# **ngrx/store**

https://segmentfault.com/a/1190000017566273

## **Store**

Strore是Angular基于Rxjs的状态管理，保存了Redux的核心概念，并使用RxJs扩展的Redux实现。使用Observable来简化监听事件和订阅等操作。  
在看这篇文章之前，已经假设你已了解[rxjs](https://cn.rx.js.org/" \t "https://segmentfault.com/a/_blank)和[redux](https://www.redux.org.cn/" \t "https://segmentfault.com/a/_blank)。

## **Tutorial**

下面这个Tutorial将会像你展示如何管理一个计数器的状态和如何查询以及将它显示在Angular的Component上。你可以通过[StackBlitz](https://stackblitz.com/fork/ngrx-start" \t "https://segmentfault.com/a/_blank)来在线测试。

1.创建actions

src/app/counter.actions.ts

import {Action} from '@ngrx/store';

export enum ActionTypes {

Increment = '[Counter Component] Increment',

Decrement = '[Counter Component] Decrement',

Reset = '[Counter Component] Reset',

}

export class Increment implements Action {

readonly type = ActionTyoes.Increment;

}

export class Decrement implements Action {

readonly type = ActionTypes.Decrement;

}

export class Reset implements Action {

readonly tyoe = Actiontypes.Reset;

}

2.定义一个reducer通过所提供的action来处理计数器state的变化。

src/app/counter.reducer.ts

import {Action} from '@ngrx/store';

import {ActionTypes} from './conter.actions';

export const initailState = 0;

export function conterReducer(state = initialState, action: Action) {

switch(action.type) {

case ActionTypes.Increment:

return state + 1;

case ActionTypes.Decrement:

return state - 1;

case ActionTypes.Reset:

return 0;

default:

return state;

}

}

3.在src/app/app.module.ts中导入 StoreModule from @ngrx/store 和 counter.reducer

import {StroeModule} from '@ngrx/store';

import {counterReducer} from './counter.reducer';

4.在你的AppModule的imports array添加StoreModule.forRoot,并在StoreModule.forRoot中添加count 和 countReducer对象。StoreModule.forRoot()函数会注册一个用于访问store的全局变量。

scr/app/app.module.ts

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

import { StoreModule } from '@ngrx/store';

import { counterReducer } from './counter.reducer';

@NgModule({

declaration: [AppComponent],

imports: [

BrowserModule,

StoreModule.forRoot({count: countReducer})

],

provoders: [],

bootstrap: [AppComponent]

})

export class AppModule {}

5.在app文件夹下新创建一个叫my-counter的Component，注入Store service到你的component的constructor函数中，并使用select操作符在state查询数据。更新MyCounterComponent template,添加添加、减少和重设操作，分别调用increment,decrement,reset方法。并使用async管道来订阅count$Observable。

src/app/my-counter/my-counter.component.html

<button (click)="increment()">Increment</button>

<div>Current Count: {{ count$ | async }}</div>

<button (click)="decrement()">Decrement</button>

<button (click)="reset()">Reset Counter</button>

更新MyCounterComponent类，创建函数并分发（dispatch）Increment,Decrement和Reset actions.

import { Component } from '@angular/core';

import { Store, select } from '@ngrx/store';

import { Observable } from 'rxjs';

import { Increment, Decrement, Reset } from '../counter.actions';

@Component({

selector: 'app-my-counter',

templateUrl: './my-counter.component.html',

styleUrls: ['./my-counter.component.css'],

})

export class MyCounterComponent (

count$: Observable<number>;

constructor(private store: Stare<{count: number}>) {

this.count$ = store.pipe(select('count'));

}

increment() {

this.store.dispatch(new Increment());

}

decrement() {

this.store.dispatch(new Decrement());

}

reset() {

this.store.dispatch(new Reset());

}

)

6.添加MyCounter component到AppComponent template中

<app-my-counter></app-my-counter>

## **Actions**

****Actions****是NgRx的核心模块之一。Action表示在整个应用中发生的独特的事件。从用户与页面的交互，与外部的网络请求的交互和直接与设备的api交互，这些和更多的事件通过actions来描述。

### **介绍**

在NgRx的许多地方都使用了actions。Actions是NgRx许多系统的输入和输出。Action帮助你理解如何在你的应用中处理事件。

### **Action接口(Action interface)**

NgRx通过简单的interface来组成Action：

interface Action {

type: string;

}

这个interface只有一个属性:type，string类型。这个type属性将描述你的应用调度的action。这个类型的值以[Source]的形式出现和使用，用于提供它是什么类型的操作的上下文和action在哪里被调度（dispatched）。您可以向actions添加属性，以便为操作提供其他上下文或元数据。最常见的属性就是payload，它会添加action所需的所有数据。  
下面列出的是作为普通javascript对象编写的操作的示例：

{

type: '[Auth API] Login Success'

