定义实体

一个EntityMetadataMap告诉NgRx关于实体的数据。向每个实体名称的集合中添加一个属性。

import { EntityMetadataMap } from '@ngrx/data';

const entityMetadata: EntityMetadataMap = {

Hero: {},

Villain: {}

};

// 因为"hero"的复数不是"heros"

const pluralNames = { Hero: 'Heroes' };

export const entityConfig = {

entityMetadata,

pluralNames

};

导出在AppModule中注册时要使用的实体配置。

注册实体存储

@NgModule({

imports: [

HttpClientModule,

StoreModule.forRoot({}),

EffectsModule.forRoot([]),

EntityDataModule.forRoot(entityConfig)

]

})

export class AppModule {}

创建实体数据服务

NgRx数据通过在你的服务类中扩展EntityCollectionServiceBase来处理服务器上的创建、检索、更新和删除数据。

@Injectable({ providedIn: 'root' })

export class HeroService extends EntityCollectionServiceBase<Hero> {

constructor(serviceElementsFactory: EntityCollectionServiceElementsFactory) {

super('Hero', serviceElementsFactory);

}

}

在组件中使用NgRx数据

要访问实体数据，组件应该注入实体数据服务。

@Component({

selector: 'app-heroes',

templateUrl: './heroes.component.html',

styleUrls: ['./heroes.component.scss']

})

export class HeroesComponent implements OnInit {

loading$: Observable<boolean>;

heroes$: Observable<Hero[]>;

constructor(private heroService: HeroService) {

this.heroes$ = heroService.entities$;

this.loading$ = heroService.loading$;

}

ngOnInit() {

this.getHeroes();

}

add(hero: Hero) {

this.heroService.add(hero);

}

delete(hero: Hero) {

this.heroService.delete(hero.id);

}

getHeroes() {

this.heroService.getAll();

}

update(hero: Hero) {

this.heroService.update(hero);

}

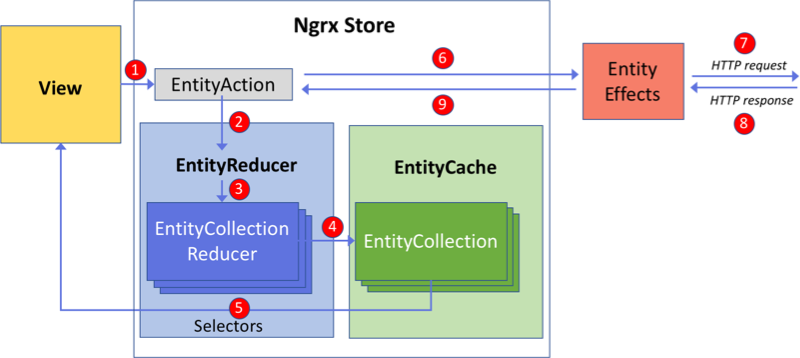
}

你用几行实体元数据向NgRx数据描述你的实体模型，然后让库来完成剩下的工作。

你的组件会注入一个NgRx Data EntityCollectionService，并调用一个或多个标准的命令方法来分派动作。

你的组件还订阅了一个或多个服务的可观察选择器，以便响应性地处理和显示由这些命令产生的实体状态变化。

NgRx数据实际上只是NgRx的外壳。数据以典型的NgRx方式流动。下面的图表演示了持久化实体操作的历程，比如Hero实体类型的QUERY\_ALL。



1 - 视图/组件调用EntityCollectionService.getAll()，它会把英雄的QUERY\_ALL实体动作分派到存储中。

2 - NgRx启动…

（1）NgRx Data EntityReducer读取动作的entityName属性(本例中是Hero)，并将动作和现有实体的集合状态转发给heroes的EntityCollectionReducer。

（2）collection reducer根据操作的entityOp (operation)属性选择一个开关箱。这个例子将动作和集合状态处理成一个新的(更新后的)英雄集合。

（3）store使用更新后的集合更新状态树中的实体缓存。

（4）NgRx可观察对象选择器会检测并向视图中的订阅者报告这些变化(如果有的话)。

3 - 原始的EntityAction然后转到entityeeffects。

4 - 效果为该实体类型选择一个EntityDataService。数据服务向服务器发送一个HTTP请求。

5 - 这个效果会把HTTP响应变成一个新的带有英雄的成功动作(如果请求失败，则会变成一个错误动作)。

6 - NgRx特效把这个动作分派给商店，这就重复了第2步，用英雄来更新集合，刷新视图。

## Entity Metadata

NgRx数据库在NgRx存储中维护了一个实体收集数据的缓存

你用实体元数据告诉NgRx数据库这些集合和它们包含的实体。

集合中的实体属于相同的实体类型。每种实体类型都以NgRx数据 EntityMetadata<T>接口的命名实例的形式出现。

您可以在EntityMetadataMap中同时为多个实体指定元数据。

下面是一个EntityMetadataMap的例子，它与演示应用中定义了Hero和Villain两个实体的元数据的例子类似。

export const appEntityMetadata: EntityMetadataMap = {

Hero: {

/\* optional settings \*/

filterFn: nameFilter,

sortComparer: sortByName

},

Villain: {

villainSelectId, // necessary if key is not `id`

/\* optional settings \*/

entityName: 'Villain', // optional because same as map key

filterFn: nameAndSayingFilter,

entityDispatcherOptions: { optimisticAdd: true, optimisticUpdate: true }

}

};

## Register metadata

你必须在NgRx数据 EntityDefinitionService中注册元数据

注册元数据最简单的方法是为整个应用定义一个单一的EntityMetadataMap，并在初始化NgRx数据库的地方指定它:

EntityDataModule.forRoot({

entityMetadata: appEntityMetadata,

})

如果你在不同 eagerly-loaded Angular modules中定义了几个实体，你就可以用multi-provider为每个模块添加元数据。

{ provide: ENTITY\_METADATA\_TOKEN, multi: true, useValue: someEntityMetadata }

这种技术不适用于惰性加载的模块。ENTITY\_METADATA\_TOKEN提供程序已经设置好，并在惰性加载模块到达时被使用。

该模块应该注入EntityDefinitionService，并直接向其中一个注册方法注册元数据。

@NgModule({...})

class LazyModule {

constructor(eds: EntityDefinitionService) {

eds.registerMetadataMap(this.lazyMetadataMap);

}

...

