从0到100搭建React Native组件库

黑湖科技移动端 LV.3
2022年10月27日 22:32 · 阅读 264

摘要：

本文是对市面上主流组件库开发方式对比分析后，结合公司业务和开发团队实际情况，经过大量实践后沉淀出来的从0到100搭建React组件库的方法论和操作指南。

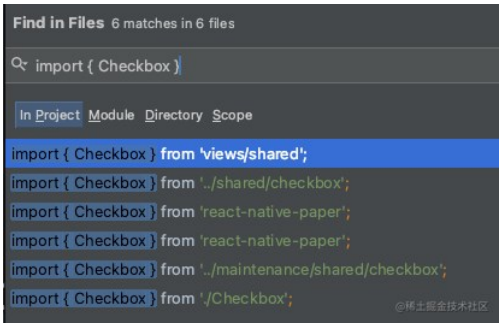
本组件库（下文带指laker-ui-demo）采用的技术栈是**React+React Hooks+Typescript+Expo+React Native Elements+Storybook+Eslint+Husky+Jest+Enzyme+rollup+npm+CICD**，融合了最近的开发语言TS，前端使用范围最广的React框架，Expo平台作为测试项目，Storybook实现组件独立开发调试预览环境，在React Native Elements基础上开发内部组件，Eslint对代码规范进行约束，Husky对代码提交规范进行约束，使用Jest+Enzyme完成单元测试功能，利用Rollup的打包速度快、配置简单等优势实现打包能力，使用npm进行包的管理，使用gitlab CICD实现全流程的持续集成和持续交付。

一、背景

1.为什么需要一个组件库？

刚刚接手公司维护了几年的主项目，引入眼帘有如下问题：

1. APP的UI一致性差：随着迭代时间的拉长，随着设计师的更换，APP页面、组件之间出现较大差异，不同时期开发的功能页面相差甚远，比如header、按钮、弹窗等均不同风格的样式。
2. 开发效率低：我自己在做第一个需求的时候，UI上需要一个checkbox，我搜了一下项目中已有的checkbox，我完全无法知道应该选择哪个，只能一个一个尝试，并去看代码解读其中的差异。因此一个最简单的UI组件我需要花很长时间去调研可用性。



3. 缺乏扩展性和复用性：入职一个月后，我们准备开始一个新APP的开发，但是我发现老项目几乎没有可以复用的资产，在新项目的开发过程中，所有的UI组件都需要从第1行代码重新开发。

2.希望组件库具备的要素

1. 规范、科学、友好的Api设计，减少使用者的学习成本。
2. 设计灵活，可扩展性强，方便维护。
3. 便捷的预览，详细的文档，方便查阅api和用法，详尽的代码示例。
4. 对业务的良好抽象，能够支撑大多数业务场景，同时和具体业务有必要的隔离，不处理特定业务逻辑。
5. 没有渲染和性能问题，这是底线，不能因为一个基础组件拖垮页面乃至整个APP。

3.组件库的设计原则和思路

1. 扩展性：在原有组件基础上可 二次封装 扩展成新的组件符合设计的 开闭原则
2. 通用性：根据组件接受的 参数 和 组件中与业务的解耦比 来衡量组件的通用性,并不是通用性占比100%的组件就是最好的组件，需要根据 不同的场景 分析
3. 健壮性：避免组件中 参数处理、函数执行过程 及 重复计算渲染 可能出现的奔溃和错误导致程序的直接挂断， 单测 以对组件内部 做好 边界处理 ，异常 错误的捕获 来衡量这一标准

二、技术栈

React: 用于构建用户界面的 JavaScript 库；

TypeScript：当下最火的前端语言，是JavaScript类型的超集；

Expo：Expo 是一个开源平台，用于使用 JavaScript 和 React 为 Android、iOS 和 Web 制作通用原生应用程序。作为组件库的测试项目集成组件库代码。

React Native Elements：一套企业级 UI 设计语言和 React 组件库；

Storybook：UI component explorer for frontend developers，辅助 UI 控件开发的工具，通过story创建独立的控件，让每个控件开发都有一个独立的开发调试环境；

Eslint & Husky - Git hooks：统一团队代码风格，规范代码编写和提交；

Jest：JavaScript 测试框架，用于组件库的单元测试；

Enzyme：一个用于 React 的 JavaScript 测试实用程序，可以更轻松地测试 React 组件的输出。您还可以在给定输出的情况下操作、遍历并以某些方式模拟运行时。Enzyme 的 API 旨在通过模仿 jQuery 的 API 来进行 DOM 操作和遍历，从而变得直观和灵活。

Rollup：一个 JavaScript 模块打包器，可以将小块代码编译成大块复杂的代码，例如 library 或应用程序。Rollup 对代码模块使用新的标准化格式，这些标准都包含在 JavaScript 的 ES6 版本中，而不是以前的特殊解决方案，如 CommonJS 和 AMD。ES6 模块可以使你自由、无缝地使用你最喜爱的 library 中那些最有用独立函数，而你的项目不必携带其他未使用的代码。ES6 模块最终还是要由浏览器原生实现，但当前 Rollup 可以使你提前体验。

What is CI/CD?：CI/CD 属于 DevOps（开发和运维的结合），结合了持续集成和持续交付的实践。CI/CD 自动化了传统上将新代码从提交到生产中所需的大部分或全部人工干预，例如构建、测试和部署，以及基础设施配置。使用 CI/CD 管道，开发人员可以对代码进行更改，然后自动测试并推送以进行交付和部署。正确获取 CI/CD 并最大限度地减少停机时间并更快地发布代码。

三、新建项目

1. monorepo和单package的取舍

- Monorepo（Monolithic Repositories）是目前比较流行的一种将多个项目的代码放在同一个库统一管理的代码管理组织方式，这种方式能够比较方便地进行版本管理和依赖管理，通常配多个 package管理的工具是 lerna 或者 yarn/npm + workspace。安装这种方式管理组件的话，每个组件都会打单独的npm包，有20个组件就会打20个包；用户在使用时按需安装自己使用的组件。比如@blacklake/laker-ui-icon，@blacklake/laker-ui-button，@blacklake/laker-ui-text等。
- 单package是指组件库整体会被打成一个npm包@blacklake/laker-ui，里面包含所有组件。业务方在使用时只需要安装一个即可；

在选择两者时考虑到如下因素：

1. 组件之间会有一些必然的依赖关系，比如button中必然会使用到text，bottomsheet中使用到button。
2. 因为我们开发的组件库核心用途不是为了开源，而是公司内部项目使用，所以理论上我们沉淀的组件就是为主项目服务的，不存在部分使用的场景。
3. 基于上述考虑，我们决定采取**单package**的方式。整个组件库打成一个npm包，既便于处理组件之间的互相依赖，又避免主项目安装过多依赖。

2.组件库开发测试方案-Expo引入

决定使用单package的方案，接下来解决的问题是如何便捷地开发测试。

1. 一开始想到的方案是，新建一个RN项目，src目录是源码，项目本身是测试项目，可以安装到手机上调试组件。但是该方案有两个缺点：1. 考虑到希望把组件库的内容部署到web端预览，RN项目目前无法支持；2. 涉及到原生组件的改动，需要频繁下载安装测试APP，开发效率较低。

