link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=122 sentences=271, words=2201

## 1.课程大纲 #

### 1.1 mobx实战篇 #

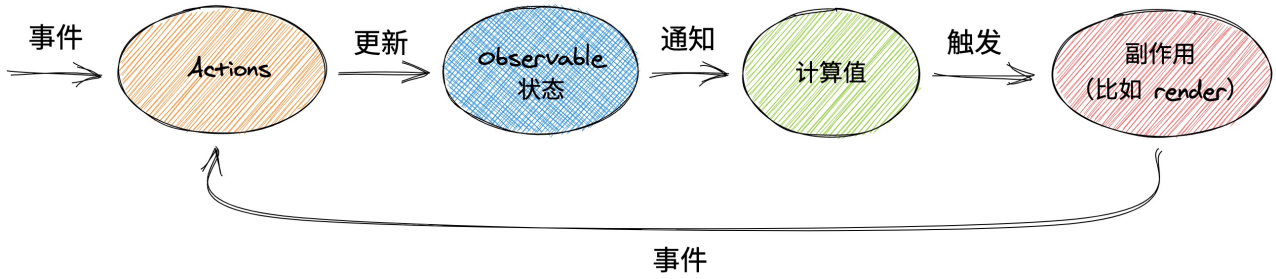
- observable、makeObservable、makeAutoObservable
- Autorun、computed、action、flow、bound、Reaction、When、runInAction
- mobx+mobx-react综合案例

### 1.2 mobx源码篇 #

- 实现observable、reaction、autorun、action
- 实现observer、useObserver、Observer、useLocalObservable

## 2.Mobx #

- mobx (https://mobx.js.org/README.html)
- 中文 (https://zh.mobx.js.org/README.html)
- 任何可以从应用状态中派生出来的值都应该被自动派生出来
- MobX 是一个身经百战的库，它通过运用透明的函数式响应编程使状态管理变得简单和可扩展




### 2.1 安装 #

```
pnpm create vite
pnpm install @babel/core @babel/plugin-proposal-decorators @babel/plugin-proposal-class-properties
pnpm install mobx mobx-react
```

### 2.2 vite.config.ts #

vite.config.ts

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
export default defineConfig({
  plugins: [react({
    babel: {
      plugins: [
        ["@babel/plugin-proposal-decorators", { legacy: true }],
        ["@babel/plugin-proposal-class-properties", { loose: true }],
      ],
    },
  })]
})
```

### 2.3 jsconfig.json #

jsconfig.json

```
{
    "compilerOptions": {
        "experimentalDecorators": true
    }
}
```

### 2.4 src\main.tsx #

src\main.tsx

