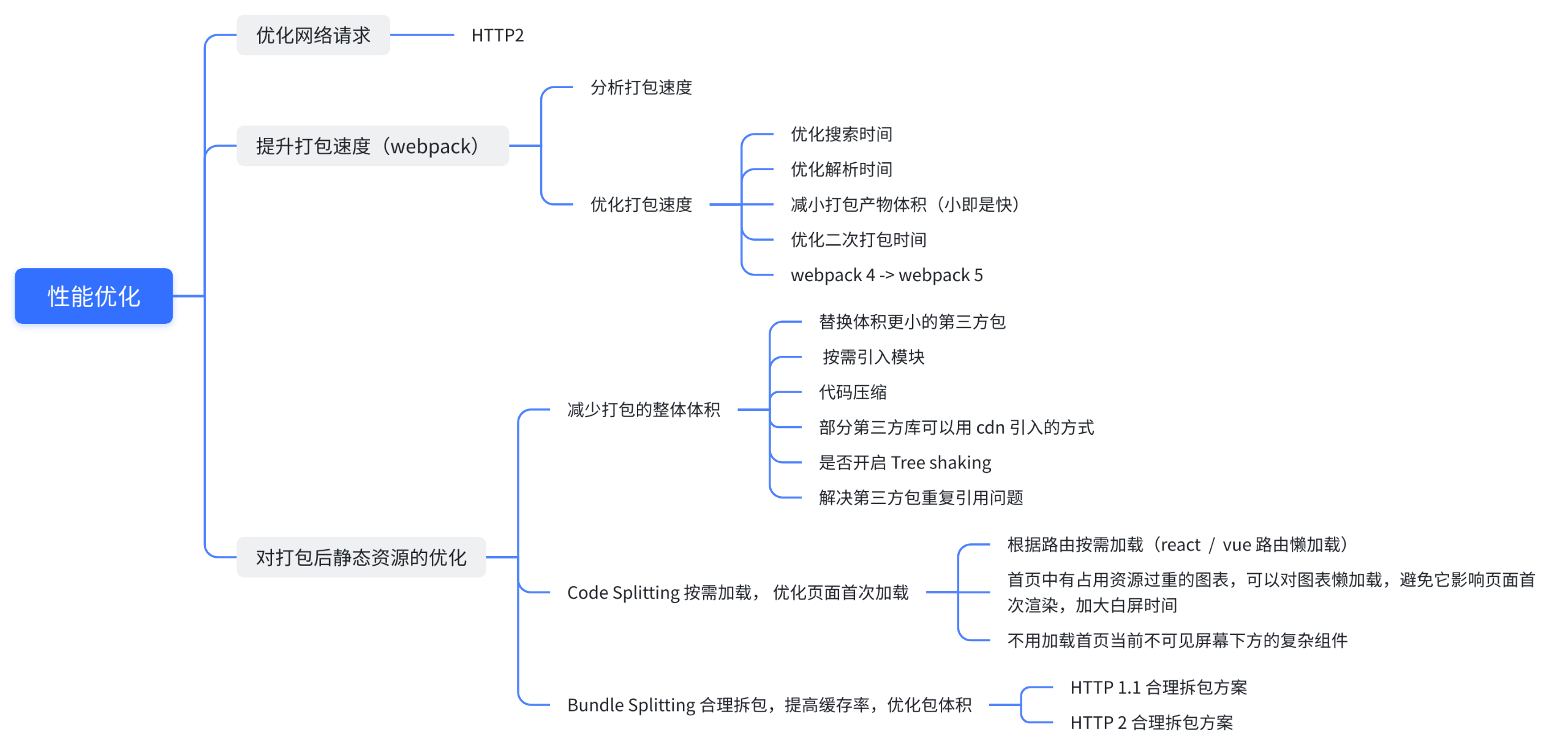
**性能优化实践指南**

|  |
| --- |
| **前言**  建议结合 [性能治理平台](https://performance.cn.goofy.app/personal) 相关治理建议 & 检测能力进行优化实践。 |

一般谈到优化主要涉及三个方面，减少网络请求时间、提高打包速度、对打包后的静态资源的优化。

**整体流程**



**减少网络请求时间**

**静态资源请求使用 HTTP2（多路复用）**

HTTP/2 的思路就是**一个域名只使用一个 TCP 长连接来传输数据**，这样整个页面加载过程只需要一次TCP 慢启动，同时可以避免了多个 TCP 连接竞争带来的带宽问题。

关于 HTTP 1.1 队头阻塞问题，等待请求完成后才能去请求下一个资源，对此 HTTP/2 **实现资源的并行请求，也就是不需要等待其他请求的完成**，然后服务器可以随时返回处理好的请求资源给浏览器。

简要总结一下，之前 HTTP 1.1 为了解决队头阻塞问题，做了每个域名下可以建立 6 个 TCP 连接优化，这样可以同时发送 6 个请求，不过这样的话又会带来带宽利用不理想问题，比如 6 次慢启动，竞争带宽等，因此 HTTP/2 做了一个域名下一个长连接，且可以并行发送请求的优化。

**提高打包速度（WIP）**

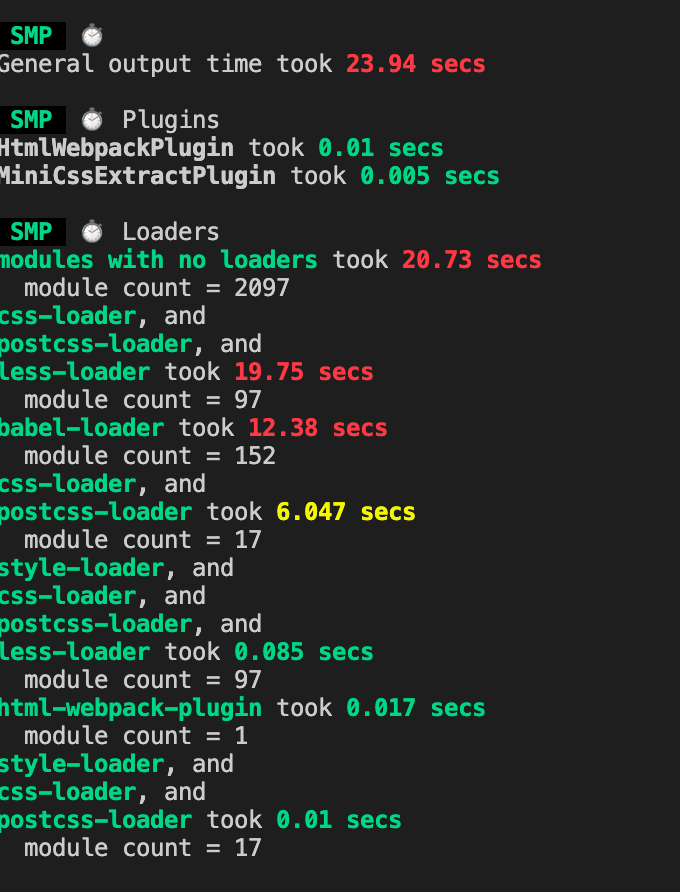
|  |
| --- |
| 构建性能优化后续待丰富，补充相关具体实践 |

**分析打包速度**

优化 webpack 构建速度的第一步是要知道耗时在哪里。我们可以通过 speed-measure-webpack-plugin 测量你的 webpack 构建期间各个阶段 loader、plugin 花费的时间：

|  |
| --- |
| TypeScript // webpack.config // 分析打包时间 const SpeedMeasurePlugin = require('speed-measure-webpack-plugin'); const smp = new SpeedMeasurePlugin();  // ...  // 耗时分析 if (env && env.speed) {  config.stats = 'errors-only'; // 只展示错误信息，方便查看speed  return smp.wrap(config); // 查看plugin和loader的耗时  } |

