PageLoader记录分析

2020年1月27日 17:47

1. 源代码分到两个目录中：public和app
2. app目录中的代码基于node模块管理，通过brunch编译到public目录的app.js 文件，在loader.ts文件中，设置了window上的全局变量，暴露了一系列给public目录中文件使用的接口
3. public目录中的ts文件采用无模块化方式，由tsc编译后直接被按顺序引入html中
4. 采用这种架构的原因
    a. 不会使用webpack，而brunch使用typescript直接加载失败（原因不明），tsc编译与brunch实时监控冲突，因此需要手动编译，为尽量减少重复编译的机会，把不需要依赖node中安装的模块的代码移动到了public目录进而跳过brunch的编译，修改后可以直接刷新测试

修改方案

- 把public目录中的文件的模块管理方式改为commonjs ，解决方案改为node
- 复制public目录中的文件到app目录下的browser文件夹中
- 把public目录中对window上全局暴露的api的引用改为从模块引入
- 修改webpack配置文件，添加tsloader
- 安装webpack，设置webpack配置文件，添加app目录中的一个entry.ts为全局入口，引入模块并暴露关键api，如loadPage，在webpack配置文件中添加entry.ts为入口文件
- 卸载brunch
- 修改index.html中的引入方式，最终只引入webpack编译好的app.js文件
- 进行最终测试

进一步修改方案

- 把html文件和assert目录移动到app目录，通过webpack的插件实现自动复制（需要研究）

引入方案

关于如何将pageloader引入blogsystem中的问题

1. pageloader作为blogsystem的一个helper存在，独立作为一个文件夹，放置在Helper/Loader 目录里
2. pageloader的node模块独立配置，考虑复制package.json过来并安装依赖（这也将作为helper的基本标准）
3. pageloader最终编译为其自身目录的index.js文件，由HelperManager扫描所有helper目录，并使用webpack动态加载加载helper目录中的index.js文件，得到其导出的load函数，放入表中供前端使用
4. 由于helpers最终会被前端site使用，其会被复制到根目录的helpers.js文件中，由site在html中引用，webpack动态加载将不能找到位置（app目录不提交），因此需要研究如

何让webpack在打包阶段就执行动态引用的问题

5.