

关于landing-pages项目发布时涉及js解析

- 项目结构
- 发布流程
- js解析

项目结构 —— monorepo

Monorepo 是管理项目代码的一个方式，指在一个项目仓库 (repo) 中管理多个模块/包 (package)，不同于常见的每个模块建一个 repo。

目前有不少大型开源项目采用了这种方式，如 babel, create-react-app, react-router

这些项目的第一级目录的内容以脚手架为主，主要内容都在 `packages` 目录中、分多个 package 进行管理。

```
├── packages
│   ├── pkg1
│   │   └── package.json
│   ├── pkg2
│   │   └── package.json
└── package.json
```

最常见的 monorepo 解决方案是 [Lerna](#) 和 yarn 的 [workspaces](#)，landing-pages 则是两者结合使用，一方面是不管你包管理器是 yarn 或 npm，都能发挥 monorepo 的作用，如果包管理工具是 yarn，则 lerna 会把依赖安装交给 yarn 处理

Landing-pages lerna.json

```

{} lerna.json > ...
1   {
2     "lerna": "2.10.1",
3     "useWorkspaces": true,
4     "command": {
5       "publish": {
6         "scope": "@lp-app/*",
7         "message": "Staged applications"
8       }
9     },
10    "version": "independent"
11  }
12

```

version: `independent` 表示各个页面版本号独立的，其他具体属性值可参看官网了解

Yarn workspaces则是在根目录下package.json下增加

```

"private": true,
"workspaces": [
  "packages/*"
],

```

Private:

根目录一般是项目的脚手架，无需发布，`"private": true`会确保根目录不被发布出去。

Workspaces: 声明workspace中package的路径

发布流程

发布命令 `yarn deploy:stageApplications qa` 执行了 `scripts`文件夹下的`stage.sh`

```

scripts > stage.sh
1  # Staging preparation
2  yarn install --frozen-lockfile && \
3  yarn bump && \
4  UNION_DEPLOYMENT_ENV=qa yarn build --env.deploymentEnvironment=qa && \
5  UNION_DEPLOYMENT_ENV=production yarn build --env.deploymentEnvironment=production && \
6
7  # Actual staging
8  yarn deploy.stageApplications.doStaging --deploymentEnvironment=qa && \
9  yarn deploy.stageApplications.doStaging --deploymentEnvironment=production
10

```

原来的发布流程：

- 1, 安装包依赖
- 2, 执行yarn bump命令（选页面，版本号）
- 3, 构建qa
- 4, 构建prod
- 5, stage全部页面

Yarn bump: `yarn -s node.babel $(which lp-project-version-bump)`

运行projects/libraries/project-version-bump下的js文件，会在终端出现选择操作，选择需要发布的页面，然后会更新对应页面下的package.json, 并且进行git操作

```

yarn run v1.22.5
$ yarn -s node.babel $(which lp-project-version-bump)
? Choose your projects (Press <space> to select, <a> to toggle all, <i> to inverse selection)
> ☐ cooking-with-cuisinart
  ☐ get-started1
  ☐ get-started2
  ☐ get-started3
  ☐ guest-list-manager-1
  ☐ guest-list-manager
  ☐ non-discrimination
(Move up and down to reveal more choices)

```

原先的发布流程是执行完这一系列操作后，就直接进入构建页面的环节，并没有把我们选中的要发布的页面传到webpack.config.deployment.js，所以导致了发布时间非常之久，就是因为webpack.config.deployment.js中把workspaces指定路径下的所有包全部构建了一遍，即使那些我们不需要发布到的

如今的发布流程：

- 1, 安装包依赖
- 2, 执行yarn bump命令（选页面，版本号）
- 3, 把选中的项目名写入buildList.json文件
- 4, webpack.config.deployment.js中读取buildList.json文件
- 5, 构建qa
- 6, 构建prod
- 7, 只stage选中的页面

(ps: 可以在landing-pages下doc文件夹查看项目架构作者的一个思路和命令解释)

