了解NgRx StoreModule背后的魔力

<https://indepth.dev/posts/1199/understanding-the-magic-behind-ngrx-store>

我写这篇文章的时候正在浏览源代码，试图理解@ngrx/store模块的主要实体是如何绑定在一起的。结果，我发现了与模块的每个组件相关的一堆有趣的东西，这就是我将在本文中描述的内容。我们将详细检查每个实体，并解释其在整个体系结构中的作用。

在开始之前，让我们快速回顾一下主要的实体是什么:

State: 保存应用程序内部状态的数据结构

Store: 数据使用者和状态之间的桥梁

Actions: 触发状态变化的机制

Reducers: 一种通过写入状态来执行状态更改的机制

Meta-reducers: a mechanism to hook into the action -> reducer pipelineWe'll start with actions.

Actions

Actions可以被认为是reducers的指令，他们也代表了一个effect的基础。它们通常从视图层(智能组件、服务等)或effect中分派。

**Creating actions**

动作可以通过3种方式创建:

const action = createAction('[Entity] simple action');

action();

const action = createAction('[Entity] simple action', props<{ name: string, age: number, }>());

action({ name: 'andrei', age: 18 });

const action = createAction('action',(u: User, prefix: string) => ({ name: `${prefix}${u.name}` }) );

const u: User = { /\* ... \*/ };

action(u, '@@@@');

在每种情况下，函数的返回值都将是一个至少包含{type: T}属性的对象。此外，类型(createAction的第一个参数)将作为属性附加到函数。知道何时创建reducer是很有用的。

function defineType<T extends string>(

type: T,

creator: Creator

): ActionCreator<T> {

return Object.defineProperty(creator, 'type', {

value: type,

writable: false,

});

}

TypeScript’s 魅力

现在我们来看看TypeScript在这里扮演了什么重要的角色。有没有想过，例如，为什么props()函数是有用的?让我们来看看!

createAction函数有3个重载:

export declare interface TypedAction<T extends string> extends Action {

readonly type: T;

}

export type ActionCreator<

T extends string = string,

C extends Creator = Creator

> = C & TypedAction<T>;

export function createAction<T extends string>(

type: T

): ActionCreator<T, () => TypedAction<T>>;

export function createAction<T extends string, P extends object>(

type: T,

config: Props<P> & NotAllowedCheck<P>

): ActionCreator<T, (props: P & NotAllowedCheck<P>) => P & TypedAction<T>>;

export function createAction<

T extends string,

P extends any[],

R extends object

>(

type: T,

creator: Creator<P, R> & NotAllowedCheck<R>

): FunctionWithParametersType<P, R & TypedAction<T>> & TypedAction<T>;

这意味着函数体必须包含一些类型保护以获得正确的类型。

ActionCreator<T, C>表示一个类型为C的函数，它的只读属性类型为T。该类型也可以用于区别联合（unions）。

让我们检查一下每个重载。

1 - 只带类型参数的createAction

const action = createAction('[Entity] simple action');

action(); // { type: [Entity] simple action }

对应于这个重载:

export function createAction<T extends string>(

type: T

): ActionCreator<T, () => TypedAction<T>>;

我们可以从上面的代码片段推断，返回类型将是一个函数，该函数将返回一个具有属性类型的对象。

下面是类型守卫，它揭示了:

export function createAction<T extends string, C extends Creator>(

type: T,

config?: { \_as: 'props' } | C

): ActionCreator<T> {

const as = config ? config.\_as : 'empty';

switch (as) {

case 'empty':

return defineType(type, () => ({ type }));

}

}

其中defineType将属性类型附加到函数(在本例中 ()=> ({type}) )。

2 - createAction with props

当你想要分派一个动作，其中包含一些对reducer有价值的数据时，可以使用这个方法。例如 userActions。add({name, age}))。

const action = createAction('[Entity] simple action', props<{ name: string, age: number, }>());

action({ name: 'andrei', age: 18 });

props<T>()所做的是返回一个对象，该对象具有一个预定义的键(\_as: 'props')和一个类型为T的键，这对于类型推断很有用。

export function props<P extends object>(): Props<P> {

return { \_as: 'props', \_p: undefined! };

}

export interface Props<T> {

\_as: 'props';

\_p: T;

