Redux套用七步驟

本章目標

- 學習Redux框架的套用步驟
- 學習Redux框架中store的基礎知識
- 學習Redux框架中reducer的基礎知識

套用七步驟

(1) 從redux模組中匯入createStore函式

(2) 撰寫一個reducer(歸納)函式

(3) 由寫好的reducer，建立出store

(4) 撰寫一個render(渲染)函式，在狀態有更動時作重新呈現

(5) 第一次調用render函式，作初始呈現

(6) 訂閱render函式到store中

(7) 觸發事件時要呼叫store.dispatch(action)方法

store詳解

store的原型

```
type Store = {
  dispatch: Dispatch
  getState: () => State
  subscribe: (listener: () => void) => () => void
  replaceReducer: (reducer: Reducer) => void
}
```

store的方法

- dispatch 用於發送action(動作)使用的方法
- getState 取得目前state的方法
- subscribe 註冊一個回調函式，當state有更動時會呼叫它
- replaceReducer 高級API，用於動態加載其它的reducer，一般情況不會用

注: 之後使用於React專用綁定套件後，subscribe方法不需使用

dispatch方法

```
dispatch: (action: A) => A
```

- 唯一可以觸發更動state的方法
- 傳入與返回類型都是一個action(動作)物件
- 傳參通常是一個Action Creator的呼叫，這種樣式稱作"函式合成"
- dispatch發送動作，store中的reducer將會同步傳入目前的狀態以及給定的action，開始計算新的狀態最後返回。回傳後更動的監聽目標將會被通知(用subscribe註冊的回調)，再調用`getState()`可得到新的狀態

合成様式示例

```
function a(x) { return x*x }
function b(x) { return x*2 }
function c(x) { return x+1 }

a(b(c(10)))
```

dispatch方法示例

```
1.  // 由todos這個reducer創建store
2.  // 第二個傳參是初始的狀態，是可選的
3.  let store = createStore(todos, [ '学习Redux' ])
4.
5.  // Action Creator(動作建立器)
6.  function addTodo(text) {
7.    return {
8.      type: 'ADD_TODO',
9.      text
10.   }
11. }
12.
13. // 用store.dispatch(action)發送動作
14. // 傳參先用Action Creator(動作建立器)創建出action物件格式
15. store.dispatch(addTodo('观看Redux教程'))
16. store.dispatch(addTodo('练习Redux示例'))
```

getState方法

```
getState(): S
```

回傳目前state樹資料，相當於reducer最後返回出來的值

reducer詳解

Reducer(歸納函式)

```
Reducer<S, A> = (state: S, action: A) => S
```

reducer要求的是純函數而且無副作用

状态模型示例–1

```
const todos = [
    {id: 1, text: '买台车'},
    {id: 2, text: '学习Redux' },
    ...
]
```

状态模型示例-2

```
const blogPosts = [
   {
    "id": "123",
    "author": {
      "id": "1",
      "name": "小张"
    },
    "title": "我的第一个博客",
    "comments": [
      {
        "id": "324",
        "commenter": {
          "id": "2",
          "name": "小杨"
        }
      }
 ]}]
```

狀態更動示例 – 添加

```
function addItem(state = [], action) {
  return [action.payload, ...state]
}
```

通用規則：拷貝原先的狀態，作更動後返回新的狀態

狀態更動示例 – 添加

```
function addItem(state = [], action) {
  // 拷貝出一個新的陣列
  // newState = [...state]語法也可以
  // newState = state.concat()也可以
  const newState = state.slice()

  // 附加新成員在新的陣列前面，
  // 注意這是一個有副作用的陣列方法
  newState.unshift(action.payload)

  // 回傳處理過的新陣列
  return newState
}
```

狀態更動示例 – 移除

```
function removeItem(state = [], action) {
  return state.filter((item, index) => index !== action.index)
}
```

`filter`會依照回調傳參布爾結果進行過濾，產生一個全新的數組

狀態更動示例 – 對象更動內容

```
const initialState = {
   fetching: false,
   list: []
}

function addUser(state = initialState, action) {
    return Object.assign({}, state, { fetching: true })
}
```

`Object.assign`可作多個對象的淺拷貝，有相同的鍵值會以後面的對象內容覆寫