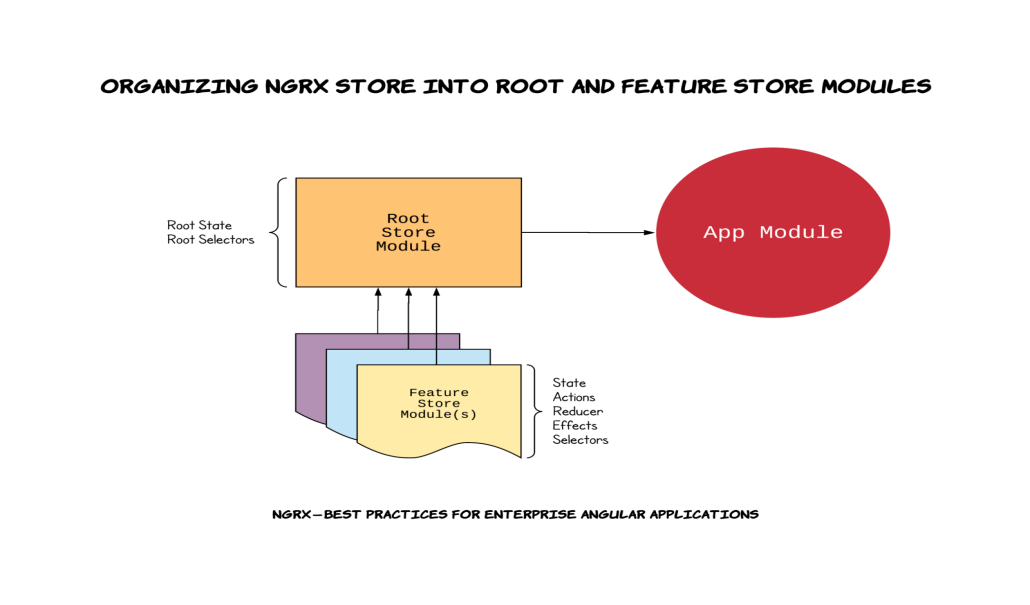
Ngrx-企业Angular应用的最佳实践

https://wesleygrimes.com/angular/2018/05/30/ngrx-best-practices-for-enterprise-angular-applications

以下链接是应用最佳实践的一个案例

<https://github.com/wesleygrimes/ticket-system> (比较全)

<https://github.com/wesleygrimes/github-user-search>



1. ***在我们开始之前***

本文不是关于NgRx的教程。现在有几种很好的资源，是由比我聪明得多的专家编写的。我强烈建议您在尝试实现这些概念之前，先花时间学习NgRx和redux模式。

原文中有链接

1. ***背景***

下面是我在日常工作中使用NgRx库构建几个企业Angular应用程序后开发的一个模式。我发现大多数在线教程都能很好地帮助您建立和运行您的存储，但是常常不能很好地说明在存储特性片、根存储（root store）和用户界面之间清晰地分离关注点的最佳实践。

使用以下模式，根应用程序状态和根应用程序状态的每个片(属性)被分离到一个RootStoreModule和功能特性MyFeatureStoreModule中。

1. ***最佳实践#1 -根存储模块***

创建一个根存储模块，作为一个合适的Angular NgModule，将NgRxstore logic（存储逻辑）捆绑在一起。功能存储模块（Feature store modules）将被导入到根存储模块中（ root store module），允许将单个根存储模块导入到应用程序的主应用程序模块中。

1 - Generate RootStoreModule

2 - Generate RootState 生成RootState接口来表示应用程序的整个状态:

这将创建一个名为RootState的接口，但是您需要在生成的.ts文件中将其重命名为State，因为我们希望稍后将其用作RootStoreState.State  
请注意:您稍后将返回并将每个功能模块作为属性添加到此接口。

1. ***最佳实践 #2 — Create Feature Store Module(s)***

创建特性存储模块（feature store modules），作为Angular NgModule的一部分，将store的特性切片（feature slices）捆绑在一起，包括状态（state）、动作（actions）、reducer、选择器（selectors）和effects。然后将功能模块导入到RootStoreModule中。这将使您的代码清晰地组织到每个feature store的子目录中。此外，如本文后面所述，动作（actions）、选择器（selectors）和状态（state）是名称空间的，并使用特性存储（feature store）前缀导出。

