有上百个，加上应用架构所包含的内容，已经是一个不小的体量，即使进行单次构建，也可能让开发者足足等上几分钟，甚至几十分钟。

时间都去哪儿了

在埋头开始优化前，首先我们必须理清楚知道一次全量构建他所包含的时间分别由什么组成，这可能让我们更加全面的去评估一个问题。

T 总 = T 下载依赖 + Twebpack

Twebpack = Tloaders + Tplugins

在如上粗略的评估中我们可以把时间归结在两大部分，一个是下载依赖耗时，还有一个是 webpack 构建耗时，而这一部分耗时可能包含了各类 loader 和 plugin 的耗时，css-loader ？babel-loader ？UglifyJsPlugin ？现在我们并不清楚

如何处理

基于如上的评估我们大概可以从三大方面来着手处理

- 从环境着手，提升下载依赖速度；
- 从项目自身着手，代码组织是否合理，依赖使用是否合理，反面提升效率；
- 从 webpack 可能存在的不足着手，优化不足，进一步提升效率。

从环境出发

一般碰到问题我们最容易想到的是升级一下依赖，把涉及构建的工具在 break change 版本之前都升级到最新，往往能带来意向不到的收益，这和机器出现问题重启往往能解决有异曲同工之处。

而事实上，也确实如此，在该项目中升级项目构建模块后提升了 10s 左右，暂时先不提其中的原由。

另外不得不吐槽 npm3 中安装依赖实在是龟速，那如何压榨安装依赖所需要的时间呢，是否有方案？或许大家已经听到过 cnpm，\

先来看看项目在不同版本的 npm 以及在 npmintall 场景下的差异：

node@4.2.4 | npm@2 (npm2)

修正后共计耗时 284s

依赖安装时长 182s

构建时长 Time: 102768ms 约为 102s

node@5.8.0 | cnpm@4 (cnpm)

共计耗时 140s

依赖安装时长 15s

初始构建所需 Time: 125271ms 约为 125s

使用 cnpm 能够达到立竿见影的效果，优化幅度达 70% - 90%

从项目自身出发

在未压缩的情况下脚本大于 1MB 变得非常普遍，甚者达到 3-4MB，这到底是因为什么？！

在这个过程中我分析了项目中的依赖，以及代码使用的状况。有几个案例非常的普遍。

案例一：依赖与依赖从属不明确

在 package.json 中

"lodash": "^4.13.1",
"lodash.clonedeep": "^3.0.2",

在这个案例中根据需求只需要保留其一即可

案例二：废弃依赖没有及时删除

很多时候我们业务变更很快，人员变动也很快，多人协同，这在项目中很容易被窥探出来

```
import xx from 'xxx';
```

然后在业务实现中并没有使用 xx

必须及时删除已经停止使用的相关库，无论是 deps 还是 devdeps，都使用 uninstall 的方式把相关依赖从 package.json 中移除，并千万记得也从源码中移除相关依赖。

案例三：为了实现小功能而引用大型 lib

webpack 强大的混淆能力，让 web 开发 和 node 开发的界线变得模糊，依赖的滥用问题异常凸显。

```
moment(key).format('YYYY-MM-DD HH:mm:ss')
// 只是使用了 moment 的 format 何必引入 moment
// 如果不想简单实现就可以使用更为专一的库来实现
// https://github.com/taylorhakes/fecha Lightweight date formatting and parsing (~2KB). Meant to replace parsing and formatting functionality of moment.js.


import isequal from 'lodash/isequal'
// 这样会导致整个 lodash 都被打入到包内
// 为何不直接使用
// import isequal from 'lodash.isequal'
```


