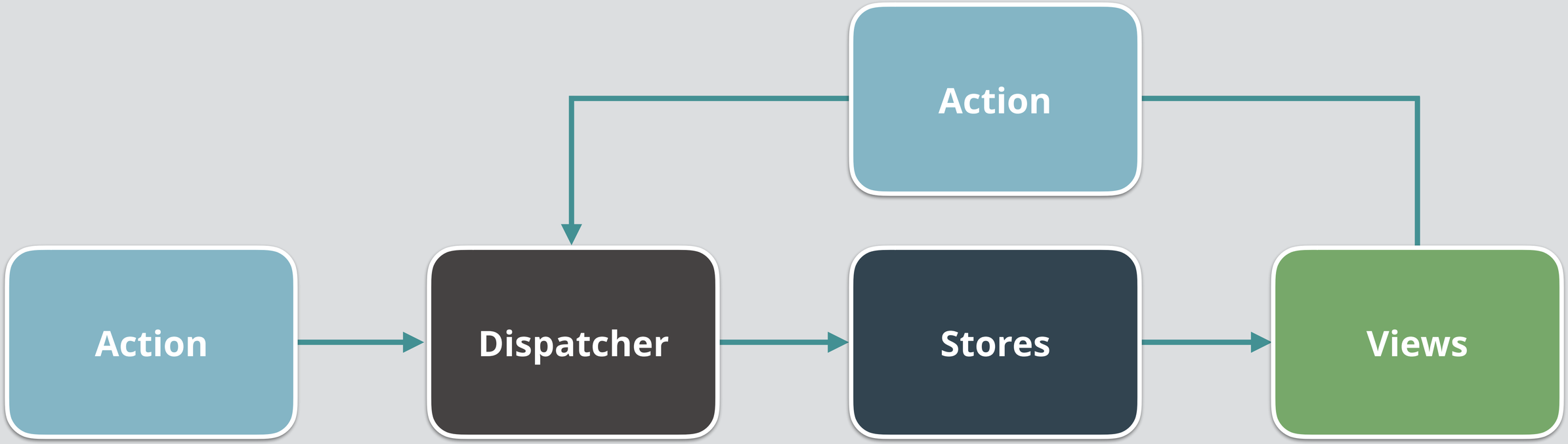
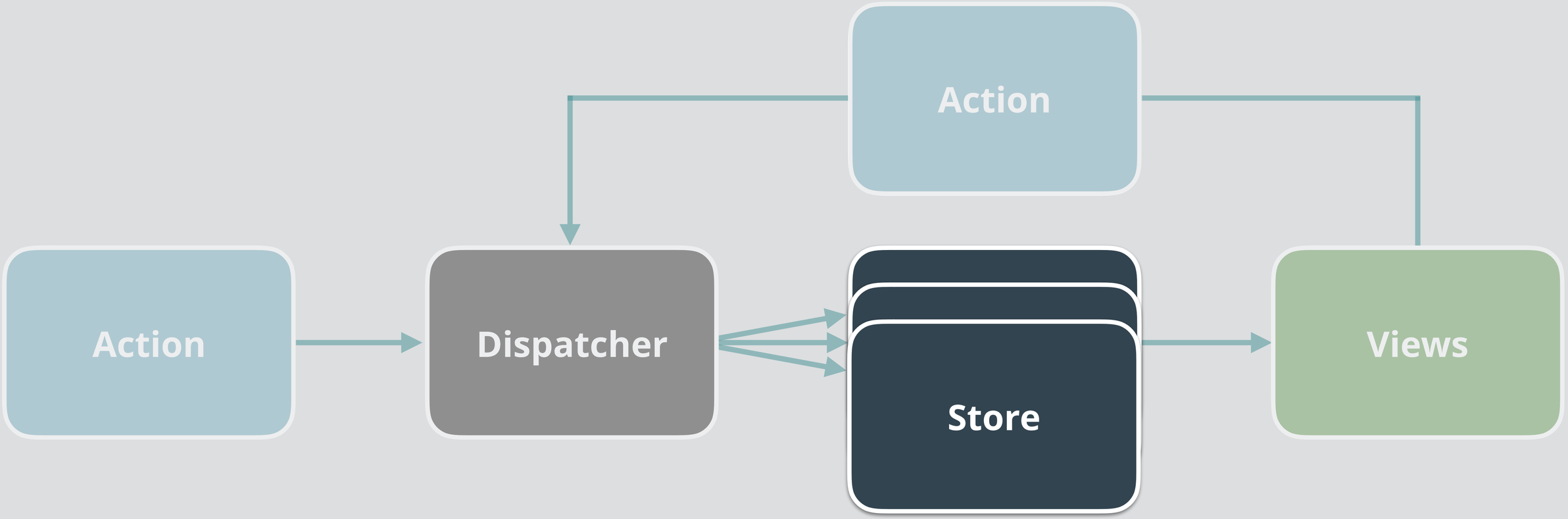
Redux

