22 为你准备了一份 Webpack 工程化最佳实践总结

更新时间：2019-06-24 09:33:23

> 世界上最快乐的事，莫过于为理想而奋斗。
>
> ——苏格拉底

介绍了这么多 Webpack 的配置和优化项目，可能大家有点迷糊了，本小节将用小幅内容来总结下 Webpack 的最佳实践，让大家在实际项目中可以直接拿本篇文章的内容作为 CheckList 手册使用！

使用 NPM Scripts 来管理开发命令

1. 使用 NPM Scripts 来配置开发命令，即 `package.json` 的 `scripts` 字段，这样即使我们修改脚本甚至切换 Webpack 到其他的打包工具，对于团队其他成员来说，使用的命令还是不变的，建议的命令包括：
    1. `npm start`：相当于 `npm run start`，用于开发命令，快速启动本地开发服务；
    2. `npm run build`：用于生产环境打包；
    3. 其他命令，类似 `npm run test/lint` 等，根据相关的需要添加即可
2. 在 `package.json` 中使用cross-env来区分环境。

下面来看个示例：

```
{
    // ...
    "scripts": {
        "start": "cross-env NODE_ENV=development webpack --config webpack.config.dev.js --mode development",
        "build": "cross-env NODE_ENV=production webpack --config webpack.config.prod.js --mode producation",
        "analyzer": "cross-env NODE_ENV=production webpack --config webpack.config.analyzer.js --mode producation",
        "lint": "lint-staged"
    }
    // ...
}
```

Webpack 区分多环境配置

区分生产环境和开发环境配置，并且封装通用配置，即将 Webpack 配置文件分为：

1. 通用配置`webpack.config.base.js`；
2. 开发环境配置`webpack.config.dev.js`；
3. 生产环境配置`webpack.config.prod.js`；

`webpack.config.base.js`

通用配置`webpack.config.base.js`用于通用的配置，例如 entry、loader 和 plugin 等，但是有些需要根据`cross-env`传入`NODE_ENV`环境变量进行相关的配置，例如：`NODE_ENV=development`的时候使用`style-loader`，而`production`的时候使用mini-css-extract-plugin的 loader 将生产环境的 CSS 生成单独的 CSS 文件；

```
// webpack.config.base.js
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const isProduction = process.env.NODE_ENV === 'production';
module.exports = {
    // ...
    module: {
        rules: [
            {
                test: /\.css?$/,
                use: [
                    {
                        loader: isProduction ? MiniCssExtractPlugin.loader : 'style-loader'
                    },
                    {
                        loader: 'css-loader',
                        options: {
                            importLoaders: 1,
                            sourceMap: !isProduction
                        }
                    },
                    {
                        loader: 'postcss-loader',
                        options: {
                            sourceMap: !isProduction
                        }
                    }
                ]
            }
        ]
    }
    // ...
};
```

`webpack.config.dev.js`

开发环境配置`webpack.config.dev.js` 主要用于开发环境配置，主要是 devServer、API mock 等相关配置，这部分配置注重的是效率，所以打包速度优化也是很重要的。

`webpack.config.prod.js`

`webpack.config.prod.js`用于生产环境配置，这部分配置注重的是线上最优打包配置，包括`splitChunks`、压缩资源、CDN 路径配置（在`output`配置）等相关配置，还可以在`terser-webpack-plugin`配置中强制去除一些忘记删除的调试信息：例如 `debugger`、`alert`。

> Tips：生产环境打包不建议生成 sourcemap，如果生成了也不要上传到线上环境，因为如果 sourcemap 上线之后，等于别人就可以通过 Chrome 等工具直接查看线上代码的源码，这是十分危险的！但是如果你项目使用类似Sentry的 JavaScript 报错收集分析平台，可以将 sourcemap 经过 Webpack 生成，上传到对应的平台之后记得删除上线包中的这些文件，防止上传到线上！

`webpack.config.analyzer.js`

除了上面的三个配置文件之外，为了方便我们分析打包是否合理做了代码拆分，我们可以增加一个`webpack.config.analyzer.js`，这个配置实际是继承`webpack.config.prod.js`然后增加webpack-bundle-analyzer插件配置。

使用 **webpack-merge** 管理配置文件关系

Webpack 配置文件拆分之后，各自之间都有依赖关系，具体关系如下：

1. `webpack.config.dev.js`是合并了`webpack.config.base.js`和自己的配置；
2. `webpack.config.prod.js`合并了`webpack.config.base.js`和自己的配置；
3. `webpack.config.analyzer.js`合并了 `webpack.config.prod.js`和自己的配置，而`webpack.config.prod.js`又是来自于`webpack.config.base.js`。

要维护这个配置关系，那么就需要使用webpack-merge这个工具库，webpack-merge 主要是提供一个 Webpack 配置对象 Merge 函数，用来合并两个配置，类似于`Object.assign`函数的功能。

拿`webpack.config.analyzer.js`来看下`webpack-merge`怎么使用：

```
const merge = require('webpack-merge');
const prodWebpackConfig = require('./webpack.config.prod.js');
const {BundleAnalyzerPlugin} = require('webpack-bundle-analyzer');

module.exports = merge(prodWebpackConfig, {
    // 增加 webpack-bundle-analyzer 配置
    plugins: [new BundleAnalyzerPlugin()]
});
```

合理使用`splitChunks`

在使用`splitChunks`主要是为了合理的划分资源大小，提高缓存命中率，从而降低资源的加载时间，在划分合理性上一定要注意把握力度，太细不能充分利用 HTTP Cache，太粗又会导致加载速度慢，这个度不好笼统的来定义，但是一般来说可以按照下面三个原则来拆分代码：