2. 查阅相关资料后，决定使用**Expo**作为组件库项目底座。官网的描述就是这样的：Expo 是通用 React 应用程序的框架和平台。它是一组围绕 react-native 和原生平台构建的功能，工具和服务，可以让你用同样的 JS/TS代码，在 iOS、Android 和 web 应用程序上开发、构建、部署和快速迭代。可以看出expo解决的问题正是我们非常关切的问题，非常适合作为组件库开发的基座，我们只需要专注于组件本身的开发，其他测试、构建、部署等工作均有expo处理，开发者不需要投入任何精力。

expo 提供了四种工具帮助我们更好的开发。

- `client` 手机端 app，用于项目的实时预览
- `expo-cli` 命令行工具，开发，构建，部署
- `snack` 沙盒工具，实时编辑，在线预览
- `SDK` 开发工具包，版本号随 RN 版本号更新

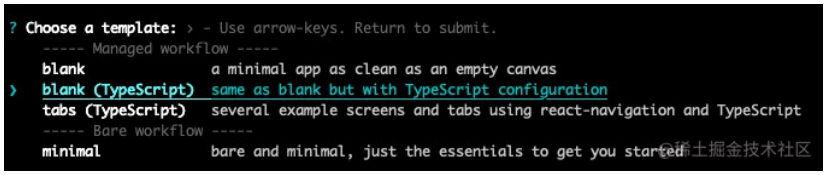
3.新建Expo项目

1. 首先，全局安装 `expo-cli` 脚手架

```
npm install expo-cli --global
```

2. 初始化项目，会让我们选择模板，我们选择 blank 空白模板

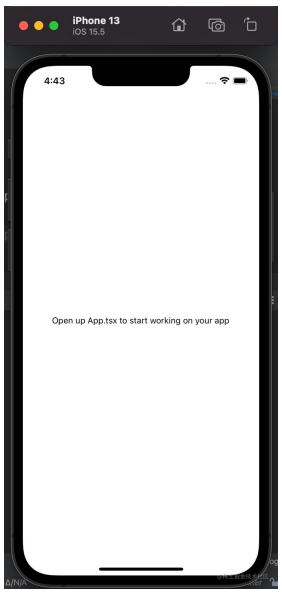
```
expo init laker-ui-demo
```



选择 blank with TypeScript configuration

3. 进入到项目目录，启动服务

```
yarn run start

输入 a，可以在android设备中运行
输入 i，可以在iOS模拟器中运行
输入 w，可以在web浏览器中运行
```



4. 组件分类

综合项目需要和对组件的理解，组件可以分为两类：基础（通用）组件（下称基础组件）和业务组件，基础组件就是通常意义上的完全UI组件，不包含任何含义，通俗来说就是任何公司任何项目都会使用的组件，比如button、text、checkbox等；业务组件就是和公司业务密切相关的组件，但是又不会掺杂过多业务逻辑的组件，更多是UI显示方面的，即数据固定、样式固定、使用方式固定，该类组件其实也是基于基础组件封装的，增加了部分业务含义。比如数量输入、联系人选择、仓位选择等。

5. 开发方式

作为一个从0开始的项目，我们有两种方式开发所有组件：

- 第一种是全部自主开发，从最基础的Button、Text开始，设计全套的API，逐个开发完成。
- 第一种是在一些成熟的第三方组件库基础上，结合自身产品UI规范进行一定程度的修改和扩展，保持API层面的基本沿用，来完成大部分组件的开发。

经过广泛调研和讨论，市面上已经有相对成熟的组件库供我们采纳借鉴，决定不再重复造轮子，采取第二种方案。对比分析了如下几种RN组件库：

Ant Design for React Native

React Native Paper

NativeBase: Universal Components for React & React Native

React Native Elements

App Builder: Create Mobile Apps | No-Code App Maker

UI Kitten - React Native UI Library based on Eva Design System

市面上还有很多其他React Nativge的组件库，各自都有优劣势，我们对比分析了其文档、版本更新频率、github star数、社区讨论范围等因素，最终采纳了React Native Elements作为基础库，在此基础上结合公司的UI规范和产品功能，拓展出自己的一套组件库。

值得一提的是，上述组件库各自都有相应的优劣势，没有好坏之分，只需要选择其中一个即可。具体的扩展能力还需要我们自行开发和维护。

6. README

一个好的项目一定要有一个好的README，让任何项目的使用者、开发者都可以第一时间获取非常明确的概要信息。本项目的README主要包含以下内容：

> 1、概述
>
> 2、运行项目
>
> 3、安装项目注意
>
> 4、参与贡献须知
>
> 5、新建分支
>
> 6、创建文件须知
>
> 7、组件命名规范
>
> 8、更新本地缓存的资源
>
> 9、本地联调
>
> 10、提交与发布代码

四、组件预览

1. 如何使用storybook实现预览

项目新建完成后，接下来需要思考的问题是，如何更高效地对组件进行预览、调试、文档撰写等，经过多方面调研，我们找到了**storybook**，这是一个非常优秀的开源工具，用于帮助前端组件开发，可以支持React, Vue和Angular等框架。正如它的名字，使用它可以写出组件库的活的用户故事（user story）。它的网站首页这样介绍的：storybook 为UI组件提供一个独立的沙箱环境，在这里无论是edge case还是难于遇到的状态都可以构建出来。可以把use case 像故事一样展现出来。

如果没有这样的工具，想单独测试一个组件的行为和表现，我们应该怎么做？大概率会建一个测试工程，写个测试页面，把组件扔进去，围绕不同的case写非常多的测试组件，再启动测试服务器，打开调试窗。

有了Storybook, 就可以变得非常高效了，通过不同的story构建不同的case，并且这些story不是仅仅用作测试代码就没有用了，它们还可以用作单元测试UT，也可以用于生成文档。这样我们写的每一行代码都可以变得更加有价值。

2. 如何安装？

核心库：@storybook/react-native

v5.3 参考：GitHub - storybookjs/react-native at v5.3.25

v6beta参考：GitHub - storybookjs/react-native: 📖 Storybook for React Native!

本项目以v6作为实例：

1. 安装如下依赖

注意安装next的tag，（笔者写这篇文章时还在beta，后期发布正式版可安装正式版，注意如果后续操作全部完成后，启动项目报一些storybook内部错误，带概率就是如下依赖版本不适配的问题，期待正式版的推出可以稳定安装运行。）

```java 复制代码
expo install @storybook/react-native@next @react-native-async-storage/async-storage react-native-safe-area
expo install @storybook/addon-ondevice-controls@next @storybook/addon-ondevice-backgrounds@next @storybook/
expo install @storybook/addon-actions@next
expo install @react-native-community/slider @react-native-community/datetimepicker
```

2. 根目录下新建.storybook目录和组件目录 components

（为什么要命名带「.」，github的issues中介绍到历史原因，详情可自行查阅storybookjs/react-native/issues/158）

3. .storybook目录下新建三个文件，分别是

```java 复制代码
//main.js
//这里需要解释一下，stories的目录是写组件代码和storybook代码，/**/是希望组件会有一个文件夹名，下文会详细介绍
module.exports = {
  stories: [
    '../components/**/*.stories.?(ts|tsx|js|jsx)'
  ],
  addons: [
    '@storybook/addon-ondevice-notes',
    '@storybook/addon-ondevice-controls',
    '@storybook/addon-ondevice-backgrounds',
    '@storybook/addon-ondevice-actions',
  ],
};
```

```javascript 复制代码
//preview.js

import {withBackgrounds} from '@storybook/addon-ondevice-backgrounds';
export const decorators = [withBackgrounds];
export const parameters = {
  backgrounds: [
    { name: "plain", value: "white", default: true },
    { name: "warm", value: "hotpink" },
    { name: "cool", value: "deepskyblue" },
  ],
  controls: {
    matchers: {
      color: /(background|color)$/i,
      date: /Date$/,
    },
  },
};
```

```javascript 复制代码
//Storybook.tsx

import { getStorybookUI } from '@storybook/react-native';
import './storybook.requires';
const StorybookUIRoot = getStorybookUI({});
export default StorybookUIRoot;
```

