Redux套用於React應用

本章目標

- 學習Redux框架如何套用到React中(不用綁定套件)
- 學習Redux框架中單向資料流的運作過程
- 學習Flux標準動作、動作類型命名與常量基礎知識

套用七步驟(承上一章)

(1) 從redux模組中匯入createStore函數

(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

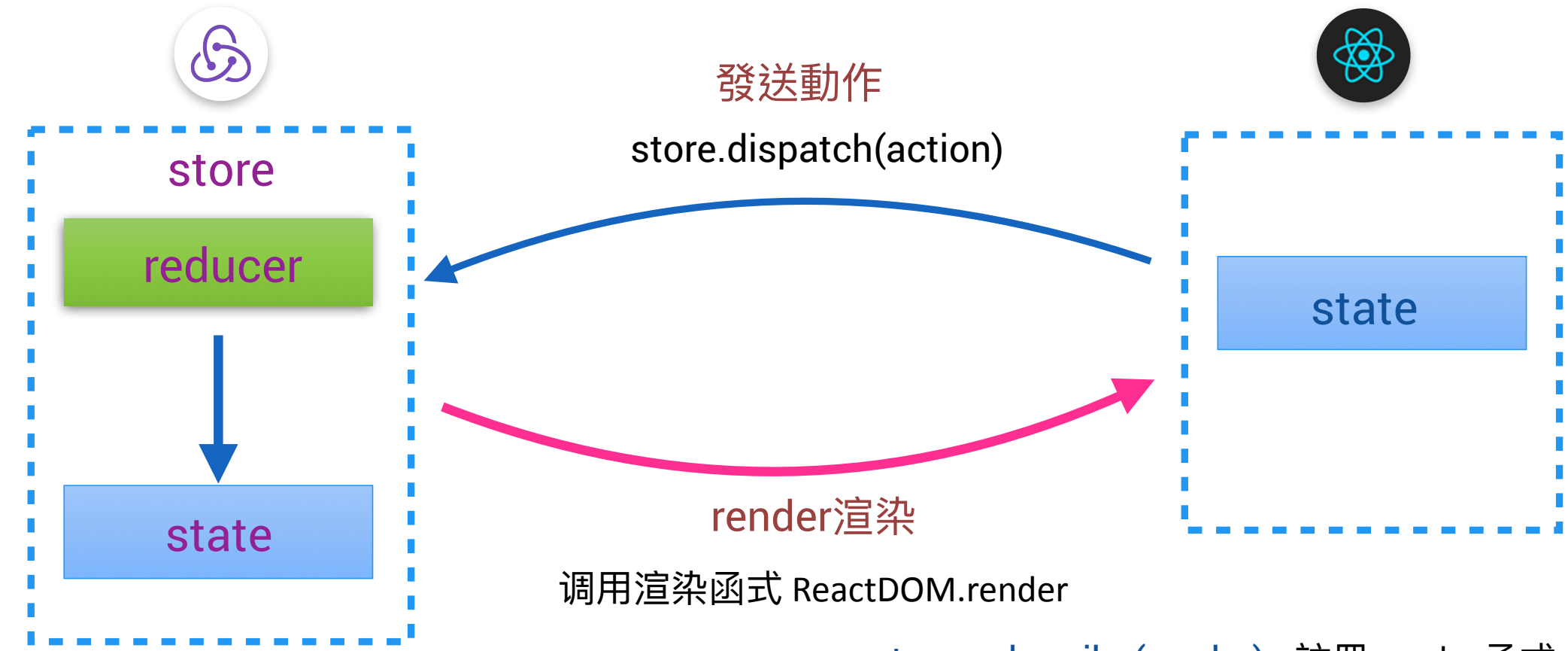
(5) 第一次調用render函式，作初始呈現

(6) 訂閱render函式到store中

(7) 觸發事件時要呼叫store.dispatch(action)方法

React組件

Redux與React整合關係圖



單向資料流詳解

單向資料流步驟

(1) 事件觸發時，調用`store.dispatch(action)`

(2) Redux的store會調用你給定的reducer函式

(3) 根reducer可以合併多個reducers為單一個狀態(state)樹 (`combineReducers`函式)

(4) Redux的store會儲存由根reducer返回的整個狀態(state)樹

事件觸發時呼叫store.dispatch(action)

只使用動作物件

```
const action = {
  type: 'ADD_TODO',
  payload: {
    id: 12345, text: '学习Redux'
  }
}

store.dispatch(action)
```

改用動作建立器

```
function addTodo(id, text) {
  return { type: ADD_TODO, payload:{ id, text } }
}

store.dispatch(addTodo(id, text))
```

