11 Redux 状态管理

状态管理概述

在页面特别复杂时，可以使用状态管理，集中、统一地管理页面数据

状态管理之 Context

Context 可以跨层级传递，类似于 Vue 中的 `provide` 和 `inject`，可用于切换主题、语言等全局配置场景。

Step1 使用 `createContext(默认值)` **创建上下文对象**

Step2 父组件中使用 `<上下文对象.Provider value={状态数据}></ThemeContext.Provider>` 包裹子组件，**提供上下文**

```tsx
// index.tsx
import { createContext, FC, useState } from 'react';
import ToolBar from './ToolBar';

const themes = {
  light: {
    fore: '#000',
    background: '#eee',
  },
  dark: {
    fore: '#fff',
    background: '#222',
  },
};

export const ThemeContext = createContext(themes.light);

const Demo: FC = () => {
  const [theme, setTheme] = useState(themes.light);
  const toDark = () => setTheme(themes.dark);
  const toLight = () => setTheme(themes.light);

  return (
    <ThemeContext.Provider value={theme}>
      <div>
```

```
        <span>Context Demo</span>  
        <button onClick={toDark}>dark</button>  
        <button onClick={toLight}>light</button>
      </div>
      <ToolBar />
    </ThemeContext.Provider>
  );
};

export default Demo;
```

Step3 子组件中使用 `useContext(上下文对象)` **使用上下文**，即父组件通过 `上下文对象.Provider` 传递的 `value` 属性的值。

```
// ToolBar.tsx
import { FC } from 'react';
import ThemeButton from './ThemeButton';

const ToolBar: FC = () => {
  return (
    <>
      <p>ToolBar</p>
      <div>
        <ThemeButton />
      </div>
    </>
  );
};

export default ToolBar;
```

```
// ThemeButton.tsx
import { FC, useContext } from 'react';
import { ThemeContext } from '.';

const ThemeButton: FC = () => {
  const context = useContext(ThemeContext);
  const style = { color: context.fore, backgroundColor: context.background };
  return <button style={style}>ThemeButton</button>;
};

export default ThemeButton;
```

状态管理之 useReducer

1. 介绍：`useReducer` 是 `useState` 的替代方案，适合管理**复杂数据类型**，是精简版的 Redux。其一般语法为，

```
const [state, dispatch] = useReducer(reducer, initialState);
// 参数
// reducer 用于更新状态数据的函数，接收 prevState 和 action，返回新的 state
// initialState 表示状态数据的初始值
// 返回值
// dispatch 用于更新状态数据的函数，接收 action，内部自动调用 reducer，此时 state 会进行更新
// state 表示最新的状态数据
```

2. `useReducer` 的核心概念
   - state/store：**存储数据**（不可变数据）
   - action：**动作**，格式为 { type: 'xxx', ... }
   - reducer：根据 action **生成新的** state
   - dispatch：**触发** action
3. `useReducer` 的注意事项
   - `useReducer + useContext` 实现**跨组件通讯**
   - state 和 dispatch 默认**没有模块化**，导致数据混在一起，不适合于复杂项目

Step1 创建 `reducer` 和 `initialState`

Step2 调用 `useReducer`，生成 `state` 和 `dispatch`

Step3 使用 `state` 渲染页面，使用 `dispatch` 更新状态数据

```
import { FC, useReducer } from 'react';

type StateType = { count: number };
type ActionType = { type: 'increment' | 'decrement' };

const initialState: StateType = { count: 0 };

const reducer = (prevState: StateType, action: ActionType): StateType => {
  switch (action.type) {
    case 'increment':
      return { count: prevState.count + 1 };
```

```
    case 'decrement':
      return { count: prevState.count - 1 };
    default:
      throw new Error();
  }
};

const CountReducerDemo: FC = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      <h3>Count: {state.count}</h3>
      <p>
        <button onClick={() => dispatch({ type: 'increment' })}>plus</button>
 
        <button onClick={() => dispatch({ type: 'decrement' })}>minus</button>
      </p>
    </>
  );
};

export default CountReducerDemo;
```

useReducer + Context 解决跨组件问题

可以在根组件使用 `useReducer` 创建 `state` 和 `dispatch`，然后使用 Context 将其传递给所有子组件，此时所有组件可以通过唯一的 `state` 和 `dispatch` 渲染页面和修改状态，从而实现统一状态管理。