Js解析

- projects/libraries/read-projects-info - 项目包信息
- projects/libraries/cli-prompts - 终端交互
- projects/libraries/project-version-bump - 更新版本号
- Webpack.config.deployment.js

projects/libraries/read-projects-info

Projects/index.js

主入口, 获取projects下每个页面和js模块信息

核心js方法

```
async getAllWorkspacesWhere(filterCallback) {  
  const { stdout: rawInfo } = await exec('yarn -s lerna ls --json');  
  const parsedInfo = JSON.parse(rawInfo);  
  const workspaceNames = parsedInfo.map(({ name }) => name);  
  
  return Promise.all(workspaceNames  
    .map((name) => {  
      const location = getLocationForPackage(name);  
  
      return {  
        ...parseProjectName(name),  
        location,  
      };  
    })  
    .filter(filterCallback)  
    .map(async data => ({  
      data,  
      packageJson: await getPackageJson(data.location),  
    })))  
},
```

通过执行 `yarn -s lerna ls --json` 获取到每个package信息

```
[  
  {  
    "name": "@lp-app/cooking-with-cuisinart",
```

```
    "version": "0.1.1",
    "private": true
  },
  ....]
```

getLocationForPackage(name)

根据传入 packageName 获取到package所在的绝对路径

返回值如：'/Users/ili/work/landing_pages/projects/apps/get-started1'

parseProjectName(name)

把传入的packageName进行type区分 返回值如下

页面：

```
{ type: 'app',
  fullName: '@lp-app/get-started1',
  scope: '@lp-app',
  name: 'get-started1',
  location: '/Users/ili/work/landing_pages/projects/apps/get-started1'
}
```

Js模块：

```
{ type: 'script',
  fullName: '@lp-script/gating-service-worker',
  scope: '@lp-script',
  name: 'gating-service-worker',
  location:
    '/Users/ili/work/landing_pages/projects/workers/gating-service-worker' }
```

Filters

是按type或name过滤package信息

getPackageJson(location)

根据传入的路径，获取到相关页面或js模块下 package.json信息并返回

packageJson:

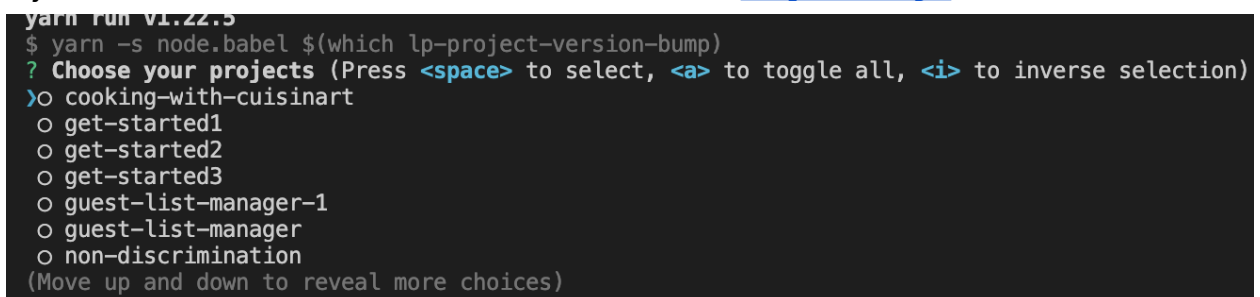
```
{ name: '@lp-app/cooking-with-cuisinart',
  private: true,
  version: '0.1.1',
  segmentWriteKey: [Object],
  'x-application': [Object],
  dependencies: [Object] }
```