}

## Metadata Properties

EntityMetadata<T>接口描述了实体类型的各个方面，告诉NgRx数据库如何管理T类型的实体数据集合。

Type T是你的应用程序在TypeScript中对该实体的表示;它可以是接口或类。

#entityName

类型的entityName是唯一必需的元数据属性。它是缓存中实体类型元数据的唯一键

它必须为单个EntityMetadata实例指定。如果在EntityMetadataMap中忽略它，则映射键就会变成entityName，如本例所示。

const map = {

Hero: {} // "Hero" becomes the entityName

};

entityName的拼写和大小写(通常是PascalCase)对于NgRx数据约定很重要。它出现在生成的实体操作、错误消息和持久性操作中。

重要的是，默认的实体dataservice从该名称的小写版本创建HTTP资源url。例如，如果entityName是"Hero"，默认的数据服务将会POST到一个URL，比如'api/ Hero '。

默认情况下，它在准备集合资源URL时生成实体名称的复数形式。

它不擅长多元化。它会为URL生成api/heros来获取所有英雄，因为它盲目地在小写实体名后面加了一个s。

当然，hero的正确复数形式是“heroes”，而不是“heros”。下面您将看到如何纠正这个问题。

#filterFn

许多应用程序允许用户过滤缓存的实体集合。

在附带的演示应用中，用户可以通过姓名来过滤英雄，也可以通过姓名或恶棍的说法来过滤恶棍。

我们觉得这个常见的场景值得构建到NgRx数据库中。所以每个实体都可以有一个可选的过滤器函数。

每个集合的filteredEntities选择器根据用户的筛选条件(保存在存储实体集合的filter属性中)对集合应用筛选函数。

如果没有筛选函数，filteredEntities选择器与selectAll选择器相同，后者返回集合中的所有实体。

过滤器函数(参见EntityFilterFn)接受实体集合和用户的过滤条件(过滤器模式)，并返回所选实体的数组。

下面是一个筛选具有name属性(其值包含搜索字符串)的实体的示例。

export function nameFilter(entities: { name: string }[], search: string) {

return entities.filter(e => -1 < e.name.indexOf(search));

}

NgRx数据库包含一个辅助函数PropsFilterFnFactory<T>，它创建一个实体筛选函数，将用户的输入作为一个不区分大小写的正则表达式，并将其应用到实体的一个或多个属性。

演示程序使用这个助手来创建英雄和恶棍过滤器。以下是应用程序如何为恶人创建nameAndSayingFilter函数。

/\*\*

\* Filter for entities whose name or saying

\* matches the case-insensitive pattern.

\*/

export function nameAndSayingFilter(entities: Villain[], pattern: string) {

return PropsFilterFnFactory<Villain> ['name', 'saying'](entities, pattern);

}

#selectId

每种实体类型都必须有一个主键，其值为整数或字符串。

NgRx数据库假设实体有一个id属性，它的值是主键。

并不是每个实体都有一个名为id的主键属性。对于某些实体，主键可以是两个或多个属性的组合值。

在这些情况下，您指定一个selectId函数，给定一个实体实例，该函数返回一个整数或字符串主键值。

在EntityCollectionReducer测试中，Villain类型有一个名为key的string主键属性。selectorId函数是这样的:

selectId: (villain: Villain) => villain.key;

#sortComparer

NgRx数据库以特定的顺序保存集合实体。

这实际上是底层NgRx实体库的一个特性。

默认顺序是实体从服务器到达的顺序。添加的实体被推到集合的末尾。

您可能更喜欢按其他顺序维护集合。当你提供sortComparer函数时，NgRx-lib会按照你的comparer所规定的顺序来保存这个集合。

在演示应用程序中，反派元数据没有比较器，所以它的实体是默认顺序。

hero元数据有一个sortByName比较器，它将集合按名称的字母顺序保存。

export function sortByName(a: { name: string }, b: { name: string }): number {

return a.name.localeCompare(b.name);

}

运行演示应用程序，尝试更改现有英雄的名字或添加新的英雄。

你的应用程序可以调用selectKey选择器来查看集合的ids属性，它会返回一个集合的主键值的数组，这个数组是按排序的。

#entityDispatcherOptions

这些选项决定集合的分派器的默认行为，该分派器将动作发送给reducer和effects。

dispatcher save命令将在发送相应的HTTP请求之前(乐观)或之后(悲观)添加、删除或更新集合。调用者可以在可选的isOptimistic参数中指定。如果调用方没有指定，则调度程序将基于默认选项进行选择。

默认值是安全的:乐观值用于删除，悲观值用于添加和更新。您可以在这里重写这些选项。

#additionalCollectionState

存储中的每个NgRx数据实体集合都有预定义的属性。

通过将additionalCollectionState属性设置为具有这些自定义集合属性的对象，您可以添加自己的集合属性。

EntitySelectors测试通过添加foo和bar集合属性来测试英雄元数据。

additionalCollectionState: {

foo: 'Foo',

bar: 3.14

}

当NgRx Data第一次在存储中创建集合时，属性值成为这些属性的初始集合值。

NgRx数据库会为这些属性生成选择器，但无法更新它们。您必须自己创建或扩展现有的reducer来实现这一点

如果您想要添加的属性来自后端，您将需要一些额外的工作，以确保属性可以从Effects正确地保存到存储中。

步骤1:实现PersistenceResultHandler将数据从后端保存到action.payload

创建一个新的类AdditionalPersistenceResultHandler，它扩展DefaultPersistenceResultHandler并覆盖handlessuccessess方法，其目的是解析从DataService接收到的数据，检索附加属性，然后将其保存到action.payload。注意，成功动作的默认减速机需要action.payload。数据是实体的数组或实体。这需要在获取附加属性后进行设置，下面的示例中没有显示。

export class AdditionalPersistenceResultHandler extends DefaultPersistenceResultHandler {

handleSuccess(originalAction: EntityAction): (data: any) => Action {

const actionHandler = super.handleSuccess(originalAction);

// 返回一个工厂以获取数据处理程序

// 从DataService解析数据并保存到action.payload

return function(data: any) {

const action = actionHandler.call(this, data);

if (action && data && data.foo) {

// save the data.foo to action.payload.foo

(action as any).payload.foo = data.foo;

}

return action;

};

}

}

步骤2:覆盖EntityCollectionReducerMethods以从操作中保存额外的属性。EntityCollection实例的有效负载

按照前面的步骤，我们已经向action.payload添加了附加属性。接下来，我们需要将其设置为reducer中的EntityCollection实例。为了实现这一点，我们需要创建一个额外的EntityCollectionReducerMethods来扩展EntityCollectionReducerMethods。此外，我们还需要覆盖该方法以匹配您的操作。例如，如果附加属性foo仅在queryMany操作中可用(由EntityCollectionService.getWithQuery触发)，则可以采用此方法。

export class AdditionalEntityCollectionReducerMethods<T>

extends EntityCollectionReducerMethods<T> {

constructor(public entityName: string, public definition: EntityDefinition<T>) {

super(entityName, definition);

}

protected queryManySuccess(

collection: EntityCollection<T>,

action: EntityAction<T[]>

): EntityCollection<T> {

const ec = super.queryManySuccess(collection, action);

if ((action.payload as any).foo) {

// save the foo property from action.payload to entityCollection instance

(ec as any).foo = (action.payload as any).foo;

}

return ec;

}

}

步骤3:注册定制的EntityCollectionReducerMethods和AdditionalPersistenceResultHandler。

最后，我们需要注册AdditionalPersistenceResultHandler和AdditionalEntityCollectionReducerMethods来替换默认的实现。

在NgModule中注册AdditionalPersistenceResultHandler，

@NgModule({

{ provide: PersistenceResultHandler, useClass: AdditionalPersistenceResultHandler },

})

注册AdditionalEntityCollectionReducerMethods，要做到这一点，我们需要创建一个AdditionalEntityCollectionReducerMethodFactory，详细信息请参见Entity Reducer

