了解@ ngrx / effects背后的魔力

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

NgRx的底层机制绝对值得探索。希望在本文结束时，你不仅能理解ngrx/effects包是如何工作的，还能理解ngrx/store和ngrx/effects是如何一起工作的。

其中最令人着迷的发现之一是在整个过程中行动的处理方式。如你所知，一个action既是一个effects的组成部分，也是一个reducer的组成部分。NgRx确保action首先由reducer处理，之后它们最终会被effects拦截。我们将在本文中进行其他有趣的发现

**[提供 effects](https://indepth.dev/understanding-the-magic-behind-ngrx-effects/" \l "providing-the-effects)**

为此，我们可以使用EffectsModule.forRoot([effectClass])、EffectsModule.forFeature([effectClass])或USER\_PROVIDED\_EFFECTS多令牌。forRoot方法应该只使用一次，因为它将实例化其他基本服务，如EffectsRunner或EffectSources。

如果使用令牌方法，还必须使用forRoot或forFeature，因为只提供令牌是不够的，因为它依赖于EffectsRootModule或EffectsFeatureModule。

一旦这些effects(类)被注册，为了建立它们，一个可观察对象就会被创建(在EffectSources的帮助下)并订阅(多亏了EffectRunner);我们将在下一节中探讨它。

这个可观察对象发出的值将是那些已注册类的实例:

// EffectsModule.forRoot(rootEffects)

{

return {

ngModule: EffectsRootModule,

providers: [

{

//确保' forRoot '静态方法只被调用一次

provide: \_ROOT\_EFFECTS\_GUARD,

useFactory: \_provideForRootGuard,

deps: [[EffectsRunner, new Optional(), new SkipSelf()]],

},

EffectsRunner,

EffectSources,

Actions,

rootEffects, // 类的数组(effects)

{

// 依赖“ROOT\_EFFECTS”

provide: \_ROOT\_EFFECTS,

// 因为' createEffects '是如何实现的，所以将它作为一个数组来提供

useValue: [rootEffects],

},

{

// 该令牌将由用户在其单独的模块中提供

provide: USER\_PROVIDED\_EFFECTS,

multi: true,

useValue: [], // [UserProvidedEffectsClass]

},

{

provide: ROOT\_EFFECTS,

useFactory: createEffects,

deps: [Injector, \_ROOT\_EFFECTS, USER\_PROVIDED\_EFFECTS],

},

],

};

}

export function createEffects(

injector: Injector,

effectGroups: Type<any>[][],

userProvidedEffectGroups: Type<any>[][]

): any[] {

const mergedEffects: Type<any>[] = [];

for (let effectGroup of effectGroups) {

mergedEffects.push(...effectGroup);

}

for (let userProvidedEffectGroup of userProvidedEffectGroups) {

mergedEffects.push(...userProvidedEffectGroup);

}

return createEffectInstances(injector, mergedEffects);

}

// 这里创建了实例

export function createEffectInstances(/\* ... \*/): any[] {

return effects.map(effect => injector.get(effect));

}

在这种情况下，EffectsRootModule将注入包含所需实例的ROOT\_EFFECTS，并将它们推入effects流:

@NgModule({})

export class EffectsRootModule {

constructor(

private sources: EffectSources,

runner: EffectsRunner,

store: Store<any>,

@Inject(ROOT\_EFFECTS) rootEffects: any[],

/\* ... \*/

) {

// 订阅 `effects stream`

// The `observer` is the Store entity

runner.start();

rootEffects.forEach(effectSourceInstance =>

// Push values into the stream

sources.addEffects(effectSourceInstance)

);

store.dispatch({ type: ROOT\_EFFECTS\_INIT });

}

addEffects(effectSourceInstance: any) {

this.sources.addEffects(effectSourceInstance);

}

}

作为附注，当根effects初始化时，你可以通过在reducer中注册rootEffectsInitaction来执行特定的状态改变:

createReducer(

initialState,

on(rootEffectsInit, (s, a) = {/\* ... \*/})

)

EffectsFeatureModule(从EffectsModule.forFeature()返回)遵循类似的方法，除了它只会将effects实例推入流:

@NgModule({})

export class EffectsFeatureModule {

constructor(

// 确保基本服务(EffectsRunner, EffectSources)

// 第一次初始化

root: EffectsRootModule,

@Inject(FEATURE\_EFFECTS) effectSourceGroups: any[][],

/\* ... \*/

) {

effectSourceGroups.forEach(group =>

group.forEach(effectSourceInstance =>

root.addEffects(effectSourceInstance)

)

);

}

}

还有一个小小的区别。FEATURE\_EFFECTS是一个多提供者令牌，这意味着，当注入时，它将包含所有提供的类的数组。

**[The effects stream](https://indepth.dev/understanding-the-magic-behind-ngrx-effects/" \l "the-effects-stream)**

如前一节所述，EffectsRootModule所做的不仅仅是实例化effects类。它还负责从新创建实例的所有effects(例如createEffect的结果)中创建一个单独的流，并订阅它

@NgModule({})

export class EffectsRootModule {

constructor(

private sources: EffectSources,

runner: EffectsRunner,

store: Store<any>,

@Inject(ROOT\_EFFECTS) rootEffects: any[],

@Optional() storeRootModule: StoreRootModule,

@Optional() storeFeatureModule: StoreFeatureModule,

@Optional()

@Inject(\_ROOT\_EFFECTS\_GUARD)

guard: any

) {

runner.start(); // Creating the stream

rootEffects.forEach(effectSourceInstance =>

sources.addEffects(effectSourceInstance)

);

store.dispatch({ type: ROOT\_EFFECTS\_INIT });

}

addEffects(effectSourceInstance: any) {

// Pushing values into the stream

this.sources.addEffects(effectSourceInstance);

}

}

@Optional() storeerootmodule: storeerootmodule和@Optional() storeFeatureModule: storeFeatureModule将确保在ngrx/store实体(StoreModule.forRoot()的结果和StoreModule.forFeature()的结果)被初始化之后，effects被初始化。这种预先初始化包括:

1 - reducers对象的创建:所有注册的reducers，包括来自功能模块的，将合并成一个大对象，代表应用程序的形状

2 - 状态实体——保存应用信息的地方，也是place操作满足reducers的地方，这意味着它是reducers被调用的地方，这可能会导致状态改变

3 - store实体——数据消费者(e。g:智能组件)和模型(State实体)

4 - ScannedActionsSubject——effects(间接)订阅的流;更多相关信息请参见actions流。

runner.start()将创建一个订阅到所有注册effects合并后产生的流的订阅(下一节将详细介绍)。换句话说，所有的影响(e。g:那些由createEffect创建的)会被合并到一个单独的observable中，它发出的值就是actions。

// EffectsRunner

start() {

if (!this.effectsSubscription) {

this.effectsSubscription = this.effectSources

.toActions()

.subscribe(this.store);

}

}

流的观察者将是store实体。这是可能的，因为它实现了Observer接口:

export class Store<T = object> extends Observable<T>

implements Observer<Action> {

next(action: Action) {

this.actionsObserver.next(action);

}

}

也就是说，任何由这些效应产生的行为都会被store所拦截，store会进一步传播这些效应，从而使状态发生变化。

**[EffectSource](https://indepth.dev/understanding-the-magic-behind-ngrx-effects/" \l "effectsources)s**

它是所有注册的effects将被合并到一个可观察对象中的地方，它发出的值(action)将被负责分派它们的Store实体截获，这样app的状态就可以被更新。它也是允许调用生命周期钩子的实体。

合并行为可以通过下面的EffectSources.toActions()来实现:

export class EffectSources extends Subject<any> {

constructor(

/\* ... \*/

private store: Store<any>,

/\* ... \*/

) {

super();

}

// Pushing an effect into the stream created by `toActions()`

addEffects(effectSourceInstance: any): void {

this.next(effectSourceInstance);

}

toActions(): Observable<Action> {

return this.pipe(

groupBy(getSourceForInstance),

mergeMap(source$ => {

return source$.pipe(groupBy(effectsInstance));

