**mycena-store  
詳細說明書**

撰寫人： Joneshong

撰寫日：2022/04/07

更新日：2022/04/12

版本：1.1.3

# 目錄

## [系統架構](#_i2oojmpdvr73)

### [目的](#_hczzj8av30zb)

### [系統運作說明](#_h4jdf4t46tyq)

#### [Store](#_appg4ghcv75k)

#### [Reducer](#_h6t4g4bpjd52)

#### [Effect](#_ph2jegmtfrtz)

#### [Action](#_ycifvl1vsg8)

#### [Selector](#_fji30efg7otw)

#### [Entity](#_61ee4w38v9of)

### [系統操作文檔](#_b2ukd06re29d)

## [測試單元](#_eh8lhq7g81zl)

## [未來預計加入功能](#_25n4vddmejnd)

## [版本更新紀錄](#_jrsamwbd8ilb)

## [依賴函式庫說明](#_z4cnq1nze74y)

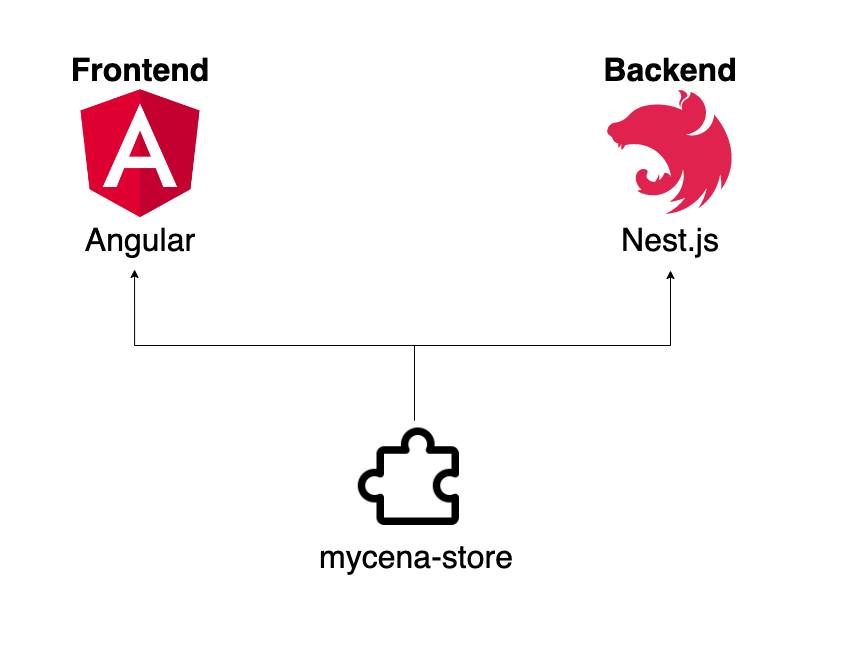
## [參考資料](#_aj5fdt8w0k5g)

## [附件/註](#_q36ywv2yqv1)

# [系統架構🔗](#_i2oojmpdvr73)

## [目的🔗](#_hczzj8av30zb)

我們致力於開發一套以 Typescript為基底不局限於前端或是後端使用的函式庫（Library），此架構是要將輔助系統打造成以**事件驅動程式設計**（Event driven），同性質的函式庫有 NgRx（[參 6.3](#_kf0m6c3p3gnq)）、Nest.js-CQRS（[參 6.2.1](#_kf0m6c3p3gnq)）、@felangel/bloc…等。  
 在程式語言 Javascript日益成熟的今日，微軟（Microsoft）公司甚至開發出一套基於 Javascript的另一套程式語言 Typescript，顧名思義提供靜態型別檢查以及類別爲基的物件導向程式語法操作，為了開發大型應用程式而設計的語言；加上 Node.js能在伺服器端運行 Javascript的跨平台執行環境。同時前端框架（Framework）：Angular、Vue.js、React …等；後端框架：Express、Nest.js…等，也以 Typescript為主體發展。