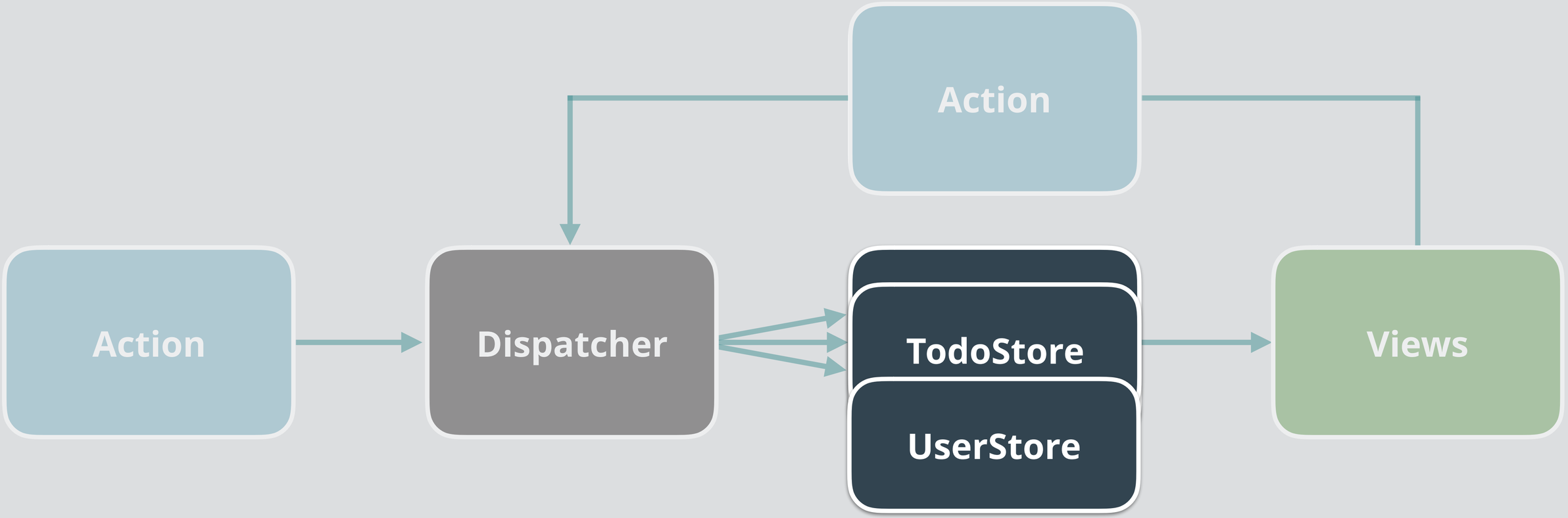
探討 Redux 前，先回顧

Flux


Action
Action
Dispatcher
Stores
Views

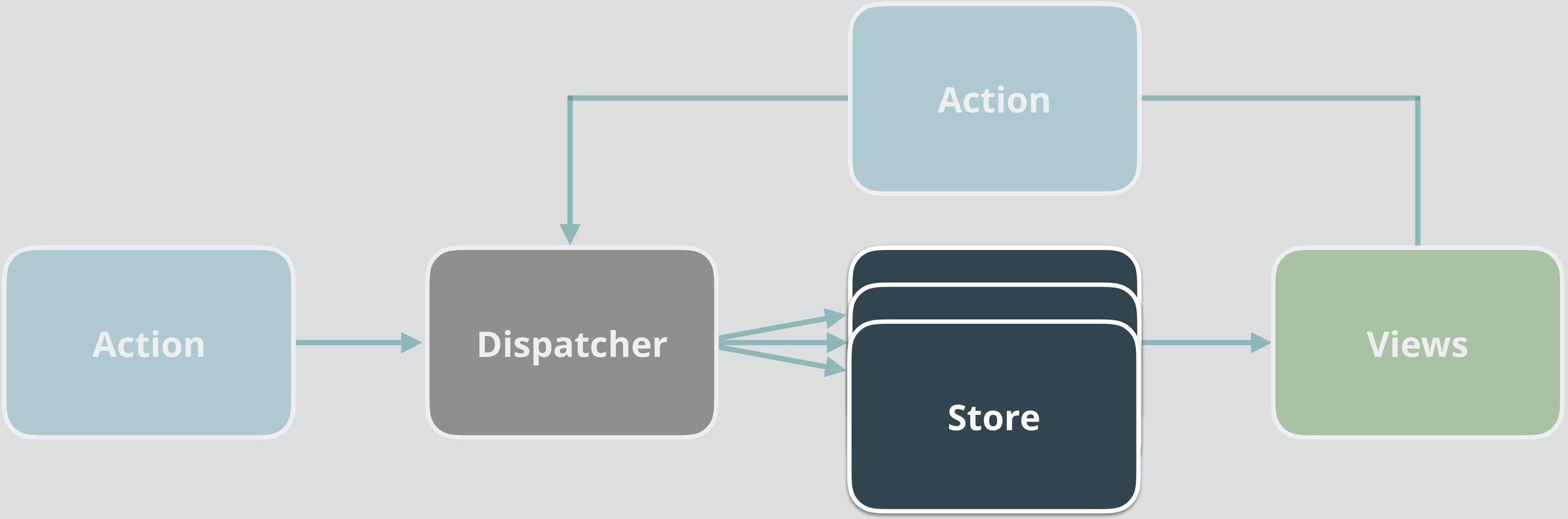


實務上，我們會根據不同業務拆分 Store；

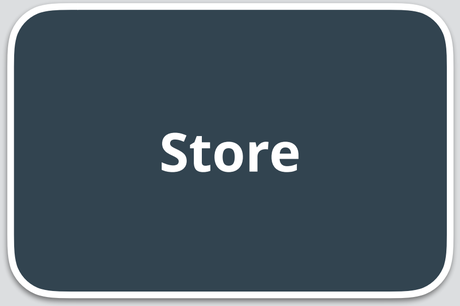


實務上，我們會根據不同業務拆分 Store；

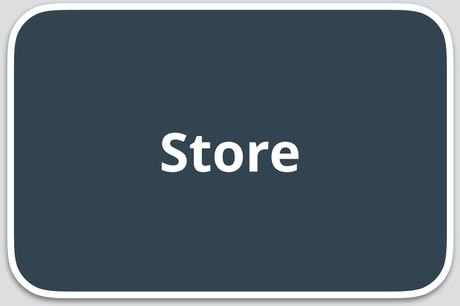
例如：TodoStore, UserStore…



那麼，你還記得 Store 做了哪些事情嗎？



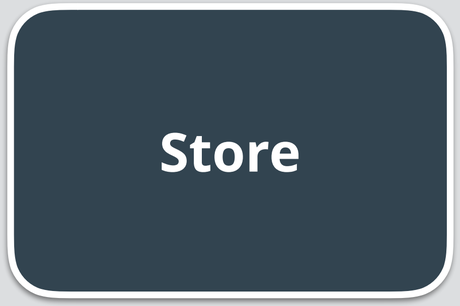
Store 有兩個主要職責：




```
let state = 0
```

Store 有兩個主要職責：

1. **儲存資料狀態**




```
let state = 0

function change(action) {
  state = action.data
}
```

Store 有兩個主要職責：

1. **儲存資料狀態**
2. **根據 Action 改變資料狀態**

使用 Flux 會遇到的

問題

Store
state: 10

```
let state = 0

function change(action) {
  state = action.data
}
```

當我們修改...