}),

mergeMap(source$ => {

const effect$ = source$.pipe(

exhaustMap(sourceInstance => {

return resolveEffectSource(

this.errorHandler,

this.effectsErrorHandler

)(sourceInstance);

}),

map(output => {

reportInvalidActions(output, this.errorHandler);

return output.notification;

}),

filter(

(notification): notification is Notification<Action> =>

notification.kind === 'N'

),

dematerialize()

);

// start the stream with an INIT action

// do this only for the first Effect instance

const init$ = source$.pipe(

take(1),

filter(isOnInitEffects),

map(instance => instance.ngrxOnInitEffects())

);

return merge(effect$, init$);

})

);

}

}

让我们来看看它通过每个有效的块到底做了什么:

1 - 根据它们的源(实例的原型，创建实例的类)将effects分组如你所知，groupBy会为找到的每个新键返回一个可观察对象。然后，如果一个关键字不是new的值到达，groupBy会使用一个已经存在的可观察对象，并将这个新值推送给它。这个事情发生在这里(groupBy(getSourceForInstance))，键是创建当前实例的类，这意味着如果有3个不同的effects类，groupBy将发出3个可观察对象。

2 - ：根据实例(effects)的标识符分组

mergeMap(source$ => { // <- `source$` an observable that is resulted from the previous `groupBy`

return source$.pipe(groupBy(effectsInstance));

}),

之所以使用mergeMap，是因为第一个groupBy可能会发出多个可观察对象，而预期的行为是处理所有这些对象。现在，如果同一个effect类被加载多次，则只会使用一个实例，因为这些类默认具有相同的标识符(每个标识符只能有一个类):

function effectsInstance(sourceInstance: any) {

if (isOnIdentifyEffects(sourceInstance)) {

return sourceInstance.ngrxOnIdentifyEffects();

}

return '';

}

使用ngrxOnIdentifyEffects(OnIdentityEffects接口要求的)，我们可以为包含这些effects的类指定一个唯一标识符。

这与groupBy(effectsInstance)以及最终的routmap一起，将确保只使用第一个惟一实例。

例如，如果我们有EffectsModule。([A, A, A])，它们具有相同的标识符(e。g: "默认情况下)，第二个groupBy将只发出一个包含3个对象的observable。

With the help of exhaustMap

mergeMap(source$ => {

const effect$ = source$.pipe(

exhaustMap(sourceInstance => { /\* ... \*/ }),

)

/\* ... \*/

}

)

3个项目中只有一个项目(在这种情况下是第一个A)将被考虑。

在内部，plustmap使用一个标志(hasSubscription)来检查是否有一个活动的内部订阅正在进行。您可以在这里查看相关代码。

合并所有的effects到一个单一的流

// `mergeMap` - for each emitted observable(emitted by the second `groupBy`)

// perform the same logic: merge all the effects(class properties)

// into one single observable

mergeMap(source$ => {

const effect$ = source$.pipe(

exhaustMap(sourceInstance => {

return resolveEffectSource(

this.errorHandler,

this.effectsErrorHandler

)(sourceInstance);

}),

/\* ... \*/

)

resolveEffectSource会合并当前实例的所有现有属性(它们是可观察的，可能是由createEffect()创建的)，并最终调用ngrxOnRunEffectslifecycle方法(OnRunEffects接口需要):

function resolveEffectSource(/\* ... \*/): (sourceInstance: any) => Observable<EffectNotification> {

return sourceInstance => {

const mergedEffects$ = mergeEffects(

sourceInstance,

errorHandler,

effectsErrorHandler

);

if (isOnRunEffects(sourceInstance)) {

return sourceInstance.ngrxOnRunEffects(mergedEffects$);

}

return mergedEffects$;

};

}

通过ngrxOnRunEffects，我们可以改变合并所有effects(实例的属性)后产生的可观察对象。

export function mergeEffects(

sourceInstance: any,

globalErrorHandler: ErrorHandler,

effectsErrorHandler: EffectsErrorHandler

): Observable<EffectNotification> {

const sourceName = getSourceForInstance(sourceInstance).constructor.name;