Step1 创建 `reducer` 和 `initialState`

```
// reducer.ts
import type { TodoType } from './store';

export type ActionType = {
  type: string;
  // eslint-disable-next-line @typescript-eslint/no-explicit-any
  payload?: any;
};

export default function (prevState: TodoType[], action: ActionType) {
  switch (action.type) {
    case 'add':
      return prevState.concat(action.payload); // 此时 payload 类型为 TodoType
```

```
    case 'delete':
      return prevState.filter(todo => todo.id != action.payload); // 此时 payload 类型为 string，表示要删除的 id
    default:
      throw new Error(`非法的 type=${action.type}`);
  }
}
```

```
// store.ts
import { nanoid } from 'nanoid';

export type TodoType = {
  id: string;
  title: string;
};

const initialState: TodoType[] = [
  { id: nanoid(5), title: '吃饭' },
  { id: nanoid(5), title: '睡觉' },
];

export default initialState;
```

Step2 父组件调用 `useReducer`，生成 `state` 和 `dispatch`，并通过 Context 进行传递

```
// index.tsx
import { createContext, FC, useReducer, Dispatch } from 'react';
import List from './List';
import InputForm from './InputForm';
import initialState, { TodoType } from './store';
import reducer, { ActionType } from './reducer';

type StateContextProps = {
  state: TodoType[];
  dispatch: Dispatch<ActionType>;
};

export const StateContext = createContext<StateContextProps>({
  state: initialState,
  dispatch: () => {},
});

const TodoListReducerDemo: FC = () => {
```

```
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <StateContext.Provider value={{ state, dispatch }}>
      <p>TodoList by Reducer</p>
      <List />
      <InputForm />
    </StateContext.Provider>
  );
};

export default TodoListReducerDemo;
```

Step3 子组件通过 Context 获取 `state` 和 `dispatch`，使用 `state` 渲染页面，使用 `dispatch` 更新状态数据

```
// List.tsx
import { FC, useContext } from 'react';
import { StateContext } from '.';

const List: FC = () => {
  const { state, dispatch } = useContext(StateContext);
  const style = { border: 'transparent', backgroundColor: 'transparent' };
  const handleDelClick = (id: string) => {
    return () => {
      confirm('确认删除? ') && dispatch({ type: 'delete', payload: id });
    };
  };

  return (
    <ul>
      {state.map(todo => (
        <li key={todo.id}>
          <span>{todo.title}</span> 
          <button style={style} onClick={handleDelClick(todo.id)}>
            ❌
          </button>
        </li>
      ))}
    </ul>
  );
};

export default List;
```

```tsx
// InputForm.tsx
import { ChangeEvent, FC, useContext, useState } from 'react';
import { nanoid } from 'nanoid';
import { StateContext } from '.';

const InputForm: FC = () => {
  const [text, setText] = useState('');
  const { state, dispatch } = useContext(StateContext);

  const handleChange = (event: ChangeEvent<HTMLInputElement>) =>
setText(event.target.value);
  const handleSubmit = (event: ChangeEvent<HTMLFormElement>) => {
    event.preventDefault();
    if (!text.trim()) return;
    const newTodo = { id: nanoid(5), title: text };
    dispatch({ type: 'add', payload: newTodo });
    setText('');
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={text}
        onChange={handleChange}
        placeholder="What needs to be done?"
      />
      <button type="submit">Add #{state.length}</button>
    </form>
  );
};

export default InputForm;
```