}

这个action描述了调用后端API成功认证的时间触发。

{

type: '[Login Page]',

payload: {

username: string;

password: string;

}

}

这个action描述了用户在登录页面点击登录按钮尝试认证用户的时间触发。payload包含了登录页面提供的用户名和密码。

### **编写 actions**

有一些编写actions的好习惯：

* 前期——在开始开发功能之前编写编写action，以便理解功能和知识点
* 分类——基于事件资源对actions进行分类
* 编写更多——action的编写容易，所以你可以编写更多的actions,来更好的表达应用流程
* 事件-驱动——捕获事件而不是命令，因为你要分离事件的描述和事件的处理
* 描述——提供针对唯一事件的上下文，其中包含可用于帮助开发人员进行调试的更详细信息

遵循这些指南可帮助您了解这些actions在整个应用程序中的流程。  
下面是一个启动登陆请求的action示例：

import {} from '@ngrx/store';

export class Login Implements Action {

readonly type = '[Login Page] Login'

constructor(public: payload: {username: string, password: string}){}

}

action编写成类，以便在dispatched操作时提供类型安全的方法来构造action。Login action 实现(implements) Action interface。在示例中，payload是一个包含username和password的object，这是处理action所需的其他元数据.  
在dispatch时，新实例化一个实例。

login-page.component.ts

click(username: string, password: string) {

store.dispatch(new Login({username:username, password: password}))

}

Login action 有关于action来自于哪里和事件发生了什么的独特上线文。

* action的类型包含在[]内
* 类别用于对特征区域的action进行分组，无论他是组件页面，后端api或浏览器api
* 类别后面的Login文本是关于action发生了什么的描述。在这个例子中，用户点击登录页面上的登录按钮来通过用户名密码来尝试认证。

### **创建action unions**

actions的消费者，无论是reducers(纯函数)或是effects(带副作用的函数)都使用actions的type来确定是否要执行这个action。在feature区域，多个actions组合在一起，但是每个action都需要提供自己的type信息。看上一个Login action 例子，你将为action定义一些额外的信息。

import {Action} from '@ngrx/store';

export enum ActionTypes {

Login = '[Login Page] Login';

}

export class Login Implememts Action {

readonly type = ActionTypes.Login;

constructor(public paylad: {username: string, password: string})

}

export type Union = Login;

将action type string放在enum中而不是直接放在class内。此外，还会使用Union类去导出Loginclass.

## **Reducers**

NgRx中的Reducers负责处理应用程序中从一个状态到下一个状态的转换。Reducer函数从action的类型来确定如何处理状态。

### **介绍**

Reducer函数是一个纯函数，函数为相同的输入返回相同的输出。它们没有副作用，可以同步处理每个状态转化。每个reducer都会调用最新的action，当前状态（state）和确定是返回最新修改的state还是原始state。这个指南将会向你展示如何去编写一个reducer函数，并在你的store中注册它，并组成独特的state。

### **关于reducer函数**

每一个由state管理的reducer都有一些共同点：

* 接口和类型定义了state的特征
* 参数包含了初始state或是当前state、当前action
* switch语句

下面这个例子是state的一组action，和相对应的reducer函数。  
首先，定义一些与state交互的actions。

scoreboard-page.actions.ts

import {Action} from '@ngrx/store';

export enum Actiontypes {

IncrementHome = '[Scoreboard Page] Home Score',

IncrementAway = '[Scoreboard Page] Away Score',

Reset = '[Scoreboard Page] Score Reset',

}

export class IncrementHome implements Action {

readonly type = ActionTypes.IncrementHome;

}

export class IncrementAway implements Action {

readonly type = ActionTypes.IncrementAway;

}

export class Reset implements Action {

readonly type = ActionTypes.Reset;

constructor(public payload: {home: number, away: number}) {}

}

export type ActionsUnion = IncrementHome | IncrementAway | Reset;

接下来，创建reducer文件，导入actions,并定义这个state的特征。

### **定义state的特征**

每个reducer函数都会监听actions,上面定义的scorebnoard actions描述了reducer处理的可能转化。导入多组actions以处理reducer其他的state转化。

scoreboard.reducer.ts

import \* as Scoreboard from '../actions/scoreboard-page.actions';

export interface State {

home: number;

away: number;

}

根据你捕获的内容来定义state的特征，它是单一的类型，如number,还是一个含有多个属性的object。

### **设置初始state**

初始state给state提供了初始值，或是在当前state是undefined时提供值。您可以使用所需state属性的默认值设置初始state。创建并导出变量以使用一个或多个默认值捕获初始state。

scoreboard.reducer.ts

export const initialState: Satate = {

home: 0,

away: 0,

};