// `getSourceMetadata(sourceInstance)` - getting all the effect class' properties

const observables$: Observable<any>[] = getSourceMetadata(sourceInstance).map(

({

propertyName,

dispatch,

useEffectsErrorHandler,

}): Observable<EffectNotification> => {

const observable$: Observable<any> =

typeof sourceInstance[propertyName] === 'function'

? sourceInstance[propertyName]()

: sourceInstance[propertyName];

// Whether it should re-subscribe if errors occur

const effectAction$ = useEffectsErrorHandler

? effectsErrorHandler(observable$, globalErrorHandler)

: observable$;

// You might not want the `Store` to intercept the action

// and trigger state changes based on it

if (dispatch === false) {

return effectAction$.pipe(ignoreElements());

}

const materialized$ = effectAction$.pipe(materialize());

return materialized$.pipe(

map(/\* ... \*/)

);

}

);

return merge(...observables$);

}

：在这里可以找到一个重现合并操作的较小示例。

const materialized$ = effectAction$.pipe(materialize())所做的是确保如果不对错误进行重新订阅，它将抑制任何传入的错误，这意味着所有合并effects的流不会被破坏。

在这个阶段，当所有的effect类的属性合并到一个observable中之后，ngrxOnInitEffects生命周期方法(OnInitEffects接口所需要的)将会被每个类调用(如果它存在的话)，这样一个action就会立即被分派:

mergeMap(source$ => {

// Merged effects

const effect$ = source$.pipe(

exhaustMap(/\* ... \*/),

/\* ... \*/

);

// `source$`'s value is an effect class instance

const init$ = source$.pipe(

// `take(1)` -> make sure the `exhaustMap`'s behavior is `replicated`

// as there is only one effect class per identifier

take(1),

filter(isOnInitEffects),

// `instance.ngrxOnInitEffects()` -> Action

map(instance => instance.ngrxOnInitEffects())

);

return merge(effect$, init$);

})

上述过程可以直观地理解为:

有动画

**[Creating Effects](https://indepth.dev/understanding-the-magic-behind-ngrx-effects/" \l "creating-effects)**

创建一个effects可以通过createEffect()函数实现:

type DispatchType<T> = T extends { dispatch: infer U } ? U : true;

type ObservableType<T, OriginalType> = T extends false ? OriginalType : Action;

export function createEffect<

C extends EffectConfig,

DT extends DispatchType<C>,

OT extends ObservableType<DT, OT>,

R extends Observable<OT> | ((...args: any[]) => Observable<OT>)

>(source: () => R, config?: Partial<C>): R & CreateEffectMetadata {

const effect = source();

const value: EffectConfig = {

...DEFAULT\_EFFECT\_CONFIG,

...config, // Overrides any defaults if values are provided

};

Object.defineProperty(effect, CREATE\_EFFECT\_METADATA\_KEY, {

value,

});

return effect as typeof effect & CreateEffectMetadata;

}

createEffect()将返回一个带有属性CREATE\_EFFECT\_METADATA\_KEY的可观察对象，该属性将保存该特定effects的配置对象。当将effects类的属性合并到一个可观察对象时，每个可观察对象(这个effect类的属性)都会根据配置稍有改变。该对象有两个属性:

1 - dispatch: boolean—结果操作是否应该被分派到store;

const observable$: Observable<any> =

typeof sourceInstance[propertyName] === 'function'

? sourceInstance[propertyName]()

: sourceInstance[propertyName];

const effectAction$ = useEffectsErrorHandler

? effectsErrorHandler(observable$, globalErrorHandler)

: observable$;

if (dispatch === false) {

return effectAction$.pipe(ignoreElements());

}

ignoreElements操作符将忽略所有内容，除了错误或完整通知。

useEffectsErrorHandler: boolean - effects是否错误(e。g:由于外部API调用)应该被处理或不处理

const observable$: Observable<any> =

typeof sourceInstance[propertyName] === 'function'

? sourceInstance[propertyName]()

: sourceInstance[propertyName];

const effectAction$ = useEffectsErrorHandler

? effectsErrorHandler(observable$, globalErrorHandler)

