**router-store**

绑定来连接Angular路由器和Store。在每个路由器导航周期中，多个动作被分派，允许你监听路由器状态的变化。然后可以从路由器的状态中选择数据，为应用程序提供额外的信息。

import { StoreRouterConnectingModule, routerReducer } from '@ngrx/router-store';

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

@NgModule({

imports: [ BrowserModule,

StoreModule.forRoot({

router: routerReducer,

}),

RouterModule.forRoot([ // routes ]),

// 连接RouterModule和StoreModule，默认使用MinimalRouterStateSerializer

StoreRouterConnectingModule.forRoot(),

],

bootstrap: [AppComponent],

})

export class AppModule {}

**1 - Router Actions**

Router Store提供了五个导航操作，它们按特定的顺序被分派。Router Store提供的routerReducer用动作给出的最新路由器状态更新其状态。默认情况下，我们建议使用我们提供的creator函数。

Actions

routerRequestAction

在每次导航开始时，路由器将分派一个ROUTER\_REQUEST操作。

routerNavigationAction

在导航期间，在任何守卫或解析器运行之前，路由器将分派一个ROUTER\_NAVIGATION动作。

如果您希望ROUTER\_NAVIGATION在警卫或解析器运行后被分派，请更改导航动作的时间。

routerNavigatedAction

成功导航之后，路由器将分派一个ROUTER\_NAVIGATED操作。

routerCancelAction

当导航被取消时，例如由于一个守卫说用户不能访问所请求的页面，路由器将发送一个ROUTER\_CANCEL操作。

该操作包含导航之前的存储状态。您可以使用它来恢复存储的一致性。

routerErrorAction

当导航过程中出现错误时，路由器将发送一个ROUTER\_ERROR操作。

该操作包含导航之前的存储状态。您可以使用它来恢复存储的一致性。

您还可以使用操作类型，它是在NgRx中引入操作创建器之前定义的方式。如果您正在寻找操作类型的示例，请访问版本7的文档。x和之前。

操作的顺序

Success case:

ROUTER\_REQUEST ROUTER\_NAVIGATION ROUTER\_NAVIGATED

Error / Cancel case (with early Navigation Action Timing):

ROUTER\_REQUEST ROUTER\_NAVIGATION ROUTER\_CANCEL / ROUTER\_ERROR

Error / Cancel case (with late Navigation Action Timing):

ROUTER\_REQUEST ROUTER\_CANCEL / ROUTER\_ERROR

**2 - Router selectors**

在@ngrx/路由器-store中提供的getSelectors方法提供了从路由器状态选择公共信息的函数。

getSelectors方法采用选择器函数作为它的唯一参数来选择存储路由器状态的状态块。下面的示例展示了如何为state对象中的顶级路由器键提供选择器。

注意:getSelectors方法与@ngrx/router-store提供的routerReducer一起工作。如果使用自定义序列化器，则需要提供自己的选择器。

reducers/index.ts

import \* as fromRouter from '@ngrx/router-store';

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

export interface State {

router: fromRouter.RouterReducerState<any>;

}

export const selectRouter = createFeatureSelector<

State, fromRouter.RouterReducerState<any>

>('router');

export const {

selectCurrentRoute, // 选择当前路由

selectQueryParams, // 选择当前路由查询参数

selectQueryParam, // 工厂函数来选择查询参数

selectRouteParams, // 选择当前路由参数

selectRouteParam, // 工厂功能选择路线参数

selectRouteData, // 选择当前路由数据

selectUrl, // 选择当前url

} = fromRouter.getSelectors(selectRouter);

export const selectSelectedCarId = selectQueryParam('carId');

export const selectCar = createSelector(

selectCarEntities,

selectSelectedCarId,

(cars, carId) => cars[carId]

);

export const selectCarsByColor = createSelector(

selectCarEntities,

selectQueryParams,

(cars, params) => cars.filter(c => c.color === params['color'])

);

Configuration Options

interface StoreRouterConfig {

stateKey?: string | Selector<any, RouterReducerState<T>>;

serializer?: new (...args: any[]) => RouterStateSerializer;

navigationActionTiming?: NavigationActionTiming;

routerState?: RouterState;