```
import {observable} from 'mobx';
console.log(observable);
```

## 3.mobx #

### 3.1 创建可观察对象 #

- 创建可观察状态 (https://zh.mobx.js.org/observable-state.html#%E5%88%9B%E5%BB%BA%E5%8F%AF%E8%A7%82%E5%AF%9F%E7%8A%B6%E6%80%81)
- 属性，完整的对象，数组，Maps 和 Sets 都可以被转化为可观察对象。 使得对象可观察的基本方法是使用 `makeObservable`为每个属性指定一个注解。 最重要的注解如下：
    - `observable` 定义一个存储 state 的可追踪字段。
    - `action` 将一个方法标记为可以修改 state 的 action。
    - `computed` 标记一个可以由 state 派生出新的值并且缓存其输出的 getter。
- 像数组，Maps 和 Sets 这样的集合都将被自动转化为可观察对象。

### 3.2 observable #

- 用法: `observable(source, overrides?, options?)`
- observable (https://zh.mobx.js.org/observable-state.html#observable)注解可以作为一个函数进行调用，从而一次性将整个对象变成可观察的。 `source`对象将会被克隆并且所有的成员都将会成为可观察的

- 由 observable 返回的对象将会使用 Proxy 包装，这意味着之后被添加到这个对象中的属性也将被侦测并使其转化为可观察对象

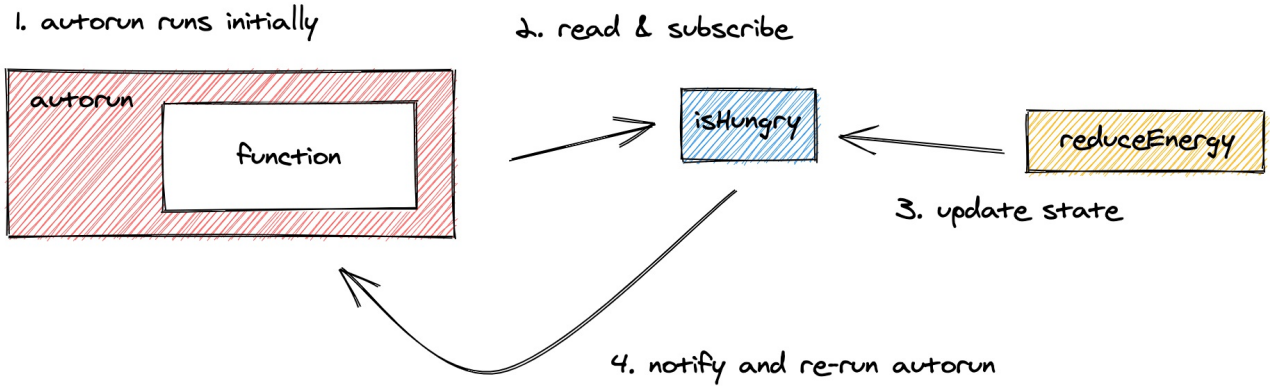
```
import {observable,reaction} from 'mobx';
let obj = {name:'1'}
let proxyObj = observable(obj);
console.log(proxyObj);
```

### 3.3 reactions #

- reactions (https://zh.mobx.js.org/reactions.html#%E4%BD%BF%E7%94%A8-reactions-%E5%A4%84%E7%90%86%E5%89%AF%E4%BD%9C%E7%94%A8-)是需要理解的重要概念，因为他可以将 MobX 中所有的特性有机地融合在一起
- reactions的目的是对自动发生的副作用进行建模。 它们的意义在于为你的可观察状态创建消费者，以及每当关联的值发生变化时，自动运行副作用

### 3.4 Autorun #

- 用法: autorun(effect: (reaction) => void)
- Autorun (https://zh.mobx.js.org/reactions.html#autorun) 函数接受一个函数作为参数，每当该函数所观察的值发生变化时，它都应该运行。 当你自己创建 autorun 时，它也会运行一次。它仅仅对可观察状态的变化做出响应，比如那些你用 observable 或者 computed 注解的
- Autorun通过在响应式上下文运行 effect来工作。在给定的函数执行期间，MobX 会持续跟踪被 effect 直接或间接读取过的所有可观察对象和计算值。 一旦函数执行完毕，MobX 将收集并订阅所有被读取过的可观察对象，并等待其中任意一个再次发生改变。 一旦有改变发生，autorun 将会再次触发，重复整个过程




```
import {observable,autorun} from 'mobx';
let obj = {name:'1'}
let proxyObj = observable(obj);
autorun(()=>{
    console.log(proxyObj.name);
})
proxyObj.name = '2';
```

### 3.5 makeObservable #

- 用法: makeObservable(target, annotations?, options?)
- makeObservable (https://zh.mobx.js.org/observable-state.html#makeobservable)函数可以捕获已经存在的对象属性并且使得它们可观察。任何 JavaScript 对象（包括类的实例）都可以作为 target 被传递给这个函数。 一般情况下，makeObservable 是在类的构造函数中调用的，并且它的第一个参数是 this