等等等等。

引入一个第三方 lib 的时候，请再三思量，问自己是否有必要，是否能简单实现，是否可以有更优的 lib 选择

案例四：忽略三方库的优化插件

这个项目是使用脚手架工具初始化而来的项目,对于已经熟悉 ant-design 的同学而言，以上这些应该都已熟悉。

在查阅项目代码时候碰到了又一个很典型的案例

在 webpack.config.js 中引用了优化引用的插件 babel-plugin-antd

```
webpackConfig.babel.plugins.push(['antd', {
```

```
  style: 'css',
}]);
```

该插件会在 babel 语法解析层面对引用关系梳理即用什么的组件就只会引用什么样组件的代码以及样式。

```
// import js and css modularly, parsed by babel-plugin-antd
import { DatePicker } from 'antd';
```

但是很让人忧心的是我们很容易在原始代码里面找到这样的踪迹

```
import 'antd/lib/index.css';
```

一个错用可能就会让包的体积大上一个量级。

如上这类优化插件比如在 babel-plugin-lodash 中也有相关实现。

如果三方库有提供优化类插件，那么请合理的使用这类插件。

案例五：babel-runtime 和 babel-polyfill

由于历史的原因 babel@5 到 babel@6，polyfill 推荐的形式也并不一样。但是在项目中我们可以发现一点是，开发人员并不清楚这其中的原委。以至于代码中我们经常可以看到的一种情形是以下两种方式共存：

```
//js 文件
require('babel-pollyfill');
"dependencies": {
  "babel-runtime": "*"
},
"devDependencies": {
  "babel-plugin-transform-runtime": "*"
},
"babel": {
  "presets": [
    "es2015",
    "stage-0"
  ],
  "plugins": [
    "add-module-exports",
    "transform-runtime"
  ]
```

}

两种方式只需要一种即可，更加推荐下一种方式，在压缩的情况下至少能给代码减少 50KB 的体积

案例六：css-module

在 atool-build 中默认会对 *.module.less 和 *.module.css 的文件使用 css-module 来处理

而 css-module 这一块的处理由 css-loader 完成。

简单来说使用 css-module 后可以保证某个组件的样式，不会影响到其他组件

在日常中经常有同学会跑过来问，为什么我的样式变成有 hash 后缀了，为什么构建文件变大了，原因就在于此。

如果你的项目使用 ant-design， 并且通过 antd-init 脚手架来生成项目，那么你所有的 less 文件都会被应用 css-module。

这本应是一种好的方式，但是在实际项目中开发者并不清楚其中的逻辑，并且在使用在也不规范，如手动直接调用大型组件的 less 文件的同时也调用其 css 文件。

应用 css-module 后会导致构建的文件体积变大，如果小项目，并且能自己管理好命名空间的情况下可以不开启，反之请开启。

另外关于 css-loader 自从版本 0.14.5 之后压缩耗时增加几十倍的问题，其实之前在本地做过相应的测试，

css-loader 分别尝试过 0.14.5 和 0.23.x

然后并没有在这个业务项目中发现问题，但不保证别的业务项目中会复现这个问题，基于此记录一笔。

案例七 发布至 npm 的包并未 es5 化

目前在 atool-build 中处理 jsx 时并不会像处理 js 一样对 node_modules 目录下的内容进行屏蔽。

而现在在内部项目中可以看到大量的场景借 jsx 核没有 es5 化，这无疑是构建性能中巨大瓶颈的一块。

发布至 npm 的包，请全部 es5 化

综上开源世界的选择很多很精彩，但回过来头来想想我们是不是有点过分的利用了这份便捷，少了些对前端本身的敬畏呢。我们要合理适度的使用三方依赖，并认真思考每一步选择背后所需要承担的结果。

通过依赖的精简，使用上的规范，在构建速度上提升了 12 秒，在代码压缩的情况下省下了约 900KB 的空间

从 webpack 不足出发

- 单进程实现在耗 cpu 计算型 loader 中表现不佳，happypack 的用武之地

在 webpack 中虽然所有的 loader 都会被 async 并发调用，但是从实质上来讲它还是运行在单个 node 的进程中，以及在同一个事件循环中。虽然单进程在处理 IO 效率上要强于 多进程，但是在一些同步并且非常耗 cpu 过程中，多进程应该是优于单进程的，比如现在的项目中会用 babel 来 transform 大量的文件。所以 happypack 的性能提升大概就来源于此。 当然也可以预见到，如果你的项目并不复杂，没有大量的 ast 语法树解析层的事情要做，那么即使用了 happypack 成效基本可视为无。