## [系統運作說明🔗](#_h4jdf4t46tyq)

工程有四個原則：單一事件來源（Single source of truth）、緊跟慣例（Stick to convention）、分而治之（Divide and conquer）、簡潔為上（Keep it simple ）。

職責分離原則（**C**ommand **Q**uery **R**esponsibility **S**egregation），在系統的請求都離不開 CRUD（Create, Read, Update, Delete），職責分離就是，請求的對象、處理業務邏輯、資料儲存...等，是不同的對象，切分程式看似是多費一份工，但在日後的管理、維護、擴充與多人協作上是一大優勢，就是分而治之又或解耦（Decoupling）。

接著細說此函式庫職責分離的角色分別是：

### [Store🔗](#_appg4ghcv75k)

函式庫的介面（Facafe），外部要做任何請求都是對它下達的，類似於櫃檯、窗口的概念。它確保了系統裡面的單一事件來源，事件就是透過它傳遞，也可以藉由訂閱它一但改變就能得到最新的狀態。

### [Reducer🔗](#_h6t4g4bpjd52)

註冊在 Store之下，根據事件（Event, Action）去處理要使得當前狀態如何改變到下一個狀態，狀態的轉換是以同步進行的，事件會按照傳入順序排隊。在此狀態盡量維持與資料庫一致，資料結構單純，需要計算的部分可以交由 Selector，

### [Effect🔗](#_ph2jegmtfrtz)

註冊在 Store之下，根據事件（Event, Action）通過網路請求到系統外部交互，以及與設備 API直接交互，預設應該回傳一個 Action，將請求的結果轉換成事件丟回 Store讓 Reducer決定是如何消化該事件，可以關閉回傳。

### [Action🔗](#_5kkje2p3er80)

事件唯一職責就是表達獨特的事件和意圖，應該確保簡潔為上，當我們定義的動作越多，在應用程序中表達的流程就越好。Action 藉由事件種類（type）驅動，事件本身可以夾帶事件的描述（payload）。

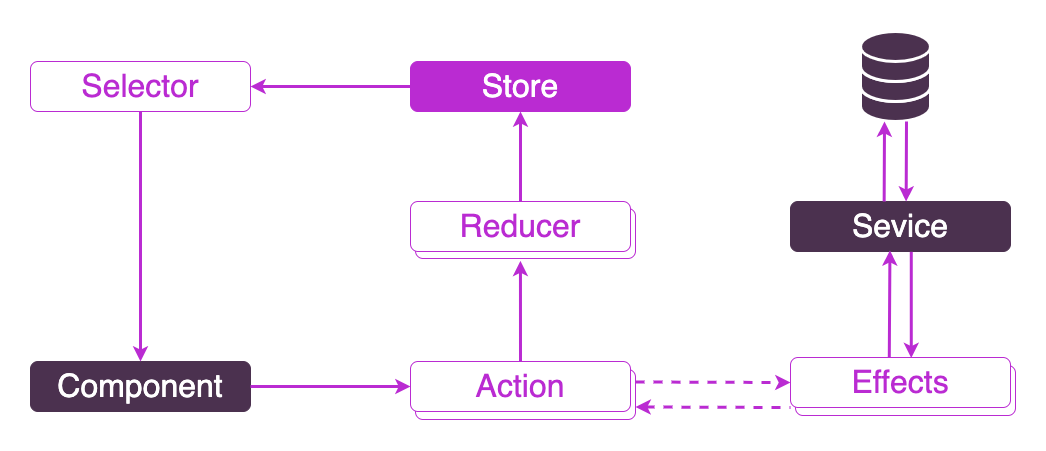
事件可由系統內部的 Reducer消化，也可以將外部交互通過網路來請求，這時就由 Effect去消化該請求，若是有回傳新的 Action，這一連串的事件其實是有某種正相關的事件鏈。

