ngrx-entity

用于管理记录集合的实体状态适配器。

实体提供了操作和查询实体集合的API。

1 - 减少用于创建管理模型集合的简化器的样板。

2 - 为管理实体集合提供performant CRUD操作。

3 - 用于选择实体信息的可扩展类型安全适配器。

详细的安装说明可以在安装页面上找到。

实体和类实例

Entity促进在管理集合时使用纯JavaScript对象。当实体在集合中管理时，ES6类实例将转换为普通的JavaScript对象。这为您在管理这些实体时提供了一些保证:

确保状态中包含的数据结构本身不包含逻辑，从而减少它们自身发生变化的机会。

状态将始终是可序列化的，允许您从浏览器存储机制(如本地存储)存储和补充。

状态可以通过Redux Devtools检查。

这是NgRx的核心原则之一。Redux文档还对这个约束提供了更多的见解。

**1 - Entity Interfaces**

EntityState<T>

实体状态是给定实体集合的预定义通用接口，接口如下:

interface EntityState<V> {

ids: string[] | number[];

entities: { [id: string | id: number]: V };

}

id:集合中所有主id的数组

entities: 由主id索引的集合中的实体字典

扩展此接口以提供实体状态的任何其他属性。

export interface User {

id: string;

name: string;

}

export interface State extends EntityState<User> {

// additional entity state properties

selectedUserId: number | null;

}

**2 - EntityAdapter<T>**

为所提供的实体适配器提供泛型类型接口。实体适配器提供了许多用于管理实体状态的收集方法。

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

Entity Adapter

createEntityAdapter<T>

用于为单个实体状态集合返回通用实体适配器的方法。返回的适配器提供了许多针对集合类型执行操作的适配器方法。该方法接受一个具有两个属性的对象进行配置。

selectId:选择集合的主id的方法。当实体的主键为id时可选

sortComparer:用于对集合进行排序的比较函数。只有在显示之前需要对集合进行排序时，才

需要使用comparer函数。设置为false以保持集合未排序，这样在CRUD操作期间性能更好。

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;

}

export function selectUserId(a: User): string {

//在本例中，这是可选的，因为主键是id

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,

});

**3 - Adapter Methods**

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函数中用于根据提供的操作管理实体集合。

getInitialState

根据所提供的类型返回实体状态的initialState。其他状态也通过提供的配置对象提供。initialState将提供给您的reducer函数。

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

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({

// additional entity state properties

selectedUserId: null,

});

const userReducer = createReducer(initialState);

export function reducer(state: State | undefined, action: Action) {

return userReducer(state, action);

}

Adapter Collection Methods

实体适配器还提供针对实体的操作的方法。这些方法可以一次将一条记录更改为多条记录。如果进行了更改，则每个方法返回新修改的状态;如果没有进行更改，则返回相同的状态。

addOne: 向集合添加一个实体

addMany: 向集合中添加多个实体

setAll: 用提供的集合替换当前集合

setOne: 添加或替换集合中的一个实体

removeOne:从集合中删除一个实体

removeMany: 按id或谓词从集合中删除多个实体

removeAll: 明确的实体集合

updateOne: 更新集合中的一个实体。支持部分更新。

updateMany: 更新集合中的多个实体。支持部分更新

upsertOne: 添加或更新集合中的一个实体。支持部分更新。

upsertMany: 添加或更新集合中的多个实体。支持部分更新。

map: 通过定义映射函数(类似于Array.map)来更新集合中的多个实体

export interface User {

id: string;

name: string;

}

import { createAction, props } from '@ngrx/store';

import { Update, EntityMap, Predicate } from '@ngrx/entity';

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

export const loadUsers = createAction('[User/API] Load Users', props<{ users: User[] }>());

export const addUser = createAction('[User/API] Add User', props<{ user: User }>());

export const setUser = createAction('[User/API] Set User', props<{ user: User }>());

export const upsertUser = createAction('[User/API] Upsert User', props<{ user: User }>());

export const addUsers = createAction('[User/API] Add Users', props<{ users: User[] }>());

export const upsertUsers = createAction('[User/API] Upsert Users', props<{ users: User[] }>());

export const updateUser = createAction('[User/API] Update User', props<{ update: Update<User> }>());

export const updateUsers = createAction('[User/API] Update Users', props<{ updates: Update<User>[] }>());

export const mapUsers = createAction('[User/API] Map Users', props<{ entityMap: EntityMap<User> }>());

export const deleteUser = createAction('[User/API] Delete User', props<{ id: string }>());

export const deleteUsers = createAction('[User/API] Delete Users', props<{ ids: string[] }>());

export const deleteUsersByPredicate = createAction('[User/API] Delete Users By Predicate', props<{ predicate: Predicate<User> }>());

export const clearUsers = createAction('[User/API] Clear Users');

port { Action, createReducer, on } from '@ngrx/store';