Store
state: 10

```
let state = 0

function change(action) {
  state = action.data + 1
}
```

當我們修改**根據 Action 改變狀態的邏輯**；

Store
state: 0

```
let state = 0

function change(action) {
  state = action.data + 1
}
```

當我們修改根據 Action 改變狀態的邏輯；

因為整支 Store 更新了，**資料狀態也會重置！**

你可以想像當你改完一個 bug，

你可以想像當你改完一個 bug，

為了測試就必須再跑一次註冊流程的痛苦嗎？

如何

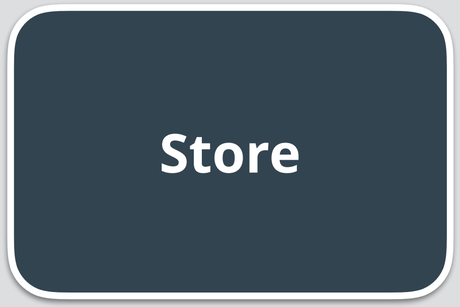
解決？

Store

```
let state = 0

function change(action) {
  state = action.data
}
```

你有發現這關鍵問題是什麼嗎？

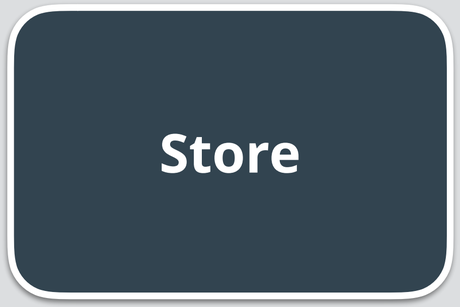



```
let state = 0

function change(action) {
  state = action.data
}
```

Store 有兩個主要職責：

1. 儲存狀態
2. 根據 Action 改變狀態




```
let state = 0

function change(action) {
  state = action.data
}
```

Store 有兩個主要職責，

我們必須把它們拆開，讓它們不會互相影響。

啟發於

Elm

在 Elm architecture 中，更新 Model 的方式是：

```
(msg, model) => model'
```

在 Elm architecture 中，更新 Model 的方式是：

```
(msg, model) => model'
```

翻成白話文就是：

```
(行為，狀態）=> 新狀態
```

在 Elm architecture 中，更新 Model 的方式是：

`(msg, model) => model'`

翻成白話文就是：

(行為，狀態) => 新狀態

再白一點就是：

透過一個函數，給予行為和狀態，回傳新狀態

所以，我們把根據 Action 改變狀態的邏輯，
定義成一個函數：

```
(action, state) => state'
```

所以，我們把根據 Action 改變狀態的邏輯，
定義成一個函數：

```
(action, state) => state'
```

我們叫這一個函數為 **reducer**。

Redux 最核心的觀念

Reducer

```
(action, state) => state'
```

Reducer 的好處是：

1. **可預期**：當參數給予一樣的，回傳結果就會一樣

```
(action, state) => state'
```

Reducer 的好處是：

1. **可預期**：當參數給予一樣的，回傳結果就會一樣
2. **函數編程**：它就只是一個簡單的函數

```
(action, state) => state'
```

Reducer 的好處是：

1. **可預期**：當參數給予一樣的，回傳結果就會一樣
2. **函數編程**：它就只是一個簡單的函數
   2.1 可以當作參數傳遞，或者擁有其他函數編程的優點

```
(action, state) => state'
```

Reducer 的好處是：

1. **可預期**：當參數給予一樣的，回傳結果就會一樣
2. **函數編程**：它就只是一個簡單的函數
   2.1 可以當作參數傳遞，或者擁有其他函數編程的優點
   2.2 可以做到 undo/redo 的功能

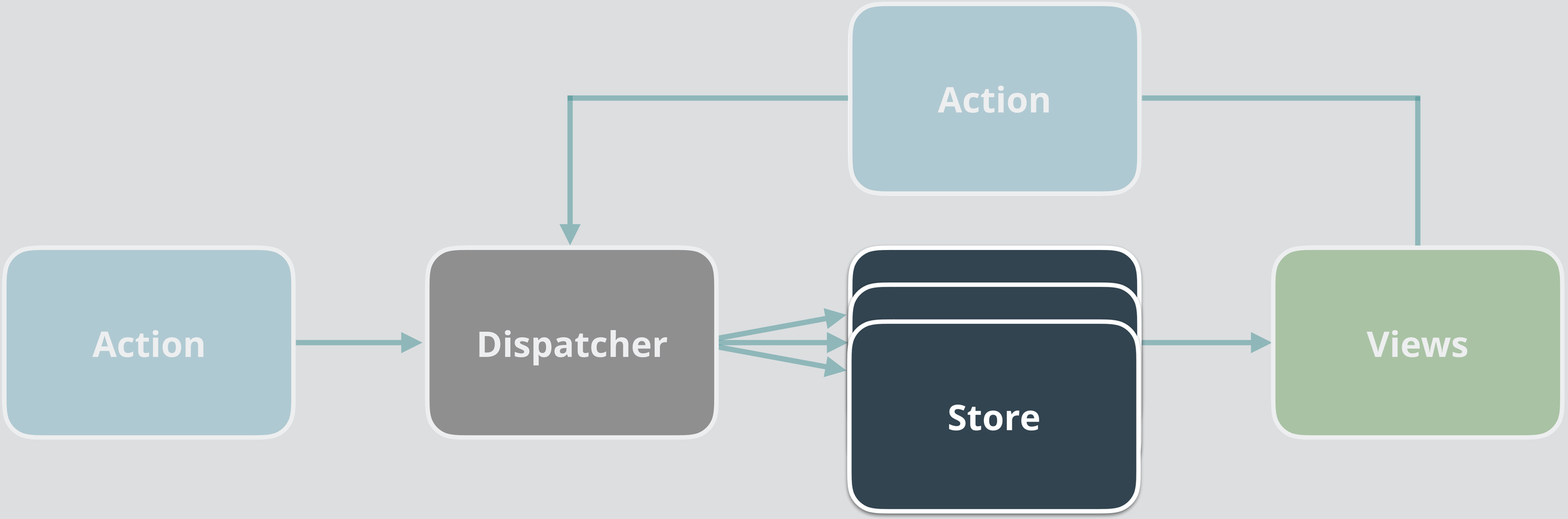
```
(action, state) => state'
```

Reducer 的好處是：

1. **可預期**：當參數給予一樣的，回傳結果就會一樣
2. **函數編程**：它就只是一個簡單的函數
   2.1 可以當作參數傳遞，或者擁有其他函數編程的優點
   2.2 可以做到 undo/redo 的功能
   2.3 可以任意改變調用函數的順序