在优化项目中，400 多模块都需要涉及 babel 加载，并且还存在 npm 包并没有 es5 的情况（bad），所以可以预见到的是会有不错的结果。

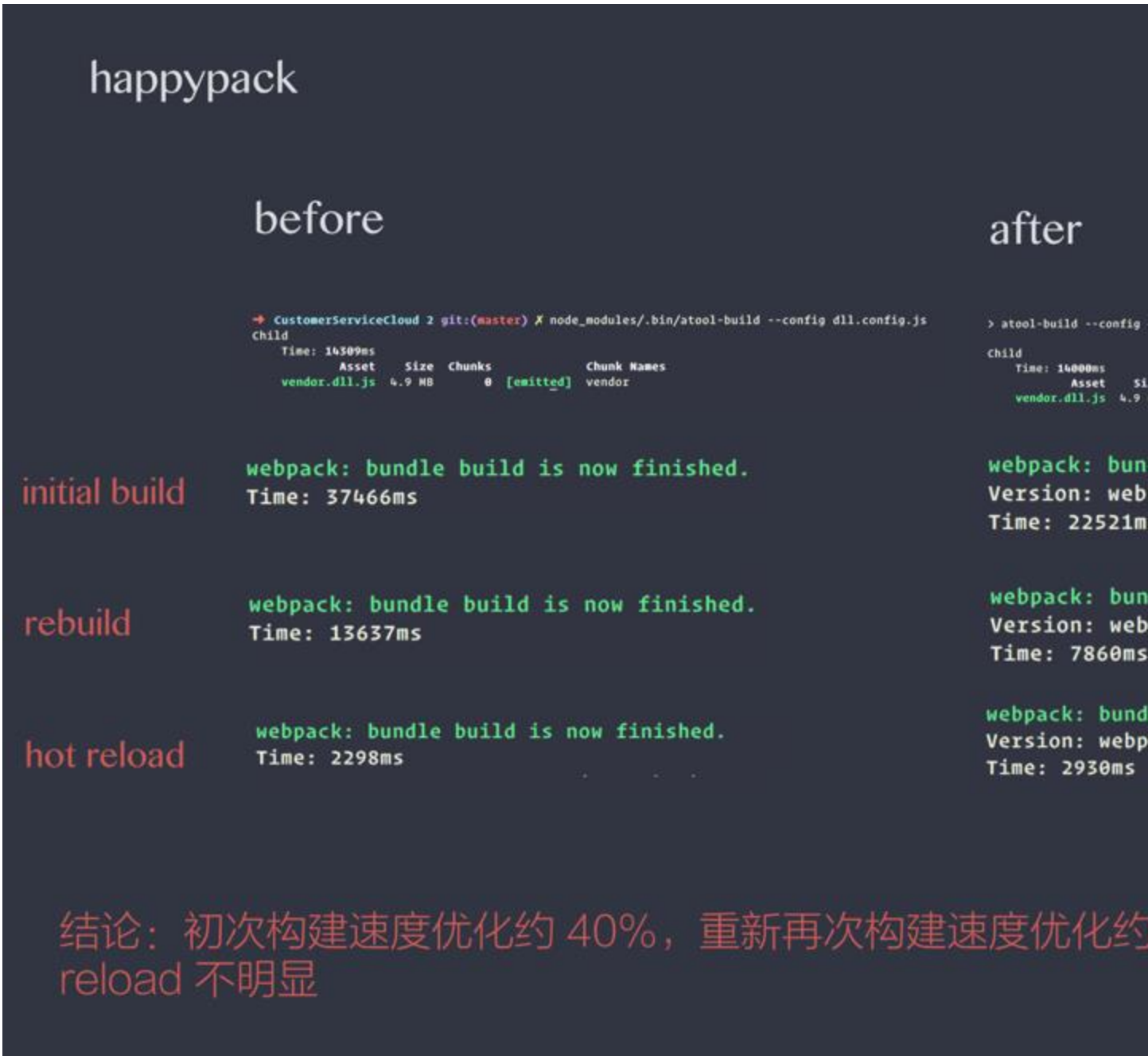
在 webpack.config.js 中添加如下代码片段

```
var babelQuery = webpackConfig.babel;
var happyThreadPool = HappyPack.ThreadPool({ size: 25 });
function createHappyPlugin(id, loaders) {
  console.log('id', id)
  return new HappyPack({
    id: id,
    loaders: loaders,
    threadPool: happyThreadPool,

    // disable happy caching with HAPPY_CACHE=0
    cache: true,

    // make happy more verbose with HAPPY_VERBOSE=1
    verbose: process.env.HAPPY_VERBOSE === '1',
  });
}
webpackConfig.module = {};
```

```
webpackConfig.module = {
  loaders: [
    {
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'happypack/loader?id=js',
    },
    {
      test: /\.jsx$/,
      loader: 'happypack/loader?id=jsx',
    },
    {
      test(filePath) {
        return /\.css$/.test(filePath) && !/\.module\.css$/.test(filePath);
      },
      loader: ExtractTextPlugin.extract('style',
'happypack/loader?id=cssWithoutModules')
    },
    {
      test: /\.module\.css$/,
      loader: ExtractTextPlugin.extract('style',
'happypack/loader?id=cssWithModules')
    },
    {
      test(filePath) {
        return /\.less$/.test(filePath) && !/\.module\.less$/.test(filePath);
      },
      loader: ExtractTextPlugin.extract('style',
'happypack/loader?id=lessWithoutModules')
    },
    {
      test: /\.module\.less$/,
      loader: ExtractTextPlugin.extract('style',
'happypack/loader?id=lessWithModules')