```
import {observable,makeObservable,autorun} from 'mobx';
class Doubler {
    value
    constructor(value) {
        makeObservable(this, {
            value: observable,
        })
        this.value = value
    }
}
const doubler = new Doubler(1);
autorun(()=>{
    console.log(doubler.value);
});
doubler.value = 2;
```

### 3.6 computed #

- computed (https://zh.mobx.js.org/computeds.html#%E9%80%9A%E8%BF%87-computeds-%E6%B4%BE%E7%94%9F%E4%BF%A1%E6%81%AF)
- 计算值可以用来从其他可观察对象中派生信息。 计算值采用惰性求值，会缓存其输出，并且只有当其依赖的可观察对象被改变时才会重新计算。 它们在不被任何值观察时会被暂时停用
- 计算值可以通过在 JavaScript getters 上添加 computed 注解来创建。 使用 makeObservable 将 getter 声明为 computed。或者如果你希望所有的 getters 被自动声明为 computed，可以使用 makeAutoObservable

```
+import {observable,makeObservable,autorun,computed} from 'mobx';
class Doubler {
    value
    constructor(value) {
        makeObservable(this, {
            value: observable,
+           double: computed,
        })
        this.value = value
    }
+   get double() {
+       return this.value * 2
+   }
}
const doubler = new Doubler(1);
autorun(()=>{
    console.log(doubler.value);
+   console.log(doubler.double);
});
doubler.value = 2;
```

### 3.7 action #

- actions (https://zh.mobx.js.org/actions.html#%E4%BD%BF%E7%94%A8-actions-%E6%9B%B4%E6%96%B0-state)
- 所有的应用程序都有 actions。action 就是任意一段修改 state 的代码。原则上，actions 总会为了对一个事件做出响应而发生。例如，点击了一个按钮，一些输入被改变了，一个 websocket 消息被送达了，等

等

- 尽管 makeAutoObservable 可以自动帮你声明一部分 actions，但是 MobX 还是要求你声明你的 actions。Actions 可以帮助你更好的组织你的代码并提供以下性能优势：
    - 它们在 transactions 内部运行。任何可观察对象在最外层的 action 完成之前都不会被更新，这一点保证了在 action 完成之前，action 执行期间生成的中间值或不完整的值对应用程序的其余部分都是不可见的
    - 默认情况下，不允许在 actions 之外改变 state。这有助于在代码中清楚地对状态更新发生的位置进行定位
- action 注解应该仅用于会修改 state 的函数。派生其他信息（执行查询或者过滤数据）的函数不应该被标记为 actions，以便 MobX 可以对它们的调用进行跟踪

```
+import {observable,makeObservable,autorun,computed,action} from 'mobx';
class Doubler {
    value
    constructor(value) {
        makeObservable(this, {
            value: observable,
            double: computed,
+           increment: action
        })
        this.value = value
    }
    get double() {
        return this.value * 2
    }
+   increment() {
+       this.value++
+       this.value++
+   }
}
const doubler = new Doubler(1);
autorun(()=>{
    console.log(doubler.value);
    console.log(doubler.double);
});
+doubler.increment();
```

### 3.8 flow #

- flow (https://zh.mobx.js.org/actions.html#%E4%BD%BF%E7%94%A8-flow-%E4%BB%A3%E6%9B%BF-async--await-) 包装器是一个可选的 async / await 替代方案，它让 MobX action 使用起来更加容易
- flow 将一个 generator 函数 作为唯一输入。 在 generator 内部，你可以使用 yield 串联 Promise（使用 yield somePromise 代替 await somePromise）。 flow 机制将会确保 generator 在 Promise resolve 之后继续运行或者抛出错误。
- 所以 flow 是 async / await 的一个替代方案，不需要再用 action 进行包装。它可以按照下面的方式使用：
    - 使用 flow 包装你的异步函数
    - 使用 function * 代替 async
    - 使用 yield 代替 await