4. 修改APP.tsx：

```javascript 复制代码
import StorybookUIRoot from "./.storybook/Storybook";
export { StorybookUIRoot as default };
```

5. 根目录下新建metro.config.js文件

```ini 复制代码
// metro.config.js

const { getDefaultConfig } = require('expo/metro-config');
const defaultConfig = getDefaultConfig(  __dirname);
defaultConfig.resolver.resolverMainFields = [
  'sbmodern',
  ...defaultConfig.resolver.resolverMainFields,
];
defaultConfig.transformer.getTransformOptions = async () => ({
  transform: {
    experimentalImportSupport: false,
    inlineRequires: false,
  },
```

```
});
defaultConfig.watchFolders = [...defaultConfig.watchFolders, './.storybook'];
module.exports = defaultConfig;
```

6. 修改package.json文件

```json
// 新增如下两个脚本
"scripts": {
  "prestart": "sb-rn-get-stories",
  "storybook-watcher": "sb-rn-watcher"
},

注意:
如果上文建立的文件夹名不是 .storybook, 而是其他, 比如 .abc, 那么脚本应该为
"scripts": {
  "prestart": "sb-rn-get-stories --config-path .abc",
  "storybook-watcher": "sb-rn-watcher --config-path .abc"
},
```

3. 如何写代码？

以一个最基础的组件Button举例:

1. 新建组件文件夹，大驼峰命名

components目录下新建一个文件夹，文件夹内存放组件本身的文件和story文件。举例来说，新建按钮组件，文件夹取名为Button。

2. 文件夹下新建文件:Button.tsx，用于书写组件代码

```typescript
//Button.tsx

import React from 'react';
import {TouchableOpacity, Text, StyleSheet} from 'react-native';
interface MyButtonProps {
  onPress: () => void;
  text: string;
}
export const MyButton = ({onPress, text}: MyButtonProps) => {
  return (
    <TouchableOpacity style={styles.container} onPress={onPress}>
      <Text style={styles.text}>{text}</Text>
    </TouchableOpacity>
  );
};
const styles = StyleSheet.create({
  container: {
    paddingHorizontal: 16,
    paddingVertical: 8,
    backgroundColor: 'violet',
  },
  text: {color: 'black'},
});
```

3. 文件夹下新建文件:Button.stories.tsx，用于书写组件story代码

```typescript
// Button.stories.tsx

import React from 'react';
import {ComponentStory, ComponentMeta} from '@storybook/react-native';
import {MyButton} from './Button';
const MyButtonMeta: ComponentMeta<typeof MyButton> = {
  title: 'MyButton',
  component: MyButton,
  argTypes: {
    onPress: {action: 'pressed the Button'},
  },
  args: {
    text: 'Hello world',
  },
};
export default MyButtonMeta;
type MyButtonStory = ComponentStory<typeof MyButton>;
export const Basic: MyButtonStory = args => <MyButton {...args} />;
```

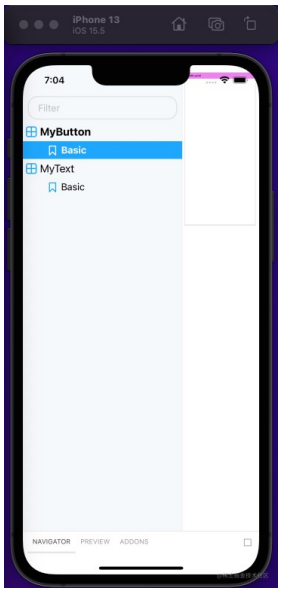
4. 统一资源导出：index.ts

在component根目录下新增index.ts文件，把所有组件export出去。

```javascript
// index.ts

export { default as BlButton } from './Button/Button';
export { default as BlText } from './Text/Text';
```

4. 启动运行

```css
yarn start

输入i，可以在iOS模拟器中预览
输入a，可以在Android设备中预览
输入w，可以在web浏览器中预览
```



5. story进阶：

仅仅展示组件的最终效果，信息也并不直观。因为增加了如下两个能力，包括把代码示例和实际UI效果放在一起显示；组件的所有参数罗列出来加以解释含义。

1. 组件代码示例和UI效果对比

```xml
```
//<Code/>是封装的展示代码的通用组件

<Code title={'Vanilla'}>{"<BlButton title={'默认样式'} />"}</Code>
<BlButton containerStyle={buttonMarginStyle} title={'默认样式'} />
<Code>{"<BlButton disabled title={'默认样式'} />"}</Code>
<BlButton containerStyle={buttonMarginStyle} disabled title={'disabled'} />
```
```

2. 组件参数列表

```less 复制代码
```
//<ParameterTable/>是封装的展示参数列表的通用组件
<ParameterTable
  table={[['autoHitSlop', 'boolean', '放大点击区域'],
    ['children', 'ReactNode', '自定义按钮内容'],
    ['clear', 'boolean', '纯文字无背景色无边框'],
    ['headerLeftButton', 'boolean', '居左显示'],
    ['headerRightButton', 'boolean', '居右显示'],
    ['inline', 'boolean', '行内'],
    ['l', 'boolean', 'minHeight: 26'],
    ['m', 'boolean', 'minHeight: 24'],
    ['outline', 'boolean', '有边框无背景色'],
    ['plain', 'boolean', '色值#333333(BlColor.grey1)'],
    ['round', 'boolean', '圆角'],
    ['s', 'boolean', 'minHeight: 17'],
    ['xl', 'boolean', 'minHeight: 30'],
    ['xxl', 'boolean', 'minHeight: 36'],
    ['warn', 'boolean', '色值#FAAD14(BlColor.error2)'],
  ]}
/>
```
```

6. 部署在线预览

本地预览打包效果：运行 `yarn build build-static-webapp` 或者：`expo build:web`；命令运行成功后，会在项目目录下生成 web-build 目录，想要预览的话，可以使用任意服务器静态托管 web-build 目录。比如，可以 cd 到 web-build 目录下，然后通过 http-server 来进行预览；

服务端打包：目前的 CI 触发逻辑为 `$CI_COMMIT_BRANCH == "feat/main"`，即当向 `feat/main` 分支合并代码时，`yarn build build-static-webapp` 命令会自动执行，然后触发 `ship-webapp` job，完成部署。

云端部署的具体工作就交给公司运维团队负责了。

五、规范代码编写 - eslint+prettier

1. 命名规范

1. 规范细则：

- 合理地为组件和 api 命名，命名前找其他人讨论。
- 在 `component` 目录下新建文件夹。文件夹名称为采用 `PascalCase`(大驼峰式命名) 的组件名，比如 `List`。

- 一个组件文件夹下的所有组件必须通过名称为 `/index.tsx?/` 的文件导出。
- 组件名以 `Bl` 开头，导出为默认。注意不要导出为默认的同时还具名导出。
- 添加组件导出语句到 `components/index.ts` 文件中。
- 在 `component/**` 目录下新建 `${componentNameWithoutBl}.stories.tsx` 采用 `PascalCase`。
- 在 `components/_tests_` 文件夹下可进行单元测试代码的编写。单元测试下的文件名必须是实际组件名，不能是 `index.test.tsx`。

