react后台系统最佳实践

本文主要讲三块内容：中后台系统的技术栈选型、hooks时代状态管理库的选型以及hooks的使用问题与解决方案。

一、中后台系统的技术栈选型

1. 要做什么

我们的目标是搭建一个适用于公司内部中后台系统的前端项目最佳实践。

2. 要求

由于业务需求比较多，一名开发人员需要负责几个后台系统。所以项目最佳实践的要求按重要性排行：1、开发效率。2、可维护性。3、性能。
总之，开发的高效率跟简单的代码结构是比较侧重的两点。

3. 技术栈怎么选

由于我司前端技术栈主要使用React，所以基础框架采用React跟React-router。项目开发内容主要是做中后台系统页面，于是选择antd作为系统的UI框架。然后ahooks提供了useAntdTable方法可以帮助我们节省二次封装的工作量，所以采用ahooks作为项目主要使用的hooks库。最后考虑到开发效率以及性能，状态管理库则采用MobX。

下面详细说一下状态管理库的选型过程。

二、hooks时代状态管理库的选型

如果使用了ahooks或者React-Query这类带数据请求方案的hooks库，那已经分担了状态管理库很大一部分工作了。剩下的部分是页面的交互状态处理问题，主要是解决跨组件通信的问题。

目前调研的状态管理方案有以下几种：

context

首先考虑的是不引入任何状态管理库，直接使用React框架提供的context方法。
React context表面上使用起来很方便，只要定义一个provider并传入数据，使用的时候用useContext获取对应的值即可。
但这个方案需要开发者考虑如何处理组件重复渲染的问题，需要开发者考虑是通过手动拆分provider的数据还是使用memo、useMemo缓存组件的方案(详情见这里)。
总的来说解决起来还是比较麻烦的，每次添加状态都要检查这个值是否要拆分、是否频繁更新以及怎么组织组件比较合理等问题。
总结：React context开发效率不高、后期维护麻烦。

```js 复制代码
// 需要拆分状态
<UserContext.Provider value={userData}>
  <MenuContext.Provider value={menuData}>
    {props.children}
  </MenuContext.Provider>
</UserContext.Provider>
// 需要缓存组件
useMemo(() => <Component value={a} />, [a])
```

redux

接下来是目前React状态管理库中下载量最高的redux。
redux这个方案首先要吐槽的是其繁琐的写法。每次使用的时候都要烦恼action怎么取名；使用reducer时要写一大堆扩展运算符，而且一个请求至少要写三个状态(发送请求、请求成功、请求失败)；异步用thunk会被嫌弃不够优雅，而saga的API又多generator写法又不好用。
官方的推出的Redux Toolkit框架解决了上面说的action的命名问题，还有reducer的要写一堆扩展运算符的问题。但状态颗粒度太细的问题还是存在，saga的写法也还是没变。
如果结合ahooks的话刚好是可以把saga节省掉，但用了这些请求库之后redux鼓吹的状态跟踪的优点也就消失了大半~~(虽然感觉这个功能也没啥作用)~~。只是单纯解决跨组件通信的话引入Redux Toolkit又感觉太重了，而且对比其他状态库Redux Toolkit使用起来还是不够简便。
总结：redux是真的繁琐繁琐繁琐。

```ts 复制代码
// 代码来源网上的[案例](https://codesandbox.io/s/react-ts-redux-toolkit-saga-knq31?file=/src/api/user.
// 这一坨代码实现的功能随便换个库就只要几行就搞定
```

```
export const createSagaAction = <
  PendingPayload = void,
  FulfilledPayload = unknown,
  RejectedPayload = Error
>(typePrefix: string): SagaAction<PendingPayload, FulfilledPayload, RejectedPayload> => {
  return {
    request: createAction<PendingPayload>(`${typePrefix}/request`),
    fulfilled: createAction<FulfilledPayload>(`${typePrefix}/fulfilled`),
    rejected: createAction<RejectedPayload>(`${typePrefix}/rejected`),
  }
}
export const fetchUserAction = createSagaAction<
 User['id'],
 User
>('user/fetchUser');
export function* fetchUser(action: PayloadAction<User['id']>) {
  try {
    const user = yield call(getUser, action.payload);
    yield put(fetchUserAction.fulfilled(user));
  } catch (e) {
    yield put(fetchUserAction.rejected(e));
  }
}
export function* userSaga() {
  yield takeEvery(fetchUserAction.request.type, fetchUser);
}
export const userSlice = createSlice({
  name: 'user',
  initialState,
  reducers: {},
  extraReducers: (builder) => (
    builder
      .addCase(fetchUserAction.request, (state) => {
        state.isLoading = true;
      })
      .addCase(fetchUserAction.fulfilled, (state, action) => {
        state.isLoading = false;
        state.user = action.payload;
      })
      .addCase(fetchUserAction.rejected, (state, action) => {
        state.isLoading = false;
        state.error = action.payload;
      })
  )
});
```