Redux

的成形

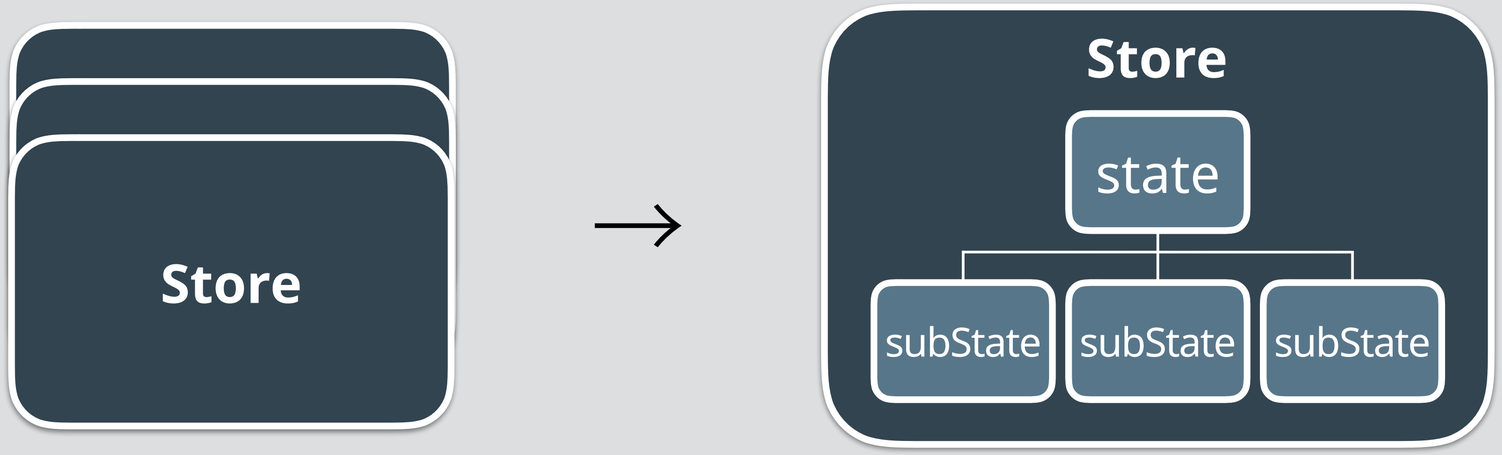


如果我們調整 Store 這個角色：

→

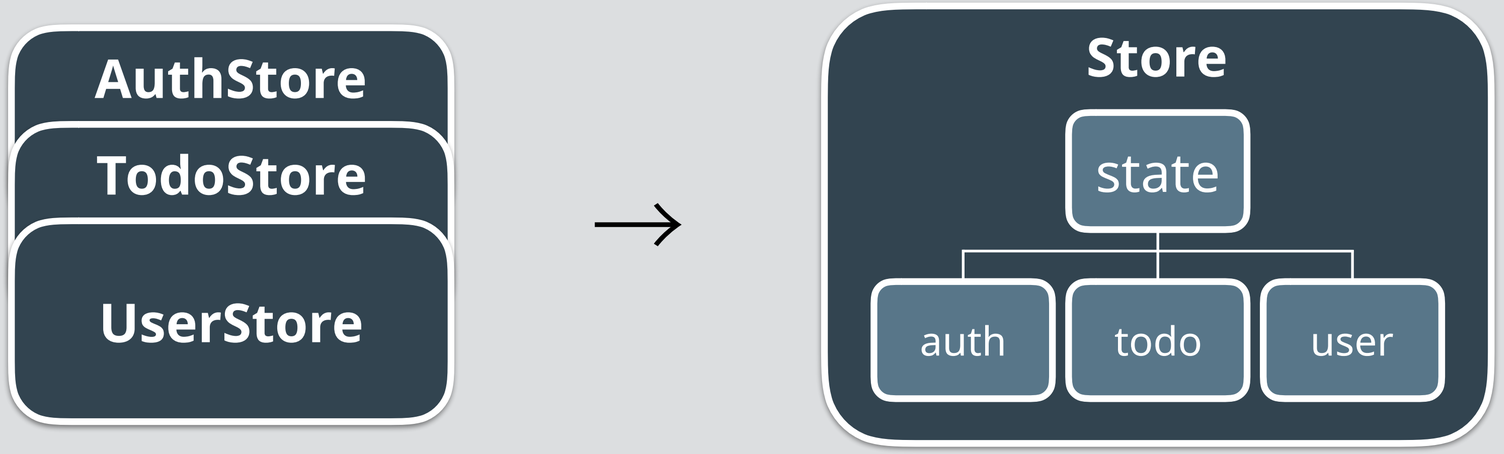
如果我們調整 Store 這個角色：

1. 將每個 Store 的業務狀態 →



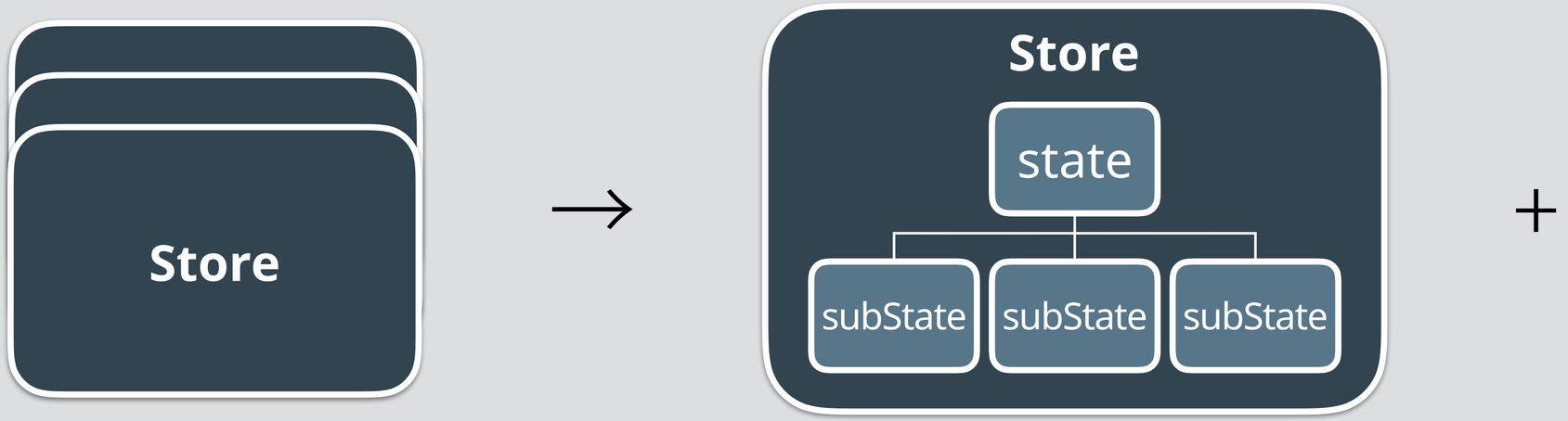