状态管理之 Redux ⭐

依赖安装

```
npm install @reduxjs/toolkit react-redux --save
```

使用方法

Step 1. 定义特定模块的 state、action、reducer

1. 这一步的核心是通过 `createSlice` **创建一个** `Slice` **对象**（看做是一个 Redux 模块），该对象接收一个配置对象，主要有以下三个字段，
   - `name` 模块**名称**
   - `initialState` 模块的**初始状态**
   - `reducers` 对象，**每一个字段取值为一个函数**，键可以看作是 `actionType`，值可以看作是相应动作的处理逻辑，用于返回一个最新的 `state`。

     > 关于 `reducers` 的字段对应的函数类型的说明 `(prevState: 状态类型, action: PayloadAction<参数类型>) => newState: 状态类型`
2. 定义好 `Slice` 对象后，需要**向外暴露当前 Redux 模块的** `action` **和** `reducer`，以便被统一管理。

   a. `Slice` 对象的 `actions` 属性是一个对象，包含着所有的 `action`。每个 `action` 是一个函数，可以理解为 `actionCreator`，接收在 `createSlice` 的 `reducers` 配置中定义的 `PayloadAction<参数类型>` 指定的数据类型。每个 `action` 执行后，才算是创建了一个动作，作为 `dispatch` 的参数，用于 `state` 的更新。

   b. `Slice` 对象的 `reducer` 属性就是要暴露出去的用于更新当前模块状态的纯函数。

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit";
import { nanoid } from "nanoid";

export type TodoItemType = {
  id: string;
  title: string;
  completed: boolean;
};

const INIT_STATE: TodoItemType[] = [
  { id: nanoid(5), title: "吃饭", completed: false },
  { id: nanoid(5), title: "睡觉", completed: true },
];

export const todoListSlice = createSlice({
  name: "todoList",
  initialState: INIT_STATE,
  reducers: {
    addTodo(state: TodoItemType[], action: PayloadAction<TodoItemType>) {
      return [action.payload, ...state];
    },
    removeTodo(state: TodoItemType[], action: PayloadAction<{ id: string }>) {
      const { id: removeId } = action.payload;
```

```
      return state.filter((todo) => todo.id !== removeId);
    },
    toggleTodo(state: TodoItemType[], action: PayloadAction<{ id: string }>) {
      const { id: toggleId } = action.payload;
      return state.map((todo) =>
        todo.id === toggleId ? { ...todo, completed: !todo.completed } : todo
      );
    },
  },
});

export const { addTodo, removeTodo, toggleTodo } = todoListSlice.actions;

export default todoListSlice.reducer;
```

Step 2. 合并所有模块

这一步的核心有两个部分

1. 通过 `configureStore` **创建一个** `Store` **对象**，并向外默认暴露。
   - 该对象接收一个配置对象，主要有 `reducer` 字段，该字段是一个对象，每一个字段的键为 Redux 模块的 `name` 配置名，值为 Redux 模块默认暴露的 `reducer` 纯函数。
   - 该对象用于管理整个应用程序的状态数据，有 `getState`、`dispatch`、`subscribe` 等原生 Redux 中的常见 API。
2. **创建一个** `StateType` **类型**，用于声明 Redux 集中管理的 `state` 的数据类型，并向外暴露。`StateType` 的键为 Redux 模块的 `name` 配置名，值为 Redux 模块对应的 `state` 的数据类型。

```
import { configureStore } from "@reduxjs/toolkit";
import countReducer from "./count";
import todoListReducer, { TodoItemType } from "./todoList";

export type StateType = {
  count: number;
  todoList: TodoItemType[];
};

export default configureStore({
  reducer: {
    count: countReducer,
    todoList: todoListReducer,
  },
});
```

Step 3. 提供 store 给整个应用程序

在程序的入口文件中，使用 `<Provider store={store}></Provider>` 包裹根组件 `<App/>`，此时 Redux 才得以管理整个应用程序的状态。

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App.tsx";
import { Provider } from "react-redux";
import store from "./store/index.ts";

console.log(store)
ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

Step 4. 访问和修改特定模块的 state

1. 通过 `useDispatch` Hook 获取到 `dispatch` 函数，结合导入的 `action`（严格来说是 `actionCreator`）来更新 `state`。