### [Selector🔗](#_fji30efg7otw)

用於獲取 Reducer狀態切片，可以在不會污染源頭資料的情況下，對資料進行計算，甚至是可以調用複數個狀態組成讀取模型。

### [Entity🔗](#_61ee4w38v9of)

實體只有數據結構，本身不包含複雜邏輯，唯用於建立關係，在記憶體中形成類似 －Neo4j 網狀資料庫的關係網，可以以任一實體為樹狀頂點往下去找關係。



### Relation🔗

描述 Entity之間的關係的敘述，點跟點之間的線，例如：有兩個 Entity是老師、課程，他們之間的關係是：一堂課程只會有一位老師，一位老師會負責很多課程，老師跟課程的關係呈現一對多，

## [系統操作文檔🔗](#_b2ukd06re29d)

此函式庫分成兩個面向的人群，第一種 Junior Engineer是基礎功能的單純使用，第二種 Architect是進階根據自身需求添加進函式庫核心。接著就以程式碼範例走過一遍。

### [基礎功能使用🔗](#_4xpgtjzcdwh3)

口訣：順序就是從小的到大的，從不依賴別人的寫道需要依賴人的部分。

model > entity > action > reducer > store(main) > effect

以下用 公司（團體Group）為範例說明，在 Angular, Nest, Unit-test有些微差異。

#### group.model.ts 🔗

export interface Group {

id: string;

name: string;

cooperationId?: string[];

description?: string;

tel: string;

address: string;

}

定義 Group的資料結構與格式。

#### group.entity.ts 🔗

import { Entity } from "../../lib/entity";

export class GroupEntity extends Entity {

\_name: string = "GroupEntity";

constructor(props) { super(props) }

}

定義 Entity時需要注意，不應該寫複雜邏輯去改變自身狀態（資料），目前函式庫版本 Entity提供的方法著重在於關係的綁定。

#### group.actions.ts 🔗

import { Action } from "../../lib/action";

export enum ActionEnum {

TestGroup = '[Group] Testing Group',

TestGroupAPI = '[Group] Testing Group API',

TestGroupAPISuccess = '[Group] Testing Group API Success',

}

export const ActionMap = {

TestGroup: ActionEnum.TestGroup,

TestGroupAPISuccess: ActionEnum.TestGroupAPISuccess,

}

export class TestGroup extends Action {

readonly type = ActionEnum.TestGroup

constructor(public payload: { message: string }) { super(); }

}

export class TestGroupAPI extends Action {

readonly type = ActionEnum.TestGroupAPI

constructor() { super(); }

}

export class TestGroupAPISuccess extends Action {

readonly type = ActionEnum.TestGroupAPISuccess

constructor(public payload: { body: any }) { super(); }

}

export type ActionUnion = TestGroup

基本 GRUD事件可參考（[參 6.3.2.](#_35jhggmig3xl)），可以直接調用全域的 Action並提供 Entity name，這段函式庫已經自動化這段了，減少重複 coding的部分。 Enum是全部的事件，而 ActionMap是會去註冊特別通道的事件

#### group.reducer.ts 🔗

import { EntityState, EntityAdapter, createEntityAdapter } from "../../lib/adapter";

import { Reducer } from "../../lib/reducer";

import { Action } from "../../lib/action";

import { ActionMap } from "./group.actions";

import { GroupEntity } from "./group.entity";

import { Group } from "./group.models";

export const FeatureKey = 'group';

export interface GroupState extends EntityState<Group> {

message: string;

apiBody: any;

}

export const adapter: EntityAdapter<Group> = createEntityAdapter<Group>();

export const initialState: GroupState = adapter.getInitialState({

message: null,

apiBody: null,

});

export class GroupReducer extends Reducer<Action, GroupState>{

\_name = "GroupReducer";

constructor() {

super(initialState, ActionMap);

*this*.setEntity(GroupEntity);

}

async\* mapEventToState(event: Action): AsyncIterableIterator<GroupState> {

let newState = *this*.defaultActionState(event);

switch (event['type']) {

case ActionMap.TestGroup: {

yield await *this*.handelTestGroup(newState, event);

}

case ActionMap.TestGroupAPISuccess: {

yield await *this*.handelTestGroupAPISuccess(newState, event);

}

default: {

yield await newState;

}

}

}

private handelTestGroup(newState, event): GroupState {

newState['message'] = event['payload']['message'];

return newState;