### 命名您的特性库（ Feature Store）

在下面的示例实现中，我们将使用特性（feature ）名称MyFeature，但是，对于生成的每个特性（feature ），这将是不同的，并且应该与之密切对应

RootState属性名。例如，如果您正在构建博客应用程序，特性名称可能是Post。

### 2 - Entity Feature Modules or Standard Feature Modules?

根据您正在创建的特性类型，您可能从实现 [NgRx Entity](https://medium.com/ngrx/introducing-ngrx-entity-598176456e15)中获益，也可能不会。如果您的存储功能片（store feature slice）将处理数组类型，那么我建议遵循下面的实体功能模块（Entity Feature Module）实现。如果构建一个不包含标准类型数组的存储功能片，那么我建议遵循下面的标准功能模块（ Standard Feature Module ）实现。

### 3 - 建议实现--实体特征模块（ Entity Feature Module）

1. Generate MyFeatureStoreModule feature module using the
2. Actions — Create an actions.ts file in the
3. State — Create a state.ts file in the app/root-store/my-feature-store director

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

import { MyModel } from '../../models';

export const featureAdapter: EntityAdapter<MyModel> = createEntityAdapter< MyModel>({

selectId: model => model.id,

sortComparer: (a: MyModel, b: MyModel): number =>

b.someDate.toString().localeCompare(a.someDate.toString())

});

export interface State extends EntityState<MyModel> {

isLoading?: boolean;

error?: any;

}

export const initialState: State = featureAdapter.getInitialState({

isLoading: false,

error: null

});

1. Reducer — Create a reducer.ts file in the
2. Selectors — Create a selectors.ts file in the

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

import { MyModel } from '../models';

import { featureAdapter, State } from './state';

export const getError = (state: State): any => state.error;

export const getIsLoading = (state: State): boolean => state.isLoading;

export const selectMyFeatureState: MemoizedSelector<object,State> = createFeatureSelector<State>('myFeature');

export const selectAllMyFeatureItems: ( state: object) =>

MyModel[] = featureAdapter.getSelectors(selectMyFeatureState).selectAll;

export const selectMyFeatureById = (id: string) =>

createSelector(

this.selectAllMyFeatureItems,

(allMyFeatures: MyModel[]) => {

if (allMyFeatures) {

return allMyFeatures.find(p => p.id === id);

} else {

return null;

}

}

);

export const selectMyFeatureError: MemoizedSelector< object, any> = createSelector(

selectMyFeatureState, getError);

export const selectMyFeatureIsLoading: MemoizedSelector< object, boolean> = createSelector(

selectMyFeatureState, getIsLoading);

1. Effects — Create an effects.ts file in the app/root-store/my-feature-store directory with the following:

### 4 - 建议实现--标准功能模块（ Standard Feature Module ）

1. Actions — Create an actions.ts file in the

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

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

export enum ActionTypes {

LOGIN\_REQUEST = '[My Feature] Login Request',

LOGIN\_FAILURE = '[My Feature] Login Failure',

LOGIN\_SUCCESS = '[My Feature] Login Success'

}

export class LoginRequestAction implements Action {

readonly type = ActionTypes.LOGIN\_REQUEST;

constructor(public payload: { userName: string; password: string }) {}

}

export class LoginFailureAction implements Action {

readonly type = ActionTypes.LOGIN\_FAILURE;

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

}

export class LoginSuccessAction implements Action {

readonly type = ActionTypes.LOGIN\_SUCCESS;

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

}

export type Actions =

| LoginRequestAction

| LoginFailureAction

| LoginSuccessAction;

1. State — Create a state.ts

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

export interface State {

user: User | null;

isLoading: boolean;

error: string;

}

export const initialState: State = {

user: null,

isLoading: false,

error: null

};

1. Reducer — Create a reducer.ts file in the

import { Actions, ActionTypes } from './actions';

import { initialState, State } from './state';

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

switch (action.type) {

case ActionTypes.LOGIN\_REQUEST:

return {

...state,

error: null,

isLoading: true

};

case ActionTypes.LOGIN\_SUCCESS:

return {

...state,

user: action.payload.user,

error: null,

isLoading: false

};

case ActionTypes.LOGIN\_FAILURE:

return {

...state,

error: action.payload.error,

isLoading: false

};

default: {

return state;