    }
  ],
}
if (!!handleFontAndImg) {
  webpackConfig.module.loaders.concat([
```

```
    { test: /\.woff(\?v=\d+\.\d+\.\d+)?$/, loader:
'happypack/loader?id=woff' },
    { test: /\.woff2(\?v=\d+\.\d+\.\d+)?$/, loader:
'happypack/loader?id=woff2' },
    { test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/, loader: 'happypack/loader?id=ttf' },
    { test: /\.eot(\?v=\d+\.\d+\.\d+)?$/, loader: 'happypack/loader?id=eot' },
    { test: /\.svg(\?v=\d+\.\d+\.\d+)?$/, loader: 'happypack/loader?id=svg' },
    { test: /\.(png|jpg|jpeg|gif)(\?v=\d+\.\d+\.\d+)?$/i, loader:
'happypack/loader?id=img' },
    { test: /\.json$/, loader: 'happypack/loader?id=json' },
    { test: /\.html?$/, loader: 'happypack/loader?id=html' }
  ])
} else {
  webpackConfig.module.loaders.concat([
    { test: /\.woff(\?v=\d+\.\d+\.\d+)?$/, loader:
'url?limit=10000&minetype=application/font-woff' },
    { test: /\.woff2(\?v=\d+\.\d+\.\d+)?$/, loader:
'url?limit=10000&minetype=application/font-woff' },
    { test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/, loader:
'url?limit=10000&minetype=application/octet-stream' },
    { test: /\.eot(\?v=\d+\.\d+\.\d+)?$/, loader: 'file' },
    { test: /\.svg(\?v=\d+\.\d+\.\d+)?$/, loader:
'url?limit=10000&minetype=image/svg+xml' },
    { test: /\.(png|jpg|jpeg|gif)(\?v=\d+\.\d+\.\d+)?$/i, loader:
'url?limit=10000' },
    { test: /\.json$/, loader: 'json' },
    { test: /\.html?$/, loader: 'file?name=[name].[ext]' }
  ])
}