```
+import {observable,makeObservable,autorun,computed,flow,action} from 'mobx';
class Doubler {
    value
    constructor(value) {
        makeObservable(this, {
            value: observable,
            double: computed,
            increment: action,
+           fetch: flow
        })
        this.value = value
    }
    get double() {
        return this.value * 2
    }
    increment() {
        this.value++
        this.value++
    }
+   *fetch() {
+       const response = yield new Promise((resolve)=>setTimeout(()=>resolve(5),1000))
+       this.value = response;
+   }
}
const doubler = new Doubler(1);
autorun(()=>{
    console.log(doubler.value);
    console.log(doubler.double);
});
doubler.increment();
+doubler.fetch();
```

### 3.9 bound #

- flow.bound (https://zh.mobx.js.org/actions.html#flowbound)注解可用于将方法自动绑定到正确的实例，这样 this 会始终被正确绑定在函数内部。 与 actions 一样，flows 默认可以使用 `autoBind` 选项

```
import {observable,makeObservable,autorun,computed,flow,action} from 'mobx';
class Doubler {
    value
    constructor(value) {
        makeObservable(this, {
            value: observable,
            double: computed,
+           increment: action.bound,
+           fetch: flow.bound
        })
        this.value = value
    }
    get double() {
        return this.value * 2
    }
    increment() {
        this.value++
        this.value++
    }
    *fetch() {
        const response = yield new Promise((resolve)=>setTimeout(()=>resolve(5),1000))
        this.value = response;
    }
}
const doubler = new Doubler(1);
autorun(()=>{
    console.log(doubler.value);
    console.log(doubler.double);
});
+const increment = doubler.increment;
+increment();
+const fetch = doubler.fetch;
+fetch();
```

### 3.10 makeAutoObservable #

- 使用 `makeAutoObservable(target, overrides?, options?)`
- makeAutoObservable (https://zh.mobx.js.org/observable-state.html#makeautoobservable) 就像是加强版的 makeObservable，在默认情况下它将推断所有的属性。你仍然可以使用 overrides 重写某些注解的默认行为
- 与使用 makeObservable 相比，makeAutoObservable 函数更紧凑，也更容易维护，因为新成员不需要显式地提及。 然而，makeAutoObservable 不能被用于带有 super 的类或 子类
- 推断规则：
    - 所有 自有 属性都成为 observable
    - 所有 getters 都成为 computed
    - 所有 setters 都成为 action
    - 所有 prototype 中的 functions 都成为 autoAction
    - 所有 prototype 中的 generator functions 都成为 flow
    - 在 overrides 参数中标记为 false 的成员将不会被添加注解。例如，将其用于像标识符这样的只读字段

```
+import {observable,makeObservable,autorun,computed,flow,action,makeAutoObservable} from 'mobx';
class Doubler {
+   PI=3.14
    value
    constructor(value) {
+       makeAutoObservable(this,{PI:false},{autoBind:true})
        this.value = value
    }
    get double() {
        return this.value * 2
    }
    increment() {
        this.value++
        this.value++
    }
    *fetch() {
        const response = yield new Promise((resolve)=>setTimeout(()=>resolve(5),1000))
        this.value = response;
    }
}
const doubler = new Doubler(1);
+autorun(()=>{
+    console.log(doubler.PI);
+});
autorun(()=>{
    console.log(doubler.value);
    console.log(doubler.double);
});
const increment = doubler.increment;
increment();
const fetch = doubler.fetch;
fetch();
+doubler.PI=3.15;
```

### 3.11 Reaction #

- 使用 `reaction(() => value, (value, previousValue, reaction) => { sideEffect }, options?)`.
- Reaction (https://zh.mobx.js.org/reactions.html#reaction)类似于 autorun，但可以让你更加精细地控制要跟踪的可观察对象。 它接受两个函数作为参数：第一个 data 函数，其是被跟踪的函数并且其返回值将会作为第二个函数，effect 函数的输入。 重要的是要注意，副作用只会对 data 函数中被访问过的数据做出反应，这些数据可能少于 effect 函数中实际使用的数据。
- 一般的模式是在 data 函数中返回你在副作用中需要的所有数据， 并以这种方式更精确地控制副作用触发的时机。 与 autorun 不同，副作用在初始化时不会自动运行，而只会在 data 表达式首次返回新值之后运行