特定的项目，都有自己特定的性能构建瓶颈，我们可以针对耗时比较严重的loader、plugin 进行优化。



**优化打包速度**

**优化搜索时间**

我们可以缩小文件搜索范围， 减小不必要的编译工作。

**优化 loader 配置**

由于 loader 对文件的转换操作很耗时，我们需要让尽可能少的文件被 Loader 处理。可以通过 test 、 include 、 exclude 三个配置项来命中 Loader 要应用规则的文件。

|  |
| --- |
| TypeScript {  test: /\.(j|t)sx?$/,  loader: 'babel-loader',  include: [  // 只对项目根目录下的 src 目录中的文件采用 babel-loader  path.resolve(\_\_dirname, './src')  ]  }, |

每个额外的 loader/plugin 都有其启动时间，尽量减少不必要的使用。

**优化 resolve.alias 配置**

resolve.alias 配置项通过别名来把原导入路径映射成一个新的导入路径，减少耗时的递归解析操作。

|  |
| --- |
| TypeScript resolve: {  alias: {  "components": path.resolve(\_\_dirname, './components'),  } }, |

**优化 resolve.extensions 配置**

在导入语句没带文件后缀时，webpack 会根据 resolve.extension 自动带上后缀后去尝试询问文件是否存在，所以在配置 resolve.extensions 应尽可能注意以下几点：

* resolve.extensions 列表要尽可能的小，不要把项目中不可能存在的情况写到后缀尝试列表中。
* 频率出现最高的文件后缀要优先放在最前面，以做到尽快的退出寻找过程。
* 在源码中写导入语句时，要尽可能的带上后缀，从而可以避免寻找过程。

|  |
| --- |
| JavaScript resolve: {  extensions: ['.ts', '.tsx', '.js', '.jsx', '.less'], }, |

同理可通过减少 resolve.modules, resolve.mainFiles, resolve.descriptionFiles 中条目数量，因为他们会增加文件系统调用的次数。

**优化解析时间**

日常开发中我们需要使用 loader 对 js ，css ，img，font 等文件做转换操作，并且转换的文件数据量也是非常大。由于 js 单线程的特性使得这些转换操作不能并发处理文件，而是需要一个个文件进行处理。当 webpack 需要打包大量文件时，打包时间就会比较漫长， 因此我们可以对耗时比较长的 loader 开启多进程。

**开启多进程打包( webpack4 官方推荐 thread-loader )**

[thread-loader](https://www.npmjs.com/package/thread-loader) 可以将非常消耗资源的 loader 分流给一个 worker pool。 将 thread-loader 放在 expensive loader 后面，不要在小开销的loader开启，反而会更慢。

|  |
| --- |
| 不要使用太多的 worker，因为 Node.js 的 runtime 和 loader 都有启动开销。最小化 worker 和 main process(主进程) 之间的模块传输。进程间通讯(IPC)是非常消耗资源的。 |

**减小打包体积（小即是快）**

减少编译结果的整体大小，以提高构建性能。尽量保持 chunk 体积小。具体优化方案可详见下面对[打包静态资源的优化方案](https://bytedance.feishu.cn/docx/doxcngTgupE8patDZDKiTPF9FGd)。

**优化二次打包时间**

**持久化缓存**

在 webpack 配置中使用 [cache](https://webpack.docschina.org/configuration/cache) 选项。使用 package.json 中的 "postinstall" 清除缓存目录。

**对打包后静态资源的优化**

对于打包资源优化的总体原则，在于**尽可能的减少模块体积或者延迟模块的引用的基础上提高缓存命中率**。大概总结了以下三点：

1. **减小打包的整体体积**
2. **Code Splitting**: 按需加载，优化页面首次加载体积。如根据路由按需加载，可根据是否可见按需加载
3. **Bundle Splitting**: 分包，根据模块更改频率分层次打包，充分利用缓存；

**减小打包的整体体积**

**替换体积更小的第三方包**

[性能治理平台](https://performance.cn.goofy.app/analyze/webpack/62aa9b4ff495f3004c348a08/w_freebox?contrastVersionId=62aa9a8ff495f3004c348a06) 通过检测当前打包产物依赖的第三方包，对功能相似的第三方包，推荐体积更小的第三方包替换，同时给出**可减少体积大小**，让开发者可以通过节省的体积大小来决定是否替换第三方包。