Redux store呼叫給定的reducer

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

前一個狀態(目前狀態) + 動作 => 新的狀態

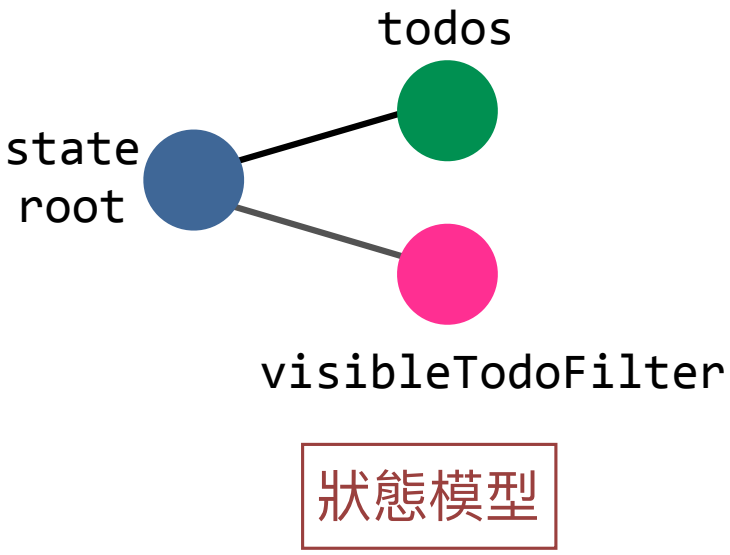
根reducer合併多個reducers為單一個狀態樹

```
// reducer
function todos(state = [], action) {
  return nextState
}

// reducer
function visibleTodoFilter(state = 'SHOW_ALL', action) {
  return nextState
}

// 合併為單一的reducer，注意傳參為合併函式為一物件
let todoApp = combineReducers({
  todos,
  visibleTodoFilter
})
```

combineReducers

- 傳參為物件，「對象字面屬性初始設定簡寫法」
- 返回是一個reducer，但與原本寫的單個reducer構造不同，是序列依次調用其中的reducer

```
let todoApp = combineReducers({
  todos,
  visibleTodoFilter,
})

let todoApp = combineReducers({
  todos: todos,
  visibleTodoFilter: visibleTodoFilter,
})
```



Redux store儲存由根reducer返回的狀態樹

- 當狀態樹有更動時，所有以`store.subscribe(listener)`註冊的監聽者(渲染函式)會被自動呼叫
- 監聽者(渲染函式)中可以透過`store.getState()`方法，獲取新的狀態值
- 如果使用的是React應用，這個時間點相當於呼叫`component.setState(newState)`

動作物件詳解

Flux標準動作(FSA)是什麼

- 用於標準統一化Action(動作)物件結構的社群標準

```
{
  type: 'ADD_TODO',
  payload: {
    text: '学Redux'
  }
}
```

一般資料

```
{
  type: 'ADD_TODO',
  payload: new Error(),
  error: true
}
```

錯誤/例外

```
{
  type: 'ADD_TODO',
  payload: {
    text: 'Do something.'
  },
  meta: {
    steps: [
      [success, failure]
    ]
  }
}
```

額外資料

動作類型的命名

- 動作_對象 (動詞_名詞)

  ADD_TODO
  DELETE_TODO
  UPDATE_TODO

- 功能/區域_動作_對象 (名詞_動詞_名詞)

  CART_ADD_ITEM
  CART_DELETE_ITEM
  PRODUCT_ADD_ITEM

- 動作_名詞_狀態 (動詞_名詞_形容詞/動詞)

  FETCH_USER_REQUEST
  FETCH_USER_SUCCESS
  FETCH_USER_FAIL
  FETCH_USER_COMPLETE

動作建立器(函式)的命名

- 通常跟隨著動作類型的名稱來命名，容易辨別

```
const addTodo = payload => ({
  type: ADD_TODO,
  payload,
})


const fetchUserRequest = payload => ({
  type: FETCH_USER_REQUEST,
  payload,
})
```

動作類型使用常數來代表字串

- 將動作類型的字串，先以常數(固定變數)代表，之後在動作建立器、reducer中使用這常數

```
// ./actionTypes.js
export const ADD_TODO = 'ADD_TODO'
export const DELETE_TODO = 'DELETE_TODO'
export const EDIT_TODO = 'EDIT_TODO'

// ./actions.js
import { ADD_TODO } from './actionTypes'

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

動作類型使用常數來代表之目的

- 字串寫錯時開發期間不會報錯(主控台/編輯工具)，難以除錯。但常數(固定變數)會報錯
- 集中所有的動作類型到同一程式碼檔案中，易於維護使用
- 養成好習慣，減少寫錯的發生機會

動作與狀態更動關係不是」1對1"，是"多對多"
(一個動作對一個狀態更動)

```
function firstReducer(state, action) {
    switch (action.type) {
        case ACTION_X:
            // 處理動作 x 的狀態更動
        case ACTION_Y:
            // 處理動作 y 的狀態更動
    }
}
```

動作與狀態更動關係不是」1對1"，是"多對多"（兩個動作對一個狀態更動）

```
function firstReducer(state, action) {
    switch (action.type) {
        case ACTION_X:
        case ACTION_Y:
            // 處理動作 x 或 y 的狀態更動
    }
}
```

動作與狀態更動關係不是」1對1"，是"多對多"（一個動作對兩個狀態更動）

```
function firstReducer(state, action) {
    switch (action.type) {
        case ACTION_X:
            // 處理動作 x 的狀態更動
    }
}

function secondReducer(state, action) {
    switch (action.type) {
        case ACTION_X:
            // 處理動作 x 的狀態更動
    }
}
```

省略用`combineReducers`合併的程式碼