webpackConfig.plugins.push(createHappyPlugin('js',
['babel?'+JSON.stringify(babelQuery)]))
webpackConfig.plugins.push(createHappyPlugin('jsx',
['babel?'+JSON.stringify(babelQuery)]))
webpackConfig.plugins.push(createHappyPlugin('cssWithoutModules',
['css?sourceMap&-restructuring!postcss']))
webpackConfig.plugins.push(createHappyPlugin('cssWithModules',
['css?sourceMap&-
restructuring&modules&localIdentName=[local]___[hash:base64:5]!postcss']))
webpackConfig.plugins.push(createHappyPlugin('lessWithoutModules',
['css?sourceMap!postcss!less-loader?sourceMap']))
```

```
webpackConfig.plugins.push(createHappyPlugin('lessWithModules',
['css?sourceMap&modules&localIdentName=[local]___[hash:base64:5]!postcss!less-
loader?sourceMap']))
if (!!handleFontAndImg) {
  webpackConfig.plugins.push(createHappyPlugin('woff',
['url?limit=10000&minetype=application/font-woff']))
  webpackConfig.plugins.push(createHappyPlugin('woff2',
['url?limit=10000&minetype=application/font-woff']))
  webpackConfig.plugins.push(createHappyPlugin('ttf',
['url?limit=10000&minetype=application/octet-stream']))
  webpackConfig.plugins.push(createHappyPlugin('eot', ['file']))
  webpackConfig.plugins.push(createHappyPlugin('svg',
['url?limit=10000&minetype=image/svg+xml']))
  webpackConfig.plugins.push(createHappyPlugin('img', ['url?limit=10000']))
  webpackConfig.plugins.push(createHappyPlugin('json', ['json']))
  webpackConfig.plugins.push(createHappyPlugin('html',
['file?name=[name].[ext]']))
}
```



