

## 20 | 大型项目：源码越来越多，项目该如何扩展？

2022-10-15 宋一玮 来自北京



天下无鱼

<https://shikey.com/>

《现代React Web开发实战》

[课程介绍 >](#)



讲述：宋一玮

时长 13:36 大小 12.42M



你好，我是宋一玮，欢迎回到 React 应用开发的学习。

上节课提到，为了应对大中型 React 项目的复杂应用逻辑，我们会分为**局部**和**整体**两个部分来学习。对于作为局部的组件逻辑，可以通过抽象来简化组件的设计和开发。我们学习了 React 中的自定义 Hooks 和组件组合这两种抽象方式，也学习了在这两种抽象基础上的代码复用，尤其是高阶组件的写法。

从局部到整体，复杂度会在代码量上直观地展现出来。在前端工程化和团队协作的基础上，大型 React 项目代码量上 10 万很常见。项目从最初的几行代码到如今的数十万代码，你可能会遇到如下问题：

- 新功能的组件、Hooks、样式要不要分文件写，源文件都放到哪里？
- Redux 的 action、reducer、store 都写到哪里？

- 公共的代码放到哪里？
- 代码文件多到找不到怎么办？

这节课我们会继续讨论 **React** 应用的整体逻辑，看看大中型 **React** 项目在代码增多后，整体扩展上会遇到的挑战，以及如何应对这些挑战。

## 几种典型的 **React** 项目文件目录结构

项目源码的文件目录结构并不等同于应用的整体逻辑，但却可以**作为把握应用整体逻辑的一张“地图”**。一个好的文件目录结构是**自解释**的，可以帮助新接触项目的开发者快速熟悉代码逻辑。

**React** 应用项目有以下五种典型的文件目录结构：

- 单文件结构；
- 单目录结构；
- 按文件职能划分目录结构；
- 按组件划分目录结构；
- 按业务功能划分目录结构。

接下来我们分别看一下。

### 单文件结构

是的，你没看错，单文件结构就是指，在单个 **React** 组件文件中开发所有业务逻辑。

你肯定亲眼见过这种结构，比如在 [第 3 节课](#)里 **CRA** 创建的 **oh-my-kanban** 项目，不算样式的话，我们当时把所有代码都写在了 `src/app.js` 中。需要注意的是，这种结构只适合代码演示或微型的 **React** 项目。

### 单目录结构

比起单文件结构，这种结构拆分了组件文件，拆分的文件都放在同一个目录下。前面 [第 12 节课](#)末尾，**oh-my-kanban** 项目完成大重构第一步时的项目结构就是这样。目录树结构（有

省略) 如下:


 复制代码

```
1  src
2  |— App.css
3  |— App.js
4  |— KanbanBoard.js
5  |— KanbanCard.js
6  |— KanbanColumn.js
7  |— KanbanNewCard.js
8  |— index.css
9  |— index.js
```

单目录结构比起单文件结构, 能支撑更多组件以及相关逻辑, 适合微型 **React** 项目。

## 按文件职能划分目录结构

顾名思义, 在这种结构下, 组件文件放一个目录, 自定义 **Hooks** 文件放一个目录, **context** 文件放一个目录, 如果使用了 **Redux** 的话, **actions**、**reducers**、**store** 各占一个目录 (或者 **Redux Toolkit** 的 **slices** 和 **store** 目录)。

 在第 13 节课, **oh-my-kanban** 项目完成大重构后, 加入的第一个 **context** 就放到了独立的 **src/context** 目录。不只 **JS** 项目, 在第 18 节课, 我们尝试为 **TS** 项目加入的类型定义 **src/types/KanbanCard.types.ts**, 也放到了专门的 **types** 目录下。目录树结构 (有省略) 如下:

 复制代码

```
1  src
2  |— components
3  |   |— App.css
4  |   |— App.tsx
5  |   |— KanbanBoard.tsx
6  |   |— KanbanCard.tsx
7  |   |— KanbanColumn.tsx
8  |   |— KanbanNewCard.tsx
9  |— context
10 |   |— AdminContext.ts
11 |— hooks
12 |   |— useFetchCards.ts
13 |— types
14 |   |— KanbanCard.types.ts
15 |— index.css
```