}

重载是这样的:

export function createAction<T extends string, P extends object>(

type: T,

config: Props<P> & NotAllowedCheck<P>

): ActionCreator<T, (props: P & NotAllowedCheck<P>) => P & TypedAction<T>>;

config将是props<P>()的一个实例，它允许P被推断，并在(props: P & NotAllowedCheck) => P & TypedAction<T>>中使用。

ActionCreator<T, (props: P & NotAllowedCheck<P>) => P & TypedAction<T>> 是一个可以带一个参数(对象)调用的函数，它的类型是P(从props<P>()推断出)，它的返回类型是一个对象，该对象包含P(P是一个对象)的所有属性和类型属性(TypedAction<T>)

下面是createAction如何建立这个:

export function createAction<T extends string, C extends Creator>(

type: T,

config?: { \_as: 'props' } | C

): ActionCreator<T> {

if (typeof config === 'function') {

/\* ... \*/

// `config.\_as` - returned from `props()`

const as = config ? config.\_as : 'empty';

switch (as) {

case 'props':

return defineType(type, (props: object) => ({

...props,

type,

}));

}

}

3 - 使用函数创建action

当您想要在数据到达reducer之前修改数据时，这就派上了用场。或者，您可能只是想根据一些更复杂的逻辑来确定操作的数据。

const action = createAction(

'action',

(u: User, prefix: string) => ({ name: `${prefix}${u.name}` })

);

const u: User = { /\* ... \*/ };

action(u, '@@@@');

它的重载如下所示:

export function createAction<

T extends string,

P extends any[],

R extends object

>(

type: T,

creator: Creator<P, R> & NotAllowedCheck<R>

): FunctionWithParametersType<P, R & TypedAction<T>> & TypedAction<T>;

Creator<P, R>只是一个函数，它占用一个P类型的形参，并返回一个R类型的对象。这将允许我们推断P和R类型。NotAllowedCheck<R>确保创建者不是一个存在的操作或数组。它必须是一个接收一些参数的函数，并基于这些参数返回一个表示操作数据的对象。

函数参数类型<P, R & TypedAction<T>> & TypedAction<T>;意味着返回类型必须是一个参数类型为P(从Creator<P, R>推断)的函数，它返回类型为R(也从Creator<P, R>推断)的对象，并具有属性类型。

这是createAction内部发生的事情:

export function createAction<T extends string, C extends Creator>(

type: T,

config?: { \_as: 'props' } | C

): ActionCreator<T> {

if (typeof config === 'function') {

return defineType(type, (...args: any[]) => ({

// `config(...args)` will return an object

...config(...args),

type, // The `type` property is always returned

}));

}

}

Reducers

Reducers是负责状态变化的纯函数。下面是描述reducer形状的界面:

export interface ActionReducer<T, V extends Action = Action> {

(state: T | undefined, action: V): T;

}

如您所见，reducer有两个参数:当前状态和已调度的当前操作。

提供reducers

reducer有两种提供方式:

1 - 在createReducer的帮助下创建的值为reducers的对象

StoreModule.forRoot({ foo: fooReducer, user: UserReducer })

每个键表示存储的一个部分。

2 - 注射令牌

const REDUCERS\_TOKEN = new InjectionToken('REDUCERS');

@NgModule({

imports: [

StoreModule.forRoot(REDUCERS\_TOKEN)

],

providers: [

{ provide: REDUCERS\_TOKEN, useValue: { foo: fooReducer } }

],

}) /\* ... \*/

reducer是如何设置的?

让我们假设reducers是这样提供的:

StoreModule.forRoot({ entity: entityReducer })

StoreModule.forRoot将返回一个ModuleWithProviders对象，其中包括以下提供商(providers):

/\* ... \*/

{

provide: \_REDUCER\_FACTORY,

useValue: config.reducerFactory

? config.reducerFactory

: combineReducers,

},

{

provide: REDUCER\_FACTORY,

deps: [\_REDUCER\_FACTORY, \_RESOLVED\_META\_REDUCERS],

useFactory: createReducerFactory,

},

/\* ... \*/

如你所见，除非你提供一个定制的reducer工厂，否则将使用combineReducers函数代替(稍后我们会看一下)。createReducerFactory主要用于添加meta-reducers。