}

private handelTestGroupAPISuccess(newState, event): GroupState {

newState['apiBody'] = event['payload']['body'];

return newState

}

}

export const reducer = new GroupReducer();

在 Reducer中，重點就是設置初始狀態，並且針對事件種類去消化，當前狀態根據事件夾帶的事件描述來判斷是否該進入一個新的狀態。

#### index.ts🔗

import { CqrsMain } from "../lib/main";

import { RelationshipByType } from "../lib/interface/relation.interface";

import \* as fromGroup from "./group/group.reducer";

import { GroupEffects } from "./group/group.effects";

import { TestGroupAPI } from "./group/group.actions";

export interface StoreState {

group: fromGroup.GroupState;

}

export const Reducers = {

group: fromGroup.reducer,

};

export const FeatureKeys = {

group: fromGroup.FeatureKey,

};

export const Effects = [

GroupEffects

];

export const RelationshipByTypeMap: RelationshipByType = {

"OneToOne": new Set([]),

"OneToMany": new Set([]),

"ManyToOne": new Set([]),

"ManyToMany": new Set([

"Group{cooperationId} to Group{group(id)}",

]),

}

export const Cqrs = new CqrsMain<StoreState, typeof Reducers>();

Cqrs.forRootReducers(Reducers);

export const Store = Cqrs.Store;

export const StoreSate = Cqrs.StoreSate;

export const Actions = Cqrs.Actions;

Cqrs.forRootEffects(Effects);

Cqrs.setRelationshipByType(RelationshipByTypeMap);

Cqrs.setRelationConfig(Relation.getInstance().fromJDL(Cqrs.relationshipByType));

Store基礎組成元素是 StoreState, Reducers，在前置作業做好以後需要將剛剛做的那些素材導入 CQRS中，在導入成功後即可從 Cqrs中調用Store出來。

#### group.effects.ts🔗

（unit-tests 版本）

import { createEffect, ofType } from "../../lib/effect";

import { from, map, mergeMap } from "rxjs";

import { Actions } from "../index";

import { ActionEnum, TestGroupAPISuccess } from "./group.actions";

import { ErrorResponse } from "../../lib/action";

import { injectable } from "inversify";

import { GroupService } from "./group.service";

@injectable()

export class GroupEffects {

constructor(private \_groupService: GroupService) { }

test$ = createEffect(() => Actions.pipe(

ofType(ActionEnum.TestGroupAPI),

mergeMap((event) => {

return from(*this*.\_groupService.getAPITest()).pipe(

map(res => {

if (200 <= res.statusCode && res.statusCode <= 299) {

return new TestGroupAPISuccess({

body: res.body

})

} else {

return new ErrorResponse({

"failedAction": event,

"error": res['error']

})

}

})

)

}),

), { dispatch: true })

}

Effect注入 Service，根據事件種類呼叫 Service提供的 Methods ，再根據 Service回傳的值決定該如何轉換成 Action，另外錯誤事件函式庫有提供一個通用版本。

#### group.service.ts🔗

（unit-tests 版本）

import { injectable } from 'inversify';

import \* as supertest from 'supertest';

@injectable()

export class GroupService {

request: supertest.SuperTest<supertest.Test>

constructor() {

*this*.request = supertest("https://httpbin.org");

}

getAPITest() {

return *this*.request.get('/get');

}

}

單純的 GET範例。

### 補充說明🔗

#### Angular

需要在 src/main.ts下，添加 **setAppModule**

platformBrowserDynamic()

.bootstrapModule(AppModule)

.then((appModule) => {

Cqrs.setAppModule(appModule);

})

.catch((err) => console.error(err));

並且在 Effect當中的裝飾器（Decorator，也就是小老鼠＠符號）

group.effects.ts🔗（Angular 版本）

import { Injectable } from "@angular/core";

@Injectable()

這邊的 Injetable要使用 Angular自帶的，才能使用 Angular中的 Service。

#### Nest.js

需要在 src/main.ts下，添加 **setAppModule**

async function bootstrap() {

const app = await NestFactory.create(AppModule, { cors: false });

// (...省略)

