对于“前端状态”相关问题，如何思考比较全面

问题的起源

有相当比例的前端从业者入行是从**学习前端框架的使用**开始的。换言之，在他们的知识体系中，最底层是**前端框架如何使用**，其他业务知识都是构建于此之上。

要以此为基础回答**前端状态**相关问题，并不容易。就比如你问组长：

- 为什么项目中用 `Redux` 而不用 `Mobx` ？
- 为什么要用 `Hooks` 而不用 `ClassComponent` ？

很多时候得到的是一个既定的事实（就是这样，没有为什么），而不是分析后的结果。

要分析这类问题，我们需要知道一些更低抽象层级的知识。

几乎所有主流前端框架的实现原理，都在践行 `UI = f(state)` 这个公式，通俗的说 —— **UI是对状态的映射。**

这应该是**前端状态**会出现的最低抽象层级了，所以我们从这个层级出发。

前端框架的实现原理

限于篇幅有限，这里我们以最常见的 `React` 与 `Vue` 举例。

在实现**UI是对状态的映射**过程中，两者的方向不同。

`React` 并不关心状态如何变化。每当调用更新状态的方法（比如 `this.setState` ，或者 `useState dispatch` ...），就会对整个应用进行 `diff` 。

所以在 `React` 中，传递给**更新状态的方法**的，是**状态的快照**，换言之，是个**不可变的数据。**

`Vue` 关心状态如何变化。每当更新状态时，都会对**与状态关联的组件**进行 `diff` 。

所以在 `Vue` 中，是直接改变状态的值。换言之，状态是个**可变的数据**。

这种底层实现的区别在单独使用框架时不会有很大区别，但是会影响上层库的实现（比如状态管理库）。

现在我们知道，通过前端框架，我们可以将状态映射到 `UI` 。那么如何管理好对应的映射关系呢？

换言之，如何将状态与**和他相关的UI**约束在一起？

我们再往更高一级抽象看。

如何封装组件

前端开发普遍采用**组件**作为**状态与UI的松散耦合单元**。

到这里我们可以发现，如果仅仅会使用前端框架，那么只能将组件看作是**前端框架中既定的设计**。

但如果从更低一层抽象（前端框架的实现原理）出发，就能发现 —— 组件是为了解决框架实现原理中**UI到状态的映射**的途径。

那么组件该如何实现，他的载体是什么呢？从软件工程的角度出发，有两个方向可以探索：

- 面向对象编程
- 函数式编程

面向对象编程的特点包括：

- 继承
- 封装
- 多态

其中**封装**这一特点使得**面向对象编程**很自然成为组件的首选实现方式，毕竟组件的本质就是**将状态与UI封装在一起的松散耦合单元**。

`React` 的 `ClassComponent`， `Vue` 的 `Options API` 都是类似实现。

但毕竟组件的本质是**状态与UI的松散耦合单元**，在考虑复用性时，不仅要考虑**逻辑的复用**（逻辑是指操作状态的业务代码），还要考虑**UI的复用**。所以**面向对象编程**的另两个特性并不适用于组件。

框架们根据自身特点，在**类面向对象编程**的组件实现上，拓展了复用性：

- `React` 通过 `HOC`、 `renderProps`
- `Vue2` 通过 `mixin`

经过长期实践，框架们逐渐发现 —— **类面向对象编程的组件实现**中**封装**带来的好处不足以抵消**复用性**上的劣势。

于是 `React` 引入了 `Hooks`，以函数作为组件封装的载体，借用**函数式编程**的理念提高复用性。类似的还有 `Vue3` 中的 `Composition API`。

不管是 `ClassComponent` 还是 `FunctionComponent`、 `Options API` 还是 `Composition API`，他们的本质都是**状态与UI的松散耦合单元**。

当组件数量增多，逻辑变复杂时，一种常见的解耦方式是 —— 将可复用的逻辑从组件中抽离出来，放到单独的 `Model` 层。 `UI` 直接调用 `Model` 层的方法。

对 `Model` 层的管理，也就是所谓的**状态管理**。

对状态的管理，是比组件中**状态与UI的耦合**更高一级的抽象。

状态管理问题

状态管理要考虑的最基本的问题是 —— 如何与框架实现原理尽可能契合？

比如，我们要设计一个 `User Model`，如果用 `class` 的形式书写：

```js
class User {
  name: String;
  constructor(name: string) {
                this.name = name;
        }
        changeName(name: string) {
```

```
                return this.name = name;
        }
}
```