}

}

}

1. Selectors — Create a selectors.ts file in the

const getError = (state: State): any => state.error;

const getIsLoading = (state: State): boolean => state.isLoading;

const getUser = (state: State): any => state.user;

export const selectMyFeatureState: MemoizedSelector< object, State>

= createFeatureSelector<State>('myFeature');

export const selectMyFeatureError: MemoizedSelector< object, any> = createSelector(

selectMyFeatureState, getError);

export const selectMyFeatureIsLoading: MemoizedSelector< object, boolean> = createSelector(

selectMyFeatureState, getIsLoading);

export const selectMyFeatureUser: MemoizedSelector< object, User> = createSelector(

selectMyFeatureState, getUser);

1. Effects — Create an effects.ts file in the

@Injectable()

export class MyFeatureStoreEffects {

constructor(private dataService: DataService, private actions$: Actions) {}

@Effect()

loginRequestEffect$: Observable<Action> = this.actions$.pipe(

ofType<featureActions.LoginRequestAction>(

featureActions.ActionTypes.LOGIN\_REQUEST

),

switchMap(action =>

this.dataService

.login(action.payload.userName, action.payload.password)

.pipe(

map(

user =>

new featureActions.LoginSuccessAction({

user

})

),

catchError(error =>

observableOf(new featureActions.LoginFailureAction({ error }))

)

)

)

);

}

#### 4 - 建议实现--实体和标准功能模块（Entity and Standard Feature Modules）

现在我们已经创建了我们的功能模块，无论是实体类型还是上面的标准类型，我们需要将这些(state, actions, reducer, effects, selectors)导入到Angular NgModule中。此外，我们将创建一个桶导出（ barrel export），以使应用程序组件中的导入整洁有序，并使用断言的名称空间。

1. Update the my-feature-store.module.ts with the following:

@NgModule({

imports: [

CommonModule,

StoreModule.forFeature('myFeature', featureReducer),

EffectsModule.forFeature([MyFeatureStoreEffects])

],

providers: [MyFeatureStoreEffects]

})

export class MyFeatureStoreModule {}

1. 创建一个应用程序/ root-store / my-feature-store /index.ts筒出口（ [barrel export](https://twitter.com/toddmotto/status/918818392680824832)）。您将注意到，我们导入了 store components，并在重新导出它们之前将它们别名化。这本质上是我们的存储组件的“名称空间”。

import \* as MyFeatureStoreActions from './actions';

import \* as MyFeatureStoreSelectors from './selectors';

import \* as MyFeatureStoreState from './state';

export { MyFeatureStoreModule } from './my-feature-store.module';

export { MyFeatureStoreActions, MyFeatureStoreSelectors, MyFeatureStoreState };

1. ***最佳实践#1 -根存储模块(续)***

#### 更新应用程序/ root-store / root-state.ts并为我们之前创建的每个特性添加一个属性:

import { MyFeatureStoreState } from './my-feature-store';

import { MyOtherFeatureStoreState } from './my-other-feature-store';

export interface State {

myFeature: MyFeatureStoreState.State;

myOtherFeature: MyOtherFeatureStoreState.State;

}

2 - 更新您的应用程序/ root-store / root-store.module。通过导入所有的功能模块，并导入以下NgRx模块:StoreModule.forRoot({})和EffectsModule.forRoot([]):

import { CommonModule } from '@angular/common';

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

import { EffectsModule } from '@ngrx/effects';

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

import { MyFeatureStoreModule } from './my-feature-store/';

import { MyOtherFeatureStoreModule } from './my-other-feature-store/';

@NgModule({

imports: [

CommonModule,

MyFeatureStoreModule,

MyOtherFeatureStoreModule,

StoreModule.forRoot({}),

EffectsModule.forRoot([])

],

declarations: []

})

export class RootStoreModule {}

3 - 创建一个应用程序/ root-store /selectors.ts文件。这将包含任何根状态级选择器( selectors)，例如（Loading）属性，甚至是聚合错误属性（or even an aggregate Error property）:

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

import { MyFeatureStoreSelectors } from './my-feature-store';

import { MyOtherFeatureStoreSelectors } from './my-other-feature-store';

export const selectError: MemoizedSelector<object, string> = createSelector(

MyFeatureStoreSelectors.selectMyFeatureError,

MyOtherFeatureStoreSelectors.selectMyOtherFeatureError,

(myFeatureError: string, myOtherFeatureError: string) => { return myFeature || myOtherFeature; }

