Step14. Webpack 构建项目进一步优化

webpack dll VS external

webpack 在打包后，生成的文件主要分为三种类型：
* 业务代码
* 外部依赖库
* webpack runtime

webpack 中的 **dll 和 external** 在本质上其实是解决的同一个问题：避免将某些外部依赖库打包进我们的业务代码，而是在运行时提供这些依赖。

一方面实现了代码拆分，以及依赖的复用，另一方面提升构建速度.

这两种方案应该是各有各的优劣，分别适用于不同的环境。

(1) dll 与 externals 的区别

dll 符合前端模块化的要求
webpack 配置上稍微复杂一些，需要预打包所需的 dll 资源，并在构建时配置相应的 plugin，
使用 dll 的前提是，这些外部依赖一般不需要发生变更。所以，如果某天发生了变更，那就需要将项目重新构建，违背了 dll 的使用前提，必然要作出相应的牺牲。
external 不太符合前端的模块化思想，所需要的外部库需要在浏览器全局环境下可访问
外部库升级的话，如果兼容之前的 API，不需要项目重新构建

webpack 配置上稍微简单些，但是同样需要将所需的外部库打包为所需要的格式，并在运行态下引用
相比较而言的话，dll 比 external 应该更加智能一些，主要体现在模块的引用和打包上。比如说如下方式去引用了 react 中的一个方法：

import AA from 'react/lib/createClass'

如果采用 dll 的方式，是不会造成重复打包的，他会将引用直接指向 dll。但是如果使用 external 的话，则会 react 中的部分代码打包进来。

(2) externals

防止将某些 import 的包(package)**打包**到 bundle 中，而是在运行时(runtime)再去从外部获取这些*扩展依赖(external dependencies)*。
例如，从 CDN 引入 jQuery，而不是把它打包
<script src="https://code.jquery.com/jquery-3.1.0.js""></script>
module.exports = {

```
//...
externals: {
  jquery: 'jQuery'
});
```

这样就剥离了那些不需要改动的依赖模块，换句话，下面展示的代码还可以正常运行：

```
import $ from 'jquery';
$('.my-element').animate(/* ... */);
```

DLLPlugin 和 DLLReferencePlugin 的使用

DLLPlugin 为什么会出现？

在使用 webpack 进行打包时候，对于依赖的第三方库，比如 vue，vuex 等这些不会修改的依赖，我们可以让它和我们自己编写的代码分开打包，这样做的好处是每次更改我本地代码的文件的时候，webpack 只需要打包我项目本身的文件代码，而不会再去编译第三方库，那么第三方库在第一次打包的时候只打包一次，以后只要我们不升级第三方包的时候，那么 webpack 就不会对这些库去打包，这样的可以快速的提高打包的速度。因此为了解决这个问题，DllPlugin 和 DllReferencePlugin 插件就产生了。

https://www.webpackjs.com/plugins/dll-plugin/#dllplugin

DLLPlugin

DLLPlugin 它能把第三方库代码分离开，并且每次文件更改的时候，它只会打包该项目自身的代码。所以打包速度会更快。
DLLPlugin 这个插件是在一个额外独立的 webpack 设置中创建一个只有 dll 的 bundle，也就是说我们在项目根目录下除了有 webpack.config.js，还会新建一个 webpack.dll.config.js 文件。
webpack.dll.config.js 作用是把所有的第三方库依赖打包到一个 bundle 的 dll 文件里面，还会生成一个名为 manifest.json 文件。
该 manifest.json 的作用是用来让 DllReferencePlugin 映射到相关的依赖上去的。

DllReferencePlugin 这个插件是在 webpack.config.js 中使用的，该插件的作用是把刚刚在 webpack.dll.config.js 中打包生成的 dll 文件引用到需要的预编译的依赖上来。什么意思呢？就是说在 webpack.dll.config.js 中打包后比如会生成 vendor.dll.js 文件和 vendor-manifest.json 文件，vendor.dll.js 文件包含所有的第三方库文件，vendor-manifest.json 文件会包含所有库代码的一个索引，当在使用 webpack.config.js 文件打包 DllReferencePlugin 插件的时候，会使用**该 DllReferencePlugin 插件读取 vendor-manifest.json 文件，看看是否有该第三方库。**
vendor-manifest.json 文件就是有一个第三方库的一个映射而已。

所以说 第一次使用 webpack.dll.config.js 文件会对第三方库打包，打包完成后就不会再打包它了，然后每次运行 webpack.config.js 文件的时候，都会打包项目中本身的文件代码，当需要使用第三方依赖的时候，会使用 DllReferencePlugin 插件去读取第三方依赖库。所以说它的打包速度会得到一个很大的提升。