}

stateKey:reducer键的名称，默认为路由器。还可以提供一个选择器函数。

serializer:如何序列化路由器快照。默认设置为DefaultRouterStateSerializer(参见默认路由器状态序列化器)。有关更多信息，请参见自定义路由器状态序列化器。

navigationActionTiming:当ROUTER\_NAVIGATION被分派时。默认为NavigationActionTiming.PreActivation。有关更多信息，请参见导航动作计时。

routerState:设置此属性以决定应该使用哪个序列化器(如果没有提供)和分派操作上的元数据。

**3 - Default Router State Serializer**

DefaultRouterStateSerializer路由器状态序列化器，序列化URL与ActivatedRouteSnapshot从Angular路由器。后者是递归序列化的，但只允许向下遍历路由，因为根和父参数被设置为undefined。

当启用串行化运行时检查时，不能使用DefaultRouterStateSerializer。启用序列化运行时检查后，必须使用MinimalRouterStateSerializer序列化器。这也适用于具有不变性运行时检查的Ivy。

**4 - Custom Router State Serializer**

在每个导航周期中，会分发一个RouterNavigationAction，并附带其有效负载中的状态快照RouterStateSnapshot。RouterStateSnapshot是一个大型的复杂结构，包含许多关于当前状态和路由器呈现内容的信息。当与存储Devtools一起使用时，这会导致性能问题。在大多数情况下，你可能只需要RouterStateSnapshot提供的一条信息。为了减少导航期间提供的RouterStateSnapshot，您为快照提供了一个自定义序列化器，以只返回您需要的内容

您的自定义序列化器应该实现抽象类RouterStateSerializer并返回一个快照，该快照应该具有扩展BaseRouterStoreState的接口。

然后通过配置提供序列化器。

In a custom serializer ts file

port { Params, RouterStateSnapshot } from '@angular/router';

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

export interface RouterStateUrl {

url: string;

params: Params;

queryParams: Params;

}

export class CustomSerializer implements RouterStateSerializer<RouterStateUrl> {

serialize(routerState: RouterStateSnapshot): RouterStateUrl {

let route = routerState.root;

while (route.firstChild) {

route = route.firstChild;

}

const {

url,

root: { queryParams },

} = routerState;

const { params } = route;

// Only return an object including the URL, params and query params

// instead of the entire snapshot

return { url, params, queryParams };

}

}

In your root reducer

export const reducers: ActionReducerMap<State> = {

router: routerReducer

};

In your AppModule

@NgModule({

imports: [

StoreModule.forRoot(reducers),

RouterModule.forRoot([

// routes

]),

StoreRouterConnectingModule.forRoot({

serializer: CustomSerializer

})

]

})

export class AppModule {}

**6 - Navigation action timing**

默认情况下，ROUTER\_NAVIGATION是在运行任何保护程序或解析器之前分派的。这可能并不总是理想的，例如，如果您依赖于在保护和解析器成功运行并且新路由将被激活之后要分派的动作。你可以通过提供相应的配置来改变调度时间:

StoreRouterConnectingModule.forRoot({

navigationActionTiming: NavigationActionTiming.PostActivation,

});

**7 - Router State Snapshot**

此属性决定应该使用哪个路由器序列化器。如果提供了自定义序列化器，它将使用提供的序列化器。routerState还在已分派的@ngrx/路由器-store操作上设置元数据。

RouterState.Full

当此属性设置为RouterState时。Full， @ngrx/路由器-store将使用DefaultRouterStateSerializer序列化器来序列化Angular路由器事件。

动作的元数据将包含Angular路由器事件，例如NavigationStart和routing recognition。

app.module.ts

StoreRouterConnectingModule.forRoot({

routerState: RouterState.Full,

});

RouterState.Minimal

RouterState。MinimalRouterStateSerializer序列化Angular路由器的RouterState和RouterEvent。

MinimalRouterStateSerializer和默认的RouterStateSerializer之间的区别在于此序列化器是完全可序列化的。为了使状态和动作可序列化，属性paramMap、queryParamMap和component被忽略。

app.module.ts

StoreRouterConnectingModule.forRoot({

routerState: RouterState.Minimal,

});