```
const dispatch = useDispatch();
```

2. 通过 `useSelector` Hook 获取到指定的 Redux 模块的数据。

```
const reduxModuleState = useSelector<StateType>(
    (state) => state.reduxModuleName
  ) as reduxModuleStateType[];
```

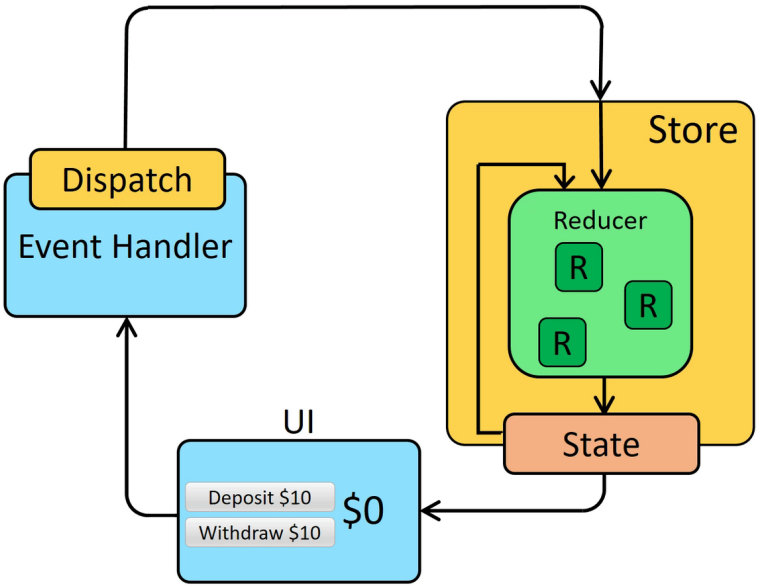
> 这里的 `state` 表示合并后的 `state`，根据其类型 `StateType`，可以获取到特定 Redux 模块的 `state` 数据。

```
import { FC, useState } from "react";
import { useDispatch, useSelector } from "react-redux";
```

```
import {
  TodoItemType,
  addTodo,
  removeTodo,
  toggleTodo,
} from "../store/todoList";
import { StateType } from "../store";
import { nanoid } from "nanoid";

const TodoList: FC = () => {
  const todoList = useSelector<StateType>(
    (state) => state.todoList
  ) as TodoItemType[];
  const dispatch = useDispatch();
  const [text, setText] = useState("");

  return (
    <div>
      <h2>Todo List Redux Demo</h2>
      <ul>
        {todoList.map((todo) => {
          const { id, title, completed } = todo;
          return (
            <li key={id}>
              <span
                style={{
                  color: completed ? "green" : "black",
                  textDecoration: completed ? "line-through" : "",
                }}
              >
                {title}
              </span>{" "}
                 
              <span
                onClick={() => dispatch(toggleTodo({ id }))}
                style={{ cursor: "pointer" }}
              >
                {completed ? "🟦" : "🔲"}
              </span>
               
              <span
                onClick={() => dispatch(removeTodo({ id }))}
                style={{
                  cursor: "pointer",
                  backgroundColor: "red",
                  borderRadius: "3px",
                  fontSize: "13px",
```

```
                  color: "white",
                }}
              >
                删除
              </span>
            </li>
          );
        })}
      </ul>
      <div>
        <input
          type="text"
          value={text}
          onChange={(event) => setText(event.target.value.trim())}
          style={{ width: "10%" }}
        />
        <button
          onClick={() => {
            text &&
              dispatch(
                addTodo({ id: nanoid(5), title: text, completed: false })
              );
            setText("");
          }}
        >
          Add #{todoList.length}
        </button>
      </div>
    </div>
  );
};

export default TodoList;
```

Redux 单向数据流



Redux 开发者工具 Redux DevTools

状态管理之 MobX

与 Redux 相比，MobX 的最大特点是**声明式**状态管理。

依赖安装