recoil

官方还推荐了一个叫recoil的状态管理库，使用了下感觉也不够简便。
定义状态有两个常用的api：atom跟selector。atom每次使用都要写key，selector用着感觉也有点冗余。调用的api还分useRecoilState跟useRecoilValue，从简便性来说被下面要讲的zustand完爆。
然后这个框架本身也比较新，npm下载量也比zustand要低不少。
总结：简便性被zustand完爆，下载量不高。

```js
// 定义分atom跟selector
const a = atom({
  key: "a",
  default: []
});
const b = selector({
  key: "b",
  get: ({ get }) => {
    const list = get(a);
    return Math.random() > 0.5 ? list.slice(0, list.length / 2) : list;
  }
});
// 调用则区分useRecoilValue、useRecoilState
const list = useRecoilValue(b);
const [value, setValue] = useRecoilState(a);
```

zustand

然后到了势头挺猛的zustand，npm的下载量已经能赶上MobX了。趋势对比
基本的调用真的挺简洁的，通过create定义store，使用的时候直接调用就好。用起来比recoil方便多了。
但是呢zustand的状态都是不可变，getState时跟redux一样要用到很多扩展运算符。官方是推荐引入immer，但这样写法又变复杂了一点。
另外zustand定义store时颗粒度需要挺细的，不然组件重复渲染的问题不好解决。不像MobX那样可以把同一个页面的store写到一个文件里，zustand拆分的维度是需要按组件渲染状态去划分。
如果能像React toolkit那样不需要用户自己引入immer的话zustand还是挺香的。因为后台系统一般来说交互类的状态并不多，拆分颗粒度过细的问题并不大。而要开发人员自己每次都手动增加immer还是挺烦的。
总结：需要引入额外的库，store拆分要求比较细。

```ts
// 定义store
import produce from 'immer';
```

```
// list这个值不拆出去的话，在组件A修改title的值会引起list所在组件B的渲染。
const useStore = create<TitleState>((set) => ({
  title: '',
  list: [],
  setTitle: (title) => set(produce((state: TitleState) => {
    state.title = title;
  })),
}));
// 组件A 使用title
const { title, setTitle } = useStore();
// 组件B 使用list
const { list } = useStore();
```

MobX

是的，说了一圈状态管理库最后还是选择MobX。
MobX使用起来很简单，主要用到useLocalStore跟useObserver两个api。store可以按照页面划分，维护起来很方便。性能也好，按照store的值去拆分组件就行。
至于说React加MobX不如用vue的说法，可能从性能上说是这样。但本质上说选择React主要是看重React衍生出来的其强大的生态环境，而不是其他原因。
举一个典型例子就是React Native。如果有APP跨端开发需求的话，那么React Native还是比较热门的解决方案。目前React从生态成熟度上来说有着其他框架都达不到的高度，前端团队可以用React这一个框架去解决web、app、服务端渲染等多个场景的开发需求。使用一个技术栈能够降低开发成本，开发人员切换开发场景的成本比较低，不需要学额外的框架语法。
所以说没必要跟其他框架攀比，既然选择了React，那就在React体系内找一个好用的状态管理库就行，不要有其他的心理负担。

```
js 复制代码
// 一个文件定义一个页面的store
class ListStore {
    constructor() {
        makeAutoObservable(this);
    }
    title = '';
    list = [];
    setList(values) {
        this.list = values;
    }
}
// 使用
const localStore = useLocalStore(() => store);
useObserver(() => (
    <div>
      {localStore.list.map((item) => (<div key={item.id}>{item.name}</div>))}
```

```
    </div>
));
```

补充：关于React组件重复渲染问题，网上有些言论是觉得无所谓。
但如果不管的话当项目随着时间而变得复杂之后很可能会遇到性能问题，到时候想改难度就变大了。
即使花大力气重构之后也面临测试问题，项目上线需要申请测试资源对业务功能进行回归测试，总得来说还是比较麻烦的。
而MobX处理组件重复渲染问题挺方便的，只要组件拆分得当就不需要开发者过多关心。

三、hooks的使用问题与解决方案

技术栈选好之后接下来就是确定React代码的开发形式了。
首先是目前在React项目中使用hooks的写法是必须的，这是官方确定的路线。

问题

但是用hooks会遇到两个比较麻烦的问题，一个是useEffect、useCallback、useMemo这些API的依赖项过多时的问题。另一个是useEffect的使用问题。