（如下图所示，替换 monaco-editor 可减少 5.88 MB，可以大大的减小项目的编译产物体积）

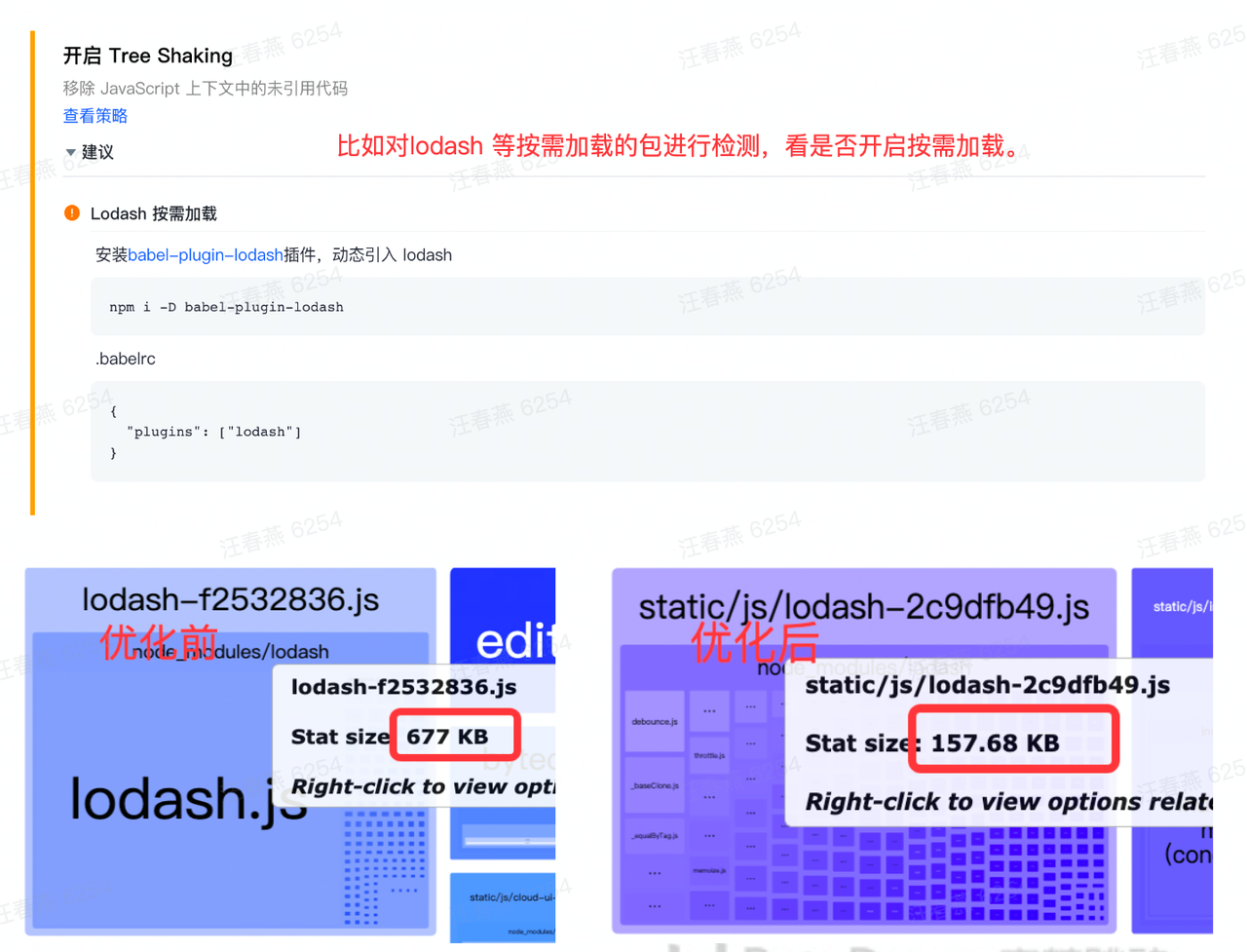


**按需引入模块**

[性能治理平台](https://performance.cn.goofy.app/analyze/webpack/62aa9b4ff495f3004c348a08/w_freebox?contrastVersionId=62aa9a8ff495f3004c348a06) 通过对按需引入的包进行检测，看是否开启按需引入。常见的如 lodash ，acro-design，echarts 等按需引入的包，例如 lodash 按需引入，我们可以根据自己所需要使用的模块单独引入：

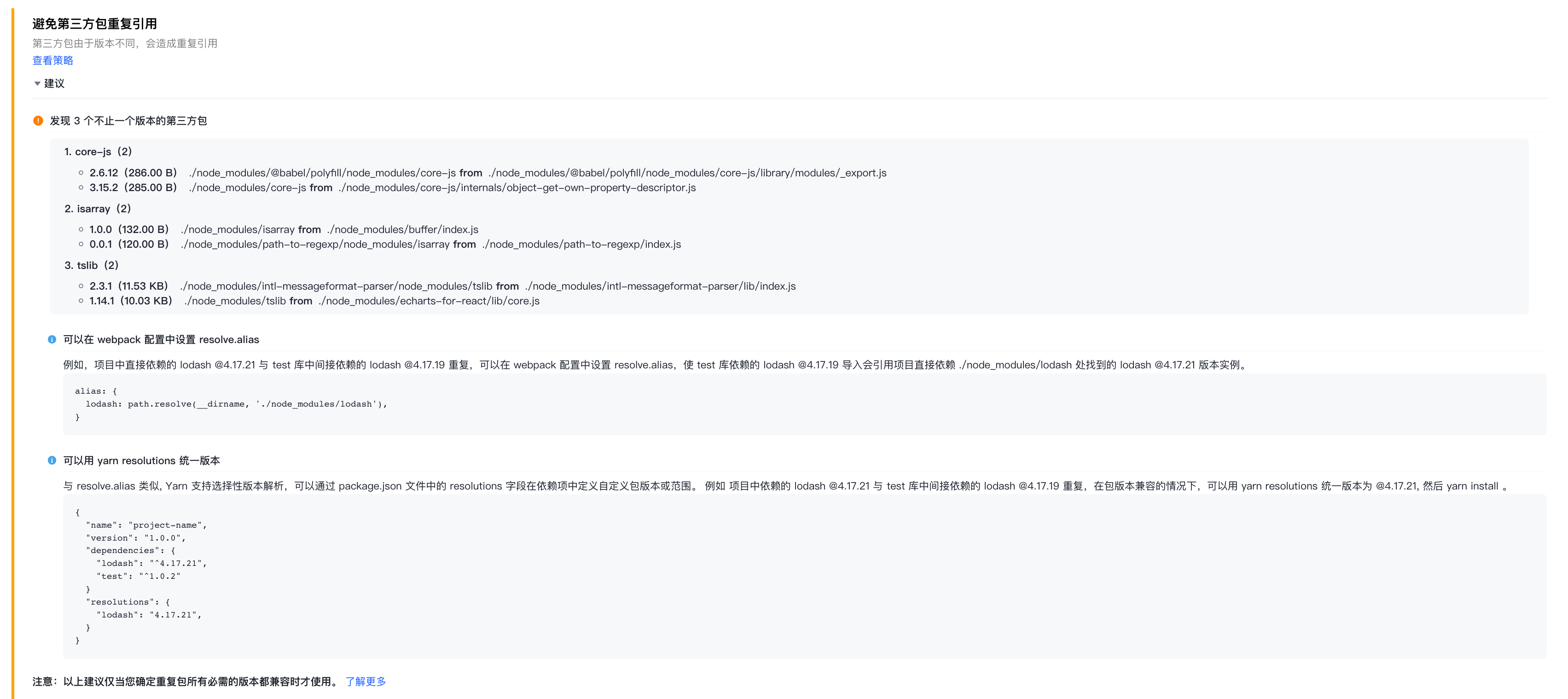
|  |
| --- |
| TypeScript import last from "lodash/last"; |

也可以利用 [babel-plugin-lodash](https://bnpm.bytedance.net/package/babel-plugin-lodash) 插件，按需引入 lodash。