如果我們調整 Store 這個角色：

1. 將每個 Store 的業務狀態 → 組成一個狀態樹



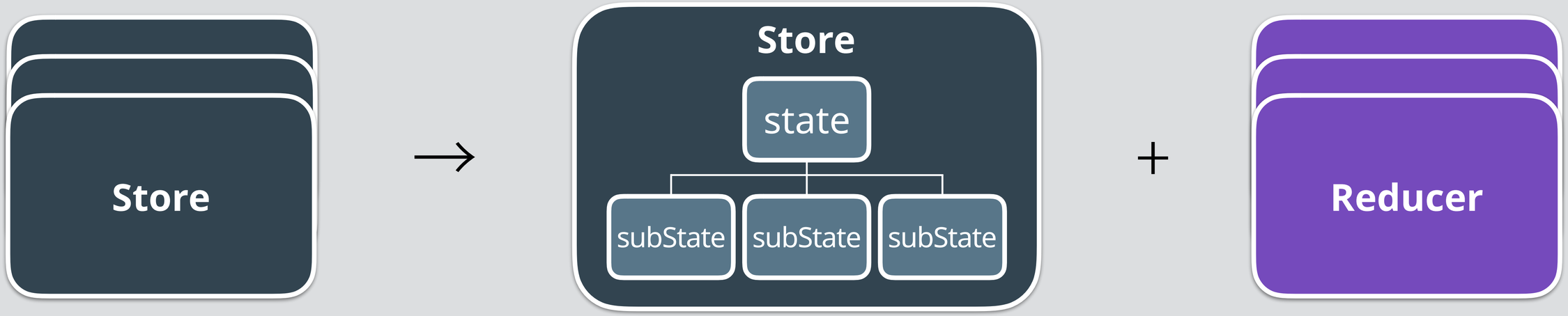
如果我們調整 Store 這個角色：

1. 將每個 Store 的業務狀態 → 組成一個狀態樹



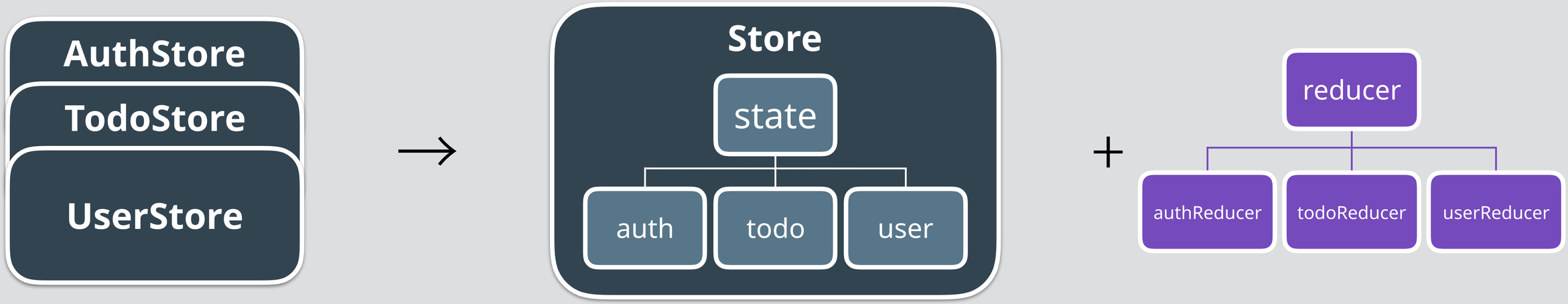
如果我們調整 Store 這個角色：