### **创建reducer函数**

reducer函数的职责是以不可变的方式处理state的更变。定义reducer函数来处理actions来管理state。

scoreboard.reducer.ts

export function reducer {

satate = initialState,

action: Scoreboard.ActionsUnion

}: State {

switch(action.type) {

case Scoreboard.ActionTypes.IncrementHome: {

return {

...state,

home: state.home + 1,

}

}

case Scoreboard.ActionTypes.IncrementAway: {

return {

...state,

away: state.away + 1,

}

}

case Scoreboard.ActionTypes.Reset: {

return action.payload;

}

default: {

return state;

}

}

}

Reducers将switch语句与TypeScript在您的actions中定义的区分联合组合使用，以便在reducer中提供类型安全的操作处理。Switch语句使用type union来确定每种情况下正在使用的actions的正确特征。action的types定在你的action在你的reducer函数的case语句。type union 也约束你的reducer的可用操作。  
在这个例子中，reducer函数处理3个actions:IncrementHome,IncrementAway,Reset。每个action都有一个基于ActionUnion提供的强类型。每个action都可以不可逆的处理state。这意味着state更变不会修改源state，而是使用spread操作返回一个更变后的新的state。spread语法从当前state拷贝属性，并创建一个新的返回。这确保每次更变都会有新的state，保证了函数的纯度。这也促进了引用完整性，保证在发生状态更改时丢弃旧引用

注意：spread操作只执行浅复制，不处理深层嵌套对象。您需要复制对象中的每个级别以确保不变性。有些库可以处理深度复制，包括[lodash](https://lodash.com/" \t "https://segmentfault.com/a/_blank)和[immer](https://github.com/mweststrate/immer" \t "https://segmentfault.com/a/_blank)。

当action被调度时，所有注册过的reducers都会接收到这个action。通过switch语句确定是否处理这个action。因为这个原因，每个switch语句中总是包含default case，当这个reducer不处理action时，返回提供的state。

### **注册root state**

state在你的应用中定义为一个large object。注册reducer函数。注册reducer函数来管理state的各个部分中具有关联值的键。使用StoreModule.forRoot()函数和键值对来定义你的state，来在你的应用中注册一个全局的Store。StoreModule.forRoot()在你的应用中注册一个全局的providers，将包含这个调度state的action和select的Store服务注入到你的component和service中。

app.module.ts

import {NgModule} from '@angular/core';

import {StoreModule} form '@ngrx/store';

import {scoreboardReducer} from './reducers/scoreboard.resucer';

@NgModule({

imports: [StoreModule.forRoot({game: scoreboardReducer})],

})

export class AppModule {}

使用StoreModule.forRoot()注册states可以在应用启动时定义状态。通常，您注册的state始终需要立即用于应用的所有区域。

### **注册Feature state**

Feature states的行为和root state相同，但是你在你的应用中需要定义具体的特征区域。你的state是一个large object，Feature state会在这个object中以键值对的形式注册。  
下面这个state object的例子，你将看到Feature state如何以递增的方式构建你的state。让我们从一个空的state开始。

app.module.ts

@NgModule({

imports: [StoreModule.forRoot({})],

})

export class AppModule {}

这里在你的应用中创建了一个空的state

{

}

现在使用scoreboardreducer和名称为ScoreboarModule的特征NgModule注册一个额外的state。

scoreboard.module.ts

import { NgModule } from '@angular/core';

import { StoreModule } from '@ngrx/store';

import { scoreboardReducer } from './reducers/scoreboard.reducer';

@NgModule({

imports: [StoreModule.forFeature('game', scoreboardReducer)],

})

export class ScoreboardModule {}

添加ScoreboardModule到APPModule。

app.module.ts

import { NgModule } from '@angular/core';

import { StoreModule } from '@ngrx/store';

import { ScoreboardModule } from './scoreboard/scoreboard.module';

@NgModule({

imports: [StoreModule.forRoot({}), ScoreboardModule],

})

export class AppModule {}

每一次ScoreboardModule被加载，这个game将会变为这个object的一个属性，并被管理在state中。

{

game: { home: 0, away: 0}

}

Feature state的加载是eagerly还是lazlly的，取决于你的应用。可以使用Feature state 随时间和不同特征区域构建状态对象。

## **select**

Selector是一个获得store state的切片的纯函数。@ngrx/store提供了一些辅助函数来简化selection。selector提供了很多对state的切片功能。

* 轻便的
* 记忆化
* 组成的
* 可测试的
* 类型安全的

当使用createSelector和createFeatureSelector函数时，@ngrx/store会跟踪调用选择器函数的最新参数。因为选择器是纯函数，所以当参数匹配时可以返回最后的结果而不重新调用选择器函数。这可以提供性能优势，特别是对于执行昂贵计算的选择器。这种做法称为[memoization](https://en.wikipedia.org/wiki/Memoization" \t "https://segmentfault.com/a/_blank)。

### **使用selector切片state**

index.ts

import {createSelector} from '@ngrx/store';

export interface FeatureState {

counter: number;

}

export interface AppSatte {

feature: FeatureState;