Cqrs.setAppModule(AppModule, app)

await app.listen(60001);

}

並且在 Effect當中的裝飾器（Decorator，也就是小老鼠＠符號）

group.effects.ts🔗（Nest.js 版本）

import { Injectable } from "@nestjs/common";

@Injectable()

這邊的 Injetable要使用 Nest.js自帶的，才能使用 Nest.js中的 Service。

### 函式庫核心說明🔗

不同於基礎功能使用完全貼上程式碼，這邊只敘述主要功能的描述。

#### Entity🔗

如同上述 Entity主要用於建立關係，所以有以下四種方法建立

type OptionsRelationDescription = {

source: RelationDescription;

target: RelationDescription;

}

type BuildRelationOptions = {

targetKey: string;

};

setOneToOne: (

theOne: Entity | Entity[],

options?: OptionsRelationDescription | BuildRelationOptions

) => void;

setOneToMany: (

theOne: Entity | Entity[],

options?: OptionsRelationDescription | BuildRelationOptions

) => void;

addManyToOne: (

theMany: Entity | Entity[],

options?: OptionsRelationDescription | BuildRelationOptions

) => void;

addManyToMany: (

theMany: Entity | Entity[],

options?: OptionsRelationDescription | BuildRelationOptions

) => void;

不管是哪種關係都離不開：一對一、一對多、多對一、多對多，而這段參考了 JHipster的關係敘述方式協助建立關係，以案場與攝影機（一對多）舉例的話就是，案場下會知道他有哪幾台攝影機，而任一攝影機也會知道他所屬的案場細節。這種建立關係的方式是利用 Javascript中 By Reference​​（[參 6.6.1](#_dco4zr6lkkw4)）的特性，在不增加記憶體負擔的情況下產生連結；需注意的也是因這特性在刪除資料時，須先解除所關聯的物件中記著自己的關係，否則會因為連結存在沒有真的被刪除。

在建立關係時同時記著自己會在對方資料中產生的欄位，以下提供方法刪除：

breakAllReferences: () => void;

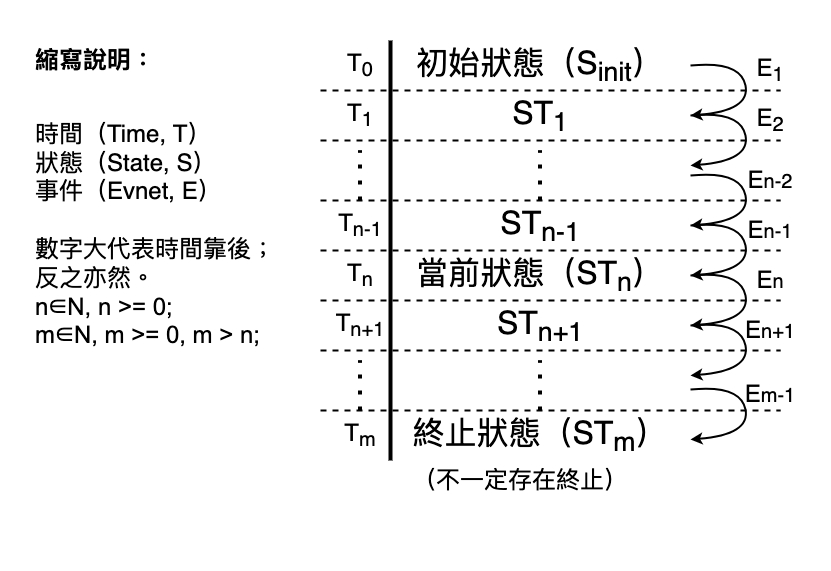
breakTargetReference: (reference: string, id: string) => void;

另外提供從 Entity在轉換成純 Object的方法，可用於資料傳輸。

toObject: ()=> object;

#### Action🔗

在參考資料 NgRx, Nest.js.Cqrs, @felangel/bloc中的事件，都像是信件，只夾帶一次性訊息，但在現實世界中事件是一連串的因果組合的，可以藉由某一個結果和事件來回推他的上一個結果，如圖