1. 將每個 Store 的業務狀態 → 組成一個狀態樹
2. 將每個 Store 改變狀態的邏輯



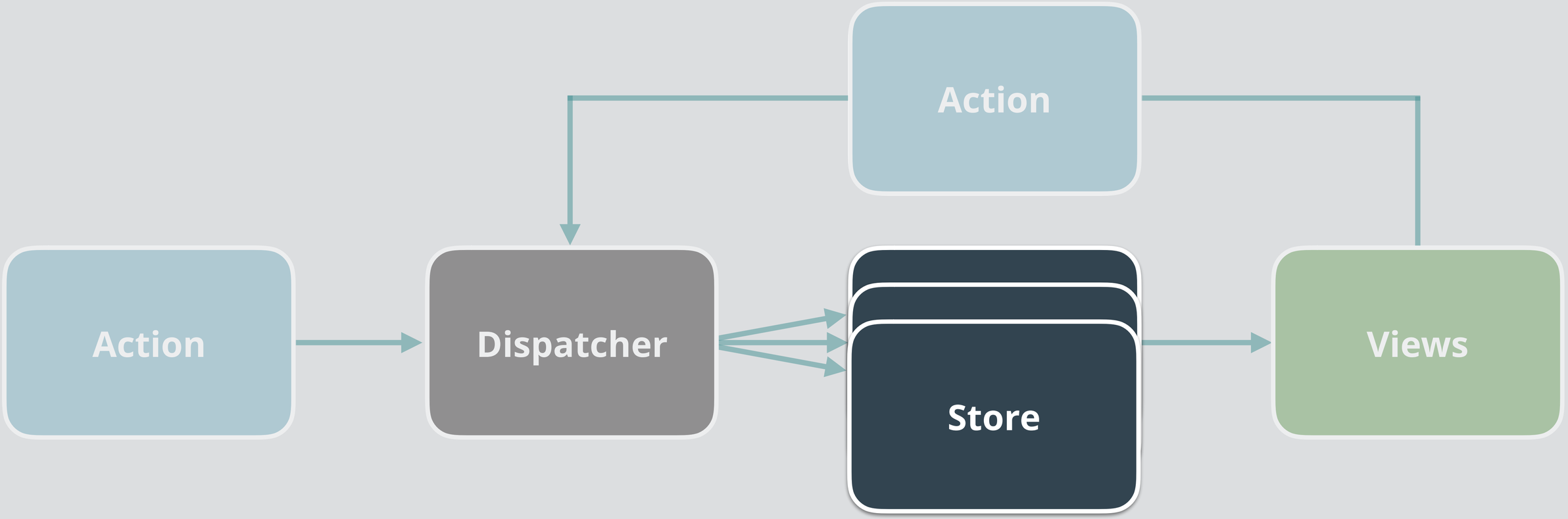
如果我們調整 Store 這個角色：

1. 將每個 Store 的業務狀態 → 組成一個狀態樹
2. 將每個 Store 改變狀態的邏輯 → 對應成每個 Reducer

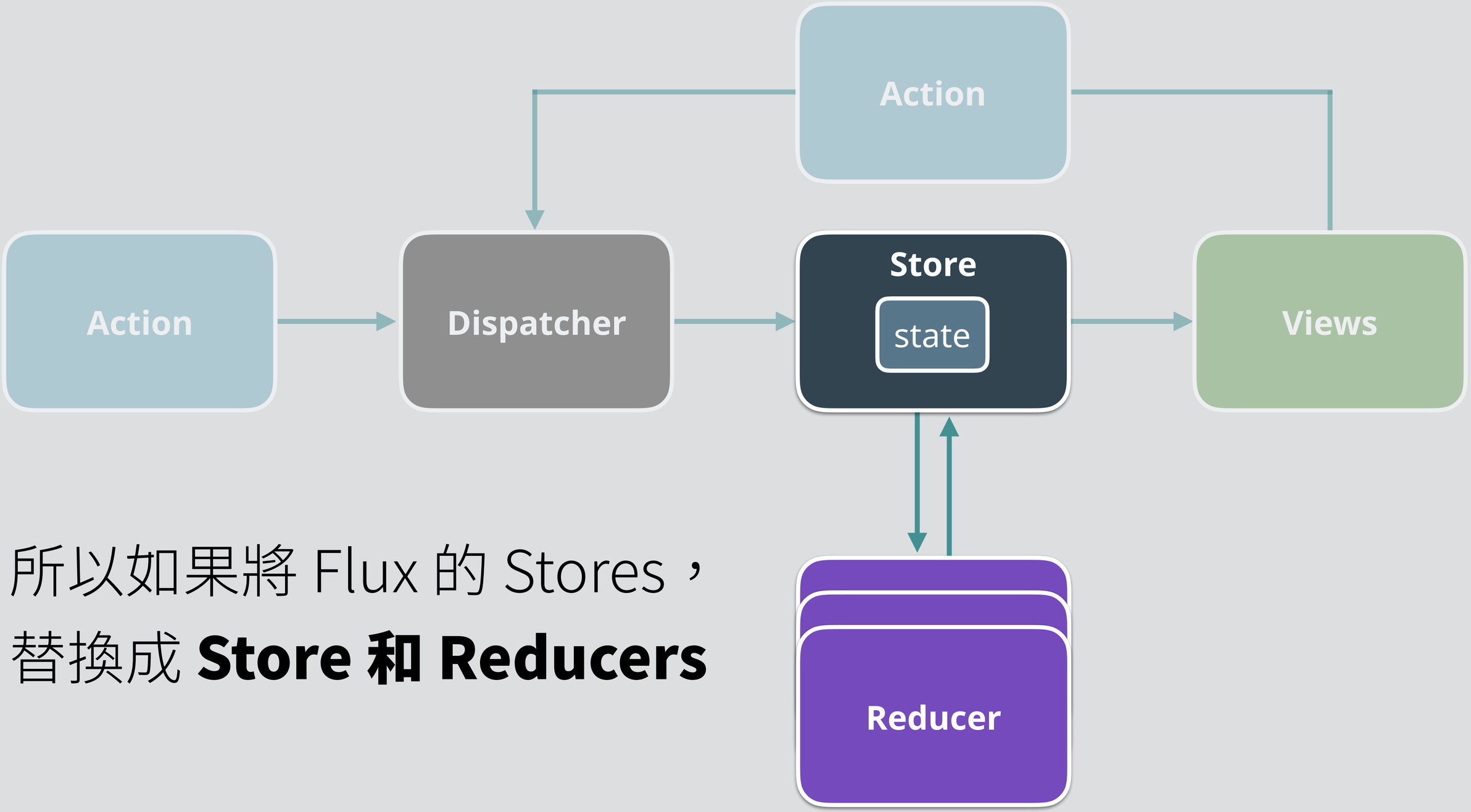