}

export const selectFeature = (state: AppState) => state.feature;

export const selectFeatureCount = createSelector(

selectFeature,

(state: FeatrureState) => state.counter

)

### **使用selectors处理多切片**

createSelector能够从基于同样一个state的几个切片state中获取一些数据。  
createSelector最多能够接受8个selector函数，以获得更加完整的state selections。  
在下面这个例子中，想象一下你有selectUser object 在你的state中，你还有book object的allBooks数组。  
你想要显示你当前用户的所有书。  
你能够使用createSelector来实现这些。如果你在allBooks中更新他们，你的可见的书将永远是最新的。如果选择了一本书，它们将始终显示属于您用户的书籍，并且在没有选择用户时显示所有书籍。  
结果将会是你从你的state中过滤一部分，并且他永远是最新的。

import {createSelecotr} from '@ngrx/store';

export interface User {

id: number;

name: string;

}

export interface Book {

id: number;

userId: number;

name: string;

}

export interface AppState {

selectoredUser: User;

allBooks: Book[];

}

export const selectUser = (state: AppSate) => state.selectedUser;

export const SelectAllBooks = (state: AppState) => state.allBooks;

export const SelectVisibleBooks = createSelector(

selectUser,

selectAllBooks,

(selectedUser: User, allBooks: Books[]) => {

if(selectedUser && allBooks) {

return allBooks.filter((book: Book) => book.UserId === selectedUser.id);

}else {

return allBooks;

}

}

)

### **使用selecotr props**

当store中没有一个适合的select来获取切片state，你可以通过selector函数的props。  
在下面的例子中，我们有计数器，并希望他乘以一个值，我们可以添加乘数并命名为prop：

index.ts

export const getCount = createSelector(

getCounterValue,

(counter, props) => counter \* props.multiply

);

在这个组件内部，我们定义了一个props。

ngOnInit() {

this.counter = this.store.pipe(select(formRoot.getCount, {multiply: 2}));

}

记住，selector只将之前的输入参数保存在了缓存中，如果你用另一个乘数来重新使用这个selector，selector总会去重新计算它的值，这是因为他在接收两个乘数。为了正确地记忆selector，将selector包装在工厂函数中以创建选择器的不同实例

index.ts

export const getCount = () => {

createSelector(

(state, props) => state.counter[props.id],

(counter, props) => counter \* props\* multiply

);

}

组件的selector现在调用工厂函数来创建不同的选择器实例:

ngOnInit() {

this.counter2 = this.store.pipe(select(fromRoot.getCount(), { id: 'counter2', multiply: 2 }));

this.counter4 = this.store.pipe(select(fromRoot.getCount(), { id: 'counter4', multiply: 4 }));

this.counter6 = this.store.pipe(select(fromRoot.getCount(), { id: 'counter6', multiply: 6 }));

}

### **选择Feature States**

createFeatureSelector是一个返回顶级feature state的便捷方法。它返回一个按特征切片的state的typed selector方法。  
****示例****

index.ts

import { createSelector, createFeatureSelector } from '@ngrx/store';

export interface FeatureState {

counter: number;

}

export interface AppState {

feature: FeatureState;

}

export const selectFeature = createFeatureSelector<AppState, FeatureState>('feature');

export const selectFeatureCount = createSelector(

selectFeature,

(state: FeatureState) => state.counter

);

下面这个selector无法被编译，因为foo不是AppState的一个特征切片。

index.ts

export const selectFeature = createFeatureSelector<AppSatte, FeatureState>('foo');

### **重置Memoized Selector**

通过调用createSelector或createFeatureSelector返回的选择器函数初始化具有memoized值null。在第一次将其memoized值存储在内存中时调用selector。如果之后使用相同的参数调用selector，则它将返回memoized值。如果之后使用不同的参数调用选择器，它将重新计算并更新其memoized值。

import { createSelector } from '@ngrx/store';

export interface State {

counter1: number;

counter2: number;

}

export const selectorCounter1 = (state: State) => state.counter1;

export const selectorCounter2 = (state: State) => state.counter2;

export const selectTotal = createSelector(

selectCounter1,

selectCounter2,

(counter1, counter2) => counter1 + counter2

); // selecotTotal是一个值为null的memoized值，因为他还没有被调用。

let state = { counter1: 3, counter2: 4};

selectTotal(state); //计算值3 + 4，返回7，selectTotal的memoized值的值为7

selectTotal(state); //不做计算，直接返回selectTotal的memoized值的值7

state = {...state, counter2: 5};

selectTotal(state); //计算值3 + 5,返回8.selectTotal的memoized值的值为8

selector memoized值会无限期的保留在内存中。如果memoized的值是一个不需要的大数据集，它可以重新设置memoized值为null，这样就可以将大数据集从内存中删除。这可以通过在selector上调用release方法来完成。

selectTotal(state); // 返回memoized值8

selectTotal.release(); //memoized的值被设为null

释放selector也会递归的释放所有的先祖选择器。

export interface State {

evenNums: number[];

oddNums: number[];