2. 脚本工具，编写eslint规则

dart 复制代码

```
const { get } = require('lodash');
const { pascalCase } = require('change-case');

module.exports = {
  rules: {
    'component-naming': {
      create: ({ report }) => {
        return {
          // accept any valid esquery selector.
          // for available selectors, refer to https://github.com/estools/esquery
          ExportSpecifier: node => {
            const name = get(node, 'exported.name');
            const isType = get(node, 'parent.exportKind') === 'type';
            if (name && !name.toLowerCase().startsWith('bl') && !isType) {
              report(
                node,
                `Component named export variable name must be starts with 'bl' or 'Bl'.`,
              );
            }
          },
          ExportDefaultDeclaration: node => {
            const name = get(node, 'declaration.id.name') || get(node, 'declaration.name');
            const loc = get(node, 'declaration.id.loc') || get(node, 'declaration.loc');
            if (name && !name.startsWith('Bl')) {
              report({ loc, message: `Component default export name must be starts with 'Bl'.` });
            }
          },
        };
      },
    },
    'index-file': {
      create: ({ report }) => {
        return {
          ImportDeclaration: node => {
            if (node.specifiers.length) {
              report(node, 'No import declaration allowed in this file.');
            }
          },
          ExportNamedDeclaration: node => {
            const exportedSpecifiers = get(node, 'specifiers', []).map(el =>
              get(el, 'exported.name'),
            );
            const exportSourceFile = get(node, 'source.value', '').match(
              '\/(?<group>\w+)\/(?<folderOrFile>\w+)(\/(?<fileOrNull>\w+))?',
            );
            const groupName = get(exportSourceFile, 'groups.group');
            const folderOrFileName = get(exportSourceFile, 'groups.folderOrFile');
            const fileOrNull = get(exportSourceFile, 'groups.fileOrNull');
            if (!folderOrFileName) {
              report({
                loc: node.source.loc,
                message: 'Component must in their own folder.',
              });
            }
            if (!fileOrNull) {
              report({
                loc: node.source.loc,
                message: 'Component file name cannot be index.',
              });
            }
            if (pascalCase(folderOrFileName) !== folderOrFileName && groupName !== 'utils') {
              report({
                loc: node.source.loc,
                message: 'Component name must be written in pascal case.',
              });
            }
            if (['Bl', 'BL'].some(el => folderOrFileName.startsWith(el)) && groupName !== 'utils') {
              report({
                loc: node.source.loc,
                message: `Component folder name cannot starts with 'Bl'.`,
              });
            }
            if (
              exportedSpecifiers.some(el => !el.startsWith('bl') && !el.startsWith('Bl')) &&
              groupName !== 'utils'
            ) {
```

```
            report(node, `Exported name must starts with 'bl' or 'Bl'.`);
          }
        },
      };
    },
  },
};
```

2. 代码规范

1. 使用eslint+prettier，对code quality和formatting进行检测

两者关系参考彻底搞清楚ESLint和Prettier的职责关系 - 掘金

因为我们是一个typescript项目，所以安装如下三个依赖：

java 复制代码

```
yarn add -D eslint prettier @react-native-community/eslint-config @typescript-eslint/eslint-plugin eslint-
```

- eslint: ESLint的核心代码
- @typescript-eslint/parser：ESLint的解析器，用于解析typescript，从而检查和规范Typescript代码
- @typescript-eslint/eslint-plugin：这是一个ESLint插件，包含了各类定义好的检测Typescript代码的规范
- eslint相关配置代码写在 .eslintrc.js 文件中

因为TS项目中同时使用了React，所以需要安装：

csharp 复制代码

```
yarn add -D eslint-plugin-react
```

因为eslint更多解决的是代码质量的问题，代码格式的问题需要用另一个更妙的工具-prettier

arduino 复制代码

```
yarn add -D prettier eslint-config-prettier eslint-plugin-prettier
```

- prettier：prettier插件的核心代码
- eslint-config-prettier：解决ESLint中的样式规范和prettier中样式规范的冲突，以prettier的样式规范为准，使ESLint中的样式规范自动失效
- eslint-plugin-prettier：将prettier作为ESLint规范来使用
- prettier相关配置代码写在 .prettierrc.js 文件中

作为一个react native项目，还需要增加如下依赖：

java 复制代码

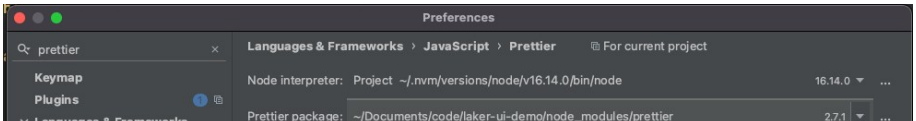
```
yarn add -D @react-native-community/eslint-config eslint-plugin-jest
```

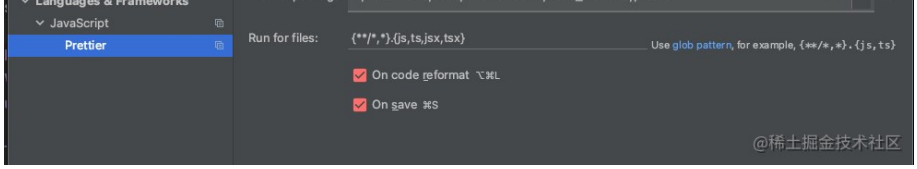
java 复制代码

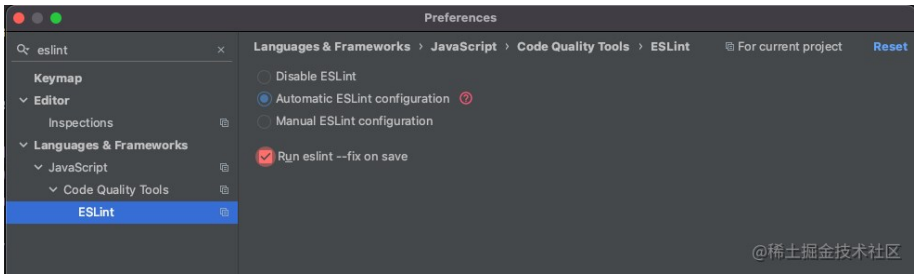
```
//eslintrc.js

module.exports = {
  parser: '@typescript-eslint/parser',
  plugins: ['@typescript-eslint', 'prettier'], // eslint-plugin-prettier的缩写
  extends: ['@react-native-community', 'plugin:react/recommended', 'plugin:prettier/recommended'],
  rules: {
    '@typescript-eslint/no-use-before-define': ['error'],
    'prettier/prettier': 'error',
  },
};
```

json 复制代码

```
//.prettierrc.js

{
  "printWidth": 100,
  "tabWidth": 2,
  "singleQuote": true,
  "trailingComma": "all",
  "bracketSpacing": true,
  "jsxBracketSameLine": true,
  "semi": true,
  "arrowParens": "avoid"
}
```







> 注意：eslint版本和webstorm版本也有很多耦合关系，比如最常见的，2022.2以下版本安装eslint8+版本，会出现 TypeError: this.cliEngineCtor is not a constructor 这个报错，具体解决办法可以参考：Bug: this.cliEngineCtor is not a constructor · Issue #15677 · eslint/eslint

2. 使用tsc-lint-filter，对ts error进行检测

安装依赖：www.npmjs.com/package/tsc...