```
npm install mobx mobx-react --save
```

核心概念

1. State：驱动你的应用程序的数据。
2. Actions：任意可以改变 *State* 的代码。
3. Derivations：任何来源是 State 并且不需要进一步交互的东西。

> Derivations 分为两类
>
> - Computed values：通过纯函数从当前的可观测的 State 中派生的数据。
> - Reactions：State 改变时需要自动执行的副作用。

使用方法

Step 1. 创建可被观察的 State —— observable

使用 `makeObservable` 在 `constructor` 中通过以下方式标记指定属性为 `observable`，使其可以被 MobX 追踪。

Step 2. 创建更新 State 的 Actions —— action

与标记 State 类似，可以使用 `action` 标识 Action。建议将所有修改 `observable` 的值的代码标记为 `action`。

Step 3. 创建响应 State 变化的 Derivations —— computed、autorun

1. 定义 `getter` 方法 + 使用 `makeObservable` 将标记指定属性为 `computed` → 创建 Computed Values。该数据会在 State 变化时自动更新。
2. 可以通过 `autorun` 等方法来自定义 Reactions，其接收一个回调函数，该回调函数会在所用 State 变化时自动执行。

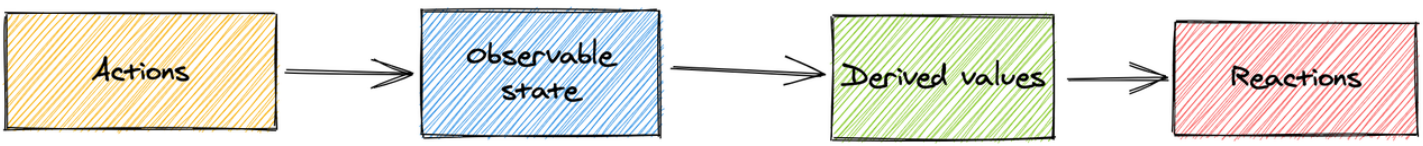
Step 4. 创建响应式 React 组件

通过 `observer` 将 React 组件包裹起来，此时组件便具有响应式特性。`observer` 将 React 组件转换为了**从数据到渲染的派生**过程，此时所有的组件在渲染时都是智能的。

```
import * as React from "react";
import { render } from "react-dom";
import { observer } from "mobx-react-lite";
import { makeObservable, observable, computed, action } from "mobx";

class Todo {
  id = Math.random();
  title = "";
  finished = false;

  constructor(title) {
    makeObservable(this, {
      title: observable, // state
      finished: observable, // state
      toggle: action // action
    });
    this.title = title;
  }

  toggle() {
    this.finished = !this.finished;
  }
}

class TodoList {
  todos = [];
  get unfinishedTodoCount() {
    return this.todos.filter(todo => !todo.finished).length;
  }
```

```
30    constructor(todos) {
31      makeObservable(this, {
32        todos: observable, // state
33        unfinishedTodoCount: computed // derivation-computedValues
34      });
35      this.todos = todos;
36    }
37  }
38
39  const TodoListView = observer(({ todoList }) => (
40    <div>
41      <ul>
42        {todoList.todos.map(todo => (
43          <TodoView todo={todo} key={todo.id} />
44        ))}
45      </ul>
46      Tasks left: {todoList.unfinishedTodoCount}
47    </div>
48  ));
49
50  const TodoView = observer(({ todo }) => (
51    <li>
52      <input
53        type="checkbox"
54        checked={todo.finished}
55        onClick={() => todo.toggle()}
56      />
57      {todo.title}
58    </li>
59  ));
60
61  const store = new TodoList([
62    new Todo("Get Coffee"),
63    new Todo("Write simpler code")
64  ]);
65  render(<TodoListView todoList={store} />, document.getElementById("root"));
```

MobX 单向数据流



- Derivations 在 State 改变时**自动且原子化地更新**。
- Derivations 默认**同步**更新。

- Derivations-computedValue 的更新是**惰性**的，任何需要其的副作用发生前都是不激活的。
- Derivations-computedValue 应该是**纯函数**，不能修改 State。

Redux 管理用户信息

useGetUserInfo

获取 Redux 中 user 状态，包括 { username, password }。