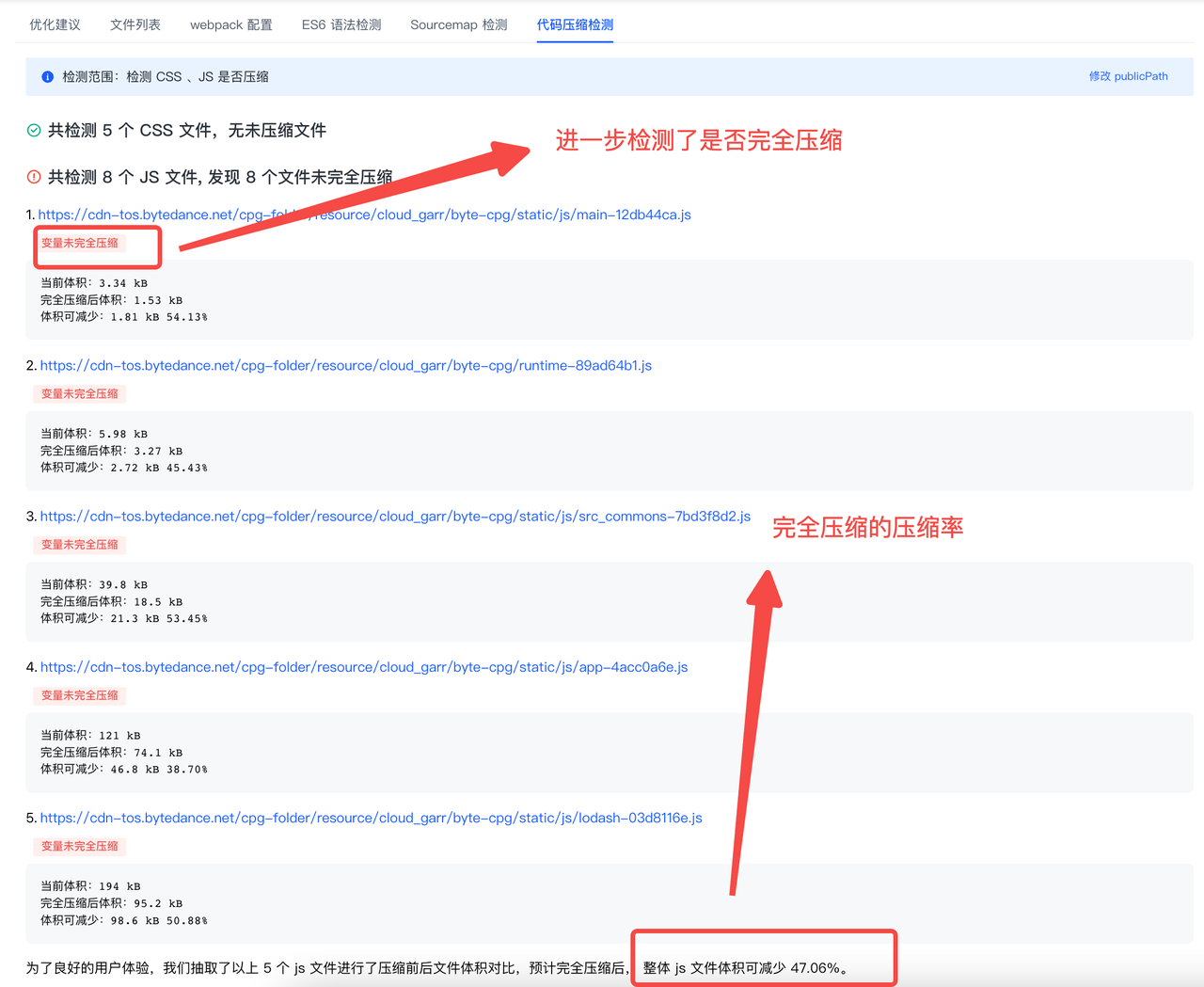
**解决第三方包重复引用问题**

[性能治理平台](https://performance.cn.goofy.app/analyze/webpack/62c2995e892ca4005501d170/w_performance_govern?contrastVersionId=62c294bc892ca4005501d168)根据构建产物，检测是否有第三方包重复引用的情况，并给出重复引用的第三方包体积。对于**版本不兼容的重复依赖**可以根据平台检测出的重复包各个版本的体积大小，看是否有解决必要。关于第三方包重复解决方案详见 [第三方包重复引用解决方案](https://bytedance.feishu.cn/docx/doxcnny3OdgyxzhnkzYeugmJZPe)



**代码压缩**

webpack 压缩插件对资源进行压缩，常见的有 js、 css 压缩。检测当前版本构建产物是否进行 CSS 压缩、JS 压缩。其中 JS 压缩除了检测是否压缩以外，进一步检测了**是否完全压缩**以及**整体压缩率** 。



**如何对 CSS 进行压缩**

可以结合 [mini-css-extract-plugin](https://github.com/webpack-contrib/mini-css-extract-plugin) & [optimize-css-assets-webpack-plugin](https://github.com/NMFR/optimize-css-assets-webpack-plugin) webpack 插件对 CSS 进行压缩，具体可以参考 [CSS 压缩文档](https://web.dev/minify-css/) 。

**如何对 JS 进行压缩**

* [terser-webpack-plugin](https://github.com/webpack-contrib/terser-webpack-plugin) （Webpack 推荐）
* [esbuild-loader](https://github.com/privatenumber/esbuild-loader) (ESBuildMinifyPlugin)
* [UglifyJS](https://github.com/mishoo/UglifyJS)

**部分第三方依赖包 external**

**部分第三方库用 cdn 引入的方式**

一些体积比较大且不常改动的稳定 npm 包，可以以 cdn 的方式去引入，打包 externals 掉， 这里需要**注意 cdn 地址安全性**，最好用内部 cdn 地址。

|  |
| --- |
| TypeScript // \*\*\*.html <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>  // webpack.config.js externals: {  jquery: 'jQuery', }, |

**优点**

减少整体打包编译时间和体积。（[关于cdn 其他优点](https://www.sitepoint.com/7-reasons-to-use-a-cdn/)）

**不适用场景**

* 需要按需引入的 npm 包， 对于按需加载的包没必要用 cdn 全量引入。
* cdn 引入的形式可能会造成全局污染问题，可根据项目实际情况适用。
* 对于体积比较大且不是首屏用到的包，用 cdn 形式引入，可能会影响首屏加载速度。

**微前端子应用**

对于一些微前端子应用，可以直接复用主应用的依赖，external 对应子应用依赖。打包的时候可以减小子应用整体打包编译时间和体积。

|  |
| --- |
| 目前字节云改造后，子应用已不提供 external 方法，具体可见 [字节云子平台历史包袱去除](https://bytedance.feishu.cn/docx/doxcni214A49GcRqWZFvLsVI6zc)。以下仅提供例子，其他微前端子应用可根据项目具体情况分析是否可以复用部分主应用依赖包。 |

如字节云(Garfish)的子应用，查看[字节云源码](https://code.byted.org/cloud-fe/cloud)可知，**vue、 vuex、 vue-router、 iview、view-design、 byted-tea-sdk、moment** 等 npm 包，可以直接复用主应用依赖，子应用 external 对应包。

|  |
| --- |
| TypeScript // 字节云源码 src/main.ts  // 主工程 external 出去包，子平台 webpack 中设置同名 external 包，打包的时候体积可以小一些 Garfish.setExternal('vue', Vue); Garfish.setExternal('vuex', Vuex); Garfish.setExternal('vue-router', Router); Garfish.setExternal('vue-i18n', VueI18n); Garfish.setExternal('iview', iView); Garfish.setExternal('view-design', iView); Garfish.setExternal('byted-tea-sdk', Tea);  // 字节元源码 src/registerProducts.ts Garfish.setExternal('moment', moment); |