只需要将这个 `Model` 的实例包装为响应式对象，就能很方便的接入 `Vue3`：

```js 复制代码
import { reactive } from 'vue'

setup() {
  const user = reactive(new User('KaSong') as User;
  return () => (
   <button onClick={() => user.changeName('XiaoMing')}>
      {user.name}
    </button>
  )
}
```

之所以这么方便，诚如本文开篇提到的 —— `Vue` 的实现原理中，状态是**可变的数据**，这与 `User Model` 的用法是契合的。

同样的 `User Model` 要接入 `React` 则比较困难，因为 `React` 原生支持的是**不可变数据**类型的状态。

要接入 `React`，我们可以将同样的 `User Model` 设计为不可变数据，采用 `reducer` 的形式书写：

```js 复制代码
const userModel = {
  name: 'KaSong'
};

const userReducer = (state, action) => {
  switch (action.type) {
    case "changeName":
      const name = action.payload;
      return {...state, name}
  }
};

function App() {
  const [user, dispatch] = useReducer(userReducer, userModel);

  const changeName = (name) => {
    dispatch({type: "changeName", payload: name});
  };
```

```
  return (
    <button onClick={() => changeName('XiaoMing')}>
      {user.name}
    </button>
  );
}
```

如果一定要接入**可变类型状态**，可以为 `React` 提供类似 `Vue` 的**响应式更新**能力后再接入。比如借用 `Mobx` 提供的响应式能力：

```
import { makeAutoObservable } from "mobx"

function createUser(name) {
    return makeAutoObservable(new User(name));
}
```


到目前为止，不管是**可变类型状态**还是**不可变类型状态**的 `Model`，都带来了**从组件中抽离逻辑**的能力，对于上例来说：

- **可变类型状态**将状态与逻辑抽离到 `User` 中
- **不可变类型状态**将状态与逻辑抽离到 `userModel` 与 `userReducer`
- 最终暴露给 `UI` 的都仅仅是 `changeName` 方法

当业务进一步复杂，`Model` 本身需要更完善的架构，此时又是更高一级的抽象。

到这一层时已经脱离前端框架的范畴，上升到纯状态的管理，比如为 `mobx` 带来结构化数据的 `mobx-state-tree`。

此时框架实现原理对 `Model` 的影响已经在更高的抽象中被抹去了，比如 `Redux-toolkit` 是 `React` 技术栈的解决方案，`Vuex` 是 `Vue` 技术栈的解决方案，但他们在使用方式上是类似的。

这是因为 `Redux` 与 `Vuex` 的理念都借鉴自 `Flux`，即使 `React` 与 `Vue` 在实现原理上有区别，但这些区别都被状态管理方案抹平了。

更高的抽象

在此之上，对于状态还有没有更高的抽象呢？答案是肯定的。

对于常规的状态管理方案，根据用途不同，可以划分出更多细分领域，比如：

- 对于表单状态，收敛到表单状态管理库中
- 对于服务端缓存，收敛到服务端状态管理库中（`React Query`、`SWR`）
- 用完整的框架收敛前后端`Model`，比如`Remix`、`Next.js`

总结

回到我们开篇提到的问题：

- 为什么项目中用`Redux`而不用`Mobx`？
- 为什么要用`Hooks`而不用`ClassComponent`？

现在我们已经能清晰的知道这两个问题的相同点与不同点：

- 相同点：都与状态相关
- 不同点：属于不同抽象层级的状态相关问题

要回答这些问题需要哪些知识呢？只需要知道问题涉及的**状态的抽象层级**，以及**比该层级更低的抽象层级**对应的知识即可。

比如回答：为什么项目中用`Redux`而不用`Mobx`？

考虑当前抽象层级

`Redux`与`Mobx`都属于`Model`的实现，前者带来一套**类Flux的状态管理理念**，后者为`React`带来**响应式更新**能力，在设计`Model`时我的项目更适合哪种类型？

或者两种类型我都不在乎，那么要不要使用更高抽象的解决方案（比如`MST`、`Redux Toolkit`）抹平这些差异？

考虑低一级抽象层级

项目用的 `ClassComponent` 还是 `FunctionComponent`？ `Redux`、 `Mobx` 与他们结合使用时哪个组合更能协调好 `UI` 与逻辑的松散耦合？

考虑再低一级抽象层级

`React` 的实现原理决定了他原生与**不可变类型状态**更亲和。 `Redux` 更契合**不可变数据**， `Mobx` 更契合**可变数据**。我的项目需要考虑这些差异么？

当了解不同抽象层级需要考虑的问题后，任何宽泛的、状态相关问题都能转化成具体的、多抽象层级问题。

从不同抽象层级出发思考，就能更全面的回答问题。