总结来说，在需要大量 cpu 计算的场景下，使用 happypack 能给项目带来不少的性能提升。从本次优化项目来看，在建立在 dll 的基础上，初次构建能再提升 40% 以上，重新构建也会提升 40% 以上。所以 dllPlugin 和 happypack 的结合可以大大优化开发环节的时间。

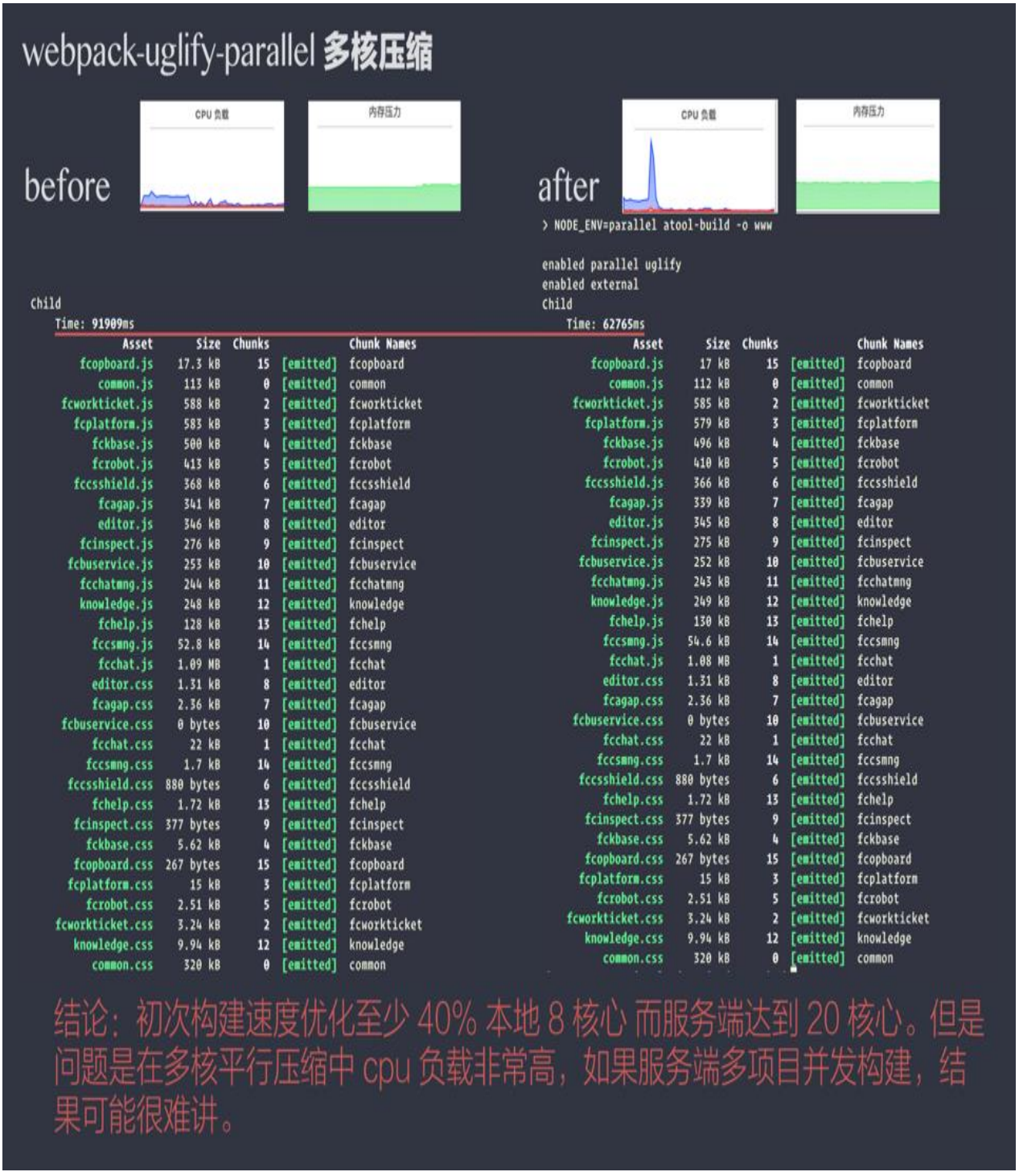
- uglifyPlugin 慢如蜗牛

uglify 过程应该是整个构建过程中除了 resolve and parse module 外最为耗时的一个环节。之前一直想要尝试在这个过程的优化，最近看社区新闻的时候，发现了 webpack-