同時也能藉由當前狀態和某個事件，來推測它會進到哪一個狀態。

因此在此函數庫中 Action會加強，它經過的點記錄起來

addTraversal: (node: string) => void;

以及像是 Effect藉由某個事件去外部請求完後如若產生新的事件，就將這兩個事件作關係綁定的

setParent: (parent: Action) => void;

addChild: (child: Action) => void;

以及將 Action轉換成純 Object的

toObject: () => object;

#### Reducer🔗

這是繼承了 @felangel/bloc中 Bloc仿作 NgRx的 Reducer。

在前面基礎操作文檔中，設置的事件列表（ActionMap）， Reducer就會根據自身的 ActionMap去 Store登記，Reducer有這幾個事件需要注意，一但 Store收到登記的 Action就會通知 對應的Reducer。

subscribeTopic: (actionTypeList: string[]) => void;

Reducer跟 Store的關係是多對一，所以當 Store在 index.ts生成時，會將素材 Reducer與 Stotre關聯起來

setStore: (store: Store<any, any>) => void;

根據 NgRx的 adpater中我們得知有一些固定的事件，既然是固定就可以將它自動化，所以此函式庫根據一些事件會調用對應的事件處理器

defaultActionState: (action: Action) => state

因為 Reducer盡量存取單純狀態，之後要綁定關係還需要轉換成 Entity，所以還需將 Entity的 class存在 Reducer中

setEntity:(entityClass: any) => void;

createEntity: (data: object) => Entity;

createEntity: (datas: any[]) => Entity[];

turnStateToEntities: () => Entity[];

#### Broker🔗

這裡是參考 Kafka（[參 6.6.2](#_8j219tldvgvt)）以 Broker去管理事件的通道（Topic）。

目前版本有兩種通道，全域廣播性質（broadcast）與針對事件名稱設定專門通道，也就是説在上述 Action章節中的 ActionMap的每一個 Action都會有一個專屬的通道，事件一來就會直接轉手通知有訂閱的 Reducer。

系統初始化以後，會等所有的 Effect載入才開始傳送資料，在那之前如果有事件發生，就會先進到緩存取直到 Broker準備好，才會按照順序去清空緩存區的事件

isReadyToDispatch$: BehaviorSubject<Action>;

dispatch: (action: Action) => void;

用事件種類增加 Topic

addTopic: (action: Action) => BehaviorSubject<Action>;

根據事件種類提取通道

getActionType: (action: Action) => BehaviorSubject<Action>;

提取廣播通道

getBroadcast: () => BehaviorSubject<Action>

#### Store🔗

# 測試單元🔗

# 未來預計加入功能🔗

# 版本更新紀錄🔗

# 依賴函式庫說明🔗

# 參考資料🔗

## Promgramming原則（[連結](https://tecky.io/zh_Hant/blog/%E5%B9%B3%E5%B8%B8%E4%BA%BA%E9%83%BD%E8%83%BD%E6%8E%8C%E6%8F%A1%E7%9A%84Programming%20%E5%8E%9F%E5%89%87/)）

## Nest

### CQRS（[連結](https://docs.nestjs.com/recipes/cqrs)）

## NgRx

### store（[連結](https://ngrx.io/guide/store)）

### adapter（[連結](https://ngrx.io/guide/entity/adapter)）

## Flutter

### BloC（[連結](https://bloclibrary.dev/#/gettingstarted)）

## JHipster

### ManagingRelationships（[連結](https://www.jhipster.tech/managing-relationships/)）

### JDLStudio（[連結](https://start.jhipster.tech/jdl-studio/)）

## 其他

### By Reference​​（[連結](https://pjchender.blogspot.com/2016/03/javascriptby-referenceby-value.html)）

### Kafka 工作原理（[連結](http://xstarcd.github.io/wiki/Cloud/kafka_Working_Principles.html)）

## 圖源

### Draw.io 自製（[連結](https://drive.google.com/file/d/1k4d1nh_otrg4BBWNmPPStUKl5HDZvjWh/view?usp=sharing)）

* 1. .

# 附件/註🔗