}

export const selectSumEvenNums = createSelector(

(state: State) => state.evenNums,

evenNums => evenNums.reduce((prev, curr) => prev + curr)

);

export const selectSumOddNums = createSelector(

(state: State) => state.oddNums,

oddNums => oddNums.reduce((prev, curr) => prev + curr)

);

export const selecTotal = createSelector(

selectorSumEvenNums,

selectorSumOddNums,

(evenSum,oddSum) => evenSum + oddSum

);

selectTotal({

evenNums: [2, 4],

oddNums: [1, 3],

})

// selectSumEvenNums: 6

// selectSumOddNums: 4

// selectTotal: 10

selectTotal.release();

// selectSumEvenNums: null

// selectSumOddNums: null

// selectTotal: null

### **高级技巧**

Selector能够为你的应用state组成一个读取模型（[CQRS](https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs" \l "solution" \t "https://segmentfault.com/a/_blank))。就CQRS架构模式而言，NgRx将读取模型（Selector）与写入模型（reducer）分开。高级技巧就是讲selector与RxJs的操作符结合起来。  
这个部分将涵盖：如何将selector与pipe操作结合；演示利用createSelector和scan来显示state的状态变化历史。

#### **使用pipe查询一个非空state**

假装我们有一个选择器叫selectValue。  
我们只使用RxJspipe操作来实现这一个功能：

app.component.ts

import {map, filter } from 'rxjs/operators';

store

.pipe(

map(state => selectValues(state)),

filter(val => val !== undefined)

)

.subscribe(/\* ... \*/)

我们可以使用select()方法重写上述内容:

app.component.ts

import { select } from '@ngrx/store';

import { map, filter } from '@rxjs/operators';

store.

pipe(

select(selectValues),

filter(val => val !== undefined)

)

.subscribe(/\* ... \*/)

#### **提取pipe操作**

我们使用pipe()方法来提取操作：

app.component.ts

import { select } from 'ngrx/store';

import { pipe } from 'rxjs';

import { filter } from 'rxjs/operators';

export const selectFilteredValues = pipe(

select(selectValues),

filter(val => val !== undefined)

);

store.pipe(selectFilteredValues).subscribe(/\* ... \*/);

#### **获取最后N个state转化**

index.ts

export const selectProjectedValues = createSelector(

selectFoo,

selectBar,

(foo, bar) => {

if(foo && bar) {

return {foo, bar};

}

return undefined;

}

);

select-last-state-transition.ts

export const selectLastStateTransitions = (count: number) => {

return pipe(

select(selectProjectedValues),

scan((acc, curr) => {

return [curr, acc[0], acc[1] ].filter(val => val !== undefined);

} [] as {foo: number; bar: string}[])

);

}

app.component.ts

store.pipe(selectLastStateTransitons(3)).subscribe(/\* ... \*/);

## **高级**

### **Meta-reducers**

@ngrx/store将多个reducers组合成一个reducer。  
开发者可以认为meta-reducers像是一个在action -> reducer管道中的钩子。Meta-reducers允许开发者在调用普通Reducer之前预处理操作。  
使用metaReducers配置meta-reducers。metaReducers会从你提供的数组中，从右到左的执行。

****注意****：Meta-reducers在NgRx中类似于Redux的中间件。

#### **使用meta-reducer来记录所有的actions**

app.module.ts

import { StoreModule, ActionReducer, MetaReducer } from '@ngrx/store';

import { reducers } from './reducers';

export function debug(reducer: ActionReducer<any>): ActionReducer<any> {

return function (state, action) {

console.log('state', state);

console.log('action', action);

return reduccer(state, action);

}

}

export const metaReducers: MetaReducer<any>[] = [debug];

@NgModule({

imports: [StoreModule.forRoot(reducers, { metaReducers })],

})

export class AppMpdule{}

## **使用依赖注入(injection)**

### **注入Reducers**

使用InjectionToken和Provider通过依赖注入注册reducer，注入root reducers到你的应用中。

app.module.ts

import { NgModule, InjectionToken } from '@angular/core';

import { StoreModule, ActionReducerMap } from '@ngrx/store';

import { SomeService } from './some.service';

import \* as fromRoot from './reducers';

export const REDUCER\_TOKEN = new InjectionToken<ActionReducerMap<fromRoot.State>>('Registered Reducers');

export function getReducers(someService: SomeService) {

return someService.getReducers();

