一、案例来自 E:\temp\steeditor-master\src\app\store\root-state.ts

https://github.com/jakipatryk/steeditor

1- 定义子状态

E:\temp\steeditor-master\src\app\store\auth-store\state.ts

import { UserData } from './models';

export interface AuthState {

currentUser: string | null;

currentUserData: UserData;

userDataLoading: boolean;

loggingOut: boolean;

}

export const initialState: AuthState = {

currentUser: null,

currentUserData: {

username: null,

about: null,

witnessesVotedFor: null

},

userDataLoading: false,

loggingOut: false

};

2- 定义父状态

import { RouterReducerState } from '@ngrx/router-store';

import { AuthState } from './auth-store';

import { PostsState } from './posts-store';

import { RouterStateUrl } from './router-store/serializer';

export interface State {

auth: AuthState;

drafts: DraftsState;

posts: PostsState;

router: RouterReducerState<RouterStateUrl>;

templates: TemplatesState;

}

二、实体集合的Reducer功能

/\*

\* 案例说明

\*/

E:\temp\steeditor-master\src\app\store\posts-store\state.ts

https://github.com/jakipatryk/steeditor

1 - 实体

export interface Post {

id: number;

entry: Partial<SteemPost>;

}

2 - 编写管理这个集合的reducer的第一步是创建一个实体适配器(entity adapter)：

export const postsAdapter: EntityAdapter<Post> = createEntityAdapter<Post>();

3 - 接下来，我们需要为我们的状态声明接口：

export interface PostsState extends EntityState<Post> {

lastCheckedId: number;

loading: boolean;

broadcasting: boolean;

}

EntityState的声明如下所示：(可索引的类型接口)

export interface EntityState<T> {

ids: string[] | number[];

entities: Dictionary<T>;

}

我们维护一个ID列表和 Entity 字典的主要原因有两点：

1. 我们希望快速查找特定的实体。 如果您只想从 Store 中选择一本书，使用 Entity 词典比搜索数组要快得多

2. 我们也想维护列表的顺序。 如果你想保持列表排序，这是特别重要的！

EntityState的声明符合所有目标。 它也是可扩展的，所以我们可以在Post集合中包含其他相关信息，例如当前选择的Post。

接下来，定义一些操作：

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

export enum BookActionTypes { ADD\_ONE = '[Books] Add One',

UPDATE\_ONE = '[Books] Update One',

DELETE\_ONE = '[Books] Delete One',

GET\_ALL = '[Books] Get All', }

export class AddOne implements Action {

readonly type = BookActionType.ADD\_ONE;

constructor(public book: BookModel) {} }

export class UpdateOne implements Action {

readonly type = BookActionType.UPDATE\_ONE;

constructor( public id: string,

public changes: Partial<BookModel> ) {

}

}

export class DeleteOne implements Action {

readonly type = BookActionType.DELETE\_ONE;

constructor( public id: string ) {}

}

export class GetAll implements Action {

readonly type = BookActionType.GET\_ALL;

constructor( public books: BookModel[] ) {}

}

export type BookActions = AddOne | UpdateOne | DeleteOne | GetAll ;

export const initialState: PostsState = postsAdapter.getInitialState({

lastCheckedId: null,

loading: false,

broadcasting: false

});

现在我们准备使用bookAdapter来创建我们的书籍 reducer：

const initialState: BookState = bookAdapter.getInitialState();

export function bookReducer(

state: BookState = initialState,

action: BookActions, ) {

switch (action.type) {

case BookActionTypes.ADD\_ONE:

return bookAdapter.addOne(action.type, state);

case BookActionTypes.UPDATE\_ONE:

return bookAdapter.updateOne({ id: action.id, changes: action.changes, }, state);

case BookActionTypes.DELETE\_ONE:

return bookAdapter.deleteOne(action.id, state);

case BookActionTypes.GET\_ALL:

return bookAdapter.addAll(action.books, state);

default: return state;

} }

新的状态可以使用新创建的 reducer 在 Store 中注册。

我们需要做的最后一件事就是生成用于处理这个状态的选择器(selector)：

export const { selectIds, selectEntities, selectAll, selectToAll } = bookAdapter.getSelectors();

那么样板代码做了哪些？

1. 不再需要显式声明状态接口(state interface)的所有属性

2. 添加，删除或更新状态实体(state entity)的实现都由适配器处理。

3. 该适配器会为您生成一组常用的选择器。