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

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

import \* as UserActions from '../actions/user.actions';

export interface State extends EntityState<User> {

// additional entities state properties

selectedUserId: number | null;

}

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

export const initialState: State = adapter.getInitialState({

// additional entity state properties

selectedUserId: null,

});

const userReducer = createReducer(

initialState,

on(UserActions.addUser, (state, { user }) => { return adapter.addOne(user, state) }),

on(UserActions.setUser, (state, { user }) => { return adapter.setOne(user, state) }),

on(UserActions.upsertUser, (state, { user }) => { return adapter.upsertOne(user, state); }),

on(UserActions.addUsers, (state, { users }) => { return adapter.addMany(users, state); }),

on(UserActions.upsertUsers, (state, { users }) => { return adapter.upsertMany(users, state);

}),

on(UserActions.updateUser, (state, { update }) => { return adapter.updateOne(update, state); }),

on(UserActions.updateUsers, (state, { updates }) => {

return adapter.updateMany(updates, state); }),

on(UserActions.mapUsers, (state, { entityMap }) => {

return adapter.map(entityMap, state); }),

on(UserActions.deleteUser, (state, { id }) => { return adapter.removeOne(id, state); }),

on(UserActions.deleteUsers, (state, { ids }) => { return adapter.removeMany(ids, state); }),

on(UserActions.deleteUsersByPredicate, (state, { predicate }) => {

return adapter.removeMany(predicate, state); }),

on(UserActions.loadUsers, (state, { users }) => { return adapter.setAll(users, state); }),

on(UserActions.clearUsers, state => {

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

);

export function reducer(state: State | undefined, action: Action) {

return userReducer(state, action);

}

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

// get the selectors

const { selectIds, selectEntities, selectAll, selectTotal,} = adapter.getSelectors();

// select the array of user ids

export const selectUserIds = selectIds;

// select the dictionary of user entities

export const selectUserEntities = selectEntities;

// select the array of users

export const selectAllUsers = selectAll;

// select the total user count

export const selectUserTotal = selectTotal;

**Entity Updates**

在使用实体适配器更新实体时，需要注意一些注意事项。

首先，updateOne和updateMany使用了如下所示的Update<T>接口。这支持部分更新

interface UpdateStr<T> { id: string; changes: Partial<T>;}

interface UpdateNum<T> { id: number; changes: Partial<T>;}

type Update<T> = UpdateStr<T> | UpdateNum<T>;

其次，upsertOne和upsertMany将执行插入或更新。如果提供了部分实体，则执行更新。

要防止部分更新，可以显式地设置所有字段，将未使用的字段设置为值未定义，或者使用setOne或setAll适配器方法。

**Entity Selectors**

创建的实体适配器返回的getSelectors方法提供了从实体选择信息的函数。

getSelectors方法将选择器函数作为其唯一的参数，用于选择已定义实体的状态块。

port { createSelector, createFeatureSelector, ActionReducerMap,} from '@ngrx/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 // shorthand for usersState => fromUser.selectUserIds(usersState)

);

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]

);

Additional Entity State Properties Update

可以向从EntityState扩展的状态添加额外属性。这些属性必须手动更新。就像在非实体状态中一样，我们可以更新减速器中添加的属性。这可以使用或不使用@ngrx/entity辅助函数来完成。

下面的步骤向您展示了如何扩展实体适配器示例。

将selectedUserId声明为接口中的附加属性。

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

export interface User {

id: string;

name: string;

}

export interface State extends EntityState<User> {

// additional state property

selectedUserId: number;

}

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

然后创建一个动作来更新selectedUserId

import { createAction, props } from '@ngrx/store';

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

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

export const selectUser = createAction('[Users Page] Select User', props<{ userId: string }>());

export const loadUsers = createAction('[User/API] Load Users', props<{ users: User[] }>());

实体适配器仅用于更新EntityState属性。附加的状态属性应该与常规状态属性一样进行更新，如下面的示例所示。

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

import { Action, createReducer, on } from '@ngrx/store';

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

import \* as UserActions from '../actions/user.actions';

export interface State extends EntityState<User> {

// additional state property

selectedUserId: number | null;

}

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

export const initialState: State = adapter.getInitialState({

// additional entity state properties

selectedUserId: null,

});

export const userReducer = createReducer(

initialState,

on(UserActions.selectUser, (state, { userId }) => {

return { ...state, selectedUserId: userId };

}),

on(UserActions.loadUsers, (state, { users }) => {

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

})

);

export function reducer(state: State | undefined, action: Action) {

return userReducer(state, action);

}