: observable$;

effectsErrorHandler映射到EFFECTS\_ERROR\_HANDLER提供的值。默认情况下，它映射到defaultEffectsErrorHandler(特效的内置错误处理程序):

export function defaultEffectsErrorHandler<T extends Action>(

observable$: Observable<T>,

errorHandler: ErrorHandler,

retryAttemptLeft: number = MAX\_NUMBER\_OF\_RETRY\_ATTEMPTS

): Observable<T> {

return observable$.pipe(

catchError(error => {

if (errorHandler) errorHandler.handleError(error);

if (retryAttemptLeft <= 1) {

return observable$; // last attempt

}

// Return observable that produces this particular effect

return defaultEffectsErrorHandler(

observable$,

errorHandler,

retryAttemptLeft - 1

);

})

);

}

如果您想知道为什么重试次数必须有一个限制，这里是这个主题解决的问题。

当错误发生时，这个可观察对象将被取消订阅。defaultEffectsErrorHandler所做的是允许我们重新订阅just-unsubscribedobservable，只要不超过允许的最大尝试次数。

例如，如果你有这样的effects:

addUser$ = createEffect(

() => this.actions$.pipe(

ofType(UserAction.add),

exhaustMap(u => this.userService.add(u)),

map(/\* Map to action \*/)

),

)

如果由于调用userService.add()而发生错误，并且在任何地方都没有处理，例如:

// `this.userService.add(u)` is a cold observable

exhaustMap(

u => this.userService.add(u).pipe(catchError(err => /\* Action \*/))

),

addUser$将从action $ stream取消订阅。defaultEffectsErrorHandler将简单地重新订阅actions$，但是还有另一件事值得一提:actions$ stream实际上是一个主题，因此我们可以确定，当重新订阅时，我们不会接收到任何先前发出的值，而只接收到更新的值。

你也可以为effects提供自定义错误处理程序:

{

provide: EFFECTS\_ERROR\_HANDLER,

useValue: customErrHandler,

},

function customErrHandler (obs$, handler) {

return obs$.pipe(

catchError((err, caught$) => {

console.log('caught!')

// Only re-subscribe once

// return obs$;

// Re-subscribe every time an error occurs

return caught$;

}),

)

}

这里customErrHandler应该是一个函数，它接受一个可观察对象$(建立在action$ observable之上的可观察对象)和一个errHandler对象。

TypeScript's Magic

现在我们来看看TypeScript在使用createEffect创建effects时所扮演的重要角色。

考虑一下这个effects:

addUser$ = createEffect(

() => this.actions$.pipe(/\* ... \*/),

)

addUser$的类型应该是:可观察对象<Action> & CreateEffectMetadata。CreateEffectMetadata是一种标识createEffect()创建的属性的方法。当属性合并到一个单一的可观察对象中时，它特别有用。

在揭示为什么存在可观察对象<Action>之前，让我们尝试写同样的effects，但这次在配置对象中指定dispatch: false:

addUser$ = createEffect(

() => of(1),

{ dispatch: false }

)

addUser$的类型将是Observable<number> & CreateEffectMetadata。

但是如果我们有:

addUser$ = createEffect(

() => of(1),

// { dispatch: false }

)

我们会得到:Type 'number'是不能分配给Type 'Action'的。这意味着分派属性对effects的类型有影响。

让我们看看这是如何实现的:

// U默认为' undefined '

type DispatchType<T> = T extends { dispatch: infer U } ? U : true;

// ' OriginalType '只在' dispatch '被显式设置为' false '时使用

type ObservableType<T, OriginalType> = T extends false ? OriginalType : Action;

export function createEffect<

C extends EffectConfig, // { dispatch?: boolean, useEffectsErrorHandler?: boolean; }

DT extends DispatchType<C>, // U(undefined | boolean) || true

OT extends ObservableType<DT, OT>, //如果' DT '为false(' dispatch '显式设置为' false ')，则使用原始类型

R extends Observable<OT> | ((...args: any[]) => Observable<OT>) // 使用' OT '来推断可观察对象的类型