按文件职能划分目录结构的优点在于，可以快速定位任何一种类型的源码，在源码之间导入导出也比较方便：

[复制代码](#)

```
1 // src/components/App.tsx
2 import AdminContext from '../context/AdminContext';
```

当其中某个或者某几个目录中的文件数不断增多时，这种结构的缺点就暴露出来了：不容易定位到直接相关的源文件。比如 `hooks/useFetchCards.ts` 目前只有 `components/App.tsx` 在用，这从目录结构上是看不出来的，必须进到源码里去看，当 `components` 目录下的文件足够多时，要花些功夫才能确认这两个文件的关联关系。

## 按组件划分目录结构

这种目录结构为每个组件都划分了一个独立、平级的目录，只要跟这个组件强相关，都往这个目录里招呼。这种设计出于两个考虑：

- React 的基本开发单元是组件；
- 同一组件的相关代码要尽量共置（**Colocation**，这里翻译成“托管”不太合适）。

目录树结构的例子如下：

[复制代码](#)

```
1 src
2 |— components
3 |   |— App
4 |     |— AdminContext.js
5 |     |— App.css
6 |     |— App.jsx
7 |     |— App.test.jsx
8 |     |— index.js
9 |     |— useFetchCards.js
10 |   |— KanbanBoard
11 |     |— KanbanBoard.css
12 |     |— KanbanBoard.jsx
13 |     |— index.js
14 |   |— KanbanCard
15 |     |— KanbanCard.css
16 |     |— KanbanCard.jsx
```

```
17 |       |   └─ KanbanNewCard.jsx
18 |       |   └─ index.js
19 |       └─ KanbanColumn
20 |           └─ KanbanColumn.css
21 |           └─ KanbanColumn.jsx
22 |           └─ index.js
23 | └─ index.css
24 | └─ index.jsx
```

在每个目录中都有一个 `index.js`，负责把当前目录的组件重新导出（**Re-export**）到目录外面去，这样其他组件在导入这个组件时，不需要关心目录里都有哪些实现，只关注作为入口的 `index.js` 就行。入口文件示意代码如下：

 复制代码

```
1 // src/components/KanbanCard/index.js
2 export { default as KanbanCard } from './KanbanCard.jsx';
3 export { default as KanbanNewCard } from './KanbanNewCard.jsx';
```

这种目录结构的优势在于，能为特定组件提供一定的封装性，在它专属的目录中能找到它强相关的所有代码。但它也有不足，面对一些跨组件复用的逻辑，可能会出现放到哪个组件目录都不太合适的窘境。