}

@NgModule({

imports: [StoreModule.forRoot(REDUCER\_TOKEN)],

providers: [

{

provide: REDUCER\_TOKEN,

deps: [SomeService],

useFactory: getReducers,

}

],

})

export class AppModule {}

通过功能模块组成state也能够注入Reducers。

feature.module.ts

import { NgModule, InjectionToken } from '@angular/core';

import { StoreModule, ActionReducerMap } from '@ngrx/store';

import \* as fromFeature from './reducers';

export const FEATURE\_REDUCER\_TOKEN = new InjectionToken<

ActionReducerMap<fromFeature.State>

>('Feature Reducers');

export function getReducers(): ActionReducerMap<fromFeature.State> {

return {};

}

@NgModule({

imports: [StoreModule.forFeature('feature', FEATURE\_REDUCER\_TOKEN)],

providers: [

{

provide: FEATURE\_REDUCER\_TOKEN,

useFactory: getReducers,

},

],

})

export class FeatureModule {}

### **注入 Meta-Reducers**

通过META\_REDUCERS和Provider来注入。

app.module.ts

import { MetaReducer, META\_REDUCERS } from '@ngrx/store';

import { SomeService } from './some.service';

import \* as fromRoot from './reducers';

export function getMetaReducers(

some: SomeService

): MetaReducer<fromRoot.State>[] {}

@NgModule({

providers: [

{

provide: META\_REDUCERS,

deps: [SomeService],

useFactory: getMetaReducers,

},

],

})

export class AppModule {}

## **Entity**

Entity State adapter用于管理记录集合。  
Entity提供了API去操作和查询entity集合。

* 创建样板reducers来管理models集合
* 提供高性能的CRUD操作来管理entity集合
* 用于选择实体信息的可扩展类型安全适配器

### **安装**

npm install @ngrx/entity --save

### **Entity Interface**

#### **EntityState<T>**

entity state 是具有以下接口的给定entity集合的预定义通用接口:

interface EntityState<V> {

ids: string[] | number[];

entities: {

[id: string | id: number]: V

};

}

ids: 集合中所有entity的id的数组集合  
entities: 以id为索引的entity合集字典  
通过提供额外的属性来扩展entity state

user.reducer.ts

export interface User {

id: string;

name: string;

}

export interface State extends EntityState<User> {

//额外的entity state 属性

selectedUserId: number | null;

}

#### **EntityAdapter<T>**

为提供的entity适配器提供一个通用的类型接口。entity适配器提供了许多的集合方法来管理entity state

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

### **Entity Adapter(entity适配器)**

#### **createEntityAdapter<T>**

一种为单个entity state集合返回通用entity适配器的方法。返回的适配器提供了许多适配器方法，用于对集合类型执行操作。这个方法的构建需要提供两个参数。

* selectId: 用于选择字典id的方法。当entity具有id属性时，可选。
* sortcomparer: 用于集合排序的比较方法。只有在显示之前需要对集合进行排序时才需要比较器功能。设置为false将会不对集合排序，这将使CRUD操作更高效。

user.reducer.ts

import { EntityState, EntityAdapter, createEntityAdapter } from '@ngrx/entity';

export interface User {

id: string;

name: string;

}

export interface State extends EntityState<User> {

selectedUserId: number;

}

export function selectUserId(a: user): string {

return a.id;

}

export function sortByName(a: User, b: User): number {

return a.name.localeCompare(b.name);

}

export const adapter: EntityAdapter<User> = createEntityAdapter<User>({

selectId: selectUserId,

sortComparer: sortByName

})

#### **Adapter方法**

这些方法由使用createEntityAdapter时返回的适配器对象提供。这些方法在reducer函数中用于根据您提供的操作管理entity集合。

##### **getInitialState**

根据提供的类型返回entity state的initialState。还通过提供的配置对象提供附加状态。initialState提供你的reducer函数。

user.reducer.ts

import { EntityState, EntityAdapter, createEntityAdapter } from '@ngrx/entity';

export interface User {

id: string;

name: string;

}

export interface State extends EntityState<User> {

// additional entities state properties

selectedUserId: number | null;

}

export const initialState: State = adapter.getInitialState({

selectedUserId: null;

});

export function reducer(state = initialState, action): State {

switch(action.type) {

defualt: {

return state;

}

}

}

#### **Adapter集合方法**

entity适配器也提供针对entity的操作方法。这些方法可以一次改变一个到多个记录。如果进行了更改，则每个方法返回新修改的state，如果未进行任何更改，则返回相同的state。

* addOne: 添加一个entity到集合中
* addMany: 添加多个entity到集合中
* addAll: 使用提供的集合更换当前的集合
* removeOne: 从集合中删除一个entity
* removeMany: 通过id或者谓词从集合中删除多个entity
* removeAll: 清空集合
* updateOne: 更新集合中的一个entity
* updateMany: 更新集合中的多个entity
* upsertOne: 在集合中添加或更新一个entity
* upsertMany: 在集合中添加或更新多个entity
* map: 通过定义的map函数来更新集合中的多个entity,类似于Array.map

user.models.ts

export interface User {

id: string;

name: string;