csharp 复制代码

```
yarn add -D tsc-files
```

开发自动检查脚本：

javascript 复制代码

```
import log from "../../scripts/utils/log.mjs";
import minimist from "minimist";
const argv = minimist(process.argv.slice(2));
const { info, error, fatal } = log("[tsc-lint]");
import exec from "../../scripts/utils/exec.mjs";
import path from "path";

const pathIsEqual = (path1, path2) => {
  path1 = path.resolve(path1);
  path2 = path.resolve(path2);
  if (process.platform === "win32")
    return path1.toLowerCase() === path2.toLowerCase();
  return path1 === path2;
};

const lint = async (files) => {
  info(`Including ${files.length} staged files.`);
  const regex = /(?<path>^.+)(\d+, ?\d+).+(?<error>TS\d{4}.+)/;
  const res = await exec(
    `tsc-files`,
    ["-p", "tsconfig.json", "--noEmit"].concat(files),
    {
      expectFailed: true,
      silentStdout: true,
      silentStderr: true,
    }
  )
    .then(() => {
      info("No errors found in staged file and related files. Nice!");
      process.exit(0);
    })
    .catch((e) => {
      if (e.stderr.includes('tsc-files: not found')) {
        throw new Error('tsc-files is not available at this time.')
      }
      return e.stdout
        .split("\n")
        .filter((el) => el.match(/error TS\d{4}/))
        .map((el) => ({ raw: el, ...el.match(regex)?.groups }));
    });
  const errors = res
    .filter((el) => el.path)
    .filter((el) => files.some((file) => pathIsEqual(file, el.path)))
    .map((el) => el.raw);
  if (errors.length) {
    errors.forEach((el) => error(el));
    fatal(`${errors.length} TypeScript errors in staged files.`);
  } else {
    info("No TypeScript errors in staged files.");
  }
};
```

```
lint(argv._);
```

配置package.json的lint-staged（下文会介绍）：

```
 "lint-staged": {
  "*.{js,jsx,ts,tsx}": [
    "eslint"
  ],
  "*.{ts,tsx}": [
    "node tsc-lint-filter.js"
  ]
}
```

六、规范代码提交-husky&commintlint

1. 概述

- `husky` 可以用于实现各种 Git Hook。这里主要用到 pre-commit这个 hook，在执行 commit 之前，运行一些自定义操作
- `lint-staged` 用于对 Git 暂存区中的文件执行代码检测
- `commitlint` 是我们自己开发的，用于校验commit message规则的脚本工具

2. lint-staged

git暂存区代码检测配置，我们的规则是对所有js,jsx,ts,tsx代码进行eslint检查；对ts,tsx代码进行ts error检查。

```sql
yarn add lint-staged --dev
```

```json
//package.json

"lint-staged": {
  "*.{js,jsx,ts,tsx}": [
    "eslint"
  ],
  "*.{ts,tsx}": [
    "node tsc-lint-filter.js"
  ]
},
```

3. commitlint工具

我们对commit的message做了一些规范，并开发脚步用于自动化检测，作为独立npm包发布：@blacklake/commitlint

类型	规范	示例
初始化项目及相关底层框架	[init]***	[init] 替换APP图标及名称
feature开发	[feat][jira号]***	[feat][GC-19371]计划生产任务增加备注
bug修复	[fix][jira号]***	[fix][GC-19489]修复下拉刷新无效问题
重构/技改	[refactor][jira号]***	[refactor]生产列表页面采用MVP架构重构
埋点数据	[data][jira号]***	[data]生产列表页面数据埋点
打包发布	[pack]***	[pack]修改版本号为5.3.1
增加文档	[doc]***	[doc]增加工程readme
UI调整	[style][jira号]***	[style]修改标题颜色
增加测试代码	[test][jira号]***	[test]增加log排查测试问题

这里只贴出部分核心代码：

```scss
const options = rcFile('bl-commitlint-rc', {configFileName: '.bl-commitlint-rc'});
```

```
if (!options.allowedTypes.includes(typeWithoutScope)) {
  error('Type must be one of the following: ' + options.allowedTypes);
  logErrorLocation(indices.type);
}
if (isRequireRef(typeWithoutScope) && !blCommitParserResult.groups.ref) {
  error(`Issue reference required in this commit type. Prefix must be one of the following: ${options.refPr
  logErrorLocation(indices.type ? [indices.type[1] + 1, 0] : []);
}
if (!blCommitParserResult.groups.subject) {
  error('Commit subject cannot be empty.');
  logErrorLocation(indices.subject);
}
if (options.forbiddenSubjects.includes(blCommitParserResult.groups.subject)) {
  error('Commit subject cannot be one of the following: ' + options.forbiddenSubjects);
  logErrorLocation(indices.subject);
}
if (hasError) {
  // noinspection JSCheckFunctionSignatures
  logExampleBlCommitFormat(blCommitParserResult);
  process.exit(1);
}
```

commit的message检测信息包括：

1. 提交类型是否在规定类型中，如 feat、fix、data 等。
2. 特定提交类型下必须跟jira（本公司项目管理工具）的编号，比如 HHZZ3-***。
3. 提交信息不可为空，不可以为简单的fix、test、update

项目根目录下新建 .bl-commitlint-rc 文件：

```java 复制代码
module.exports = {
requireRefByType: ["fix", "feat", "data"],
refPrefixes: ["HHZZ3-", "SCM"],
  reportMergeMessageAsError: false,
  allowedTypes: [
    "fix",
    "feat",
    "build",
    "chore",
    "ci",
    "docs",
    "style",
    "refactor",
    "perf",
    "test",
    "revert",
    "init",
    "pack",
    "data",
    "BREAKING CHANGE",
  ],
  forbiddenSubjects: ["test", "update", "fix"],
  // see https://www.npmjs.com/package/conventional-commits-parser/v/3.2.1#referenceactions
  referenceActions: [
    "jira",
    "close",
    "closes",
    "closed",
    "fix",
    "fixes",
    "fixed",
    "resolve",
    "resolves",
    "resolved",
  ],
};
```

4. husky-Husky - Git hooks

husky 能够帮你阻挡住不好的代码提交和推送,首先我们安装husky：

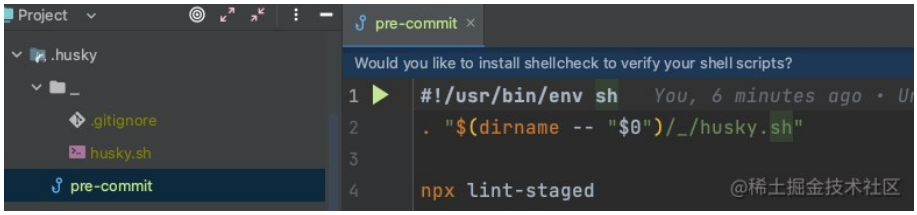
```sql 复制代码
```
yarn add husky --dev
npx husky install
npm pkg set scripts.prepare="husky install"
```
```

上述操作完成后，项目根目录下会生成一个.husky文件夹，package.json文件增加一行脚本：

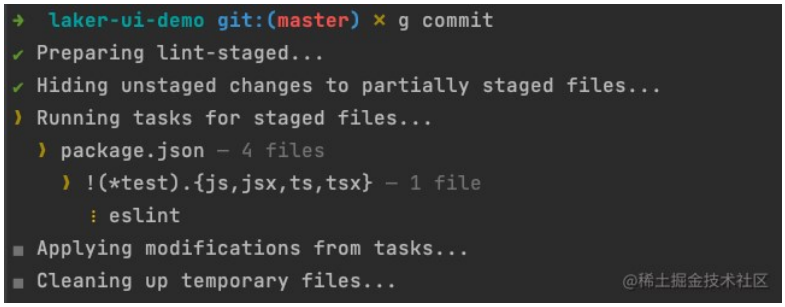
```
//package.json

{
  "scripts": {
    "prepare": "husky install"
  }
}
```

接下来设置代码commit之前需要执行的动作，通过配置husky的pre-commit达到效

```
npx husky add .husky/pre-commit "npx lint-staged"
```