```ts
import { useSelector } from 'react-redux';
import { StateType } from '@/store';
import { UserStateType } from '@/store/user';

/**
 * @description 获取 Redux 中 user 状态，包括 { username, password }
 * @returns {UserStateType} 用户状态对象
 */
export default function useGetUserInfo(): UserStateType {
  const userState = useSelector<StateType>(state => state.user) as UserStateType;
  return userState;
}
```

useEnsure

确保 Redux 中已存储用户信息，否则尝试获取用户信息。

```ts
import { getUserInfoService } from '@/service/user';
import { loginReducer, UserStateType } from '@/store/user';
import { useRequest } from 'ahooks';
import { useEffect } from 'react';
import { useDispatch } from 'react-redux';
import useGetUserInfo from './useGetUserInfo';

/**
 * @description 确保 Redux 中已存储用户信息，否则尝试获取用户信息。
 */
export default function useEnsureUserData() {
  const dispatch = useDispatch();
  const { run: loadUserData, loading: isUserDataLoading } = useRequest(getUserInfoService, {
    manual: true,
    onSuccess(res) {
```

```
      dispatch(loginReducer(res as UserStateType));
    },
  });
  const { username } = useGetUserInfo(); // 从 Redux 中获取用户信息，根据是否存在决定是否请求用户数据
  useEffect(() => {
    if (!username) {
      loadUserData();
    }
  }, [loadUserData, username]);
  return isUserDataLoading;
}

/*
  注：关于 useEnsureUserData 和 useGetUserInfo 的使用时机
  1. useEnsureUserData 用于确保 Redux 中已经存在用户数据，或在没有时完成用户数据的请求。
  2. useGetUserInfo 用于从 Redux 中获取用户数据。
  3. 使用场景：
    - useEnsureUserData 在页面加载时执行，可以在顶级的 Layout 组件中执行（例如 MainLayout、QuestionLayout），确保用户数据已被请求。
    - useGetUserInfo 在需要使用用户信息的组件中执行，例如 UserInfo 组件等。
*/
```

useNavPage

```
import { useLocation, useNavigate } from 'react-router-dom';
import useGetUserInfo from './useGetUserInfo';
import { useEffect } from 'react';
import { HOME_PATHNAME, LOGIN_PATHNAME, MANAGE_LIST_PATHNAME, REGISTER_PATHNAME } from '@/router';

/**
 * 自定义 Hook 用于根据用户的认证状态处理页面导航。
 *
 * @description
 * 此 Hook 使用用户的认证状态和当前 URL 路径名来确定是否需要将用户重定向到其他页面。
 * 如果用户的数据仍在加载中，则不执行任何操作。数据加载完毕后：
 * - 如果用户已认证，且当前处于登录或注册页面，则重定向到管理列表页面。
 * - 如果用户未认证，且不在登录、注册或主页页面，则重定向到登录页面。
 *
 * @param {boolean} isUserDataLoading - 表示用户数据是否仍在加载中。
 * @returns {void}
 */
export default function useNavPage(isUserDataLoading: boolean): void {
```

```
  const { username } = useGetUserInfo(); // Hook 获取当前用户信息
  const { pathname } = useLocation(); // Hook 获取当前 URL 路径名
  const nav = useNavigate(); // Hook 用于编程式导航到不同页面

  useEffect(() => {
    if (isUserDataLoading) return; // 如果用户数据仍在加载中，则不执行任何操作

    if (username) {
      // 如果用户已认证且当前处于登录或注册页面，则重定向到管理列表页面
      if ([LOGIN_PATHNAME, REGISTER_PATHNAME].includes(pathname))
nav(MANAGE_LIST_PATHNAME);
    } else {
      // 如果用户未认证且不在登录、注册或主页页面，则重定向到登录页面
      if (![LOGIN_PATHNAME, REGISTER_PATHNAME,
HOME_PATHNAME].includes(pathname))
        nav(LOGIN_PATHNAME);
    }

    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [isUserDataLoading, username, pathname]);
}
```