}

user.actions.ts

import { Action } from '@ngrx/store';

import { Update } from '@ngrx/entity';

import { User } from '../models/user.model';

export enum UserActionTypes {

LOAD\_USERS = '[User]Load User',

ADD\_USER = '[User]Add User',

UPSERT\_USER = '[User]Upsert User',

ADD\_USERS = '[User] Add Users',

UPSERT\_USERS = '[User] Upsert Users',

UPDATE\_USER = '[User] Update User',

UPDATE\_USERS = '[User] Update Users',

MAP\_USERS = '[User] Map Users',

DELETE\_USER = '[User] Delete User',

DELETE\_USERS = '[User] Delete Users',

DELETE\_USERS\_BY\_PREDICATE = '[User] Delete Users By Predicate',

CLEAR\_USERS = '[User] Clear Users',

}

export class LoadUsers implements Action {

readonly type = UserActiontypes.LOAD\_USERS;

constructor(public payload: { users: User[] }) {}

}

export class AddUser implements Action {

readonly type = UserActiontypes.ADD\_USER;

constructor(public payload: { user: User }) {}

}

export class UpsertUser implements Action {

readonly type = UserActiontypes.UPSERT\_USER;

constructor(public payload: { users: User }) {}

}

export class AddUsers implements Action {

readonly type = UserActiontypes.ADD\_USERS;

constructor(public payload: { users: User[] }) {}

}

export class UpsertUsers implements Action {

readonly type = UserActiontypes.UPSERT\_USERS;

constructor(public payload: { users: User[] }) {}

}

export class UpdateUser implements Action {

readonly type = UserActiontypes.UPDATE\_USER;

constructor(public payload: { users: User }) {}

}

export class UpdateUsers implements Action {

readonly type = UserActiontypes.UPDATE\_USERS;

constructor(public payload: { users: User[] }) {}

}

export class MapUsers implements Action {

readonly type = UserActiontypes.MAP\_USERS;

constructor(public payload: { users: User[] }) {}

}

export class DeleteUser implements Action {

readonly type = UserActiontypes.DALETE\_USER;

constructor(public payload: { id: string }) {}

}

export class DeleteUsers implements Action {

readonly type = UserActiontypes.DELETE\_USERS;

constructor(public payload: { ids: string[] }) {}

}

export class DeleteUsersByPredicate implements Action {

readonly type = UserActiontypes.DELETE\_USERS\_BY\_PREDICATE;

constructor(public payload: { predicate: Predicate<User> }) {}

}

export class ClearUsers implements Action {

readonly type = UserActiontypes.CLEAR\_USERS;

}

export type UserActionsUnion =

| LoadUsers

| AddUser

| UpsertUser

| AddUsers

| UpsertUsers

| UpdateUser

| UpdateUsers

| MapUsers

| DeleteUser

| DeleteUsers

| DeleteUsersByPredicate

| ClearUsers;

user.reducer.ts

import { EntityState, EntityAdapter, createEntityAdapter } from '@ngrx/entity';

import { User } from '../models/user.model';

import { UserActionsUnion, UserActiontypes } from '../actions/user.actions';

export interface State extends EntityState<User> {

selectedUserId: number | null;

}

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

export conmst initialState: State = adapter.getInitialState({

selectedUserId: null;

});

export function reducer(state = initialState, action: UserActionsUnion): State {

switch (action.type) {

case UserActionTypes.ADD\_USER: {

return adapter.addOne(action.payload.user, state);

}

case UserActionTypes.UPSERT\_USER: {

return adapter.upsertOne(action.payload.user, state);

}

case UserActionTypes.ADD\_USERS: {

return adapter.addMany(action.payload.users, state);

}

case UserActionTypes.UPSERT\_USERS: {

return adapter.upsertMany(action.payload.users, state);

}

case UserActionTypes.UPDATE\_USER: {

return adapter.updateOne(action.payload.user, state);

}

case UserActionTypes.UPDATE\_USERS: {

return adapter.updateMany(action.payload.users, state);

}

case UserActionTypes.MAP\_USERS: {

return adapter.map(action.payload.entityMap, state);

}

case UserActionTypes.DELETE\_USER: {

return adapter.removeOne(action.payload.id, state);

}

case UserActionTypes.DELETE\_USERS: {

return adapter.removeMany(action.payload.ids, state);

}

case UserActionTypes.DELETE\_USERS\_BY\_PREDICATE: {

return adapter.removeMany(action.payload.predicate, state);

}

case UserActionTypes.LOAD\_USERS: {

return adapter.addAll(action.payload.users, state);

}

case UserActionTypes.CLEAR\_USERS: {

return adapter.removeAll({ ...state, selectedUserId: null });

}

default: {

return state;