@Injectable()

export class AdditionalEntityCollectionReducerMethodsFactory {

constructor(private entityDefinitionService: EntityDefinitionService) {}

/\*\* Create the {EntityCollectionReducerMethods} for the named entity type \*/

create<T>(entityName: string): EntityCollectionReducerMethodMap<T> {

const definition = this.entityDefinitionService.getDefinition<T>(entityName);

const methodsClass = new AdditionalEntityCollectionReducerMethods(entityName, definition);

return methodsClass.methods;

}

}

把AdditionalEntityCollectionReducerMethodsFactory注册到NgModule中，

@NgModule({

{

provide: EntityCollectionReducerMethodsFactory,

useClass: AdditionalEntityCollectionReducerMethodsFactory

},

})

现在你可以从后端获得foo，就像另一个EntityCollection级别的属性一样。

##复数实体名称

NgRx数据的DefaultDataService依赖于HttpUrlGenerator来为每种实体类型创建常规的HTTP资源名(url)。

按照约定，针对单个实体项的HTTP请求包含实体类型名称的小写单数版本。例如，如果实体类型entityName是"Hero"，默认的数据服务将会POST到一个URL，比如'api/ Hero '。

按照约定，针对多个实体的HTTP请求包含实体类型名称的小写、复数版本。获取所有英雄的GET请求的URL应该是类似于“api/heroes”的URL。

HttpUrlGenerator本身不能将实体类型名称复数化。它委托给一个注入的多元化类，称为Pluralizer。

Pluralizer类有一个pluralize()方法，它接受单数字符串并返回复数字符串。

默认的复数形式处理许多常见的英语复数规则，比如附加一个's'。这对于Villain类型(变成了“Villains”)，甚至对于Company(变成了“Companies”)来说都没问题。

它远非完美。例如，它错误地将Hero变成了Heros而不是Heroes。

幸运的是，默认的Pluralizer也注入了一个从单数到复数字符串的映射(使用PLURAL\_NAMES\_TOKEN)。

它的pluralize()方法查找映射中的单数实体名，如果找到，则使用相应的复数值。否则，它将返回实体名称的默认复数形式。

如果这个方案适合你，为例外情况创建一个单数到复数实体名的映射:

export const pluralNames = {

// Case matters. Match the case of the entity name.

Hero: 'Heroes'

};

然后在配置NgRx数据库时指定这个映射。

EntityDataModule.forRoot({

pluralNames: pluralNames

})

如果你在单独的Angular模块中定义实体模型，你就可以在multi-provider中递增地添加一个复数名称映射。

{ provide: PLURAL\_NAMES\_TOKEN, multi: true, useValue: morePluralNames }

如果此方案不适用于您，请用您自己的发明替换Pluralizer类。

{ provide: Pluralizer, useClass: MyPluralizer }

Entity Actions

当你调用EntityCollectionService的一个命令来查询或更新缓存集合中的实体时，它会将一个EntityAction分派给NgRx store。

##Action and EntityAction

一个普通的NgRx动作是一个消息。该消息描述了可以更改存储中的状态的操作。

操作的类型标识操作。它是可选的有效负载，携带执行操作所需的消息数据。

EntityAction是NgRx动作的超集。它有额外的属性来指导NgRx数据对动作的处理。这是完整的界面。

export interface EntityAction<P = any> extends Action {

readonly type: string;

readonly payload: EntityActionPayload<P>;

}

export interface EntityActionPayload<P = any> extends EntityActionOptions {

readonly entityName: string;

readonly entityOp: EntityOp;

readonly data?: P;

// EntityActionOptions (also an interface)

readonly correlationId?: any;

readonly isOptimistic?: boolean;

readonly mergeStrategy?: MergeStrategy;

readonly tag?: string;

error?: Error;

skip?: boolean

}

type - action name, 通常由标记和entityOp生成.

entityName - 实体类型的名称.

entityOp - 实体操作的名称.

data? - 操作的消息数据.

correlationId? - 用于关联相关操作的可序列化对象(通常是字符串).

isOptimistic? - 如果应该乐观地执行操作(在服务器响应之前)，则为true

mergeStrategy - 如何将实体合并到缓存中。看到变更跟踪。

tag? - 要在生成的类型中使用的标记。如果没有指定，实体名就是标签。

error? - 意外的操作处理错误。

skip? -如果下游消费者应该跳过处理操作，则为true.

type 是NgRx唯一需要的属性。它是一个字符串，在所有可以分派到存储区的操作类型集合中唯一标识操作。

NgRx数据不关心类型。它关注的是entityName和entityOp属性。

entityName是实体类型的名称。它在NgRx数据缓存中识别这个动作应用的实体集合。这个名称对应于该集合的NgRx数据元数据。一个实体接口或类名，比如'Hero'，就是一个典型的实体名。

entityOp标识要在实体集合上执行的操作，它是entityOp枚举的一种，对应于NgRx数据可以在集合上执行的近60种不同的操作之一。

data在概念上是消息的主体。其类型和内容应符合所要执行的操作的要求。

可选correlationId吗?是一个可选的可序列化对象(通常是一个GUID)，它关联两个或多个动作，比如发起服务器动作的动作(“获取所有英雄”)，以及服务器动作完成后的后续动作(“成功获取英雄”或“获取英雄时出错”)。

可选的merge strategy告诉NgRx Data如何将动作的结果“合并”到缓存中。大多数情况下，这是一个指令到变更跟踪子系统。

当EntityActionFactory创建这个EntityAction时，可选的tag出现在生成的type文本中。

entityName是格式化类型中出现在方括号之间的默认标记，例如'[Hero] NgRx Data/query-all'。您可以设置这个标记来标识操作的目的和“谁”分派了它。NgRx Data会把你的标签放在格式化类型的括号中。

error属性指示在处理操作时出现了错误。请参阅下面的更多。

skip属性告诉下游的动作接收者，它们应该跳过通常的动作处理。这个标志通常是缺失的，并且默认为false。请参阅下面的更多。

##EntityAction consumers

NgRx库会忽略Action.type。所有的NgRx数据库行为都是由entityName和entityOp属性单独决定的。

NgRx Data EntityReducer根据entityName将一个动作重定向到一个EntityCollectionReducer上，而这个reducer根据entityOp处理这个动作。

如果一个操作的entityOp是一个持久化的EntityAction的小集合，EntityEffects将拦截该操作。entityOp名字。effect为该操作的entityName选择正确的数据服务，然后告诉服务发出适当的HTTP请求并处理响应。

##Creating an EntityAction

如果你愿意，你可以手动创建EntityAction。NgRx数据库认为任何带有entityName和entityOp属性的动作都是一个EntityAction。

create()方法帮助您创建一个一致的格式良好的EntityAction实例，该实例的类型是由标签(默认情况下是entityName)和entityOp组成的字符串。

例如，生成的默认操作。在服务器上查询所有英雄的操作类型是'[Hero] NgRx Data/query-all'。

EntityActionFactory.create()方法调用工厂的formatActionType()方法生成操作。字符串类型。

因为NgRx Data忽略了类型，如果你喜欢不同的格式，你可以用自己的方法替换formatActionType()，或者提供并注入你自己的EntityActionFactory。

注意，每个实体类型对于每个操作都有自己唯一的操作，就像您手工单独创建它们一样。

##Tagging the EntityAction

一种形式良好的行动类型可以告诉读者是什么改变了，是谁改变了它。

NgRx数据库不会查看EntityAction的类型，只查看它的entityName和entityOp。因此，只要它们共享相同的entityName和entityOp，您就可以从几个不同的操作获得相同的行为，每个操作都有自己的信息类型。