>(source: () => R, config?: Partial<C>) { }

记住这一点，在这个代码片段中

addUser$ = createEffect(

() => of(1),

// { dispatch: false }

)

我们有(从下面开始):

R - Observable<number>

OT - initially的类型是number

DT - false，因为dispatch被显式设置为false

OT extends ObservableType<DT, OT> 决定了OT的最终类型: false extends false ? number : Action -> 我们得到数量

相反，在这个片段中

addUser$ = createEffect(

() => of(1),

// { dispatch: false }

)

我们有 OT extends ObservableType<DT, OT>，它可以被视为 undefined extends false ? number : Action。所以，OT的类型将是Action，我们会得到错误，因为addUser$的类型是Observable<number>，而它本应该是Observable<Action>。

**[The actions stream](https://indepth.dev/understanding-the-magic-behind-ngrx-effects/" \l "the-actions-stream)**

在本文中，您可能多次看到actions$。在本节中，我们将了解它是什么，它做什么，以及TypeScript如何确保无缝的开发体验。

通常情况下，你可以这样将inject actions$注入到一个effects类中:

constructor(private actions$: Actions) {}

<>

export interface Action {

type: string;

}

@Injectable()

export class Actions<V = Action> extends Observable<V> {

constructor(@Inject(ScannedActionsSubject) source?: Observable<V>) {

super();

if (source) {

this.source = source;

}

}

lift<R>(operator: Operator<V, R>): Observable<R> {

const observable = new Actions<R>();

observable.source = this;

observable.operator = operator;

return observable;

}

}

ScannedActionsSubject来自@ngrx/store，它是一个Subject(也就是一个可观察对象)，它会在action被分派时发出，但只有在状态变化被处理后才会发出。因此，当一个操作被分派时(Store.dispatch())，状态实体将首先根据该操作和当前状态在reducers的帮助下更新应用程序状态，然后它将该操作推入ScannedActionsSubject创建的操作流。

通过将action的源设置为ScannedActionsSubject，每次我们有类似这样的东西时，观察者就会成为ScannedActionsSubject的观察者列表的一部分，这意味着当subject发出一个action(e. pipe().subscribe(observer))。g:主题。下一个(action))，所有注册的观察员都会收到。这应该可以解释为什么所有的effects都会收到相同的操作，但是在ofType的帮助下，这些操作可以被过滤掉。

以下是ScannedActionsSubject通知其活动观察者的方式:

// State

/\* ... \*/

const stateAndAction$: Observable<{

state: any;

action?: Action;

}> = withLatestReducer$.pipe(

scan<[Action, ActionReducer<T, Action>], StateActionPair<T>>(

reduceState, // Handling state changes

seed

)

);

this.stateSubscription = stateAndAction$.subscribe(({ state, action }) => {

this.next(state); // `state` -> the new state, after reducers have been invoked

scannedActions.next(action);

});

ofType

为了确定哪些action应该触发哪些effects，使用了自定义操作符ofType:

export function ofType(

...allowedTypes: Array<string | ActionCreator<string, Creator>>

): OperatorFunction<Action, Action> {

return filter((action: Action) =>

allowedTypes.some(typeOrActionCreator => {

if (typeof typeOrActionCreator === 'string') {

// Comparing the string to type

return typeOrActionCreator === action.type;

}

// We are filtering by ActionCreator

return typeOrActionCreator.type === action.type;

})

);

}

如您所见，它在内部使用RxJs过滤操作符，其谓词函数的返回值取决于当前发出的操作是否在提供给oftype的值中。

这里真正吸引人的是TypeScript的力量是如何被利用的。

当涉及到ofType的类型推断时，有两种可能:

1 - 你可以提供createAction()创建的action，它符合ActionCreator类型;

export type ActionCreator<

T extends string = string,

C extends Creator = Creator // `Creator` -> a function that returns an object

> = C & TypedAction<T>; // A function that has a readonly property `type`, which also returns an object