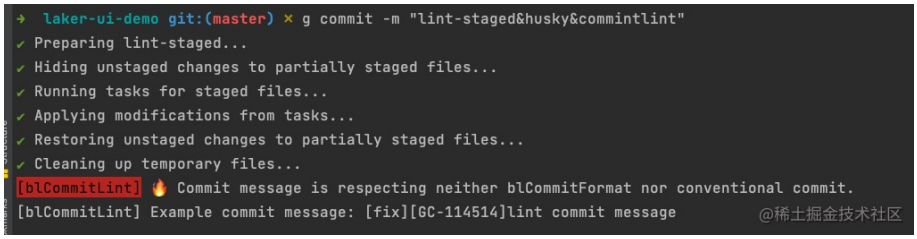


配置成功后，本地代码有改动后，执行git add .，此时代码进入了暂存区，接下来执行git commit，会出发lint-staged制定的规则检查。



接下来设置代码commit信息的自动检查，通过配置husky的commit-msg达到效

```
yarn add -D @blacklake/commitlint
npx husky add .husky/commit-msg

打开自动创建的 commit-msg文件:把undefined改成:
node ./node_modules/@blacklake/commitlint "$1"
```



七、单元测试-Jest & enzyme

查阅资料过程中读到了这篇文章如何设置Jest和Enzyme来测试React Native应用 - 掘金，按照示例初步搭建了单元测试的框架，步骤如下：

```
yarn add jest enzyme jest-environment-enzyme jest-enzyme enzyme-adapter-react-16 --save-dev
yarn add -D react-test-renderer
```

修改package.json文件：

```
"scripts": {
  "test": "jest"
},
"resolutions": {
  "jest-environment-jsdom": "27.4.6"
},
"jest": {
  "preset": "react-native",
```

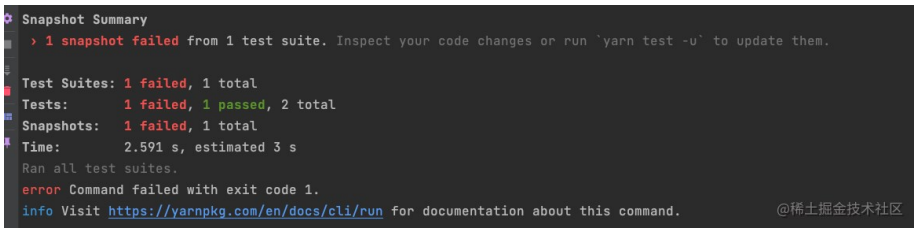
```
  "setupFilesAfterEnv": [
    "jest-enzyme"
  ],
  "testEnvironment": "enzyme",
  "testEnvironmentOptions": {
    "enzymeAdapter": "react16"
  }
},
```

项目根目录下新建__tests__目录，然后新建一个组件测试代码，Button.test.js:

javascript 复制代码

```
import 'react-native';
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';
import { MyButton } from '../components/Button/Button';

it('renders correctly, test using Jest', () => {
  renderer.create(<MyButton />);
});

// Using Jest + Enzyme
describe('<MyButton />', () => {
  it('renders correctly, test using Jest + Enzyme', () => {
    expect(shallow(<MyButton />)).toMatchSnapshot();
  });
});
```

至此，执行yarn test，可以开始运行单元测试。但是此种安装方式会安装各自依赖的最新版本，Jest从v27+后会出现一些问题，本项目暂时将Jest库将至 v27.5.1。修改了Button组件的代码后运行单测可以看到效果：



备注：本项目由于目前需求迭代紧张，人力资源有限，单元测试覆盖率不高，后续资源和时间允许的情况下，酌情补充上去。

八、国际化

国际化是一款成熟APP必备的能力，针对组件库内预设文案也要做国际化适配，我们的技术方案是，使用Context能力封装一个获取对应语言文案的工具，目前仅支持中英文，后续有其他语言也可以做相应扩展。市面上也有大量封装好的国际化工具，可以自行参考使用。

typescript 复制代码

```
//I18nProvider.tsx

export const I18nContext = React.createContext<I18nTranslates | null>(null);
I18nContext.displayName = 'I18nContext';

export const I18nProvider: React.FC<{ t: I18nTranslates; children: React.ReactNode }> = ({
  t,
  children,
}) => {
  const ref = React.useRef<I18nTranslates | null>(null);
  useEffect(() => {
    if (isEqual(ref.current, t)) {
      console.warn(
        'I18nProvider: A re-render has been requested by React.js but translation data has not actually ch
      );
    } else {
      ref.current = t;
    }
  }, [t]);
  return <I18nContext.Provider value={t}>{children}</I18nContext.Provider>;
};

export const useTranslate = <T extends keyof I18nTranslates>(group: T) => {
  const ctx = useContext(I18nContext);
  return useMemo(
    () => (key: keyof I18nTranslates[T], params?: { [key: string]: string | number }) => {
      return (
        (ctx?.[group]?.[key] || defaultTranslates?.[group]?.[key]) as unknown as string
      ).replace(/${([^}]*)}/g, (_, parameterKey) => {
        return params?.[parameterKey].toString() || '';
      });
    },
```

```
    [ctx, group],
  );
};
```

ini 复制代码

```ini
// lang-keys.ts

const defaultTranslates = {
  Timeline: {
    today: '今天',
  },
};

export type I18nTranslates = typeof defaultTranslates;
```

javascript 复制代码

```javascript
const BlTimeline: React.FC<{
  data: TimelineData[];}> = ({}) => {
      const t = useTranslate('Timeline');
      return(
          <BlText f17 color={'grey1'} style={styles.styleToday}>
              {t('today')}
          </BlText>
      );
  }
```

xml 复制代码

```xml
  <I18nProvider t={t}>
    <SomeWhatRootComponent>
      <BlTimeline
          data={[
            {
              date: dayjs()
            },
          ]}
/>
    </SomeWhatRootComponent>
  </I18nProvider>
```

九、分支及npm包管理策略

鉴于实际开发流程和项目管理方式，我们的项目是多分支多版本需求并行开发的场景，制定了如下的分支及包管理策略。

1. 分支：

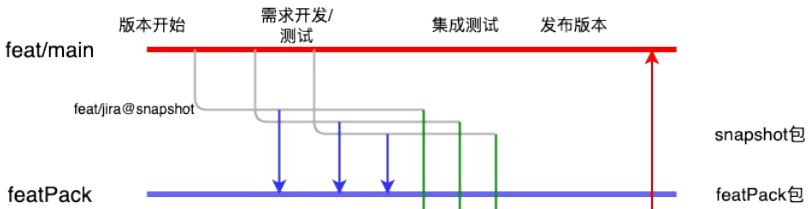
- ***@snapshot：需求开发分支，用于snapshot打包，一般从feat/main或者上一个 beta 分支拉出，合并入下一个 beta 分支。
- featPack：需求测试分支：用于featPack打包，包含未来所有版本的特性，总是从feat/main拉出，每个版本发布后重新拉取，所有snapshot分支代码均需合并到featPack上进行需求测试。
- beta/***：集成测试分支，用于beta打包，包含即将发布的未来版本的特性，总是从 `feat/main` 拉出，所有snapshot分支代码均需合并到 beta 上进行集成测试。
- feat/main: 稳定分支，用于 release 打包，包含正式版本特性，总是接收 beta 分支的合并。

