**Reducers**

NgRx中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函数根据[actions](https://ngrx.io/guide/store/actions) 的类型，确定要处理哪些[actions](https://ngrx.io/guide/store/actions) 来完成这些转换。

Reducers 是纯函数，因为它们对给定的输入产生相同的输出。它们没有副作用，并同步处理每个状态转换。每个reducer函数采用最新分派的动作、当前状态，并确定是返回新修改的状态还是返回原始状态。本指南向您展示了如何编写reducer函数、在存储库中注册它们以及编写特性状态。

Reducers 管理的每个状态部分都有几个一致的部分。

一个接口或类型定义state的形状

参数， 包括 初始状态或当前状态和 当前 action.

The action(s). 相关 函数 , 处理 状态 改变

下面是处理记分牌状态和相关的reducer函数的一组操作示例。

首先，定义一些与状态块交互的操作。

export const homeScore = createAction('[Scoreboard Page] Home Score');

export const awayScore = createAction('[Scoreboard Page] Away Score');

export const resetScore = createAction('[Scoreboard Page] Score Reset');

export const setScores = createAction('[Scoreboard Page] Set Scores', props<{game: Game}>());

接下来，创建一个reducer文件，该文件导入actions并为state块定义一个状态。

1 - 定义state接口 (Defining the state shape)

每个reducer函数都是actions的侦听器。上面定义的记分牌action描述了reducer处理的可能的转换。导入多个actions集来处理reducer中的其他state转换

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

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

export interface State { home: number; away: number; }

您可以根据捕获的对象来定义state的形状，捕获的对象是单一类型(如数字)，还是具有多个属性的更复杂的对象。

2 - 设置初始状态

初始状态为状态提供一个初始值，或者在当前状态未定义时提供一个值。使用所需状态属性的默认值设置初始状态。

创建并导出一个变量，以捕获具有一个或多个默认值的初始状态。

export const initialState: State = { home: 0, away: 0, };

状态的home和away属性的初始值为0。

3 - 创建reducer 函数

reducer函数的职责是以不可变的方式处理状态转换。创建一个reducer函数，该函数处理使用createReducer函数管理记分牌状态的操作。

const scoreboardReducer = createReducer(

initialState,

on(ScoreboardPageActions.homeScore, state => ({ ...state, home: state.home + 1 })),

on(ScoreboardPageActions.awayScore, state => ({ ...state, away: state.away + 1 })),

on(ScoreboardPageActions.resetScore, state => ({ home: 0, away: 0 })),

on(ScoreboardPageActions.setScores,

(state, { game }) => ({ home: game.home, away: game.away }))

);

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

return scoreboardReducer(state, action);

}

***导出的reducer函数是必需的***，因为AOT编译器不支持函数调用。

在上面的例子中，reducer函数的职责是以***不可变的方式***处理状态转换处理4个action。每个action都是强类型的。每个action都不可更改地处理状态转换。这意味着状态转换没有修改原始状态，而是使用*spread操作符*返回一个新的状态对象。扩展语法将属性从当前状态复制到对象中，创建一个新的引用。这就确保了每个变化都会产生一个新的状态，保持了变化的纯粹性。这还可以促进引用完整性，保证在发生状态更改时丢弃旧引用。

spread操作符只做浅复制，不处理深度嵌套的对象。您需要复制对象中的每个级别，以确保不变性。有处理深度复制的库，***包括lodash和immer***。

当一个action被分派时，所有注册的reducer都会收到这个action。它们是否处理action由将一个或多个action与给定state 更改关联的on函数决定。

您还可以使用switch语句编写reducer，这是在NgRx中引入reducer创建器之前定义的方法。如果您正在寻找使用switch语句的简化器示例，请访问version 7的文档。x和之前。

**Registering root state**

应用程序的状态被定义为一个大对象。注册reducer函数来管理状态的某些部分只会在对象中定义具有关联值的键。要在应用程序中注册全局存储，请使用StoreModule.forRoot()方法，并使用定义状态的键/值对映射。forroot()为您的应用程序注册全局提供程序，包括您注入到组件和服务中的存储服务，以分派操作和选择状态片段。

import \* as fromScoreboard from './reducers/scoreboard.reducer';

@NgModule({

imports: [

StoreModule.forRoot({ game: fromScoreboard.reducer })

],

})

export class AppModule {}

向StoreModule.forRoot()注册状态可以确保在应用程序启动时定义状态。通常，您注册的根状态总是需要立即对应用程序的所有区域可用。

**Register feature state**

feature state（特性状态）的行为方式与根状态相同，但允许您使用应用程序中的特定特性区域来定义它们。您的状态是一个大对象，特性状态在该对象中注册了附加的键和值。

查看一个示例状态对象，可以看到特性状态如何允许增量地构建状态。

让我们从一个空状态对象开始。

@NgModule({

imports: [

StoreModule.forRoot({})

],

})

export class AppModule {}

这将为您的应用程序注册一个用于根状态的空对象。

现在使用记分牌reducer和一个名为ScoreboardModule的模块来注册额外的状态。

export const scoreboardFeatureKey = 'game';

import \* as fromScoreboard from './reducers/scoreboard.reducer';

@NgModule({

imports: [

StoreModule.forFeature(fromScoreboard.scoreboardFeatureKey, fromScoreboard.reducer)

], })

export class ScoreboardModule {}

*注意:建议在注册特性状态和调用createFeatureSelector时抽象一个特性键字符串，以防止硬编码字符串。*

将ScoreboardModule添加到AppModule以快速加载状态。

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

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

export class AppModule {}

一旦ScoreboardModule被加载，游戏键就成为对象中的一个属性，并且现在在状态中被管理。

{ game: { home: 0, away: 0 }}

feature state（特性状态）是迫切加载还是延迟加载取决于应用程序的需要。您可以使用特性状态随着时间的推移以及通过不同的特性区域来构建状态对象。