REDUCER\_FACTORY令牌只会被注入到ReducerManager类中:

export class ReducerManager /\* ... \*/ {

constructor(

@Inject(INITIAL\_STATE) private initialState: any,

@Inject(INITIAL\_REDUCERS) private reducers: ActionReducerMap<any, any>,

@Inject(REDUCER\_FACTORY)

private reducerFactory: ActionReducerFactory<any, any>

) {

super(reducerFactory(reducers, initialState));

}

/\* ... \*/

}

一旦发生这种情况，createReducerFactory函数将被调用，这意味着reducerFactory属性将保存其返回值，该返回值是一个接受reducers对象(下面的reducers参数)的函数，还可以选择接受initialState:

export function createReducerFactory<T, V extends Action = Action>(

reducerFactory: ActionReducerFactory<T, V>,

metaReducers?: MetaReducer<T, V>[]

): ActionReducerFactory<T, V> {

if (Array.isArray(metaReducers) && metaReducers.length > 0) {

(reducerFactory as any) = compose.apply(null, [

...metaReducers,

reducerFactory,

]);

}

// `ReducerManager.reducerFactory` will hold this function! - it is immediately invoked in the constructor

return (reducers: ActionReducerMap<T, V>, initialState?: InitialState<T>) => {

const reducer = reducerFactory(reducers);

return (state: T | undefined, action: V) => {

// This function is the value resulted from `super(reducerFactory(reducers, initialState));`(takes place inside `ReducerManager`'s constructor)

state = state === undefined ? (initialState as T) : state;

return reducer(state, action);

};

};

}

调用super(reducerFactory(reducers, initialState))将所有的reducers组合成一个对象，其key表示存储的切片key对应一个reducer:

export function combineReducers(

reducers: any,

initialState: any = {}

): ActionReducer<any, Action> {

const reducerKeys = Object.keys(reducers);

const finalReducers: any = {};

for (let i = 0; i < reducerKeys.length; i++) {

const key = reducerKeys[i];

if (typeof reducers[key] === 'function') {

finalReducers[key] = reducers[key];

}

}

/\*

Remember from the previous snippet: `const reducer = reducerFactory(reducers)`

Now, the `reducer` will be the below function.

\*/

return function combination(state, action) {

state = state === undefined ? initialState : state;

let hasChanged = false;

const nextState: any = {};

for (let i = 0; i < finalReducerKeys.length; i++) {

const key = finalReducerKeys[i];

const reducer: any = finalReducers[key];

const previousStateForKey = state[key];

const nextStateForKey = reducer(previousStateForKey, action);

nextState[key] = nextStateForKey;

hasChanged = hasChanged || nextStateForKey !== previousStateForKey;

}

return hasChanged ? nextState : state;

};

}

此外，在上面的代码片段中，我们可以看到为什么存储的数据必须是不可变的。如果一个reducer返回一个对象的相同引用，但是属性改变了，这不会反映到UI中，因为nextStateForKey !== previousStateForKey会失败。

要点如下:

/\* ... \*/

//下面的函数是

// `@Inject(REDUCER\_FACTORY) private reducerFactory: ActionReducerFactory<any, any>`

return (reducers: ActionReducerMap<T, V>, initialState?: InitialState<T>) => { // #Fn1

const reducer = reducerFactory(reducers); // <-

return (state: T | undefined, action: V) => { // #Fn2

state = state === undefined ? (initialState as T) : state;

// `reducer` = `combination` 函数;调用时，将遍历现有的reducer  
//使用当前的state和action调用它们

return reducer(state, action);

};

};

super(reducerFactory(reducers, initialState))将导致上面的reducerFactory被调用，这将最终合并所有的reducers。

在const reducer = reducerFactory(reducers)中调用reducerFactory后，reducers将代表组合函数进行操作。当被调用时，它将遍历reducer并使用给定的状态和动作调用它们。

调用reducer的函数将在每次分派一个动作时被调用，这意味着reducer将被组合(在Fn1调用时)一次。当然，如果以后添加/删除了其他reducer, reducer对象将被正确地重新创建(Fn1再次被调用)。

createReducer helper

了解@ ngrx / effects背后的魔力

https://indepth.dev/posts/1206/understanding-the-magic-behind-ngrx-effects