1. 变更频次；
2. 页面 Router；
3. 动静分离。

变更频次

代码按照变更频次来使用`splitChunks`进行拆分，即将这些不经常修改的通用框架和库放到一起作为`common`代码，然后把业务代码按照页面间公共部分和私有部分进行拆分

页面 **Router**

不经常变动的框架和库代码拆分完之后，剩下的是业务代码，业务代码可以根据不同的页面之间公共代码拆分到一起，这样可以保证访问一个页面就可以将框架代码和公共代码缓存到浏览器中，再访问第二个页面就不会增加框架代码和公共代码页面请求了。

动静分离

动静分离指的是页面内使用频次不高或者需要**动态异步加载**（使用`import()`或者`require.ensure()`）的模块代码可以单独拆分到各自的 chunks，这样保证了页面首屏展现速度，还记得之前介绍过的一个 Case 是页面的播放器代码不经常用吗，那就是根据这个原则来拆分代码的。

另外类似 Vue、React 这类单页应用，页面 Router 之间的代码也是可以异步加载的，整个页面第一个入口就将大框架和当前页面的代码加载进来了，等点击跳到二级页面的时候只需要动态加载对应 Router 的代码即可。

多页应用 Entry 和 HTML 管理

这个之前在 《Webpack 中打包 HTML 和多页面配置》中提到过最佳实践，想继续了解的可以点击链接复习一下。

指定 chunk 的哈希值

在生产环境打包，一定要配置文件`filename`的`hash`，推荐`hash`配置规则如下：

1. JavaScript 文件使用：`[chunkhash]`；
2. CSS 文件使用：`[contenthash]`；
3. 其他静态资源用：`[hash]`，例如图片、字体等，在`url-loader`中配置`[hash]`

语法层面的最佳实践

1. 使用 ES6 Modules 语法，以保证 Tree-Shaking 起作用；
2. 合理使用 Ployfill，推荐使用`@babel/preset-env`的`useBuiltIns='usage'`方案，这部分可以参考 Babel 部分的 Polyfill 配置介绍Polyfill 配置介绍；
3. 合理使用 Webpack 的魔法注释（magic comments），比如：动态加载的模块要用`webpackChunkName`进行命名，还可以重要资源使用`webpackPrefetch`提前预加载；
4. 框架或者类库使用合理的版本，举例说明：

    1. Lodash 使用`lodash-es`版本，并且按模块使用；
    2. Momentjs 使用 date-fns代替，并且按模块使用；
    3. 移动页面使用 Zepto 代替 jQuery；
    4. Vue、San、React 这类框架库根据实际情况选择合适构建版本，以 Vue 为例，其实构建版本包含浏览器版本、ESM 版本、UMD 版本、完整版等多个版本，详细可以参考 Vue 构建版本部分的文档

其他 Webpack 配置的最佳实践

1. 生产环境使用mini-css-extract-plugin导出 CSS 文件；
2. 生产环境使用压缩功能，包括 JavaScript、CSS、图片、SVG 等；
3. 合理配置查找路径，减少查找时间，比如设置`alias`、添加项目路径、排查`node_modules`查找等；
4. 在 rule 配置上，有`test`、`include`、`exclude`三个可以控制范围的配置，最佳实践是:

    - 只在`test`和 文件名匹配 中使用正则表达式；
    - 在`include`和`exclude`中使用绝对路径数组；
    - 尽量避免`exclude`，更倾向于使用`include`。

5. icon 类图片文件太多使用 CSS Sprite 来合并图片，防止设置`url-loader`和`svg-url-loader`的 limit 值不合理，导致 icon 文件都以 Base64 方式引入 CSS 文件中，导致 CSS 文件过大。

其他方面最佳实践

1. 规范化 Git 工作流：
    1. 使用 Git Hook，类似 Husky这类 Git Hook 库，可以帮助我们在每次提交之前（`pre-commit`）自动做 lint 检查；
    2. 使用Commitizen来规范 Git 的提交 Commit Log；
2. 组件化开发，公共 UI 组件、公共函数库建设；这里说的比较抽象，需要具体项目来划分，例如我们可以将多个页面常用的 UI 组件抽象出来；也可以将通用的工具函数库建设起来，类似 Lodash 这类库；
3. 选择一个顺手的 CSS 预处理语言，Sass、Less、Stylus，只需要团队使用顺手即可；
4. 指定规则约定，包括代码规范、目录结构，文档规范等；
5. 前后端分离，选择合适的 Mock 方案；
6. 将最佳实践做成标准项目的脚手架，新项目使用脚手架工具来创建；
7. 抽象解决方案，融合到 Webpack 配置中，甚至是基于 Webpack 做自己的最佳实践工具链！这个部分在实战篇会用具体案例来介绍一些实用的解决方案。

总结

本小节主要是对之前的课程进行总结，是一篇系统性的最佳实践总结，通过本篇文章可以让你回顾之前所有的知识点，并且针对关键的知识点进行回顾，本篇文章也是一本操作手册，拿着这个手册来检查自己的项目是否还有可以改进的地方。本篇文章不再是从「配置工程师」角度来学习 Webpack 的配置，而是升华到工程化解决方案的角度，更是起到承上启下的作用，想要打造自己的工程化解决方案，需要掌握基础，然后深入内核原理，最后结合项目实践做到融会贯通，才能利用所学知识解决业务痛点。希望本文可以让大家学会多总结、多抽象业务问题的解决方案的好习惯。

> 本小节 Webpack 相关面试题：
>
> 你有什么 Webpack 的最佳实践可以介绍吗？

21 使用 Webpack 的 Tree-Shaking

怎么调试 Webpack？