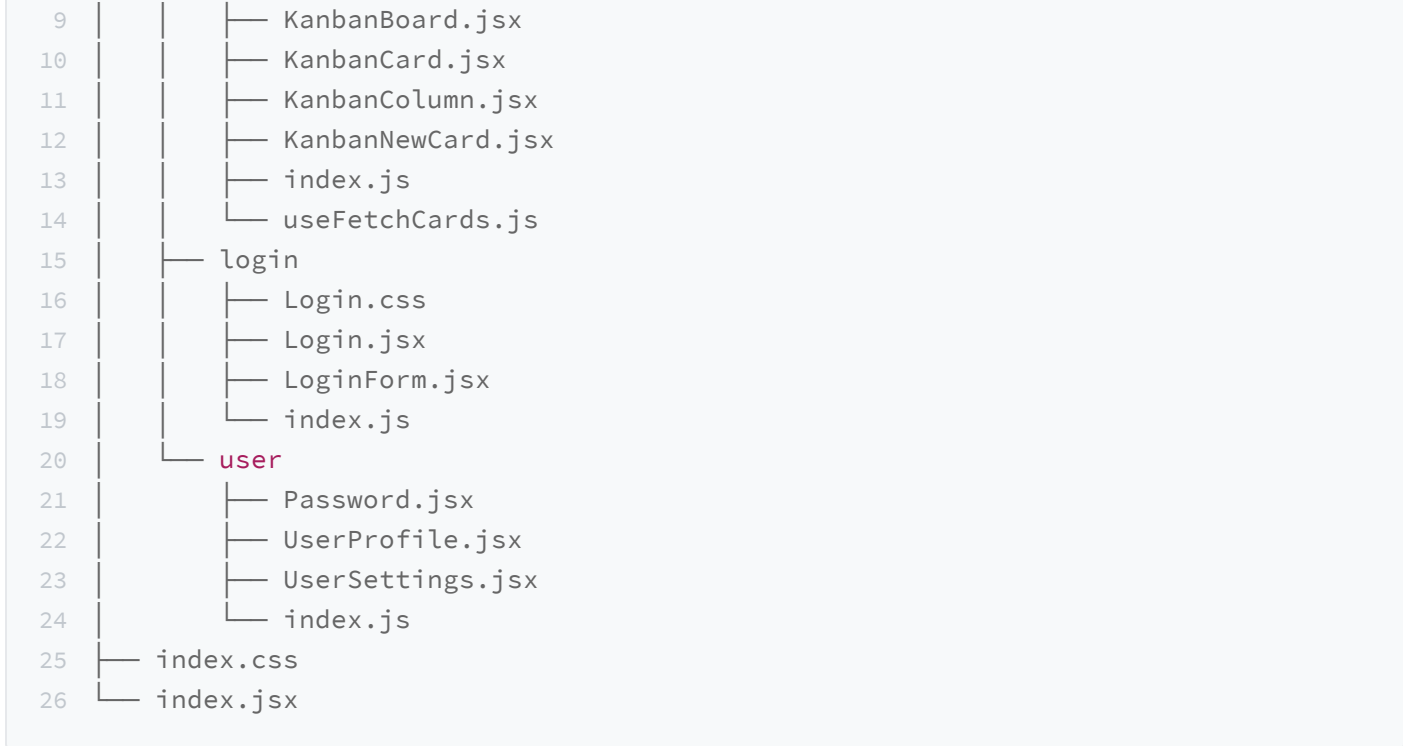
## 按业务功能划分目录结构

按业务功能划分目录结构，它与我们刚刚讲过的结构都不同，意味着目录划分的主要依据不再是具体框架中的某个具体技术概念（包括 **React** 的组件、**Hooks**、**context**，也包括 **Redux** 的 **action**、**reducer**、**store**）。这使得按业务功能划分目录结构成为一个框架无关的方案，也就是说，其他框架的应用也可以利用这种目录结构。

目录树结构的例子如下：

 复制代码

```
1 src
2 | └─ features
3 |     └─ admin
4 |         └─ AdminContext.js
5 |         └─ AdminDashboard.jsx
6 |         └─ AdminSettings.jsx
7 |         └─ index.js
8 | └─ kanban
```



按业务功能划分目录结构可以说，它是这五种结构中最适合大中型 **React** 项目的。它既强调了相关源文件的共置，也在增加业务功能时具有良好的可扩展性。但它也具有与按组件划分目录结构类似的缺点，面对一些跨业务功能复用的逻辑，放在哪个业务目录下都不太合适。