**Code Splitting 按需加载，优化页面首次加载**

通过 Code Splitting 可以只加载当前所需要的资源，比如以下几种常见情形：

1. 首页中有占用资源过重的图表，可以对图表懒加载，避免它影响页面首次渲染，加大白屏时间
2. 不用加载首页当前不可见屏幕下方的复杂组件
3. 根据路由按需加载（如 react 路由懒加载）

主要实现方式：

* 使用[import()](https://v8.dev/features/dynamic-import) 动态加载模块
* 使用 [React.lazy()](https://reactjs.org/docs/code-splitting.html#reactlazy) 动态加载组件
* 使用[lodable-component](https://loadable-components.com/docs/getting-started/) 动态加载路由，组件或者模块

**Bundle Splitting 合理拆包， 提高缓存命中率**

|  |
| --- |
| **原则：尽可能减小模块体积或者延迟模块的引用的基础上提高缓存命中率。** |

|  |
| --- |
| ❓ [什么是缓存命中率](https://doc.bytedance.net/docs/7401/12965/239281/)？ |

**使用 contenthash 定义输出文件名**

使用 contenthash 定义输出文件名，contenthash 将根据资源内容创建出唯一 hash。当资源内容发生变化时，contenthash 也会发生变化。

|  |
| --- |
| TypeScript // js 文件 output: {  filename: "[name]-[contenthash:8].js" } |

它只影响最初加载的输出文件。对于按需加载的 chunk 文件，请使用 [output.chunkFilename](https://webpack.docschina.org/configuration/output/#outputchunkfilename) 选项来控制输出。

**固定 moduleIds 、chunkIds**

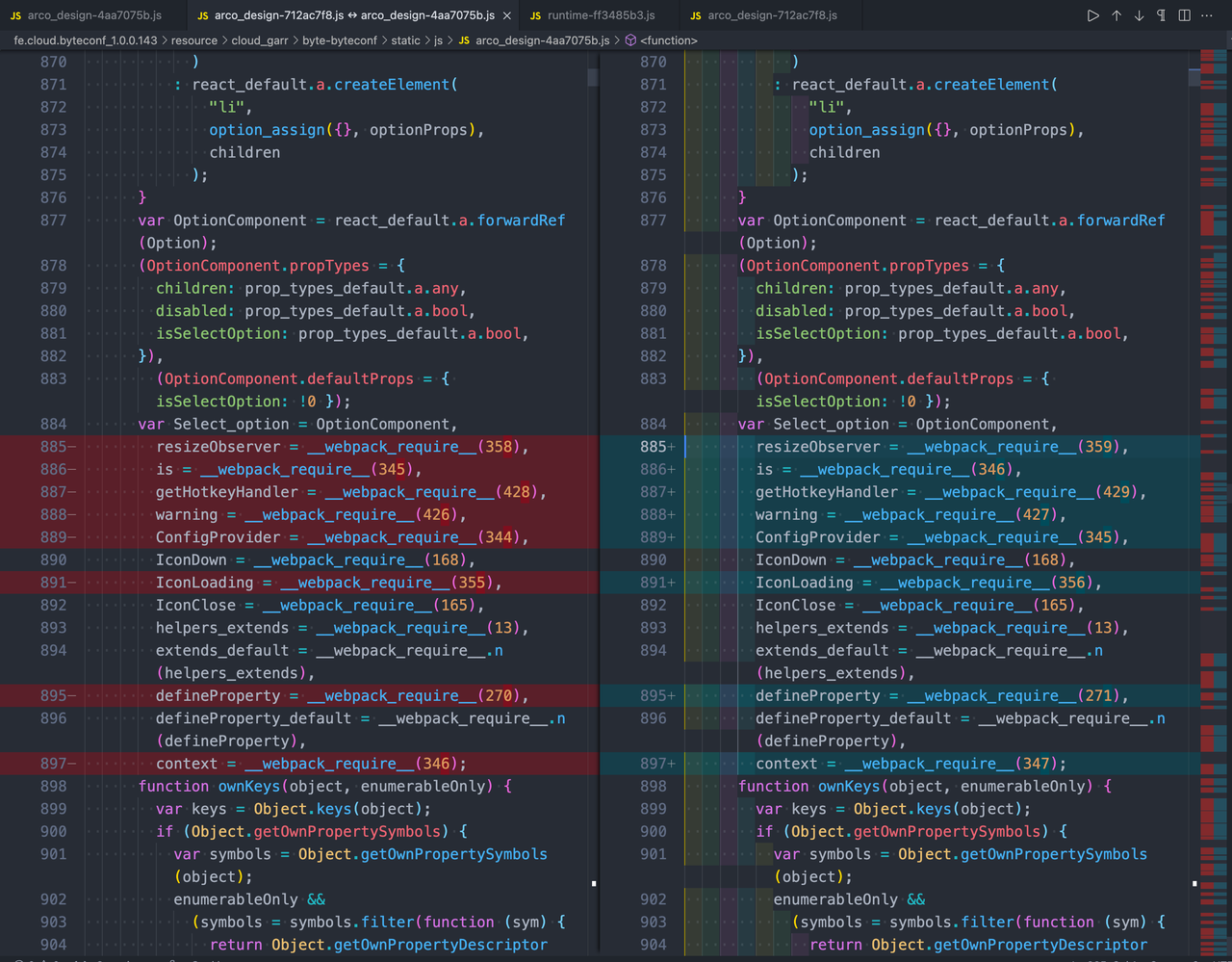
**Webpack 4 配置**

由 [webpack4 相关文档](https://v4.webpack.docschina.org/configuration/optimization/#optimization-moduleids)，webpack 4 默认 moduleIds, chunkIds 是按解析顺序增量的数字，也就是说，当解析顺序发生变化，id 也会随之改变，导致缓存失效。

|  |
| --- |
| TypeScript optimization: {  moduleIds: 'hashed', // 短哈希作为 id 以获得更好的长期缓存  chunkIds: 'named', // 不仅便于调试 且 便于长期缓存  ... } |

**为什么不固定 moduleIds 可能导致缓存失效**

当我们增减模块时，可能会导致其他没有更改过的文件内容变化（[module.id](https://webpack.docschina.org/api/module-variables/#moduleid-commonjs) 变化），导致没有更改过的文件也缓存失效，这明显是不符合预期的。因此我们需要固定 moduleIds， 便于我们资源的长期缓存。（这里结合 [webpack 示例](https://webpack.docschina.org/guides/caching/#module-identifiers) 更好理解）