EntityActionFactory.create()方法的可选标记参数可以很容易地生成有意义的entityaction。

您不必指定标记。entityName是格式化类型中出现在方括号之间的默认标记，例如'[Hero] NgRx Data/query-all'。

下面是一个使用可注入的EntityActionFactory来构造默认的“query all heroes”动作的例子。

const action = this.entityActionFactory.create<Hero>(

'Hero',

EntityOp.QUERY\_ALL

);

store.dispatch(action);

由于NgRx数据的影响，这会在日志中产生两个动作，第一个是发起请求，第二个是成功响应:

[Hero] ngrx/data/query-all

[Hero] ngrx/data/query-all/success

这个默认的entityName标记标识操作的目标实体集合。但是您无法从这些日志条目中理解操作的上下文。你不知道是谁发动了这次行动，也不知道为什么。动作类型太泛了。

您可以通过提供一个标记来创建一个信息更丰富的操作，该标记可以更好地描述正在发生的事情，也可以更容易地找到代码将该操作分派到何处。

比如：

const action = this.entityActionFactory.create<Hero>(

'Hero',

EntityOp.QUERY\_ALL,

null,

{ tag: 'Load Heroes On Start' }

);

store.dispatch(action);

操作日志现在看起来像这样:

[Load Heroes On Start] ngrx/data/query-all

[Load Heroes On Start] ngrx/data/query-all/success

##Handcrafted EntityAction

您不必使用EntityActionFactory创建实体操作。任何带有entityName和entityOp属性的操作对象都是实体操作，如下所述。

下面的示例手动创建初始化的“查询所有英雄”动作。

const action = {

type: 'some/arbitrary/action/type',

entityName: 'Hero',

entityOp: EntityOp.QUERY\_ALL

};

store.dispatch(action);

它通过NgRx数据效果触发HTTP请求，就像前面的例子一样。

请注意，NgRx Data effects使用EntityActionFactory来创建第二个success动作。如果没有tag属性，它将生成一个通用的成功操作。

这两种操作类型的日志如下所示:

some/arbitrary/action/type

[Hero] NgRx Data/query-all-success

##Where are the EntityActions?

在一个NgRx数据应用中，NgRx数据库为你创建和调度实体动作。

在调用EntityCollectionService API时，entityaction在很大程度上是不可见的。你可以在NgRx store开发工具中看到它们的实际应用。

#为什么这很重要

在一个普通的NgRx应用程序中，你手工为存储中的每个状态编写每个动作以及处理这些动作的reducer。

它需要很多动作，一个复杂的减速机，以及NgRx效果的帮助来管理单个实体类型的查询和保存。

NgRx实体库让这一工作变得简单多了。

NgRx数据库内部将大部分繁重的工作委托给了NgRx实体。

但是，您仍然必须为每种实体类型编写大量代码。你需要为每个实体类型创建8个动作，并通过调用NgRx EntityAdapter的8个方法来写一个reducer来响应这8个动作。

这些构件只处理缓存的实体集合。

您可以编写多达18个附加操作来支持典型的异步CRUD(创建、检索、更新、删除)操作。你必须将它们发送到商店，在那里你将使用更多的reducer方法和效果来处理它们，你也必须手工编写代码。

使用普通的NgRx，您将对每种实体类型进行此练习。这需要编写、测试和维护大量代码。

在NgRx数据的帮助下，您无需编写任何数据。NgRx数据创建响应这些动作的动作、调度程序、减少程序和效果。

##EntityAction.error

EntityAction.error属性指示在处理操作时发生了不好的事情。

EntityAction应该是不可变的。EntityAction。error属性是唯一的例外，它是NgRx数据系统的内部属性。您应该很少(如果有的话)自己设置。

error的主要用例是捕获减速机异常。如果减少器中有一个抛出异常，NgRx将停止订阅。捕获减速机异常允许应用程序继续运行。

NgRx数据会捕捉由EntityCollectionReducer抛出的错误，并设置EntityAction。属性设置为捕获的错误对象。

当错误操作是持久性操作(例如SAVE\_ADD\_ONE)时，error属性很重要。EntityEffects将看到这样的操作有一个错误，并立即返回相应的失败操作(SAVE\_ADD\_ONE\_ERROR)，而不尝试HTTP请求。

这是我们发现的防止坏动作通过效果触发HTTP请求的唯一方法。

##EntityAction.skip

skip属性告诉下游的动作接收者，它们应该跳过通常的动作处理。这个标志通常是缺失的，并且默认为false。

当你试图删除一个未保存的新实体时，NgRx数据集skip=true。当EntityEffects。当持久化$方法在EntityAction信封上看到这个标志设置为true时，它跳过HTTP请求并使用原始请求有效负载分派一个适当的\_SUCCESS动作。

当你试图删除一个已经添加到集合但没有保存的实体时，这个特性允许NgRx Data避免发出删除请求。这样的请求在服务器上可能会失败，因为没有要删除的实体。

有关变更跟踪的更多信息，请参阅EntityChangeTracker页面。

##EntityCache-level actions

有一些操作将实体缓存作为一个整体作为目标。

SET\_ENTITY\_CACHE用action有效负载中的对象替换整个缓存，有效地将实体缓存重新初始化到已知状态。

MERGE\_ENTITY\_CACHE将当前实体缓存中的特定实体集合替换为动作有效负载中的那些集合。它不影响其他当前集合。

在“EntityReducer”文档中了解它们

##Entity Collection

NgRx数据库为NgRx存储中的每种实体类型维护了一个实体集合的缓存(EntityCache)。

entity\_collection为实体类型实现了EntityCollection接口。

PROPERTY MEANING

ids 默认排序顺序的主键值

entities 主键到实体数据值的映射

filter 用户的过滤条件

loaded 集合是否由QueryAll填充;清除后强制false

loading 当前是否正在等待来自服务器的查询结果

changeState 当启用更改跟踪时，未保存实体的更改属性

您可以通过实体元数据使用附加属性扩展实体类型。

##EntityCollectionService

EntityCollectionService是一个建立在NgRx数据分派器和选择器$之上的facade，它管理缓存在NgRx存储中的实体集合。

Dispatcher提供了命令方法来将实体的动作分派到NgRx存储中。这些命令要么直接更新实体集合，要么触发对服务器的HTTP请求。当服务器响应时，NgRx数据库会用响应数据分派新的动作，这些动作会更新实体集合。

EntityCommands接口列出了所有的命令以及它们的作用。

你的应用程序调用这些命令方法来更新NgRx存储中的缓存实体集合。

Selectors$是返回selector observable的属性。每个可观察对象都在缓存的实体集合中观察特定的变化，并发出变化后的值。

EntitySelectors$ interface列出了所有预定义的selector observable属性，并解释了它们观察的集合属性。

您的应用程序订阅选择器可观察对象，以便处理和显示集合中的实体。

##示例来自演示应用程序

下面是演示应用的HeroesComponent的一些简化摘录，展示了该组件如何调用命令方法和订阅selector observable。

constructor(EntityCollectionServiceFactory: EntityCollectionServiceFactory) {

this.heroService = EntityCollectionServiceFactory.create<Hero>('Hero');

this.filteredHeroes$ = this.heroService.filteredEntities$;

this.loading$ = this.heroService.loading$;