## 如何选取合适的文件目录结构？

可以参考以下表格：

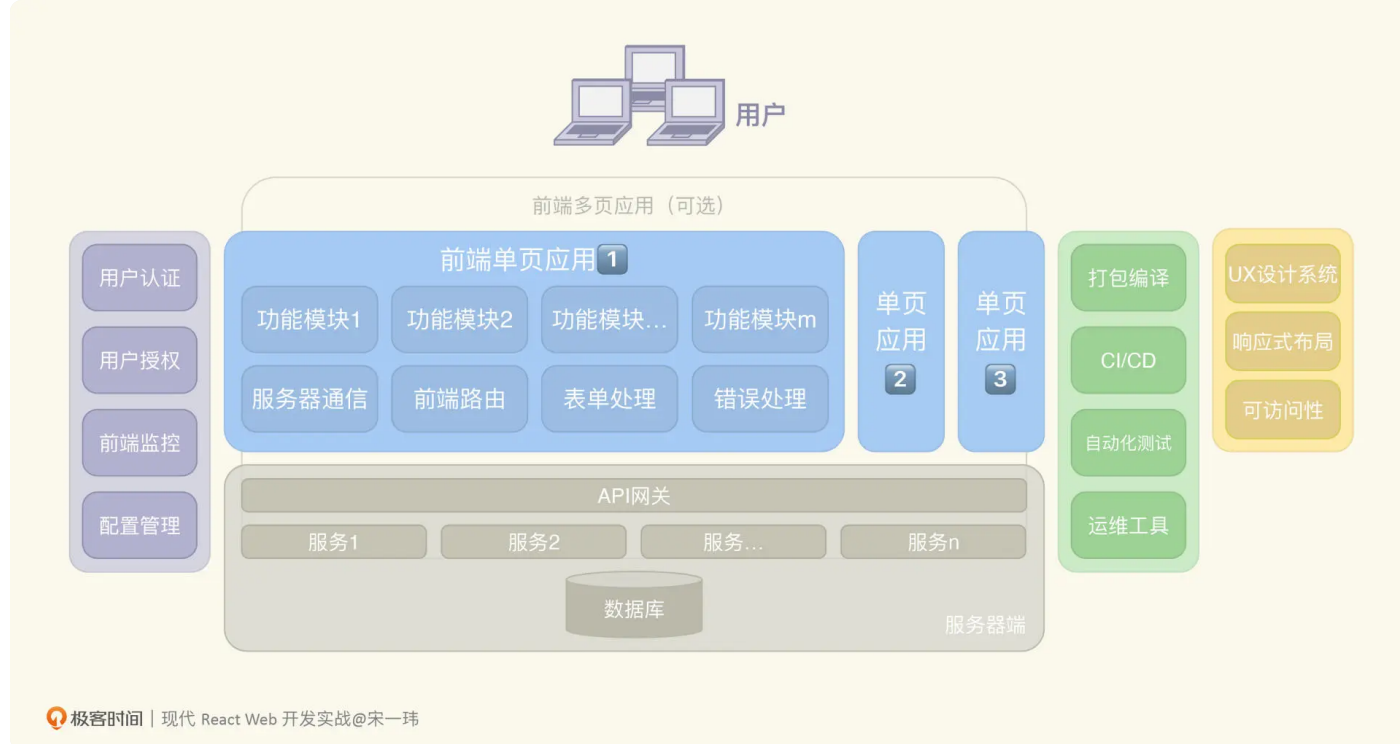
	优势	劣势	适合项目规模
单文件结构	代码集中	单文件代码量有上限	代码演示或微型React项目
单目录结构	代码集中	单目录下文件总数不宜过多	微型或小型React项目
按文件职能划分目录结构	方便定位特定类型的代码	不容易定位相关代码；任一职能目录下文件总数不宜过多	中小型React项目
按组件划分目录结构	代码共置	跨组件的公共逻辑缺少合适的地方	中小型React项目
按业务功能划分目录结构	代码共置 便于添加新业务功能	跨业务功能的公共逻辑缺少合适的地方	大中型React项目

极客时间 | 现代 React Web 开发实战@宋一玮

## 前端应用逻辑架构的功用

当提到 **React** 应用的整体逻辑时，不知你是否还记得 [第 2 节课](#) 这张应用逻辑架构图：





我工作早些年间，流行的开发流程是瀑布式开发（**Waterfall Model**），当时的概要设计阶段和详细设计阶段，对设计文档尤其是架构图的要求非常严格（我画的这种五颜六色的图……基本会被驳回重画）。近些年软件行业追求效率，敏捷开发已经成为主流，但敏捷开发并没有拒绝文档化，也绝不应该被当作拒绝架构设计的借口。

无论是否精确、美观，这样的架构图**有助于我们把握项目的整体走向**，对于大中型 **React** 项目而言是一个值得的先期投入。

也许你会问：“架构图应该是由架构师来画吧？”

我会这样理解这个事情：架构设计是一项工作、一项技能，而架构师是一个职位，两者间没有直接的等号。以我个人的经历举例，我参加工作后画的第一张架构图，那是在工作第二年。后来那张图有幸被用到了几场技术评审中，在与技术 **Leaders** 交流的过程中起了大作用，反过来也让我学到了很多。当然，这纯粹是我的个人理解，具体情况因人而异。

你的下一个问题大概是：“好的，我知道架构图有用了，那具体来说，画架构图对文件目录结构有什么用？”