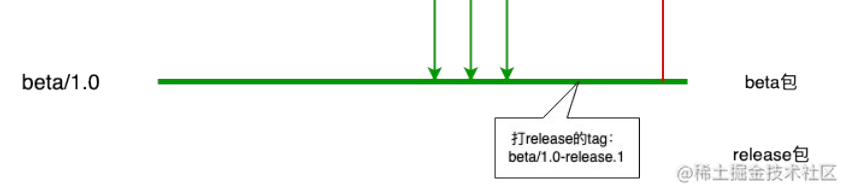
2. 包：

通过设置不同tag，满足不同场景下打出不同的npm包。

- snapshot：小范围验证，版本号规则：0.0.YYYYMMDD-HHmmss-snapshot，更新命令：yarn add @blacklake/laker-ui@snapshot
- featPack：需求测试阶段更新的包，版本号规则：0.0.YYYYMMDD-HHmmss-featpack，更新命令：yarn add @blacklake/laker-ui@featPack
- beta：集成测试阶段更新的包，版本号规则：x.x.x-beta.YYYYMMDD-HHmmss，更新命令：yarn add @blacklake/laker-ui@beta
- release：正式发布的包，版本号规则：x.x.x，更新命令：yarn add @blacklake/laker-ui@latest

工作流程大体如下：





工作步骤：

1. 识别所开发需求上线版本，比如1.0。
2. 基于feat/main checkout一个snapshot分支，如：hhzz3-12345@snapshot。
3. 开发完成后，hhzz3-12345@snapshot merge 到beta/1.0分支，ci会自动创建一个merge到featPack分支的MR，并且把beta的MR设为draft，开发者需要手动完成featPack的merge操作，此时会更新featPack包。
4. 主项目发布集成测试环境时，对当前版本的处于draft状态的MR进行merge操作，此时会更新beta包。
5. 主项目发布生产环境时，在beta/1.0分支上打一个release的tag，此时会更新release包。
6. 发布完成后，beta/1.0需要merge到feat/main，保证feat/main是一个稳定代码分支。

十、打包-rollup

1. 为什么选择rollup

随着模块化技术的发展，加上多种不同浏览器的涌现，不同浏览器对于js的支持是不同的，需要使用polyfill 或打包器/后处理器来抹平这些差异。打包器最早就只是为了做模块化这一件事，但由于它在前端工作流中越来越被赋予了极其重要的地位，于是就出现了各种模块和五花八门的插件，通常所说的打包器有如下几种：

- browserify：可以让你使用类似于 node 的 require() 的方式来组织浏览器端的 Javascript 代码，通过预编译让前端 Javascript 可以直接使用 Node NPM 安装的一些库。2011 年 2 月推出。
- Webpack：代码编译工具，有入口、出口、loader 和插件。webpack 是一个用于现代JavaScript 应用程序的静态模块打包工具。当 webpack 处理应用程序时，它会在内部构建一个依赖图(dependency graph)，此依赖图对应映射到项目所需的每个模块，并生成一个或多个 *bundle*。2012 年 12 月推出。
- Rollup：一个Js模块打包器，采用 ESM 的，可以将小块代码编译成大块复杂的代码。现在已经有很多类库都在使用 rollup 进行打包了，比如：react, vue, preact, three.js,moment, d3 等。2015 年 5 月推出。
- esbuild：极速 JavaScript 打包器，使用 golang 编写的 Blazing Fast 编译器（相比之下，其他打包器几乎都是用 node 编写的，性能较差），同时极大的利用了并行性，完全自己编写而不使用第三方库，从 0 开始就考虑到性能。但不易于使用和扩展。2017 年 11 月推出。
- Parcel：支持多核处理、自带缓存、无需任何配置的打包器。2017 年 12 月推出。

esbuild 提供的、编译 Three.js 十次的测速结果：

Bundler	Time	Relative slowdown	Absolute speed	Output size
esbuild	0.54s	1x	1013.8 kloc/s	5.83mb
rollup + terser	40.48s	75x	13.5 kloc/s	5.80mb
webpack	46.46s	86x	11.8 kloc/s	5.97mb
parcel	124.65s	231x	4.4 kloc/s	5.90mb
fuse-box@next	172.56s	320x	3.2 kloc/s	6.55mb

综合对比分析，考虑打包速度、配置难度等因素，rollup更适合于lib的打包，本项目是组件库，决定采取**rollup。**

2. 安装配置

```sql
npm i rollup -g
yarn add -D rollup @rollup/plugin-typescript @rollup/plugin-sucrase @rollup/plugin-node-resolve rollup-plu
```

上述以来和插件的作用，请自行google了解。修改根目录下的 tsconfig.json 文件：

```json
//tsconfig.json

            {
              "extends": "expo/tsconfig.base",
              "compilerOptions": {
                "strict": true,
                "declaration": true,
                "outDir": "./dist",
                "declarationDir": "./dist/dts",
```

```
    "target": "ES6",
    "rootDir": "src",
    "noEmit": true,
  },
}
```

根目录下新建rollup配置文件：rollup.config.js 如果本地安装的rollup是v3.2，node16环境下不支持ES Module语法，必须使用commonjs语法。如果rollup是v2.78，可以使用ES Module。

```php 复制代码
const typescript = require('@rollup/plugin-typescript');
const sucrase = require('@rollup/plugin-sucrase');
const nodeResolve = require('@rollup/plugin-node-resolve');
const autoExternal = require('rollup-plugin-auto-external');

module.exports = {
  input: './src/index.ts',
  output: {
    dir: './dist',
  },
  plugins: [
    autoExternal(),
    typescript({ tsconfig: './tsconfig.json' }),
    nodeResolve(),
    sucrase({
      exclude: ['node_modules/**'],
      transforms: ['jsx'],
      disableESTransforms: true,
    }),
  ],
};
```

经过上述配置后，命令行执行如下命令，即可成功打包，打包后的文件位于根目录下的dist目录。

```r 复制代码
rollup -c
```

3. 打包脚本

最终真正投入使用的打包脚本，应当包含如下逻辑：

- 静态资源配置管理
- package.json更新
- 版本号的更新和判断

```javascript 复制代码
// build.js

const bundle = (output, input) => {
  return [
    exec('rollup', [`${output}=${input}`, '-c']),
    exec('rollup', [`${output}=${input}`, '-c', '--silent'], {
      env: { ...process.env, GENERATE_DTS: true },
    }),
  ];
};

await Promise.all(bundle('index', './src/index.ts').concat(bundle('utils', utilsIndexFile)))
  .catch(e => {
    console.error(e.message);
    process.exit(1);
  })
  .finally(() => fsj.remove(utilsIndexFile));
fsj.remove('./dist/dts');
fsj.copy('./src/assets', './dist/assets');
const packageJson = JSON.parse(fsj.read('package.json'));
['private', 'scripts', 'jest', 'lint-staged', 'devDependencies'].forEach(el =>
  Reflect.deleteProperty(packageJson, el),
);
packageJson.main = './index.js';
packageJson.name = packageName;
packageJson.version = nextVersion || randomVer;
fsj.write('./dist/package.json', packageJson);
fsj.copy('README.md', './dist/README.md');
```

十一、本地调试

1. 概述

组件库是给业务项目使用的，在新需求开发过程中，如何高效开发调试组件库代码是我们需要解决的问题。最糟糕的方式是：组件库开发一个新feature，开发完成打包编译上传npm，业务项目更新组件库的依赖，重新安装使用。这样效率极低，即便打包速度再快，也避免不了开发过程中需要反复调试带来的消耗。我们采用expo作为组件库项目基架，可以非常方便地在组件开发时进行调试，可以保障组件内逻辑的高效开发。那么组件库和业务代码之间的调试，我们需要实现本地快速打包、并更新业务项目依赖版本号，甚至可以添加配置项，在本地调试时可以使用源码打包，便于方便调试。实现思路是，开发一个脚本，功能是laker-ui本地打包，然后更新项同级目录下的业务项目的package.json文件，同时yarn完成，即可立刻获得组件库新特性。