);

export const selectIsLoading: MemoizedSelector< object, boolean> = createSelector(

MyFeatureStoreSelectors.selectMyFeatureIsLoading,

MyOtherFeatureStoreSelectors.selectMyOtherFeatureIsLoading,

(myFeature: boolean, myOtherFeature: boolean) => { return myFeature || myOtherFeature; }

);

4 - Create an app/root-store/index.ts barrel export for your store with the following:

import { RootStoreModule } from './root-store.module';

import \* as RootStoreSelectors from './selectors';

import \* as RootStoreState from './state';

export \* from './my-feature-store';

export \* from './my-other-feature-store';

export { RootStoreState, RootStoreSelectors, RootStoreModule };

1. ***将根存储模块连接到应用程序***

现在我们已经构建了由特性存储模块组成的根存储模块，让我们将它添加到主app.module中。并显示如何整齐和清洁的布线过程是。

1 - Add RootStoreModule to your application’s NgModule.imports array. Make sure that when you import the module to pull from the barrel export:

将RootStoreModule添加到应用程序的NgModule中。imports array（进口数组）。确保当你导入模块从桶中拉出时:

import { RootStoreModule } from './root-store';

2 - 用例:

@Component({

selector: 'app-my-feature',

styleUrls: ['my-feature.component.css'],

templateUrl: './my-feature.component.html'

})

export class MyFeatureComponent implements OnInit {

myFeatureItems$: Observable<MyModel[]>;

error$: Observable<string>;

isLoading$: Observable<boolean>;

constructor(private store$: Store<RootStoreState.State>) {}

ngOnInit() {

this.myFeatureItems$ = this.store$.select(

MyFeatureStoreSelectors.selectAllMyFeatureItems

);

this.error$ = this.store$.select(

MyFeatureStoreSelectors.selectUnProcessedDocumentError

);

this.isLoading$ = this.store$.select(

MyFeatureStoreSelectors.selectUnProcessedDocumentIsLoading

);

this.store$.dispatch(new MyFeatureStoreActions.LoadRequestAction());

}

}

1. ***完成应用程序结构***

一旦我们完成了上述最佳实践的实现，我们的Angular应用程序结构应该看起来像这样:

├── app

│ ├── app-routing.module.ts

│ ├── app.component.css

│ ├── app.component.html

│ ├── app.component.ts

│ ├── app.module.ts

│ ├── components

│ ├── containers

│ │ └── my-feature

│ │ ├── my-feature.component.css

│ │ ├── my-feature.component.html

│ │ └── my-feature.component.ts

│ ├── models

│ │ ├── index.ts

│ │ └── my-model.ts

│ │ └── user.ts

│ ├── root-store

│ │ ├── index.ts

│ │ ├── root-store.module.ts

│ │ ├── selectors.ts

│ │ ├── state.ts

│ │ └── my-feature-store

│ │ | ├── actions.ts

│ │ | ├── effects.ts

│ │ | ├── index.ts

│ │ | ├── reducer.ts

│ │ | ├── selectors.ts

│ │ | ├── state.ts

│ │ | └── my-feature-store.module.ts

│ │ └── my-other-feature-store

│ │ ├── actions.ts

│ │ ├── effects.ts

│ │ ├── index.ts

│ │ ├── reducer.ts

│ │ ├── selectors.ts

│ │ ├── state.ts

│ │ └── my-other-feature-store.module.ts

│ └── services

│ └── data.service.ts

├── assets

├── browserslist

├── environments

│ ├── environment.prod.ts

│ └── environment.ts

├── index.html

├── main.ts

├── polyfills.ts

├── styles.css

├── test.ts

├── tsconfig.app.json

├── tsconfig.spec.json

└── tslint.json

1. ***一个完整的案例 — Chuck Norris Joke Generator***

Github

<https://github.com/wesleygrimes/angular-ngrx-chuck-norris>

演示

You can see the live demo at [https://angular-ngrx-chuck-norris.stackblitz.io](https://angular-ngrx-chuck-norris.stackblitz.io/) and here is the [Stackblitz](https://stackblitz.com/)editor:

1. ***额外的资源***我强烈推荐您注册Angular的终极课程，尤其是NgRx课程。这是非常值得的钱，我已经把它作为一个培训工具，为新的Angular开发人员。点击下面的链接注册。  
   终极课程:JavaScript、Angular、NGRX、TypeScript等专业在线课程