因为前面提到了：

项目源码的文件目录结构并不等同于应用的整体逻辑，但却可以作为把握应用整体逻辑的一张“地图”。

那么应用逻辑架构图就可以当作是“**地图**”的“**地图**”。

下面马上来了解一下，我们为大中型 **React** 项目推荐的文件目录结构，也看看跟上面的应用逻辑架构图有什么样的对应关系。

## 大中型 **React** 项目推荐的文件目录结构

当 **React** 项目规模属于中型或大型时，文件目录结构需要满足以下几个目标：

- 便于横向扩展（即增加新功能点或视图）；
- 易于定位相关代码；
- 鼓励代码复用；
- 有利于团队协作。

为了满足上面的目标，我推荐你以**按业务功能划分**为主，结合**按组件**、**按文件职能**的方式，划分目录结构。

参考的目录树结构（有省略）如下：

 复制代码

```
1  src
2  |— components
3     |— Button
4     |— Dialog
5     |— ErrorBoundary
6     |— Form
7         |— Form.css
8         |— FormField.jsx
9         |— Form.jsx
10        |— index.js
11        |— ...
12        |— Tooltip
13 |— context
14     |— ...
15     |— UserContext.js
16 |— features
17     |— admin
```



```
18 |   └── dashboard
19 |       ├── activies
20 |       │   └── ActivityList.jsx
21 |       ├── charts
22 |       │   └── ...
23 |       ├── news
24 |       │   ├── news.png
25 |       │   ├── NewsDetail.jsx
26 |       │   └── NewsList.jsx
27 |       ├── Dashboard.css
28 |       ├── Dashboard.jsx
29 |       └── index.js
30 |   └── kanban
31 |       ├── KanbanBoard.jsx
32 |       ├── index.js
33 |       └── useFetchCards.js
34 |   └── home
35 |   └── login
36 |   └── ...
37 |   └── user
38 | └── hooks
39 |     ├── ...
40 |     └── useLocation.js
41 | └── servies
42 |     ├── kanbanService.js
43 |     ├── ...
44 |     └── userService.js
45 | └── index.css
46 | └── index.jsx
```

对应上面的例子，首先建立 **features** 目录，**features** 下面的一级目录都对应一个相对完整的业务功能，目录中有实现这一功能的各类代码。

对于部分体量比较大的功能，可以根据需要在一级目录下加入二级目录，每个二级目录都对应一个相对独立的子功能（业务），目录内部是实现子功能的各类代码。必要时还可以加入三级、四级目录，但总体目录层级不应过深。所以我们说，在 **features** 目录，可以从横向、纵向两个方向扩展功能点。

在 **features** 目录之外，为公用的代码建立一系列职能型的目录，包括可重用组件的 **components** 目录、可重用 Hooks 的 **hooks** 目录；**context** 目录的主要目的不是重用，而是跨业务功能使用 **context**；**services** 目录下，集中定义了整个应用会用到的远程服务，避免四散到各个业务模块中，甚至硬编码（**Hardcode**）。这些公用代码的目录层级不宜太深，以一到二级为主。

从代码的导入导出关系来看，在 **features** 目录下，原则上同级目录间的文件不应互相导入，二级、三级目录只应被直接上一级目录导入，不能反过来被下一级目录导入。**features** 目录的代码可以导入公用目录的代码，反过来公用目录的代码不能导入 **features** 目录的代码。在任何时候都应该避免循环导入（**Circular Import**）。



来对应一下前面的逻辑架构图：

- 图中的功能模块 1、功能模块 2 再到功能模块 m 这一行，可以对应到 **features** 目录；
- 图中的服务器通信、前端路由、表单处理、错误处理这一行，可以分别对应到公用的 **services** 目录、**hooks** 目录、**components** 目录；
- 虽然在目录结构中没有体现，但用户认证、用户授权、前端监控等，也可以放到公用目录中实现。

这节课到目前为止讲到的目录结构，都是以单个 **React** 项目为前提的。

根据实际项目需要，也有很多 **React** 项目使用了多项目或者 **monorepo** 的方式来开发和扩展，虽然编译构建、**CI/CD** 更加复杂了，但更有利于多个团队的协作，提高整体开发效率。在这样的实践中，可以把追加功能点到同一个 **React** 项目（或 **monorepo** 的包）看作纵向扩展，把特定模块、可复用组件和逻辑抽取为独立 **React** 项目（或 **monorepo** 的包）看作横向扩展。

## 模块导入路径过长怎么办？

在大中型 React 项目中，有时会遇到这样的 import 语句：

 复制代码

```
1 // src/a/b/c/d/e/f/g/h/MyComponent.jsx
2 import Dialog from '../../../../../../../../components/Dialog';
```