2. 代码实现：

ini 复制代码

```
const targetFolderPath = '../phoenix';
const installSourceFolderPath = '../laker-ui/dist';

info(`Rebuilding ${packageName}`);
await build();
info(`Packing`);
const res = await exec(`npm pack ./dist`);
const packagePath = `${targetFolderPath}/node_modules/${packageName}`;
info(`Removing folder '${packagePath}'`);
fsj.remove(packagePath);
info(`Installing from local`);
await exec(`yarn add ${installSourceFolderPath} --no-lockfile`, undefined, {
  cwd: targetFolderPath,
});
fsj.write(`${packagePath}/isInstalledFromLocal`, []);
info(`Cleaning up`);
const artifactName = res.stdout.match(artifactPatternRegex)[0];
fsj.remove(artifactName);
```

十二、发布-npm

我们使用npm管理组件库包，并使用公司自建的npm仓库。发布npm需要处理的工作包括：

- 判断包的tag以发布不同的包。
- 自动更新release note
- 发布成功后接入飞书通知

css 复制代码

```
await exec(
  `yarn publish ./dist --non-interactive --verbose ${selectCase(
    [channel === 'release', ''],
    [!isNextBetaFlag, 'futureBeta'],
    [true, `--tag ${channel}`],
  )}`,
);

info('Sending upgrade notification');
sendMessageCardViaCustomBot(......)
export const sendMessageCardViaCustomBot = (
  title: string,
  template: LarkCardTemplate,
  content: LarkCardContent,
) => {
  got.post(env.WEBHOOK_URL, {
    json: {
      msg_type: 'interactive',
      card: {
        header: {
          title: {
            tag: 'plain_text',
            content: title,
          },
          template,
        },
        elements: content,
      },
    },
  });
};
```

十三、配置CICD

作为工程化极其重要的一部分，CICD是提升效率的不二法宝。

Gitlab CI/CD 是一款用于持续集成(CI)，持续交付(CD)的工具，相似的工具有Jenkins、Travis CI、GoCD等。

持续集成，即Continuous Integration，即在源代码变更后(git push)后自动检测(code lint)、构建和进行单元测试的过程，持续集成的目标是快速校验开发人员新提交的代码是没有问题的，并且适合在代码库中进一步使用。

持续交付，即Continuous Delivery，通常是指整个流程链(管道)，它自动监测源代码变更并通过构建、测试、打包和相关操作运行它们以生成可部署的版本(可以是apk打包，也可以是网站部署)，基本不需要人为干预。它包括持续集成，持续测试(保证代码质量)，持续部署(自动发布版本，供用户使用)。

Gitlab的CI/CD配置工作比较简单，只需要依靠一份".gitlab-ci.yml"，将该文件随代码上传，Gitlab就会自动执行相应的任务，从而实现CI/CD。

本项目的CICD分为如下三个stages：MR操作、验证代码有效性、打包；其中打包又分为打组件库npm包和打storybook预览web网页包。

yaml 复制代码

```yaml
stages:
  - mr operations
  - validate
  - ship

cache:
  paths:
    - .yarn-ci-cache/
  key: "$CI_BUILD_REF_NAME"

variables:
  INCLUDE: "(error|fatal|ERR!|Error:|TS\d{4}|TypeError:)"
  EXCLUDE: "(sourcemap|at.{0,10}error)"
  FF_USE_LEGACY_KUBERNETES_EXECUTION_STRATEGY: "true"

default:
  tags:
    - docker
  image: ***
  before_script:
    - yarn install --frozen-lockfile --cache-folder .yarn-ci-cache --prefer-offline
  after_script:
    - npm exec --yes -- @blacklake/watchcat@latest --include $INCLUDE --exclude $EXCLUDE

duplicate merge request:
  tags:
    - docker
  stage: mr operations
  # overwrite
before_script:
    - ''
  allow_failure: true
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_PIPELINE_SOURCE == "push"
  script:
    - node ./scripts/ci/duplicateMR.js

validate:
  tags:
    - docker
  stage: validate
  rules:
    # only if happens not on master or current default branch
- if: $CI_COMMIT_REF_NAME != "feat/main"
  script:
    # make sure package is compilable
- yarn run build
    # cancel when newer version of code available
interruptible: true

ship:
  tags:
    - docker
  stage: ship
before_script:
    - |
      { echo $NPM_NAME; sleep 2; echo $NPM_PASS; sleep 2; echo $NPM_MAIL; } | npm login
    - yarn install --frozen-lockfile --cache-folder .yarn-ci-cache --prefer-offline
  rules:
    - if: $CI_COMMIT_REF_NAME =~ /^(.+@snapshot|featPack|beta/.+)$/
    - if: $CI_COMMIT_BRANCH == "feat/main" && $CI_COMMIT_TITLE =~ /^Merge branch 'beta/.+' into.*/
  script:
    - node ./scripts/ci/ship.js
  resource_group: ship
```

```
build-webapp:
  tags:
    - docker
  stage: ship
  rules:
- if: $CI_COMMIT_BRANCH == "feat/main"
  script:
    - yarn run build-static-webapp
    - pwd && ls -alh
  artifacts:
    paths:
      - web-build

ship-webapp:
  tags:
    - docker
  stage: ship
  # overwrite
before_script:
    - ''
  needs: ['build-webapp']
  rules:
- if: $CI_COMMIT_BRANCH == "feat/main"
  inherit:
default: false
  image: ***
  variables:
    APP_NAME: 'laker-ui'
  script:
    - /ci/scripts/build_laker_ui.sh
    - /ci/scripts/deploy_laker_ui.sh
 resource_group: ship-webapp
```

十四、附录:核心组件

a. 基础组件

Avatar、Badge、BottomSheet、**Button**、Canlendar、Cascader、**CheckBox**、DateTimePicker、DataEmpty、**Dialog**、Draggable、**Drawer**、FileExplorer、Hairline、**Header**、HighlightText、**Icon**、ImageViewer、**Input**、NumberInput、ListItem、Loading、LongText、MessageBox、Modal、Notice、**PagingList**、Popover、ProgressBar、**SearchBar**、Selector、Slider、Step、Switch、Tab、**Tag**、 **Text**、Timeline、**Toast**、Tooltip、Webview

b. 业务组件

AmountInput、AmountRange、RemarkInput、SelectContact、SelectStorage、QcStatus

十五、写在最后-代致谢

行文至此，从0到100搭建React Native组件库的全部内容基本就结束了。为什么称从0到100，因为整个项目在设计到最后落地过程中，我们不仅仅是实现了一个组件库，而是利用前端的工程化思维、效能工具的开发，最后完成了一个兼具组件功能、本地调试、组件预览、单元测试、代码规范、国际化、打包发布、持续交付等多种综合能力的大项目。组件库项目主体框架搭建过程基本结束了，但是我们持续关注UI一致性的工作没有结束，还将持续下去。组件库还在持续不断地迭代中，近期我们又接到一个大需求，需要开发HD的项目，那么如何将组件库兼容到HD端甚至web端，是我们正在思考并解决的问题。

组件库的开发离不开黑湖科技移动端团队的每一位同学。我为整个团队感到骄傲。

作者信息:贾明磊，黑湖科技移动端团队负责人，邮箱:jiaminglei@blacklake.cn, jiaminglei1993@163.com