DLLPlugin 和 DLLReferencePlugin 用某种方法实现了拆分 bundles，同时还大大提升了构建的速度。

(1)配置 webpack.dll.config.js:

```
var path = require("path");
var webpack = require("webpack");
var SRC_PATH = path.resolve(__dirname,'./src');
module.exports = {
 // 要打包的模块的数组
 entry: {
  vendor: ['vue/dist/vue.esm.js','vue-router']
 },
 output: {
  path: path.join(__dirname, './static/dll'), // 打包后文件输出的位置
  filename: '[name].dll.js',// vendor.dll.js 中暴露出的全局变量名。
  library: '[name]_library' // 与 webpack.DllPlugin 中的`name: '[name]_library',`保持一致。
 },
 plugins: [
  new webpack.DllPlugin({
   path: path.join(SRC_PATH,'./static/dll/[name]-manifest.json'),
   name: '[name]_library',
   context: __dirname
  }),
 ]
};
```

DllPlugin 插件有三个配置项参数如下:
context(可选)： manifest 文件中请求的上下文，默认为该 webpack 文件上下文。
name: 公开的 dll 函数的名称，和 output.library 保持一致。
path: manifest.json 生成文件的位置和文件名称。

上面就是 webpack.dll.conf.js 的主要配置。执行之后会在 static 文件夹（在 vue-cli 生成的项目中用于存放不需要 webpack 构建的静态文件【@vue/cli 中的目录名为 Public】）下生成两个文件夹（lib 文件夹和 mainfest 文件夹）。其中 lib 下的文件为我们已经打包好的组件库，mainfest 下的文件在引入项目时有用（是一个 JSON 文件）。

在 package.json 的 scripts 里加上：

"dll": "webpack --config webpack.dll.config.js",

运行 npm run dll 在 static 下生成 dll 文件夹,dll 文件夹里包括 js 文件和 json 文件

(2) 在 webpack.base.conf.js 里加上：

// 添加 DllReferencePlugin 插件

```
plugins: [
  new webpack.DllReferencePlugin({
    context: __dirname,
    manifest: require('./static/dll/vendor-manifest.json')
  })
],
```

DllReferencePlugin 项的参数有如下：

context: manifest 文件中请求的上下文。
manifest: 编译时的一个用于加载的 JSON 的 manifest 的绝对路径。
mainfest: 请求到模块 id 的映射(默认值为 manifest.content)
name: dll 暴露的地方的名称(默认值为 manifest.name)
scope: dll 中内容的前缀。
sourceType: dll 是如何暴露的 libraryTarget。

(3)在 index.html 中引入 vendor.dll.js：

由于动态链接库我们一般只编译一次，除非依赖的三方库更新，之后就不用编译，因此入口的 index.js 文件中不包含这些模块，所以要在 index.html 中单独引入。

① 第一种方式， 一种是根据打包的路径手动添加

```
<div id="app"></div><script src="./static/js/vendor.dll.js"></script>
```

② 第二种方式，用 add-asset-html-webpack-plugin 或者

html-webpack-include-assets-plugin 插入到 html 中，简单自动化

下面着重讲下 add-asset-html-webpack-plugin 与 html-webpack-include-assets-plugin 插件的使用，项目中使用 add-asset-html-webpack-plugin

安装大同小异

npm 安装：
npm install add-asset-html-webpack-plugin -D
npm install html-webpack-include-assets-plugin -D

Yarn 安装:

yarn add add-asset-html-webpack-plugin -D
yarn add html-webpack-include-assets-plugin -D

- add-asset-html-webpack-plugin 的使用

```
const AddAssetHtmlPlugin = require('add-asset-html-webpack-plugin');

module.exports = {
  ...,
  plugins: [
    ...,
    # 给定的 JS 或 CSS 文件添加到 webpack 配置的文件中，并将其放入资源列表 html webpack 插件注入到生成的 html 中。
    new AddAssetHtmlPlugin([
      {
        # 要添加到编译中的文件的绝对路径
        filepath: path.resolve(__dirname,'./static/dll/dll_vendor.js'),
        outputPath: 'dll',
        publicPath: 'dll',
        includeSourcemap: false
      }
    ])
  ]
}
```

- html-webpack-include-assets-plugin 的使用

```
const HtmlWebpackIncludeAssetsPlugin = require('html-webpack-include-assets-plugin');

module.exports = {
    ...,
    plugins: [
        ...,
        // 给定的 JS 或 CSS 文件添加到 webpack 配置的文件中，并将其放入资源列表 html webpack 插件注入到生成的 html 中。
    new HtmlWebpackIncludeAssetsPlugin(
    {
```

```
            assets: ['dll/asset.dll.js','dll/vendor.dll.js'],
            append: false
        }
      )
]
```

至此，配置之后的：

可以看到 npm run build 后的时间大幅度减少，在 dist 打包体积上也比之前的小。在项目优化中，可以很大程度上加快项目的构建速度和减少项目的打包体积。