}

getHeroes() { this.heroService.getAll(); }

add(hero: Hero) { this.heroService.add(hero); }

deleteHero(hero: Hero) { this.heroService.delete(hero.id); }

update(hero: Hero) { this.heroService.update(hero); }

#使用工厂创建EntityCollectionService

该组件注入NgRx数据实体实体集合服务factory，并为英雄实体创建一个实体集合服务实体。

我们将在稍后的指南中进入工厂内部。

#Create the EntityCollectionService as a class

另外，你也可以在其他地方创建一个HeroEntityService，比如在AppModule中，然后把它注入到组件的构造函数中。

有两种创建服务类的基本方法。

1 - 来自EntityCollectionServiceBase < T >

2 - 用你需要的API编写一个HeroEntityService。

当HeroEntityService派生自EntityCollectionServiceBase<T>时，它必须将EntityCollectionServiceFactory注入到它的构造函数中。在演示应用程序中有这种方法的例子。

当用有限的API定义HeroEntityService时，你也可以注入EntityCollectionServiceFactory作为你要公开的功能的源。

让你喜欢的风格和应用程序需要决定你选择哪种创建技术。

#设置组件selector$属性

selector$ property是一个可观察对象，当所选的state属性发生变化时，它就会触发。

示例组件有两个这样的属性，它们公开了两个EntityCollectionService选择器observable: filteredEntities$和loading$。

filteredEntities$ observable会生成一个数组，其中包含当前缓存的英雄实体，这些实体满足用户的过滤条件。如果用户改变了过滤器，或者某个动作改变了缓存集合中的英雄，这个可观察对象就会生成一个新的英雄数组。

当数据服务正在等待服务器上的英雄时，正在加载的$ observable会生成true。它在服务器响应时生成false。这个演示应用程序订阅了loading$，这样它就可以打开和关闭一个可视的加载指示器。

这些component和EntityCollectionService选择器属性名以'$'结尾，这是返回可观察对象的属性的常见约定。EntityCollectionService的所有选择器可观察属性都遵循这个约定。

#选择器可观察对象与选择器函数

选择器$ observable(以'$'结尾)不同于同名且密切相关的选择器函数(没有'$'后缀)

选择器是从实体集合中选择状态片的函数。当状态发生变化时，selector$ observable会发出该状态片。

NgRx Data通过把selector函数传递给NgRx select操作符，并把它管道到NgRx存储中，来创建一个selector$ observable，如下面的例子所示:

loading$ = this.store.select(selectLoading);

#Using selectors$

组件类不订阅这些selector$属性，但组件模板订阅了。

模板绑定到它们，并把它们的可观察对象转发给Angular的AsyncPipe，后者会订阅它们。以下是filteredHeroes$绑定的摘录。

<div \*ngIf="filteredHeroes$ | async as heroes">

...

</div>

#调用命令方法

大多数HeroesComponent方法委托给了EntityCollectionService命令方法，比如getAll()和add()。

有两种命令:

1 - 触发对服务器请求的命令。

2 - 仅缓存命令，用于更新缓存的实体集合

服务器命令是简单的动词，如“add”和“getAll”。它们将触发异步请求的操作分派到远程服务器。

仅缓存命令方法是更长的动词，如“addManyToCache”和“removeOneFromCache”，它们的名称都包含单词“cache”。它们立即(同步地)更新缓存的集合。

大多数应用程序调用服务器命令是因为它们想要查询和保存实体数据。

应用程序很少调用仅缓存命令，因为当应用程序关闭时，实体集合的直接更新会丢失。

许多EntityCollectionService命令方法都有一个值。这个值是有类型的(通常是Hero)，所以你不会因为传入了错误类型的值而出错。

在内部，实体服务方法创建与方法意图相对应的实体动作。操作的有效负载要么是传递给方法的值，要么是该值的适当导数。

不可变性是还原模式的核心原则。一些命令方法有实体参数，比如Hero。实体参数永远不能是缓存的实体对象。它可以是缓存的实体对象的副本，通常也是这样。演示应用程序总是使用实体数据的副本来调用这些命令方法。

所有的命令方法返回void。redux模式的核心原则是命令永远不会返回值。他们只会做一些有副作用的事。

与其期待命令的结果，不如订阅反映命令效果的selector$属性。如果命令做了一些您关心的事情，选择器$ property应该能够告诉您。

##EntityCollectionServiceFactory

NgRx数据EntityCollectionServiceFactory的create<T>()方法产生了一个EntityCollectionServiceBase<T>类的新实例，它实现了实体类型为T的EntityCollectionService接口。

##Entity DataService

NgRx数据库希望通过调用一个类似rest的web api来持久化实体数据，每个实体类型都有端点。

EntityDataService维护一个服务类注册表，这些服务类专用于为特定的实体类型持久化数据。

当NgRx数据库看到一个实体持久化操作的动作时，它会向EntityDataService请求注册的数据服务，该数据服务会对该实体类型进行HTTP调用，并调用相应的服务方法。

数据服务是实现EntityCollectionDataService的类的实例。该接口支持实体的基本CRUD操作集。返回observable的每一个:

add(entity: T): Observable<T> 添加一个新的实体 POST /api/hero/

delete(id: number | string)

: Observable<number | string> 按主键值删除实体 DELETE /api/hero/5

getAll(): Observable<T[]> 获取此实体类型的所有实例 GET /api/heroes/

getById(id: number | string)

: Observable<T> 通过主键获取实体 GET /api/hero/5

getWithQuery(

queryParams: QueryParams

| string)

: Observable<T[]> 获取满足查询的实体 GET /api/heroes/?name=bombasto

update(update: Update<T>)

: Observable<T> 更新现有的实体 PUT /api/hero/5

QueryParams是一个参数名/值映射，您也可以提供查询字符串本身。HttpClient安全地将两者编码到一个编码过的查询字符串中。

Update<T>是一个具有实体属性严格子集的对象。它必须包括参与主键的属性(例如，id)。update属性值是要更新的属性;未提及的属性应保留其当前值。

默认的数据服务方法会返回相应的Angular HttpClient方法返回的observable。

你的API应该以每个数据服务方法的返回类型的形式返回一个对象。例如:当调用.add(entity)时，你的API应该创建实体，然后返回匹配T的完整实体，因为它的值将被设置为实体主键的存储记录。与其他方法不同的一个方法是delete。delete需要实体主键的响应类型，string | number，而不是已删除的完整对象T。

如果您创建了自己的数据服务替代方案，它们应该返回类似的observable。

##Register data services

默认情况下，EntityDataService注册表为空。

您可以通过创建这些类的实例，并通过以下两种方式之一向EntityDataService注册它们，从而向其添加自定义数据服务。

1 - 使用registerService()方法按实体名称注册单个数据服务。

2 - 通过使用实体名称/服务映射调用registerServices，同时注册多个数据服务。

您可以创建并导入一个模块，该模块注册您的自定义数据服务，如EntityDataService测试所示

如果你决定注册一个实体数据服务，一定要在你要求NgRx data为这个实体执行持久化操作之前注册。

否则，NgRx数据库将为该实体类型创建并注册一个默认数据服务DefaultDataService<T>的实例。

##The DefaultDataService

演示应用程序没有注册任何实体数据服务。它完全依赖于由注入的DefaultDataServiceFactory为每个实体类型创建的DefaultDataService。