**为什么不固定 chunkIds 可能导致缓存失效**

Webpack 默认 chunkname 为 chunkIds, 如果 chunkIds 是按解析顺序增量的数字（optimization.chunkIds = 'natural'）, 那么增删 chunk 可能会导致其他 chunk 的 id 跟着一起变，如下图所示，文件内容变化进而导致文件缓存失效。

|  |  |
| --- | --- |
| optimization.chunkIds = 'natural' | optimization.chunkIds = 'named' |

**Webpack 5 配置**

|  |
| --- |
| webpack 5 可以不单独设置 moduleIds， chunkIds。 |

与 webpack 4 不同，[webpack 5](https://webpack.docschina.org/configuration/optimization/#optimizationmoduleids) 新增了 optimization.moduleIds = 'deterministic' 默认配置，由模块哈希转化成的短数字。

同时也新增了 optimization.chunkIds= 'deterministic', 在**不同的编译中不变的短数字 id**，有益于长期缓存，**在生产模式中会默认开启**。在开发环境 optimization.chunkIds 会被设置成 'named'。

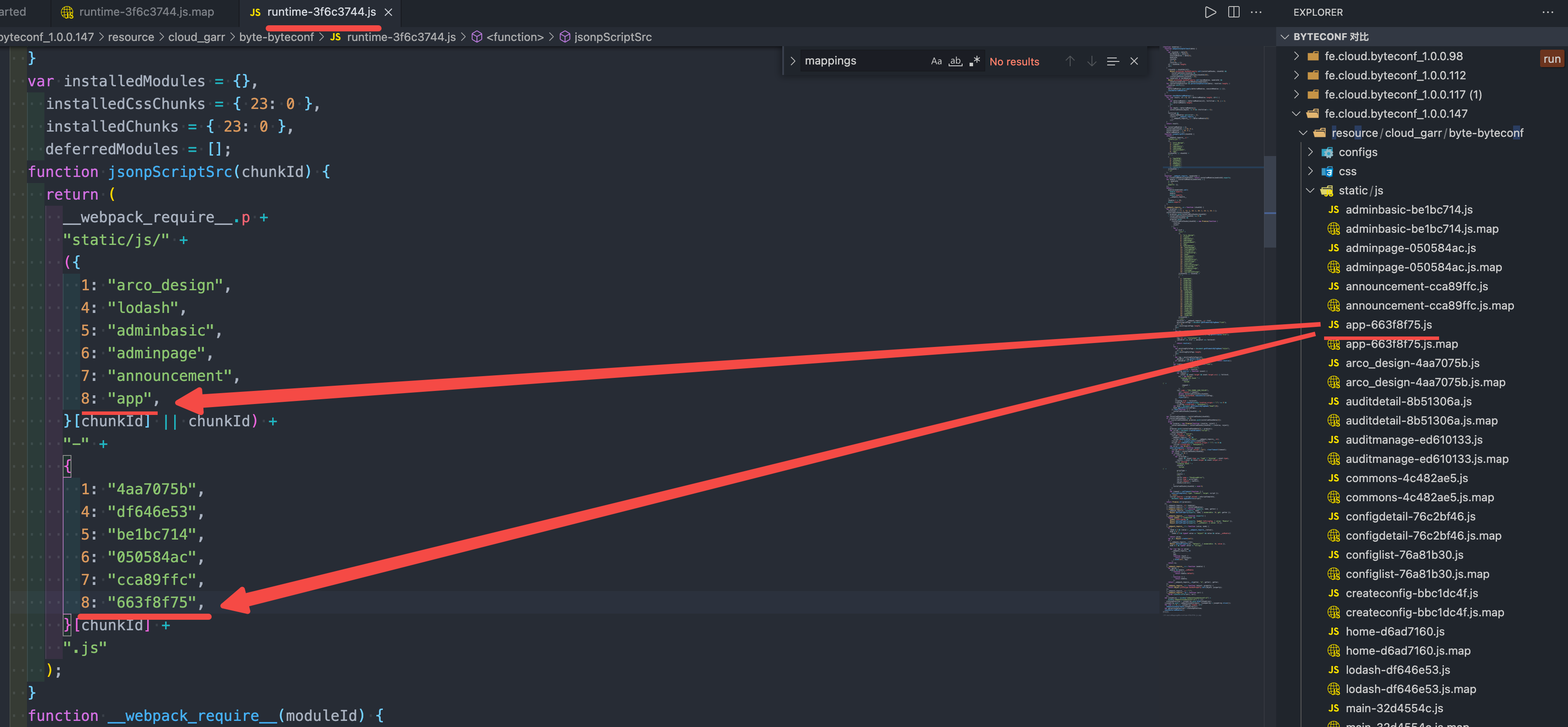
|  |
| --- |
| TypeScript optimization: {  moduleIds: 'deterministic',   chunkIds: 'deterministic', // 生产环境默认配置，开发环境默认为 named  ... } |

**分离 runtime**

Webpack 在编译时 compiler 会收集所有模块的关键信息（Manifest）。在运行时通过 Manifest 来解析和加载模块。无论选择哪种[模块语法](https://www.webpackjs.com/api/module-methods)，那些 import 或 require 语句现在都已经转换为 \_\_webpack\_require\_\_ 方法，此方法指向模块标识符(module identifier)。通过使用 manifest 中的数据，[runtime](https://www.webpackjs.com/concepts/manifest/#runtime) 将能够查询模块标识符，检索出背后对应的模块。

|  |
| --- |
| TypeScript optimization: {  runtimeChunk: {  name: 'runtime',  } |

如下图是webpack runtime 文件中 manifest 的代码，它包含入口文件与异步加载文件的映射关系，一般会打进入口文件里。 runtime 是比较容易经常变动缓存失效的（比如新增/减少异步资源等等情况），因此可以考虑单独抽取出来，避免其他没有改动的内容缓存失效从而重新加载整个包。



|  |
| --- |
| 对于简单的项目，分离的 runtime 文件比较小，可能不足1KB, 如果是用 HTTP1.1 请求资源的话，可能会增加一次额外请求的负担，可以考虑内联 webpack 的 runtime 方式，节省额外的 HTTP 请求。 |

**合理拆包*方案（适用于 SPA）***

**方案一（比较适合 HTTP 2）**

考虑 HTTP 2 的多路复用，初始页面的静态资源受资源请求数量的影响不大，因此为了更好的缓存效果以及按需加载我们可以适量加大资源请求数量。

**分包策略（基于页面路由懒加载基础上）**

1. 将**同步加载**的 npm 包拆分，对每个同步的 npm 包单独打包，设置 minSize: 4000; (40KB以下的npm 包不单独打包， 这里 minSize 可根据具体项目实际情况决定)
2. 将**异步加载**的 npm 包拆分，对每个异步的npm 包单独打包，设置 minSize: 4000; (40KB以下的npm 包不单独打包，这里 minSize 可根据具体项目实际情况决定)

|  |
| --- |
| Tips: 将比较大的 npm 包单独打包，每次增减第三方包依赖不会导致其他资源缓存失效。 |

1. 兜底配置，对剩余的 npm 包打包

* 对剩余的**同步加载**的 npm 包打在一起（40KB以下的依赖）
* 对剩余的**异步加载**的 npm 包打在一起（40KB以下的依赖）

|  |
| --- |
| Tips: node\_modules 和业务逻辑代码混合，业务逻辑更改，导致整个包将会重新下载，缓存命中降低 |

1. 对业务代码重复4次及以上的代码抽取出来，避免重复打包。

* 将业务代码中出现四次及以上且体积大于20000B的模块单独抽取出来, 避免多次打包增加包体积，这里 minChunks、minSize 可根据实际项目情况调整。