```
+import {makeAutoObservable,reaction} from 'mobx';
class Doubler {
    PI=3.14
    value
    constructor(value) {
        makeAutoObservable(this,{PI:false},{autoBind:true})
        this.value = value
    }
    get double() {
        return this.value * 2
    }
    increment() {
        this.value++
        this.value++
    }
    *fetch() {
        const response = yield new Promise((resolve)=>setTimeout(()=>resolve(5),1000))
        this.value = response;
    }
}
const doubler = new Doubler(1);
+reaction(
+    () => doubler.value,
+    value => {
+        console.log('value',value);
+    }
+)
+doubler.value=2;
```

### 3.12 When #

- 使用方式 `when(predicate: () => boolean, effect?: () => void, options?)`
- when (https://zh.mobx.js.org/reactions.html#when)会观察并运行给定的 predicate 函数，直到其返回 true。 一旦 predicate 返回了 true，给定的 effect 函数就会执行并且自动执行器函数将会被清理掉
- 如果你没有传入 effect 函数，when 函数返回一个 Promise 类型的 disposer，并允许你手动取消

```
+import {makeAutoObservable,reaction,when} from 'mobx';
class Doubler {
    PI=3.14
    value
    constructor(value) {
        makeAutoObservable(this,{PI:false},{autoBind:true})
        this.value = value
    }
    get double() {
        return this.value * 2
    }
    increment() {
        this.value++
        this.value++
    }
    *fetch() {
        const response = yield new Promise((resolve)=>setTimeout(()=>resolve(5),1000))
        this.value = response;
    }
}
const doubler = new Doubler(1);
+when(
+    () => doubler.value === 3,
+    () => {
+        console.log('value',doubler.value);
+    }
+)
+doubler.value++;
+doubler.value++;
+doubler.value++;
```

### 3.13 runInAction #

- 使用方式 `runInAction(fn)`
- 使用runInAction (https://mobx.js.org/actions.html#runinaction)来创建一个会被立即调用的临时 `action`。在异步进程中非常有用

```
import {makeAutoObservable,reaction,when,autorun,runInAction} from 'mobx';
class Doubler {
    PI=3.14
    value
    constructor(value) {
        makeAutoObservable(this,{PI:false},{autoBind:true})
        this.value = value
    }
    get double() {
        return this.value * 2
    }
    increment() {
        this.value++
        this.value++
    }
    *fetch() {
        const response = yield new Promise((resolve)=>setTimeout(()=>resolve(5),1000))
        this.value = response;
    }
}
const doubler = new Doubler(1);
+autorun(()=>console.log(doubler.value));
+runInAction(()=>{
+    doubler.value++;
+    doubler.value++;
+    doubler.value++;
+});
```

## 4.mobx-react #

- mobx-react (https://github.com/mobxjs/mobx/tree/main/packages/mobx-react)
- mobx-react-lite (https://github.com/mobxjs/mobx/tree/main/packages/mobx-react-lite)

### 4.1 observer #

#### 4.1.1 main.jsx #

src\main.jsx

```
import { createRoot } from "react-dom/client";
import Counter from "./Counter";
const rootElement = document.getElementById("root");
const root = createRoot(rootElement);
root.render(<Counter/>);
```

#### 4.1.2 observer #

- 使用 observer(baseComponent: FunctionComponent ): FunctionComponent 
- 将React组件、React类组件或独立渲染函数转换为React组件的函数。转换后的组件将跟踪其有效渲染使用的观察值，并在其中一个值更改时自动重新渲染组件
- React.memo自动应用于提供给观察者的功能组件
- 当使用React类组件时， this.props和 this.state变得可观察，因此组件将对渲染使用的属性和状态的所有更改作出反应

src\Counter.jsx