DefaultDataService<T>会使用Angular的HttpClient对服务器的web api进行类似rest的调用。

它从根路径(参见下面的“配置”)和实体名称组成HTTP url。

例如,

1 如果持久化操作是删除id=42和

2 根路径是'api'和

3 那么，实体名称是'Hero'

4 删除请求的URL将是'api/hero/42'。

当持久性操作涉及多个实体时，DefaultDataService用实体类型名称的复数形式替换资源名称。

QUERY\_ALL获取所有英雄的动作会导致一个指向URL 'api/heroes'的HTTP get请求。

DefaultDataService不知道如何将实体类型名称改为复数。它甚至不知道如何创建基本资源名。

实体元数据指南解释了如何配置默认的Pluralizer。

#配置DefaultDataService

集合级数据服务为HTTP调用构建自己的url。它们通常依赖于共享的配置信息，比如每个资源URL的根。

共享配置值几乎总是特定于应用程序，并且可能根据运行时环境而变化。

NgRx数据库定义了一个DefaultDataServiceConfig来传递共享配置到实体集合数据服务。

最重要的配置属性root返回每个web api URL的根，即实体资源名称之前的部分。如果您正在使用远程API，该值可以包括协议、域、端口和根路径，如https://my-api-domain.com:8000/api/v1。

对于DefaultDataService<T>，它的默认值是'api'，它会产生像api/heroes这样的url。

timeout属性设置ng-lib持久化操作放弃接收服务器应答并取消操作之前的最大时间(单位为ms)。默认值是0，这意味着请求不会超时。

delete404OK标志告诉数据服务，如果服务器以404 - Not Found响应删除请求，该做什么。

通常，找不到要删除的资源是没有危害的，您可以通过将此标志设置为true来避免忽略delete 404错误，这是DefaultDataService<T>的默认值。

在本地运行演示应用程序时，服务器的响应速度可能比在生产环境中更快。您可以通过设置getDelay和saveDelay属性来模拟真实世界。

#提供自定义配置

首先，创建一个DefaultDataServiceConfig类型的自定义配置对象:

const defaultDataServiceConfig: DefaultDataServiceConfig = {

root: 'https://my-api-domain.com:8000/api/v1',

timeout: 3000, // request timeout

}

在一个eagerly-loaded的NgModule中提供它，比如示例应用中的EntityStoreModule:

@NgModule({

providers: [{ provide: DefaultDataServiceConfig, useValue: defaultDataServiceConfig }]

})

##Custom EntityDataService

虽然NgRx数据库提供了一个配置对象来修改DefaultDataService的某些方面，但你可能想要进一步自定义当你为一个特定的集合保存或检索数据时会发生什么。

例如，您可能需要修改获取的实体以将字符串转换为日期，或向实体添加额外的属性。

您可以通过创建一个自定义数据服务并向EntityDataService注册该服务来实现这一点。

为了说明这一点，示例应用程序在Hero实体中添加了一个dateLoaded属性，以便记录何时将英雄从服务器加载到NgRx-store实体缓存中。

export class Hero {

readonly id: number;

readonly name: string;

readonly saying: string;

readonly dateLoaded: Date;

}

为了支持这个特性，我们将创建一个HeroDataService类，它实现了EntityCollectionDataService<T>接口。

在样例应用中，HeroDataService派生自NgRx的数据DefaultDataService<T>，以利用它的基本功能。它只会覆盖它真正需要的东西。

@Injectable()

export class HeroDataService extends DefaultDataService<Hero> {

constructor(http: HttpClient, httpUrlGenerator: HttpUrlGenerator, logger: Logger) {

super('Hero', http, httpUrlGenerator);

logger.log('Created custom Hero EntityDataService');

}

getAll(): Observable<Hero[]> {

return super.getAll().pipe(map(heroes => heroes.map(hero => this.mapHero(hero))));

}

getById(id: string | number): Observable<Hero> {

return super.getById(id).pipe(map(hero => this.mapHero(hero)));

}

getWithQuery(params: string | QueryParams): Observable<Hero[]> {

return super.getWithQuery(params).pipe(map(heroes => heroes.map(hero => this.mapHero(hero))));

}

private mapHero(hero: Hero): Hero {

return { ...hero, dateLoaded: new Date() };

}

}

这个HeroDataService挂钩到get操作中来设置英雄。在已获取的英雄实体上加载日期。它还会告诉记录器何时创建它(请参阅正在运行的示例的控制台输出)。

最后，我们必须告诉NgRx Data这个新的数据服务。

这个示例应用提供了HeroDataService，并通过调用应用的entity store模块中的EntityDataService的registerService()方法来注册它:

import { EntityDataService } from '@ngrx/data'; // <-- import the NgRx Data data service registry

import { HeroDataService } from './hero-data-service';

@NgModule({

imports: [ ... ],

providers: [ HeroDataService ] // <-- provide the data service

})

export class EntityStoreModule {

constructor(

entityDataService: EntityDataService,

heroDataService: HeroDataService,

) {

entityDataService.registerService('Hero', heroDataService); // <-- register it

}

}

##一个自定义DataService

您不必重写DefaultDataService的成员。您可以编写一个完全自定义的替代方案，通过您选择的任何机制查询和保存实体。

只要它符合接口，您就可以用同样的方式注册它。

// Register custom data service

entityDataService.registerService('Hero', peculiarHeroDataService);

Entity Effects

Work in Progress

Effects是用动作触发副作用的一种方式。

一个常见的、理想的副作用是对远程服务器的异步HTTP调用，以获取或保存实体数据。

你可以在NgRx效果库的帮助下实现一个或多个效果。

分派给NgRx存储的动作可以被你的effect方法检测和处理。处理之后，无论是同步的还是异步的，您的方法都可以将新的操作分派到存储区

NgRx数据库在它的entityeeffects类中实现了一个名为persist$的效果。

persist$方法过滤特定的EntityAction.op的价值观。这些值将转换为带有实体数据的HTTP GET、PUT、POST和DELETE请求。当服务器响应时(不管是顺利响应还是出现错误)，persist$方法将新的EntityActions发送到存储区，并提供相应的响应数据。

#取消Cancellation

您可以尝试取消保存，方法是使用要取消的持久化操作的相关id分派一个CANCEL\_PERSIST EntityAction。

EntityCache。取消此操作的$ watches，并通过管道进入EntityCache。持久化$，它可以尝试取消实体集合查询或保存操作，或者至少阻止服务器响应更新缓存。

对于保存操作来说，这显然不是一个好主意。您不能以这种方式告诉服务器取消，也不能知道服务器是否保存了。也不能指望在客户端接收到服务器响应并将更改应用到服务器或缓存之前处理取消请求。

如果在服务器结果到达之前取消，entityeeffect将不会尝试用迟到达的服务器结果更新缓存的集合。这个效果将发出一个CANCELED\_PERSIST操作。EntityCollection reducer会忽略这个操作，但是你可以在store操作中监听它，从而知道取消操作在客户端上生效了。