**分包配置**

整体分包参考配置如下，可根据实际项目情况进行调整：

|  |
| --- |
| TypeScript cacheGroups: {  initialPkg: {  // 拆分依赖，避免单文件过大拖慢页面展示  // 得益于HTTP2多路复用，不用太担心资源请求太多的问题  name(module) {  // 拆包  const packageName = module.context.match(/[\\/]node\_modules[\\/](.\*?)([\\/]|$)/)?.[1];  return `initial.pkg.${packageName?.replace('@', '')}`; // 部分服务器不允许URL带@  },  minSize: 40000, // 40K以下的依赖不做拆分  test: /[\\/]node\_modules[\\/]/,  priority: 5,  chunks: 'initial',  },  asyncPkg: {  name(module) {  // 拆包  const packageName = module.context.match(/[\\/]node\_modules[\\/](.\*?)([\\/]|$)/)?.[1];  return `async.pkg.${packageName?.replace('@', '')}`; // 部分服务器不允许URL带@  },  minSize: 40000, // 40K以下的依赖不做拆分  test: /[\\/]node\_modules[\\/]/,  priority: 4,  chunks: 'async',  },  // 兜底配置  initialVendors: {  // 同步其他第三方包  test: /[\\/]node\_modules[\\/]/,  minChunks: 1,  name: 'initialVendors',  minSize: 0,  chunks: 'initial',  priority: 3,  },  asyncVendors: {  // 异步其他第三方包  test: /[\\/]node\_modules[\\/]/,  minChunks: 1,  name: 'asyncVendors',  minSize: 0,  chunks: 'async',  priority: 2,  },  resource: {  // 业务公共代码重复打包4次以上的抽取出来  chunks: 'all',  minChunks: 4,  name: 'resource',  minSize: 20000,  priority: 1,  }, }, |

**方案二 (比较适合 HTTP 1.1)**

对于 HTTP 1.1，考虑到存在带宽利用不理想问题，我们在需要控制初始页面的静态资源受资源请求数量，对于体积比较小的包尽量不单独再发起请求请求一次。

**分包策略**

**体积比较大 & 非按需加载的第三方包， 单独打包**

对于体积比较大 & 非按需加载的（比如体积大于 800KB， 根据实际项目决定）的第三方包可以单独打包，**同时对于体积较大且不常用（比如 antv/ g6 等）的包可以用异步加载的方式去加载**，优化页面首次加载。

|  |
| --- |
| JavaScript  'antv': {  test: /[\/]node\_modules[\/]@antv[\/]/,  minChunks: 1,  chunks: 'all',  name: 'antv',  priority: 10,  }, |

**按需加载的第三方包， 单独打包**

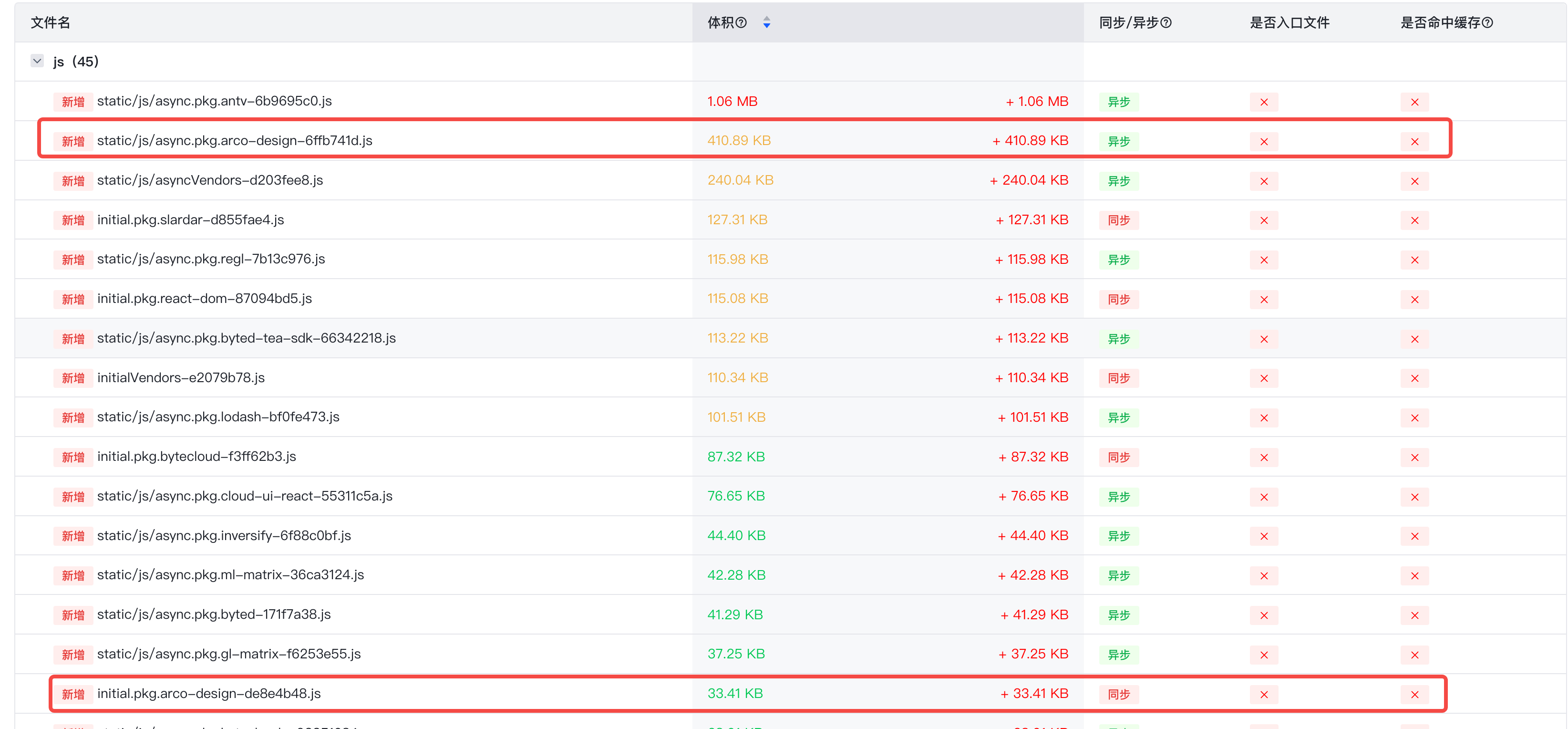
对于按需加载的第三方包我们可以单独打包，比如 acro ， antd、 lodash 等按需加载的包，每次引入新的模块功能，都会导致包变化，导致资源缓存失效，因此对于此类包我们可以单独打包，避免影响与它打在一起的别的资源缓存失效。

|  |  |  |
| --- | --- | --- |
| npm 包 | 类型 | 抽包方式（initial | async | all） |
| arco-design  antd  @bytecloud/react-ui  @byted/hooks  cloud-ui-react  lodash  echarts 等 | 按需加载 | 体积比较大的可以按同步 | 异步分开抽包的方式，根据实际项目决定 |