```
import {makeAutoObservable} from 'mobx';
import {observer} from 'mobx-react';
class Store {
    number=1
    constructor(){
        makeAutoObservable(this,{},{autoBind:true});
    }
    add(){
        this.number++;
    }
}
let store=new Store();
export default observer(function () {
    return (
        <div>
            <p>{store.number}p>
            <button onClick={store.add}>+button>
        div>
    )
});
```

### 4.2 observer class #

#### 4.2.1 Counter.jsx #

src\Counter.jsx

```
import React from 'react';
import {makeAutoObservable} from 'mobx';
import {observer} from 'mobx-react';
class Store {
    number=1
    constructor(){
        makeAutoObservable(this,{},{autoBind:true});
    }
    add(){
        this.number++;
    }
}
let store=new Store();
+@observer
+export default class Counter extends React.Component{
  render(){
    return (

            {store.number}
            +

    )
  }
+}
```

### 4.3 Observer #

- 使用 <observer>{renderFn}</observer>
- Observer (https://github.com/mobxjs/mobx/tree/main/packages/mobx-react#observer)
- Observer是一个React组件，它将观察者应用于组件中的匿名区域。它将单个无参数函数作为子函数，该函数应只返回一个React组件。将跟踪函数中的渲染，并在需要时自动重新渲染

#### 4.3.1 Counter.jsx #

src\Counter.jsx

```
import React from 'react';
import {makeAutoObservable} from 'mobx';
+import {observer,Observer} from 'mobx-react';
class Store {
    number=1
    constructor(){
        makeAutoObservable(this,{},{autoBind:true});
    }
    add(){
        this.number++;
    }
}
let store=new Store();
export default function () {
    return (
+
            {
                ()=>(
                  <>
                    {store.number}
                    +
                  </>
                )
            }
+
    )
}
```

### 4.4 useObserver #

- useObserver (https://github.com/mobxjs/mobx/tree/main/packages/mobx-react-lite#useobservertfn---t-basecomponentname--observed-options-iuseobserveroptions-t-deprecated)允许您使用类似观察者的行为，但仍然允许您以任何方式优化组件（例如，使用自定义areEqual的memo，使用forwardRef等），并准确声明观察到的部分（渲染阶段）

#### 4.4.1 Counter.jsx #

src\Counter.jsx

```
import React from 'react';
import {makeAutoObservable} from 'mobx';
import {observer,Observer,useObserver} from 'mobx-react';
class Store {
    number=1
    constructor(){
        makeAutoObservable(this,{},{autoBind:true});
    }
    add(){
        this.number++;
    }
}
let store=new Store();
export default function () {
+    return useObserver(()=>(
        <>
          {store.number}
          +
        </>
+     ));
}
```

### 4.5 useLocalObservable #

- 使用 `useLocalObservable<t>(initializer: () => T, annotations?: AnnotationsMap<t>): T</t></t>`
- 当使用useLocalObservable (https://github.com/mobxjs/mobx/tree/main/packages/mobx-react-lite#uselocalobservabletinitializer---t-annotations-annotationsmapt-t)时，返回对象的所有属性都将自动可观察，getter将转换为计算属性，方法将绑定到存储并自动应用mobx事务

#### 4.5.1 Counter.jsx #

src\Counter.jsx

```
import React from 'react';
import {makeAutoObservable} from 'mobx';
import {observer,Observer,useObserver,useLocalObservable} from 'mobx-react';
export default function () {
+   const store = useLocalObservable(()=>({
+       number:1,
+       add(){
+           this.number++;
+       }
+   }));
    return useObserver(()=>(
        <>
          {store.number}
          +
        </>
      ));
}
```

### 4.6 todos #

#### 4.6.1 main.jsx #

```
import { createRoot } from "react-dom/client";
import App from "./App";
const rootElement = document.getElementById("root");
const root = createRoot(rootElement);
root.render(<App/>);
```

#### 4.6.2 App.jsx #

src\App.jsx