Entity Reducer

实体Reducer是存储在实体缓存中的所有实体集合的主Reducer。

该库没有命名的实体缩小器类型。相反，它依赖于EntityCacheReducerFactory.create()方法来产生这个reducer，它是一个NgRx ActionReducer<EntityCache, EntityAction>。

这样的Reducer函数接受一个EntityCache状态和一个EntityAction动作，并返回一个EntityCache状态。

Reducer要么响应一个entitycache级别的动作(罕见)，要么响应一个以实体集合为目标的EntityAction(通常情况下)。所有其他类型的操作都会被忽略，而Reducer只返回给定的状态。

Reducer专门针对操作的entityType属性进行筛选。它将任何带有entityType属性的操作视为EntityAction。

实体Reducer的主要工作是

1 从状态中提取操作的实体类型的EntityCollection。

2 如果需要，创建一个新的、初始化的实体集合。

3 获取或创建该实体类型的EntityCollectionReducer。

4 使用集合和动作调用实体集合Reducer

5 用实体集合缩减器返回的新集合替换EntityCache中的实体集合。

##EntityCollectionReducers

EntityCollectionReducer将动作应用到NgRx存储中的EntityCache中的EntityCollection中。

对于给定的实体类型，总是有一个reducer。EntityCollectionReducerFactory维护它们的注册表。如果它找不到实体类型的Reducer，它会在注入的EntityCollectionReducerFactory的帮助下创建一个Reducer，并注册这个Reducer，以便下次再次使用。

#Register custom reducers

你可以为一个实体类型创建一个自定义的Reducer，并直接注册到EntityCollectionReducerRegistry.registerReducer()。

通过调用EntityCollectionReducerRegistry.registerReducers(reducerMap)，您可以同时注册几个自定义reducers，其中reducerMap是reducers的散列，以entity-type-name为键值。

##Default EntityCollectionReducer

EntityCollectionReducerFactory根据应用程序的实体元数据创建了一个默认的reducer，它利用了NgRx EntityAdapter的功能。

默认的Reducer根据EntityAction决定要做什么。属性，它期望其字符串值是EntityOp enum的成员

许多EntityOp值被忽略;Reducer只返回给定的实体集合。

例如，某些面向持久化的操作需要由NgRx数据持久化$ effect来处理。它们不更新收集数据(可能除了翻转加载标志之外)。

其他人则从集合中添加、更新和删除实体。

记住，不可变对象是redux/NgRx模式的核心原则。这些缩减器实际上不会改变原始集合或其中的任何对象。它们创建集合的副本，并且只更新集合内对象的副本。

请参阅NgRx Entity EntityAdapter收集方法，了解默认的Entity collection reducer执行的缓存更改操作的基本指南。

EntityCollectionReducerFactory类及其测试是默认reducer实际工作方式的权威。

##Initializing collection state

NgRxDataModule为NgRx数据存储添加了一个空的EntityCache。此缓存中没有集合。

如果主EntityReducer无法找到操作的实体类型的集合，它将在EntityCollectionCreator的帮助下创建一个新的、初始化的集合，该集合被注入到EntityCacheReducerFactory中。

创建者从实体的实体定义中的initialState返回一个已初始化的集合。如果实体类型没有定义或定义没有initialState属性值，创建者将返回一个EntityCollection。

然后，实体reducer在实体集合reducer的state参数中传递新的集合。

##定制实体Reducer行为

您可以通过注册一个自定义替代来替换任何实体集合减少器。

可以通过提供EntityCollectionReducerFactory的自定义替代来替换默认的entity reducer。

您甚至可以通过提供EntityCacheReducerFactory的自定义替代来替换主实体Reducer。

但通常你会想要用一些额外的Reducer逻辑来扩展集合Reducer逻辑，这些逻辑在集合Reducer逻辑之前或之后运行。

##EntityCache-level actions

有一些操作将实体缓存作为一个整体作为目标。

SET\_ENTITY\_CACHE用action有效负载中的对象替换整个缓存，有效地将实体缓存重新初始化到已知状态。

MERGE\_ENTITY\_CACHE将当前实体缓存中的特定实体集合替换为动作有效负载中的那些集合。它不影响其他当前集合。

看到entity-reducer.spec。ts为这些操作的示例。

这些操作可能是您计划的一部分，以支持脱机场景或同时回滚多个集合的更改。

例如，您可以订阅EntityServices。entityCache选择美元。当缓存更改时，您可以将缓存序列化到浏览器本地存储。你可能想要反弹几秒钟以减少流失。

稍后，当重新启动应用程序时，您可以分派SET\_ENTITY\_CACHE操作来初始化实体缓存，即使在断开连接时也是如此。或者您可以分派MERGE\_ENTITY\_CACHE来将选定的集合回滚到错误恢复或“假设”场景中的已知状态。

重要提示:MERGE\_ENTITY\_CACHE用其有效负载中的实体集合替换当前缓存的集合。它没有像名称所暗示的那样将有效负载集合实体合并到现有集合中。也许以后会重新考虑并这样做。

如果您想要创建和减少额外的、缓存范围的操作，请考虑EntityCache MetaReducer，下一节将描述它。

##MetaReducers

NgRx/store支持MetaReducers，它可以检查和处理流经存储的动作，并可能改变存储中的状态。

一个MetaReducer是一个接受一个reducer并返回一个reducer的函数。NgRx将这些reducer与其他reducer组成一个责任链。

NgRx调用由MetaReducer返回的reducer，就像它调用任何reducer一样。它使用一个状态对象和一个操作来调用它。

MetaReducer可以对状态和动作做它想做的事情。它可以记录操作、自己处理操作、委托给传入的reducer、后处理更新后的状态，或者以上所有操作。

记住，动作本身是不可变的。不要改变行动!

像每个reducer一样，传递给MetaReducer的reducer的状态只是在这个reducer的作用域中存储的部分。

NgRx数据支持两个级别的MetaReducer

1 EntityCache MetaReducer，作用域到整个实体缓存

2 ntityCollection MetaReducer，作用域为特定的集合。

#Entity Cache MetaReducers

EntityCache MetaReducer帮助您检查和应用影响整个实体缓存的操作。您可以添加自定义操作和EntityCache MetaReducer来同时更新多个集合。

一个EntityCache MetaReducer必须满足以下三个要求:

1 总是返回整个实体缓存。

2 同步返回(不等待服务器响应)。

3 永远不要改变原来的动作;克隆它来改变它。

我们打算在文档更新中解释如何更新。现在，请参阅NgRx Data entity-data.module.spec。ts为例子。

# Entity Collection MetaReducers

实体集合的MetaReducer以一个实体集合的reducer作为它的reducer参数，并返回一个新的实体集合的reducer。

新的Reducer将接收EntityCollection和EntityAction参数，这些参数将被发送到原来的Reducer。

它可以使用这些参数做它想做的事情，例如:

1 记录操作，

2 将操作转换为不同的操作(对于相同的实体集合)，

3 调用原始Reducer，

4 对原始Reducer的结果进行后处理。

新的实体集合Reducer必须满足三个要求

1 总是返回同一个实体的EntityCollection。

2 同步返回(不等待服务器响应)。

3 永远不要改变原来的动作;克隆它来改变它。

#与Store MetaReducers相比

虽然实体集合的MetaReducer是基于NgRx Store的MetaReducer(“Store MetaReducer”)建模的，但它在几个方面有很大的不同。