}

}

}

export const getSelectedUserId = (state: State) => state.selectedUser.id;

const {

selectIds,

selectEntities,

selectAll,

selectTotal

} = adapter.getSelectors();

export const selectUserIds = selectids;

export const selectUserEntities = selectEntities;

export const selectAllUsers = selectAll;

export const selectUserTotal = selectTotal;

#### **entity选择器**

创建的entity适配器返回的getSelectors方法提供了从entity中选择信息的功能。  
getSelectors方法将选择器函数作为其唯一参数，以选择已定义entity state片段。

index.ts

import { createSelector, createFeatureSelector, ActionReducerMap, } from '@mgrx/store';

import \* as fromUser from './user.reducer';

export interface State {

users: fromUser.State;

}

export const reducers: ActionReducerMap<State> = {

users: fromUser.reducer,

};

export const selectUserState = createFeatureSelector<fromUser.State>('users');

export const selectUserIds = createSelector(

selectUserState,

fromUser.selectUserIds

);

export const selectUserEntities = createSelector(

selectUserState,

fromUser.selectUserEntities

);

export const selectAllUsers = createSelector(

selectUserState,

fromUser.selectAllUsers

);

export const selectUserTotal = createSelector(

selectUserState,

fromUser.selectUserTotal

);

export const selectCurrentUserId = createSelector(

selectUserState,

fromUser.getSelectedUserId

);

export const selectCurrentUser = createSelector(

selectUserEntities,

selectCurrentUserId,

(userEntities, userId) => userEntities[userId]

);

## **Effects**

Effects是Rxjs的副作用模型。Effects使用流来提供新的操作源来和外部交流（例如网络请求，webScoket消息和基于时间的事件）  
****介绍****  
在基于服务的angualr应用中，component直接通过service来和外部资源进行沟通。相反的，effects提供了与服务的交互来将他们与组件隔离。effects是处理任务的地方，例如获取数据，生成多个事件的长时间运行的任务，以及组件不需要明确了解这些交互的其他外部交互。  
****关键概念****

* Effects隔离component的副作用，允许component有更多的纯的select state和actions调用。
* Effects是一个监听store每一个actions调用的Observerable的长期运行的服务。
* Effects根据他们所需要监听的action来过滤actions。
* Effects执行同步或异步的任务并返回一个新的action。

****和基于组件的副作用比较****  
在基于服务的应用中，component通过许多不同的service与数据交互，这些service通过属性和方法公开数据。这些服务可能又依赖于管理另外一些数据的服务。你的component使用这些service来完成任务。  
想象一下，你的应用用来管理电影。下面这个component用于获取并展示电影列表。

movies-page.component.ts

@component({

template: `

<li \*ngFor="let move of movies">{{movie.name}}</li>

`

})

export class MoviesPageComponent {

movies: Movie[];

consatructor(private movieService: MoviesService){}

ngOnInit() {

this.movieService.getAll().subscribe(movies => this.movies = movies);

}

}

你还需要相应的service来提取电影。

@Injectable({

providenIn: 'root'

})

export class MovieService {

constructor (private http: HttpClient) {}

getAll() {

return this.http.get('/movies');

}

}

这个component有多个职责：

* 管理电影的状态
* 使用service来执行副作用，调用外部API来获取电影
* 更改component中电影的状态

使用Effects可以减少component的职责。在更大的应用程序中，这变得更加重要，因为您有多个数据源，需要多个service来获取这些数据，而service可能依赖于其他service。  
Effects处理外部数据和交互，使service只执行与外部交互相关的任务。接下来，重构component以将电影数据放入Store中。Effects处理获取电影数据。

@Component({

template: `

<div \*ngFor="let movie of movies$ | async">

{{ movie.name }}

</div>

`

})

export class MoviesPageComponent {

movies$: Observable = this.store.selet(state => state.movies);

constructor(private store: Store<{movies: Movie[]}>) {}

ngOnInit() {

this.store.dispatch({type: '[movies Page]Load Movies'})

}

}

电影仍然通过MoviesService获取，但该组件不再关注如何获取和加载电影.它只负责声明加载电影和使用选择器访问电影列表数据的意图。Effect通过异步的方式获取电影。  
****撰写Effects****  
要隔离component的副作用，必须创建一个Effects类来侦听事件和执行任务。  
Effects是一个拥有多个不同部分的可注入服务：

* 一个可注入的Action service，提供一个在所有Action调度后，最新的状态改变的Observable刘
* 使用Effects装饰器，用元数据装饰Observable流。元数据用于注册订阅Store流，然后，从Effects返回的任何操作都会重新回到Store流。
* Effects订阅了Store的Observable。
* service注入Effects，完成外部API交互并处理流。

为了说明如何处理上面示例中的加载电影，让我们看一下MovieEffects。

movie.effects.ts

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

import { Actions, Effect, ofType } from '@ngrx/effects';

import { EMPTY } from 'rxjs';

import { map, mergeMap } from 'rxjs/operators';

@Injectable()

export class MovieEffects {

}