依赖项问题：
先说依赖项问题，项目中遇到useEffect、useCallback、useMemo这些API最头疼的是后面跟着好几个依赖项，当你要去修改里面的功能时你必须查看每个依赖项的具体作用，了解它们的更新时期。新增加的状态需要考虑是用useState还是用useRef，又或者是两者并存。总之心智负担还是挺高的。

```js 复制代码
// 需要查看每个依赖项的更新逻辑
const onChange = useCallback(() => {
    if (a) {
      setValue(b);
    }
}, [ a, b ]);
```

useEffect问题：
再来就是useEffect的使用问题，不管在项目里看到一个useEffect跟着多个依赖项还是多个useEffect跟着不同的依赖项，都是很头疼的事情。

当你需要增加或者修改里面的代码逻辑时你需要把代码都理解一遍，然后再决定你新的代码逻辑是写在现有的useEffect里还是再新增一个useEffect去承接。

```js
// 一个useEffect里有多个依赖项
useEffect(() => {}, [a, b, c])
// 多个useEffect跟着各自的依赖项
useEffect(() => {}, [a])
useEffect(() => {}, [b])
useEffect(() => {}, [c])
```


解决方案

前面决定了mobx作为状态管理库，所以这两个问题的解决方案就是尽量不要使用useState，服务端的接口请求使用ahooks去解决，剩下的交互状态使用mobx处理。

依赖项多问题：
首先看依赖项过多的解决方案，当使用mobx的状态之后依赖项只需要写store一个依赖就行(不写也行)，这个时候在useEffect、useCallback这些API里面获取的都是store里最新的值，不需要担心状态更新问题。

```js
// 只需要写localStore一个依赖，里面的a、b值永远都是最新的
const onChange = useCallback(() => {
    if (localStore.a) {
      localStore.setValue(localStore.b);
    }
}, [ localStore ]);// 也可以用[]
```


useEffect的使用问题：
然后是useEffect的使用问题，解决方案就是不使用useEffect。
跟上面依赖项多的解决方式一样，服务端的接口请求都使用ahooks去解决，然后组件渲染状态采用mobx结合ahooks提供的其他hooks方法(ahooks文档)，基本上就用不到useEffect了。
如果有监听某个值然后渲染层级嵌套比较深的组件的需求，比如父组件某个状态变更之后需要孙子组件的form表单执行清空动作的场景，那这个时候可以使用MobX的reaction去处理。

```js
// 当状态变更之后触发
reaction(
    () => localStore.visible,
    visible => {
      if (visible) {
        formRef.current?.resetFields(); // 清空表单
```

```
      }
    }
);
```

补充1：有人质疑不用useState、useEffect这两个API的方案，这里再展开说下。
首先是只要项目用了第三方hooks库或者自定义hooks，那useState跟useEffect的使用频率就很低了。最理想的方案就是直接使用useXXX去解决问题，而不是在业务代码里写一大堆的useState、useEffect。
另外是在以前使用class组件的时代，只要你用了状态管理库那setState方法也一样可以完全不使用。项目的组织形式有很多，选择满足自身需求的方案就好。

补充2：MobX组件复用问题可以参考官方文档提供的写法，通过传入一个返回不同状态值的函数去解决。

```js 复制代码
// 官方推荐写法
const GenericNameDisplayer = observer(({ getName }) => <DisplayName name={getName()} />)
const MyComponent = ({ person, car }) => (
    <>
        <GenericNameDisplayer getName={() => person.name} />
        <GenericNameDisplayer getName={() => car.model} />
        <GenericNameDisplayer getName={() => car.manufacturer.name} />
    </>
)
```

总结

1、系统的技术栈是React、React-router、antd、ahooks跟MobX。
2、状态管理库选择MobX可以兼顾开发效率、后期维护跟性能问题。
3、hooks问题的解决方案主要是用ahooks处理服务端状态，然后用MobX处理剩下的交互状态；尽量少使用useState，不使用useEffect。
4、后续会补充一个代码模板，把一些常用的后台系统页面的具体代码组织形式补充进来。
5、最佳实践的意义在于团队内部统一一个代码写法，以此实现降低项目开发成本以及同事之间协作成本的目标。因为代码结构的一致可以方便项目后期的维护。假设说React官方推出了一个新的代码组织形式，那么一个结构统一的项目就能够快速迁移到新写法上面(最理想情况是写一个脚本批量把代码进行替换)。而且团队的开发人员也能快速理解不同项目的结构跟功能，不会出现某个项目只有某个同事能开发的情况。
6、本文这个最佳实践是根据自身团队情况设计的，如果也比较看中开发效率跟后期维护可以参考这个模式。