使用 happypack 提升 Webpack 项目构建速度

提示：由于 HappyPack 对 file-loader、url-loader 支持的不友好，所以不建议对该 loader 使用。

webpack 打包哪一步最耗时？可能要数 loader 对文件的转换操作了，我们前面说过，我们使用 loader 将文件转换为我们需要的类型，文件数量巨大，webpack 执行又是单线程的，转换的操作只能一个一个的处理，不能多件事一起做。
我们需要 Webpack 能同一时间处理多个任务，发挥多核 CPU 电脑的威力，HappyPack 就能让 Webpack 做到这点，我们将需要通过 loader 处理的文件先交给 happypack 去处理，happypack 在收集到这些文件的处理权限后，统一分配 CPU 资源.

happypack 工作原理

happypack 通过 new HappyPack(),去实例化一个 HappyPack 对象，其实就是告诉 Happypack 核心调度器如何通过一系列 loader 去转换一类文件，并且可以指定如何为这类转换器作分配子进程。
核心调度器的逻辑代码在主进程里，也就是运行 webpack 的进程中，核心调度器会将一个个任务分配给当前空闲的子进程，子进程处理完后会将结果发送给核心调度器，它们之间的数据交换是通过进程间的通讯 API 实现的。
核心调度器收到来自子进程处理完毕的结果后，会通知 webpack 该文件已经处理完毕

参考：https://www.qdtalk.com/2018/11/16/webpack4plugin-2/

安装：

yarn add happypack --dev

使用：

```
// 引入 happypack
const HappyPack = require('happypack');

// 创建 happypack 共享进程池，其中包含 6 个子进程
const happyThreadPool = HappyPack.ThreadPool({ size: 6 });



module.exports = {
//省略部分配置
  module: {
    rules: [
      {
        test: /\.js$/,
        //把对.js 的文件处理交给 id 为 happyBabel 的 HappyPack 的实例执行
          use: 'happypack/loader?id=happyBabel',
        //排除 node_modules 目录下的文件，合理的使用排除可以事半功倍
        exclude: path.resolve(__dirname,'node_modules')
      },
    ]
  },
plugins: [
    new HappyPack({
        //用 id 来标识 happypack 处理那里类文件
      id: 'happyBabel',
      //如何处理 js 文件  用法和 loader 的配置一样
      loaders: [{
        loader: 'babel-loader?cacheDirectory=true',
      }],
       //使用共享进程池中的自进程去处理任务
      threadPool: happyThreadPool,
      //允许 HappyPack 输出日志，默认为 true
      verbose: true,
    })
  ]
}
```

在 Loader 配置中，所有文件的处理都交给了 happypack/loader 去处理，使用紧跟其后的 querystring ?id=babel 去告诉 happypack/loader 去选择哪个 HappyPack 实例去处理文件。

在 Plugin 配置中，新增了两个 HappyPack 实例分别用于告诉 happypack/loader 去如何处理 .js 和 .css 文件。选项中的 id 属性的值和上面 querystring 中的 ?id=babel 相对应，选项中的 loaders 属性和 Loader 配置中一样。

对应的参数

id：String
用唯一的标识符 id 来代表当前的 HappyPack 是用来处理一类特定的文件.
loaders: Array
用法和 webpack Loader 配置中一样.
threads: Number
代表开启几个子进程去处理这一类型的文件，默认是 3 个，类型必须是整数。
verbose: Boolean
是否允许 HappyPack 输出日志，默认是 true。
threadPool: HappyThreadPool
代表共享进程池，即多个 HappyPack 实例都使用同一个共享进程池中的子进程去处理任务，以防止资源占用过多。
verboseWhenProfiling: Boolean
开启 webpack --profile ,仍然希望 HappyPack 产生输出。
debug: Boolean
启用 debug 用于故障排查。默认 false。

https://blog.csdn.net/zgd826237710/article/details/88172290#_HappyPack_36

注意

注意，webpack4 中的 happypack 要使用 5.0.0 版本，如果你是从 webpack3 升级到 webpack4，记得升级 happypack
上面的 loader 中出现一个陌生词 cacheDirectory：
cacheDirectory 默认值为 false。
当有设置时，指定的目录将用来缓存 loader 的执行结果。之后的 webpack 构建，将会尝试读取缓存，来避免在每次执行时，可能产生的、高性能消耗的 Babel 重新编译过程(recompilation process)。如果设置了一个空值 (loader: 'babel-loader?cacheDirectory') 或者 true (loader: babel-loader?cacheDirectory=true)，loader 将使用默认的缓存目录 node_modules/.cache/babel-loader，如果在任何根目录下都没有找到 node_modules 目录，将会降级回退到操作系统默认的临时文件目录。