```
import React from "react";
import store from "./store";
import StoreContext from './context';
import Todos from './Todos';
import User from './User';
const App = () => {
  return (
    <StoreContext.Provider value={store}>
      <User />
      <hr />
      <Todos/>
    StoreContext.Provider>
  );
};
export default App;
```

#### 4.6.3 context.jsx #

src\context.jsx

```
import React from "react";
const StoreContext = React.createContext();
export default StoreContext;
```

#### 4.6.4 User.jsx #

src\User.jsx

```
import { useContext,useRef } from 'react';
import StoreContext from './context';
import { observer } from "mobx-react";
const User = observer(function () {
  const { userStore } = useContext(StoreContext);
  const ref = useRef(null);
  return (

     {userStore.isLogin?(
      <>
      {userStore.username} userStore.logout()}>退出
      </>
     ):(
      <>

       userStore.login(ref.current.value)}>登录
      </>
     )}

  );
});
export default User;
```

#### 4.6.5 Todos.jsx #

src\Todos.jsx

```
import { useContext ,useRef} from 'react';
import StoreContext from './context';
import { observer } from "mobx-react";
import {TodoStore} from './store';
const Todo = observer(function ({todo}) {
  return (
    <li>
      <input
        type="checkbox"
        checked={todo.completed}
        onChange={() => todo.toggle()}
      />
      {todo.text}
    li>
  );
});

const TodoList = observer(function () {
  const { todoStore } = useContext(StoreContext);
  return (
    <div>
      <ul>
        {todoStore.list.map((todo, index) => (
          <Todo todo={todo} key={index} />
        ))}
      ul>
    div>
  );
});

const TodoLeft = observer(function () {
  const { todoStore } = useContext(StoreContext);
  return <>未完成: {todoStore.unCompletedCount}</>;
});

const AddTodo = observer(function AddTodo() {
  const { todoStore } = useContext(StoreContext);
  const ref = useRef(null);
  return (
    <>
      <input ref={ref} type="text" />
      <button
        onClick={() => {
          const item = new TodoStore(ref.current.value);
          todoStore.add(item);
          ref.current.value = "";
        }}
      >新增button>
    </>
  );
});
export default observer(function () {
  return (
    <>
      <AddTodo />
      <TodoList />
      <TodoLeft />
    </>
  );
});
```

#### 4.6.6 user\index.jsx #

src\store\user\index.jsx

```
import { makeAutoObservable } from "mobx";
class UserStore {
  username='';
  constructor() {
    makeAutoObservable(this, {}, { autoBind: true });
  }
  get isLogin() {
    return this.username.length > 0;
  }
  login(username) {
    this.username = username;
  }
  logout() {
    this.username = "";
  }
}
const userStore = new UserStore();
export default userStore;
```

#### 4.6.7 todos\index.jsx #

src\store\todos\index.jsx

```
import { makeAutoObservable } from "mobx";
class TodoStore {
  list = [];
  get unCompletedCount() {
    return this.list.filter((todo) => !todo.completed).length;
  }
  constructor() {
    makeAutoObservable(this,{},{autoBind: true});
  }
  add(todo) {
    this.list.push(todo);
  }
}
const todoStore = new TodoStore();
export default todoStore;
```

#### 4.6.8 todo.jsx #

src\store\todos\todo.jsx

```
import { makeAutoObservable } from "mobx";
export class TodoStore {
  text = "";
  completed = false;
  constructor(text) {
    makeAutoObservable(
      this,{},{autoBind: true}
    );
    this.text = text;
  }
  toggle() {
    this.completed = !this.completed;
  }
}
```

#### 4.6.9 store\index.jsx #

src\store\index.jsx

```
import todoStore from "./todos";
import userStore from "./user";
const store = { todoStore, userStore };
export { TodoStore } from "./todos/todo";
export default store;
```