如下面这种情况：

acro-design 包体积比较大， 且部分页面路由懒加载，直接抽取出来会导致页面初始加载需要加载 700k+ 的 acro-design包，这种情况可以分别对同步 | 异步的 acro-design 打包，避免页面首次加载的时候加载到初始页面不需要的 acro-design 包。

|  |
| --- |
| JavaScript initialAcro : {  test: /[\/]node\_modules[\/]@arco-design[\/]/,  minChunks: 1,  name: 'initialAcro',  chunks: 'initial',  priority: 3,  },  asyncAcro: {  test: /[\/]node\_modules[\/]@arco-design[\/]/,  minChunks: 1,  name: 'asyncAcro',  chunks: 'async',  priority: 2,  }, |



**对于常用 & 改动频率比较低 & 非按需加载的第三方包打在一起**

对于这种常用 & 改动频率比较低 & 非按需加载的第三方包可以打在一起，抽取出来可以减少请求次数以及重复打包。

|  |  |  |
| --- | --- | --- |
| npm 包 | 类型 | 抽包方式（initial | async | all） |
| react  react-dom  react-router  redux  mobx  corejs  classnames  tslib  dayjs  byted-tea-sdk  @slardar | 常用  改动频率比较低 非按需加载 | all |

|  |
| --- |
| TypeScript commonVendors: {  // 改动频率比较低且使用频率高的包（大部分页面都用到）打到一起  test: /[\/]node\_modules[\/](@slardar|react|react-dom|react-router|core-js|classnames|tslib|dayjs|byted-tea-sdk)[\/]/,  minChunks: 1,  chunks: 'all',  name: 'commonVendors',  priority: 4,  }, |

**剩余第三方包兜底配置**

考虑到项目部分页面用到了**路由懒加载**，是**异步加载**的，因此对于剩余第三方包的兜底配置，采取了同步 | 异步 分开打包的方式，这样同步加载的时候不会加载到异步的第三方包。

|  |
| --- |
| TypeScript initialVendors: {  // 同步加载的其他第三方包  test: /[\/]node\_modules[\/]/,  minChunks: 1,  name: 'initialVendors',  minSize: 0,  chunks: 'initial',  priority: 3,  },  asyncVendors: {  // 异步加载的其他第三方包  test: /[\/]node\_modules[\/]/,  minChunks: 1,  name: 'asyncVendors',  minSize: 0,  chunks: 'async',  priority: 2,  }, |

**抽取业务代码重复模块**

将业务代码中出现四次及以上且体积大于20000B的模块单独抽取出来, 避免多次打包增加包体积，这里 minChunks、minSize 可根据实际项目情况调整。

|  |
| --- |
| TypeScript cacheGroups: {  resource: { // // 业务公共代码  chunks: 'all',  minChunks: 4,   name: 'resource',  minSize: 20000,  priority: 1,   },  } |

**分包配置**

整体分包参考配置如下，可根据实际项目情况进行调整：

|  |
| --- |
| TypeScript  ...  // 合理拆包  ...  cacheGroups: {  'antv': {  test: /[\/]node\_modules[\/]@antv[\/]/,  minChunks: 1,  chunks: 'all',  name: 'antv',  priority: 10,  },  'arco-design': {  // 体积大于1MB 或者 动态加载的 npm 包单独拆包  test: /[\/]node\_modules[\/]@arco-design[\/]/,  minChunks: 1,  chunks: 'all',  name: 'arco-design',  priority: 9,  },  bytecloud: {  test: /[\/]node\_modules[\/]@bytecloud[\/]/,  minChunks: 1,  chunks: 'all',  name: 'bytecloud',  priority: 8,  },  byted: {  test: /[\/]node\_modules[\/]@byted[\/]/,  minChunks: 1,  chunks: 'all',  name: 'byted',  priority: 7,  },  'cloud-ui-react': {  test: /[\/]node\_modules[\/]@cloud-ui-react[\/]/,  minChunks: 1,  chunks: 'all',  name: 'cloud-ui-react',  priority: 6,  },  lodash: {  test: /[\/]node\_modules[\/]lodash[\/]/,  minChunks: 1,  chunks: 'all',  name: 'lodash',  priority: 5,  },  commonVendors: {  // 改动频率比较低且使用频率高的包（大部分页面都用到）打到一起  test: /[\/]node\_modules[\/](@slardar|react|react-dom|react-router|core-js|classnames|tslib|dayjs|byted-tea-sdk)[\/]/,  minChunks: 1,  chunks: 'all',  name: 'commonVendors',  priority: 4,  },  initialVendors: {  // 其他第三方包  test: /[\/]node\_modules[\/]/,  minChunks: 1,  name: 'initialVendors',  minSize: 0,  chunks: 'initial',  priority: 3,  },  asyncVendors: {  // 其他第三方包  test: /[\/]node\_modules[\/]/,  minChunks: 1,  name: 'asyncVendors',  minSize: 0,  chunks: 'async',  priority: 2,  },  resource: {  // 业务公共代码重复打包4次以上的抽取出来  chunks: 'all',  minChunks: 4,  name: 'resource',  minSize: 20000,  priority: 1,  },  },  ... |

**性能优化实践示例**

👍 [TTI 优化101](https://bytedance.feishu.cn/wiki/wikcnpS7IYwYPLL4QlinMoLRoed)

👍  [freebox 平台性能优化(字节云子应用webpack 5)](https://bytedance.feishu.cn/docx/doxcnEKftug09G764jwFEkjT6ob)

[👍 byteconf 平台性能优化（字节云子平台& webpack 4）](https://bytedance.feishu.cn/docx/doxcnbiltrxuDXHkqYjzhULqr2g)

👍 [aidp 平台性能优化（webpack4）](https://bytedance.feishu.cn/docx/doxcnggZ5kOMo2WRO07Ws2uPY1b)

更多详见《[性能治理专栏](https://tech.bytedance.net/article/series/detail/7132725180245213192)》 ，如果有相关优化实践文章，欢迎投稿性能治理专栏，给其他小伙伴提供参考～