通过使用ofType(action1, action2，…)，它的返回类型将是由action1, action2和…的返回类型组成的联合。actionN:

export function ofType<

AC extends ActionCreator<string, Creator>[],

U extends Action = Action, // A created action

V = ReturnType<AC[number]>

// `U` - the type of the incoming observable

// `V` - the type of the returned observable

>(...allowedTypes: AC): OperatorFunction<U, V>;

我们特别感兴趣的是V = ReturnType<AC[number]>部分。AC是一个ActionCreator(createAction的结果)的数组。

AC[number]将返回AC所有元素的并集。例如:

type Action<T extends string = string> = { readonly type: T; }

function createAction<P extends object, T extends string>(t: T, payload: P): P & Action<T> {

return {

...payload,

type: t,

};

}

const actions = [createAction('type1', { name: 'andrei' }), createAction('type2', { age: 123 })];

// `(typeof actions)[number]` -> a union of types

const action: (typeof actions)[number] = {

// We can discriminate unions with the help of the `type` property

// because `createAction` returns an object with one `readonly` property,

// namely `type`

type: 'type2',

age: 123,

// name: 'John' -> ? error

}

同样，由AC[number]产生的联合可以通过type属性来区分。

接下来，我们有ReturnType<Union>。这与ReturnType<Union\_M1 | Union\_M2 |…>(Union\_Mn表示union的n-thmember)。它所做的是确定每个操作的返回类型。这个联合将是ofType返回的可观察对象的类型。

2 - 你可以提供表示实际action类型的字符串;

然而，由于ofType得到的只是一个字符串列表，为了推断正确的类型，必须手动指定一个类型的联合，期望与提供的操作类型匹配。

让我们来看看Observable.pipe的重载:

export class Observable<T> implements Subscribable<T> {

/\* ... \*/

pipe<A, B>(op1: OperatorFunction<T, A>, op2: OperatorFunction<A, B>): Observable<B>;

/\* ... \*/

}

其中OperatorFunction<T， >指定了一个函数的类型，它接收一个observable作为参数，并返回另一个observable:

export interface UnaryFunction<T, R> { (source: T): R; }

export interface OperatorFunction<T, R> extends UnaryFunction<Observable<T>, Observable<R>> {}

因此，从上面的代码片段中我们可以注意到，管道函数中的第一个操作符将是一个函数，它的单个参数的类型将是一个类型为T的observable(其中T是observable的类型参数)。

考虑到这一点，让我们来看看Actions类，它提供了一系列action，这些action取决于effects的作用:

@Injectable()

export class Actions<V = Action> extends Observable<V> {

constructor(@Inject(ScannedActionsSubject) source?: Observable<V>) { }

}

有趣，所以Actions<V = Action>也是一个Observable<V>，这意味着管道中第一个操作符(函数)的形参类型将是V类型。

让我们再看看其他ofType的重载:

export function ofType<

E extends Extract<U, { type: T1 }>,

AC extends ActionCreator<string, Creator>,

T1 extends string | AC,

U extends Action = Action,

V = T1 extends string ? E : ReturnType<Extract<T1, AC>>

>(t1: T1): OperatorFunction<U, V>;

请暂时忽略<>之间的内容，以便注意ofType的返回类型，以及Observable.pipe中第一个操作符的类型:

ofType(): OperatorFunction<U, V> <---> pipe<A>(op1: OperatorFunction<T, A>)

从这里我们可以推断出，ofType的U类型参数将是T，即V(从Actions<V extends Action> extends Observable<V>)。

这就是为什么当你在effects类中注入actions observable时，你必须提供一个actions的联合，否则就不可能推断出这个effects感兴趣的action的返回类型。通过提供这个联合，我们现在可以让TypeScript发挥它的作用了。

export function ofType<

E extends Extract<U, { type: T1 }>,

AC extends ActionCreator<string, Creator>,

T1 extends string | AC,

U extends Action = Action,

V = T1 extends string ? E : ReturnType<Extract<T1, AC>>

>(t1: T1): OperatorFunction<U, V>;

因此，我们已经确定U将是V类型(action的V <V =action>)，当注入时，它将等于提供的action的并集。E将是抽取的action，基于单例类型(type属性)我们知道，action(由createAction创建)是一个函数，它有一个称为type的只读属性.这将允许我们推断真正的action(它是被注入的action<V>的联合V的一部分)，因为每个action都扩展了<{type: aSingletonType}>。