如果我們調整 Store 這個角色：

1. 將每個 Store 的業務狀態 → 組成一個狀態樹
2. 將每個 Store 改變狀態的邏輯 → 對應成每個 Reducer

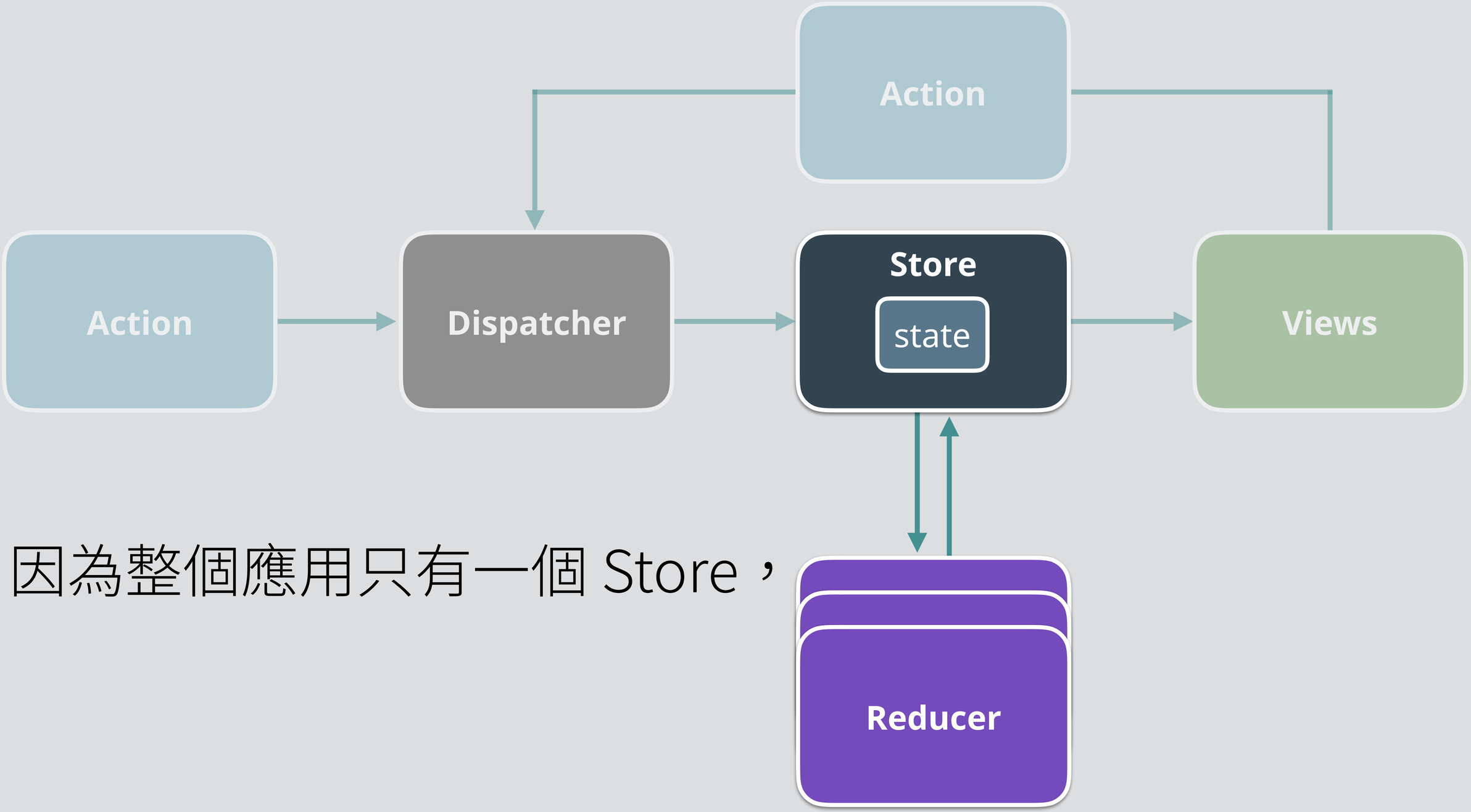


所以如果將 Flux 的 Stores，

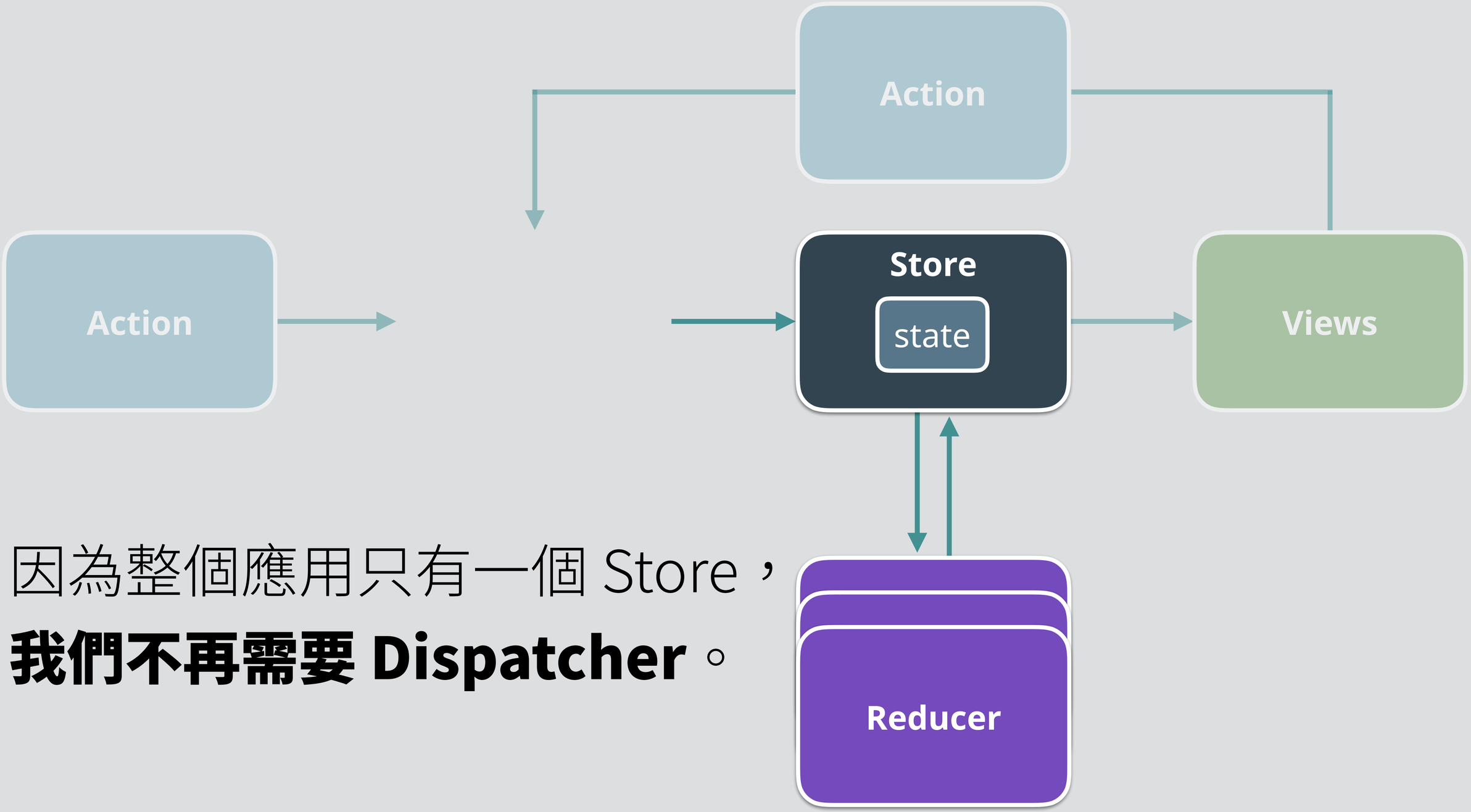