最终返回值如下

```
[ { data:
  { type: 'app',
    fullName: '@lp-app/cooking-with-cuisinart',
    scope: '@lp-app',
    name: 'cooking-with-cuisinart',
    location:
      '/Users/ili/work/landing_pages/projects/apps/cooking-with-cuisinart' },
  packageJson:
    { name: '@lp-app/cooking-with-cuisinart',
      private: true,
      version: '0.1.1',
      segmentWriteKey: [Object],
      'x-application': [Object],
      dependencies: [Object] },
  ...}]
```

projects/libraries/cli-prompts

该js实现了，在终端选择项目的交互操作，主要调用了[Inquirer.js](#)，如下效果



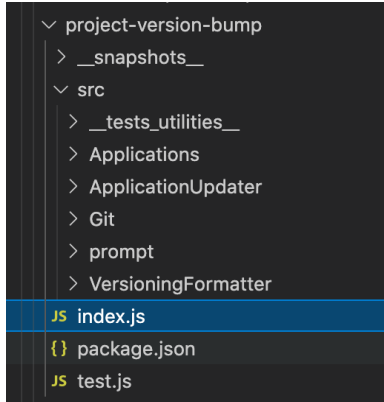
```
yarn run v1.22.5
$ yarn -s node.babel $(which lp-project-version-bump)
? Choose your projects (Press <space> to select, <a> to toggle all, <i> to inverse selection)
> ☒ cooking-with-cuisinart
  ☐ get-started1
  ☐ get-started2
  ☐ get-started3
  ☐ guest-list-manager-1
  ☐ guest-list-manager
  ☐ non-discrimination
(Move up and down to reveal more choices)
```

该js返回了选中的项目名，此处不多介绍该js，请自行参看github了解

projects/libraries/project-version-bump

该js在发布流程中起到更新项目版本号作用，是构建项目目前的核心部分

该目录结构如下，入口js为index.js



下面分析请结合项目代码理解

首先, `require.main === module`判断了执行当前js的环境是否来自node.js环境

```
if (require.main === module) {  
  main().catch((error) => {  
    console.error(error);  
    process.exitCode = 1;  
  });  
}
```

再执行了异步方法`main`，在该方法中实现功能如下

1. 获取所有项目信息
2. 在终端显示选择项目操作
3. 写入buildList.json
4. 项目版本号更新
5. git操作

1, 获取所有项目信息

```
const applications = Applications.withPackageJsonPaths(  
  await Projects.getAllProjects(),  
);
```

该步骤中调用了`projects/libraries/read-projects-info`的`Projects`方法获取所有项目包信息，详细请查看上面的解析，获取到的返回值后传入`Applications.withPackageJsonPaths`中，该方法为每个项目包信息对象中加入了`packageJsonPath`属性，值是每个项目的`package.json`的绝对路径，作用是为了后面更新项目版本号提供路径

2, 在终端进行项目选择操作

```
const requestedApplicationUpdates = await prompt(applications);
```

在prompt/index.js中首先调用了projects/libraries/cli-prompts, 在终端中选择所要发布的项目名称, 并选择项目要发布的版本号, 也是调用了Inquirer.js

```
? Choose a bump strategy for cooking-with-cuisinart -- Current version is 0.1.1 (Use arrow keys)
> major: 1.0.0
  minor: 0.2.0
  patch: 0.1.2
  premajor: 1.0.0-0
  preminor: 0.2.0-0
  prepatch: 0.1.2-0
  prerelease: 0.1.2-0
  (Use arrow keys to move between options)
```

此处也调用了semver.js, 一个语义化版本号管理的模块, 可以实现版本号的解析和比较, 规范版本号的格式, 比如 当前版本号是0.1.1, 那它的下一个版本major情况是1.0.0, minor版本是0.2.0, semver就是提供这样的一个规范格式, 其alpha号是取当前git log的第一个提交的哈希值

3, 写入buildList.json

题外话: 在原先的发布流程中是没有该步骤的, 但是为了解决只发布选中的项目, 增加了该步骤。在一开始的想法中, 原本是打算通过获取终端中交互选中的值传入环境变量, 但是经过多次尝试, 并不能很好实现, 故放弃, 选择了通过读写文件的方式传值。

```
function writeFs(appsToBump) {
  let params = {
    list: appsToBump,
  };
  params = JSON.stringify(params);
  const root = process.cwd();
  const listPath = path.join(root, 'buildList.json');
  // eslint-disable-next-line no-console
  write(listPath, params, err => console.error(err));
}
```