Store MetaReducer广泛地针对Store reducers。它包装存储缩减器，查看所有操作，并可以更新其范围内的任何状态。

但是Store MetaReducer既不会看到也不会包装实体集合reducer。这些实体集合缩减器是注册到NgRx数据特性的EntityCache缩减器内部的。

实体集合元Reducer只关注单个目标实体集合的操作。它封装了所有实体集合缩减器。

注意，它不能访问其他集合、实体缓存或存储中的任何其他状态。如果您需要一个跨集合的MetaReducer，请尝试上面描述的EntityCache MetaReducer。

#Provide Entity MetaReducers to the factory

创建一个或多个实体集合元简化器，并将它们添加到数组中。

为这个数组提供ENTITY\_COLLECTION\_META\_REDUCERS注入令牌，在这里你可以导入NgRxDataModule。

EntityCollectionReducerRegistry将其注入，并将元reducers数组组合到单个元metareducer中。较早的metareducer将较晚的metareducer封装在数组中。

当工厂注册一个EntityCollectionReducer时，包括它创建的reducer，它会在将其添加到注册表之前将该reducer包装在元元reducer中

所有分派到存储的entityaction在进出特定于实体的reducer时都要经过这个包装器。

我们打算在文档更新中解释如何创建和提供实体集合MetaReducers。目前，请参阅entity-reducer.spec。ts为例子。

EntityServices

EntityServices是NgRx Data services and the NgRx Data EntityCache.的facade。

##Registry of EntityCollectionServices

EntityServices主要是EntityCollectionServices的注册中心。

调用它的EntityServices.getEntityCollectionService(entityName)方法来获得该实体类型的单例EntityCollectionService。

这里有一个组件可以做到这一点。

@Component({...})

export class HeroesComponent implements OnInit {

heroesService: EntityCollectionService<Hero>;

constructor(entityServices: EntityServices) {

this.heroesService = entityServices.getEntityCollectionService('Hero');

}

}

##Create a custom EntityCollectionService

正如EntityCollectionService文档中所解释的那样，您通常会创建带有额外功能和便利成员的自定义EntityCollectionService类

@Injectable()

export class HeroesService extends EntityCollectionServiceBase<Hero> {

constructor(elementsFactory: EntityCollectionServiceElementsFactory) {

super('Hero', elementsFactory);

}

// ... your special sauce here

}

当然，在使用这个自定义服务之前，你必须先提供它，通常是在Angular的NgModule中。

...

import { HeroesService } from './heroes.service';

@NgModule({

imports: [...],

declarations: [...],

providers: [HeroesService]

})

export class HeroesModule {}

下面的替代示例使用了首选的“可摇树”Injectable()来在根模块中提供服务。

@Injectable({ providedIn: 'root' })

export class HeroesService extends EntityCollectionServiceBase<Hero> {

...

}

您可以将该定制服务直接注入到组件中。

@Component({...})

export class HeroesComponent {

heroes$: Observable<Hero[]>;

loading$: Observable<boolean>;

constructor(public heroesService: HeroesService) {

this.heroes$ = this.heroesService.entities$;

this.loading$ = this.heroesService.loading$;

}

...

}

到目前为止没有什么新东西。但我们希望能够从EntityServices.getEntityCollectionService()中获取HeroesService，就像我们获取默认的集合服务一样。

当应用程序有大量的收集服务时，这种一致性就会得到回报

##Register the custom EntityCollectionService

当您用EntityServices注册一个自定义EntityCollectionService实例时，EntityServices. getentitycollectionservice()的其他调用者将获得该自定义服务实例。

你会想要在任何东西试图通过EntityServices获取它之前这样做。

一种解决方案是将自定义集合服务注入到提供它们的模块的构造函数中，并在那里注册它们。

@NgModule({ ... })

export class AppModule {

// Inject the service to ensure it registers with EntityServices

constructor(

entityServices: EntityServices,

// custom collection services

hs: HeroesService,

vs: VillainsService

){

entityServices.registerEntityCollectionServices([hs, vs]);

}

}

## 子类实体服务方便应用程序类

另一个有用的解决方案是创建EntityServices的子类，它既注入了自定义集合服务，又为应用程序添加了便利成员

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

import { EntityServicesBase, EntityServicesElements } from '@ngrx/data';

import { SideKick } from '../../model';

import { HeroService, VillainService } from '../../services';

@Injectable()

export class AppEntityServices extends EntityServicesBase {

constructor(

elements: EntityServicesElements,

// 注入自定义服务，在EntityServices中注册它们，并在API中公开.

readonly heroesService: HeroesService,

readonly villainsService: VillainsService

) {

super(elements);

this.registerEntityCollectionServices([heroesService, villainsService]);

}

/\*\* get the (default) SideKicks service \*/

get sideKicksService() {

return this.getEntityCollectionService<SideKick>('SideKick');

}

}

AppEntityService首先注入EntityServicesElements助手，并直接将其传递给基类的构造函数。“元素”包含基类需要制作和管理元数据中描述的实体的成分

然后，它会注入你的两个自定义集合服务HeroesService和VillainsService，并将它们作为方便的属性直接公开给消费者，以便他们访问这些服务。

在本例中，我们不需要SideKick实体的自定义集合服务。默认的服务就可以了。

尽管如此，我们添加了一个sideKicksService属性，它为SideKick获取或创建一个默认服务。消费者会发现它比getEntityCollectionService()更容易发现，也更容易调用。

当然，基类EntityServices成员，如getEntityCollectionService()、entityCache$和registerEntityCollectionService()都是可用的

接下来，在Angular的NgModule中，把AppEntityServices作为它自己(AppEntityServices)和实体服务(EntityServices)的别名来提供

通过这种方式，应用程序类引用这个相同的AppEntityServices服务实例，无论它注入的是AppEntityServices还是EntityServices。

@NgModule({

imports: [ ... ],

providers: [

AppEntityServices,

{ provide: EntityServices, useExisting: AppEntityServices },

...

]

})

export class EntityStoreModule { ... }

##访问多个EntityCollectionServices

一个复杂的组件可能需要访问多个实体集合。EntityServices注册表使这变得很容易，即使EntityCollectionServices是为每种实体类型定制的。

您只需要一个注入的构造函数参数，即EntityServices。

@Component({...})

export class CharacterContainerComponent implements OnInit {

heroesService: HeroService;

sideKicksService: EntityCollectionService<SideKick>;

villainService: VillainService;

heroes$: Observable<Hero>;

...

constructor(entityServices: EntityServices) {

this.heroesService = entityServices.getEntityCollectionService('Hero');

this.sidekicksService = entityServices.getEntityCollectionService('SideKick');

this.villainService = entityServices.getEntityCollectionService('Villain');

this.heroes$ = this.heroesService.entities$;

...

}

...

}

EntityServices的一个特定于应用程序的子类，比如上面的AppEntityServices，使它看起来更好一些。

import { AppEntityServices } from '../../services';

@Component({...})

export class CharacterContainerComponent implements OnInit {

heroes$: Observable<Hero>;

...

constructor(private appEntityServices: AppEntityServices) {

this.heroes$ = appEntityServices.heroesService.entities$;

...

}

...

}