uglify-parallel 深入看了这个库的组织和实现，因为它就是基于 uglifyPlugin 修改而来。

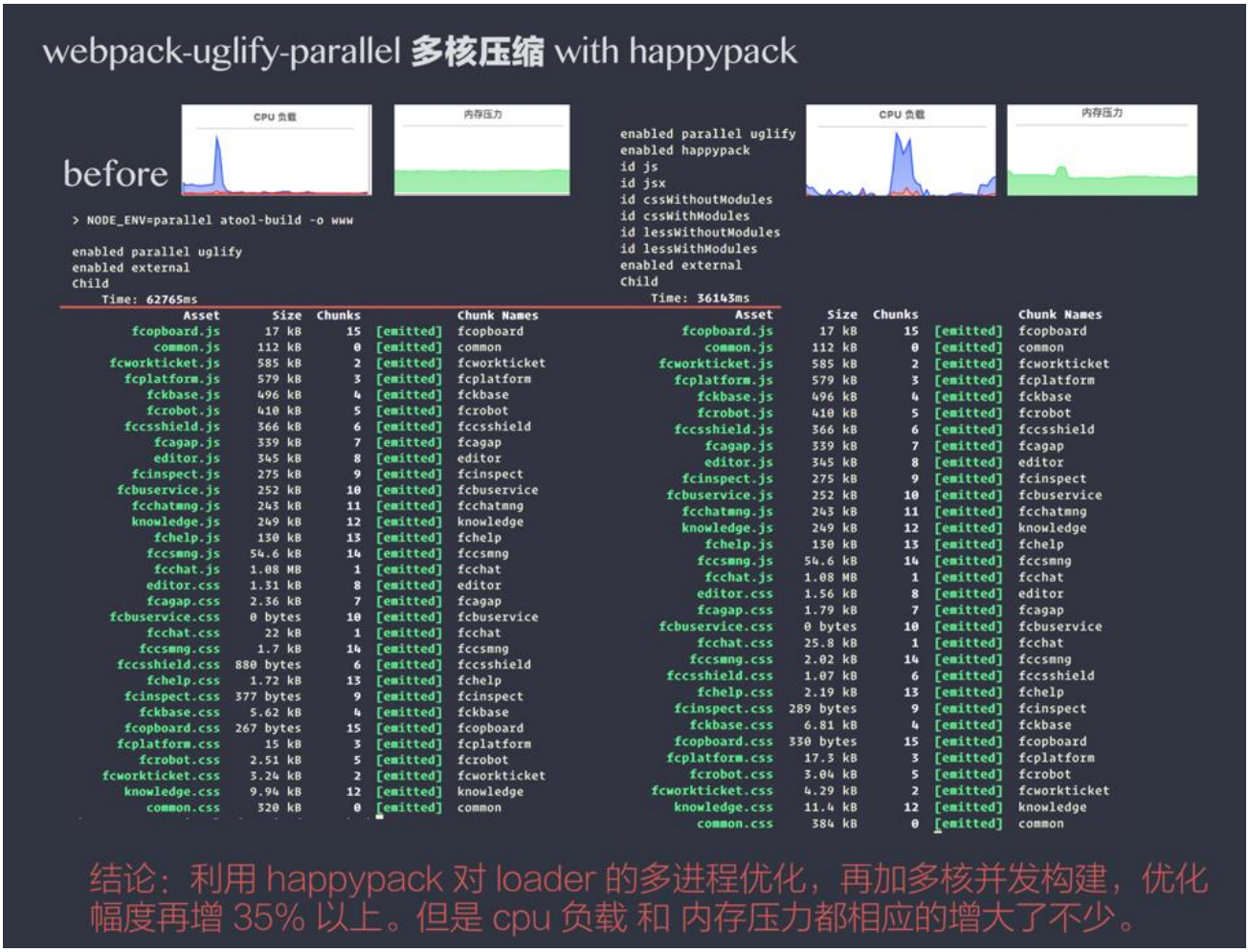
在 webpack.config.js 中启用 webpack-uglify-parallel 多核并行压缩

```
webpackConfig.plugins.some(function(plugin, i) {
  if (plugin instanceof webpack.optimize.UglifyJsPlugin) {
    webpackConfig.plugins.splice(i, 1);
    return true;
  }
});
var os = require('os');
var options = {
  workers: os.cpus().length,
  output: {
    ascii_only: true,
  },
  compress: {
    warnings: false,
  },
  sourceMap: false
}
var UglifyJsParallelPlugin = require('webpack-uglify-parallel');
webpackConfig.plugins.push(
  new UglifyJsParallelPlugin(options)
);
```



结论：初次构建速度优化至少 40% 本地 8 核心 而服务端达到 20 核心。但是问题是在多核平行压缩中 cpu 负载非常高，如果服务端多项目并发构建，结果可能很难讲。需要有节制使用多核的并行能力。

- 最后大家可能很好奇，webpack-uglify-parallel 和 happypack 和 external 的混搭会带来怎么样的化学反应。



如上这个组合拳非常适合在生成环节下使用，而 dll + happypack 则更加适合在开发环节。