最后，返回类型将是V(V of ofType)，其值基于二进制判定:

E(推断创建的action)，因为T1是一个单例类型，它允许TypeScript推断实际的action;

这里有一个模仿这种行为的例子:

// Can be thought of as actions

type A = { type: 'andrei' };

type J = { type: 'john' };

type JA = { type: 'jane' };

// E extends Extract<U, { type: 'andrei' | 'john' | 'jane' }>,

type Names = { type: 'andrei' | 'john' | 'jane' };

// === `createAction('john', props<{ age: number }>())`

type JSub = J & { age: number };

// === `createAction('john', props<{ city: string }>())`

type ASub = A & { city: string };

type R = Extract<ASub | JSub, A | JA | J>;

type R2 = Extract<ASub | JSub, Names>;

// After choosing the value of the `type` property

// the unions will be discriminated

const o: R = { type: 'andrei', city: 'city', };

const o2: R2 = { type: 'john', age: 18 };

由于T1类型的值不是string子类型，所以它必须是一个action创建器，所以我们只想获取它的返回类型

**[连接 ngrx/store with ngrx/effects](https://indepth.dev/understanding-the-magic-behind-ngrx-effects/connecting-ngrx-store-with-ngrx-effects)**

有了这篇文章的知识，以及理解了NgRx的StoreModule (@ngrx/store)背后的神奇之处，我们现在可以看到幕后发生了什么:

Store.dispatch()

它指示需要改变状态的事件从UI(e。g智能组件)。Store.dispatch()将action(事件)推入action流(与属于effects的流不同):

// Store

dispatch(action) {

this.actionsObserver.next(action);

}

这里还值得一提的是，Store类是一个可观察对象，它的源是State, State也是一个可观察对象，更准确地说，是一个BehaviorSubject:

// Store

constructor(

state$: StateObservable, // The `State` class

private actionsObserver: ActionsSubject,

) {

super();

this.source = state$;

}

这很有用，因为来自UI层的组件可以简单地订阅Store类，以便在状态发生更改时进行更新。

1 截取State类中的操作

// State

constructor(

actions$: ActionsSubject, // Receive the actions dispatched from `Store`

reducer$: ReducerObservable,

scannedActions: ScannedActionsSubject, // The `actions stream` that belong to effects

@Inject(INITIAL\_STATE) initialState: any

) {

const actionsOnQueue$: Observable<Action> = actions$.pipe(

observeOn(queueScheduler)

);

const withLatestReducer$: Observable<

[Action, ActionReducer<any, Action>]

> = actionsOnQueue$.pipe(withLatestFrom(reducer$));

const seed: StateActionPair<T> = { state: initialState };

const stateAndAction$: Observable<{

state: any;

action?: Action;

}> = withLatestReducer$.pipe(

scan<[Action, ActionReducer<T, Action>], StateActionPair<T>>(

// a)

reduceState, // Invoke the reducers -> the result will be a new state

// =====

seed

)

);

this.stateSubscription = stateAndAction$.subscribe(({ state, action }) => {

// b)

this.next(state); // Send the new state to the data consumer(e.g: a smart component)

// =====

// c)

scannedActions.next(action); // Notify effects that an action ocurred

// =====

});

}

1 使用当前action和当前状态调用reducer，产生一个新状态

2 结束数据消费者的新状态;

记住，State是Store的来源，这意味着this.next(State);将使状态可以在Storeclass中访问，可以订阅到:Store.select()或Store.pipe(select())

3 在状态更改被处理并发送给数据消费者之后，将action发送给effects;

如果action被任何已注册的effects拦截，一个新的action就会出现，而Store()又会拦截这个action，导致步骤a)、b)、c)被重复:

this.effectSources

.toActions() // The action resulted from all the merged effects

.subscribe(this.store);

这是可能的，因为商店也可以作为订阅者:

// Store

next(action: Action) {

this.actionsObserver.next(action);

}