这种情况，首先要确定 MyComponent.jsx 是否真有必要放到这么深的路径下。如果发现这在项目中是个普遍情况，那可以利用 Node.js 的 [Subpath Imports](#) 功能（[Vite](#) 中尚未支持），或是由前端构建工具提供的非标准的 module 别名（Alias）功能。

 复制代码

```
1 // Subpath Imports
2 import Dialog from '#components/Dialog';
3
4 // Alias
5 import Dialog from '@components/Dialog';
```

## 小结

今天这节课，我们了解了如何从整体层面应对大中型 React 应用逻辑的扩展，学习了 5 种典型的 React 项目文件目录结构，对比了它们各自的优劣势和适用项目规模。

然后学习了以按业务功能划分为主，结合按组件、按文件职能划分的目录结构的方式，来应对大中型 React 项目。同时也强调了前端应用逻辑架构对应用逻辑扩展的指导作用。

专栏的第三模块到目前为止，我们已经学习了大中型 React 项目的数据流、局部和整体逻辑的相关实践。

下节课我们会暂时从应用业务开发中跳出来，了解一下常见的 React 性能问题和优化方案。这些性能优化方案对各种规模的 React 应用基本都适用，目标都是保证优秀的用户体验。

## 思考题

1. 其实在业界，**React** 项目中经常会有一个名叫 **common** 的目录。如果在这节课里讲到的，大中型 **React** 项目推荐的文件目录结构中，设置一个这样的 **common** 目录，你会往这个目录里放什么文件？不放什么文件？为什么？
2. 在 [第 5 节课](#) 的思考题中：

除了浏览器，你在电脑上最常用的桌面应用是什么？是不是 macOS 的 **Finder** 或 Windows 的**资源管理器**？

如果是的话就好办了。请你尝试把 **Finder** 或资源管理器当作要用 **React** 开发的 **Web** 应用，按自己的理解做一遍组件拆分。

现在已知 **Finder** 或资源管理器是大型 **React** 项目，请你为这个项目设计一套文件目录结构，记得把自己在第 5 节课设计的组件文件都放进去。

分享给需要的人，Ta购买本课程，你将得 **18** 元

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 19 | 代码复用：如何设计开发自定义Hooks和高阶组件？

[下一篇](#) 21 | 性能优化：保证优秀的用户体验

## 精选留言 (3)

 写留言



船长

2022-10-18 来自上海

**common** 文件夹顾名思义是放公共文件、可复用逻辑的地方，比如网站的 **Header**，**Footer**，公共 **util** 等。

比如本次 **Finder** 项目的 左上角最大最小化按钮就应属于 **common** 目录的一部分，应为不止 **Finder** 用到了，**macos** 下所有的窗口都有这部分





船长

2022-10-18 来自上海

在写这个 demo 的时候想起之前困惑的一个问题，想请教下宋老师，即在 jsx 中 `<childComponent/>` 与 `childComponent()` 这 2 种调用组件的方式有什么区别？

这是一个 demo，上面的输入框是用 `<childComponent/>` 这种方式调用的，在输入时会有个输入框失焦的问题，下面用 `childComponent()` 调用的就没这个问题。

demo 地址：<https://codesandbox.io/s/fervent-ishizaka-mwusdq?file=/App.tsx>），

共 3 条评论 >



船长

2022-10-18 来自上海

模拟的 Finder 目录结构，原本想采用按【业务功能】拆分的结构的，但做的过程中发现项目比较简单，采用这种模式反而会复杂，于是采用了按【文件职能】拆分的结构，结构如下：

YeahMyKanBan

```
| |   └─ FinderSimulate
| |       └─ components
| |           └─ Column.tsx
| |           └─ CommonCard.tsx
| |           └─ SystemOperate.tsx
| |       └─ context
| |           └─ myContext.ts
| |       └─ index.tsx
| |       └─ pages
| |           └─ HeaderMenu.tsx
| |           └─ LeftMenu.tsx
| |           └─ MainContent.tsx
```

后面用了 vercel 进行了部署（codesanbox 部署 umi 项目有问题）

在线预览：<https://react-learn2-orphin.vercel.app/YeahMyKanBan/FinderSimulate>

源码地址：<https://github.com/TanGuangZhi/ReactLearn/tree/main/src/pages/YeahMyKanBan/FinderSimulate>