在该方法中实现了把之前命令行操作中选中的项目名字, 写入到根目录下的buildList.json文件中, 在之后构建项目跑webpack.config.deployment.js时再读取出来按需发布。该文件只会存在本地中, 不会被上传到github

4, 更新项目版本号

在上述操作后, 会最终在主入口获取到选中要发布项目的信息和新的版本号, 如下

```
[[ { data: [Object],
```



```
packageJson: [Object],
packageJsonPath:
  '/Users/ili/work/landing_pages/projects/apps/cooking-with-cuisinart/package.json' },
'0.1.2-31490a78.0' ]]
```

如果未选中，则退出操作；

更改项目版本号的方法：

```
updateVersions(applicationUpdates) {
  return Promise.all(applicationUpdates.map(([application, newVersion]) => {
    const { packageJson, packageJsonPath } = application;

    const newPackageJson = {
      ...packageJson,
      version: newVersion,
    };

    return write(packageJsonPath, JSON.stringify(newPackageJson, null, 2));
  }));
}
```

由前面一开始获取到的项目package.json的绝对路径，然后重新复写该文件来实现修改项目版本号

5, git操作

在完成了上述操作后，会在把前面复写了的package.json文件在本地提交，打标签

在**Git/index.js**中调用了[child_process](#) 和 [util](#), `child_process`主要作用是在node.js中执行一些shell操作，`util`则把原来的异步回调方法改成返回 Promise 实例，具体的可点击链接进行查看

```
import { promisify } from 'util';
import child from 'child_process';

const exec = promisify(child.exec);
```

```
const shouldPush = await promptToPush();
```

当执行到此处时，会在终端跳出选择？Should I push the tags for you? (Y/n) 如果选择了y, 则会把本地提交直接push到GitHub上，所以在本地尝试发布的时候，记得选n

当执行完此处操作后，yarn bump的操作就到此结束了，发布的操作就会跳到webpack.config.deployment.js构建项目

Webpack.config.deployment.js

- 1, 读取buildList.json
- 2, 配置webpack的一些plugins
- 3, 传入build-system的standardBuildConfig进行打包构建

读取buildList.json，获取待构建项目名，赋值给env.projectAllowlist, 等待后续的发布

```
const root = process.cwd();
const listPath = path.join(root, 'buildList.json');
let buildList;
try {
  const data = fs.readFileSync(listPath, 'utf8');
  buildList = JSON.parse(data).list;
} catch (err) {
  // eslint-disable-next-line no-console
  console.error(err);
}
```

配置webpack的一些plugins:

```
const plugins = [
  new ExtractTextPlugin({
    ignoreOrder: true,
    filename: '[name].css',
  }),
  new CleanPlugin(path.join('dist', deploymentEnvironment)),
  new DotEnvPlugin({
    path: path.join(__dirname, `..env.${deploymentEnvironment}`),
    systemvars: true,
  }),
];

if (compress) {
  plugins.push(new CompressionPlugin());
}
```

extract-text-webpack-plugin: 它会将所有的入口 chunk(entry chunks)中引用的 *.css, 移动到独立分离的 CSS 文件。因此, 样式将不再内嵌到 JS bundle 中, 而是会放到一个单独的 CSS 文件当中。

Clean-webpack-plugin: 每次构建时删除dist/[qa | prod]文件夹

Dotenv-webpack: 将环境变量从.env文件加载到process.env中, 方便统一管理各个环境下的全局变量和切换环境

compression-webpack-plugin: 压缩静态资源

其他补充

构建中其他问题及解决方案:

- <https://theknotww.atlassian.net/browse/ANG-17369> (只stage选中的)
- <https://theknotww.atlassian.net/browse/ANG-17397> (发布流程优化)
- <https://theknotww.atlassian.net/browse/ANG-17312> (union picture升包)