所以如果將 Flux 的 Stores，

替換成 **Store 和 Reducers**

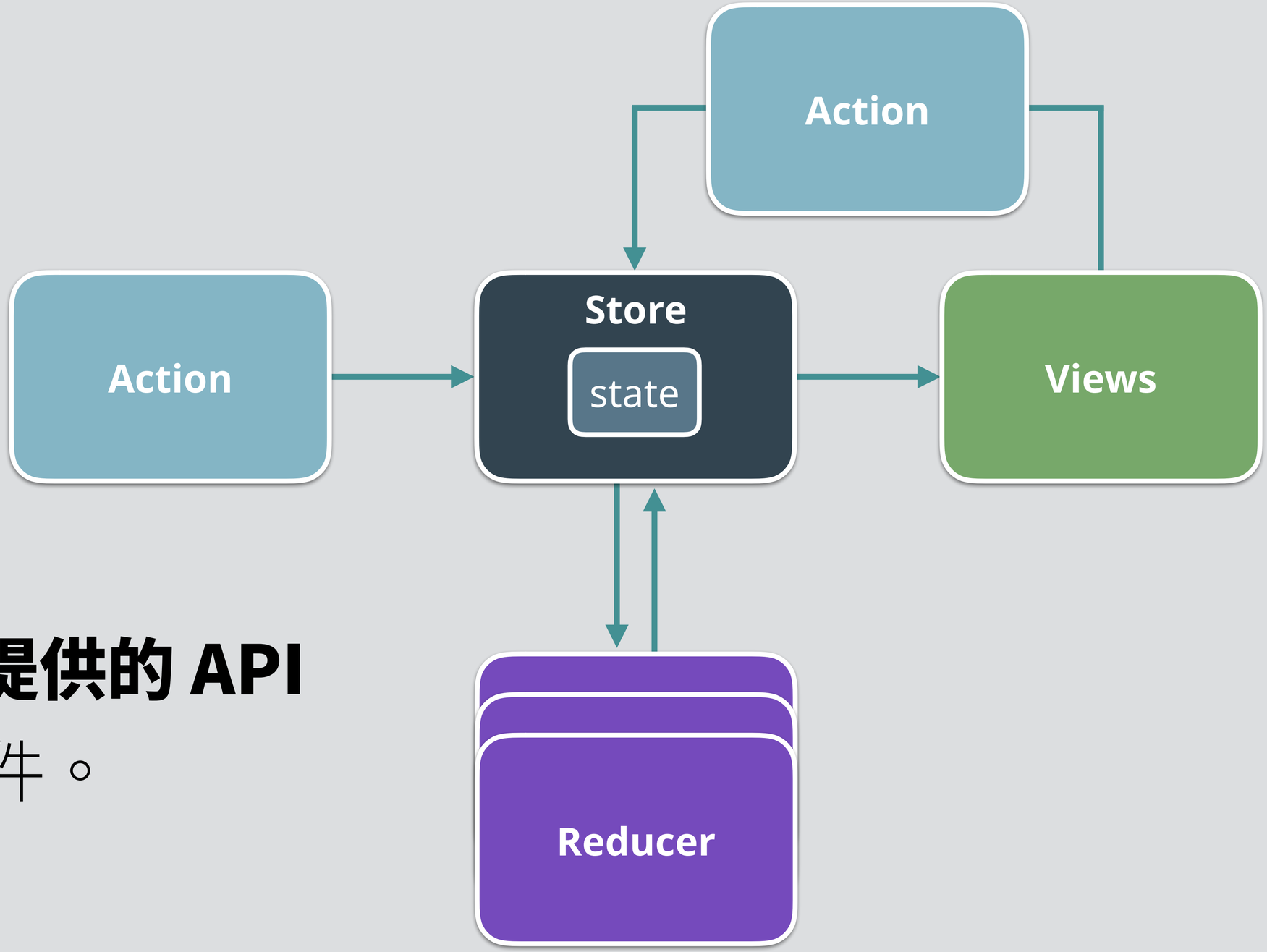


因為整個應用只有一個 Store，



因為整個應用只有一個 Store，

我們不再需要 Dispatcher。



而**透過 Store 提供的 API**

傳遞 action 物件。

這就是 Redux！