* 为每个实体类型自动的创建actions, reducers, effects, dispatchers, and selectors。
* 为每种实体类型提供默认的HTTP GET、PUT、POST和DELETE方法
* holds entity data as collections within a cache which is a slice of NgRx store state.
* 支持乐观和悲观的保存策略
* enables transactional save of multiple entities of multiple types in the same request.
* makes reasonable default implementation choices
* offers numerous extension points for changing or augmenting those default behaviors.

NgRx数据的目标是管理持久的实体数据，如客户和订单，许多应用程序查询并保存到远程存储。这是它的最佳位置。

它不适用于非实体数据。使用标准的NgRx可以更好地管理值类型、枚举、会话数据和高度特殊的数据。现实世界中的应用程序将受益于NgRx技术的组合，它们共享一个公共的存储。

Entity

实体是具有长期数据值的对象，您可以从数据库中读取数据值并将其写入数据库。实体指的是应用程序域中的某些“东西”。示例包括Customer、Order、LineItem、Product、Person和User。

在Ngrx数据中，我们通过主键来维护实体对象的标识。NgRx数据中的每个实体都必须有一个主键。主键通常是对象的单个属性。例如，“Sally”实体对象可能是“Customer”实体类型的实例，该实例永久不变的主键是id属性，值为42。

主键不必是单个属性。如果需要该特性，它可以由对象的多个属性组成。重要的是，主键在相同类型的实体的永久集合中惟一地标识该对象。只能有一个id为42的客户实体，该实体为“Sally”。

Entity Collection

实体集合的概念也是NgRx数据的基础。所有实体都属于同一实体类型的集合。Customer实体属于Customers集合。

即使实体类型只有一个实例，它也必须保存在实体集合中:可能是包含单个元素的集合。

Defining the entities

EntityMetadataMap告诉NgRx Data有关您的实体。为每个实体名称在集合中添加一个属性。

const entityMetadata: EntityMetadataMap = {

Hero: {},

Villain: {}

};

// because the plural of "hero" is not "heros"

const pluralNames = { Hero: 'Heroes' };

export const entityConfig = {

entityMetadata,

pluralNames

};

导出在AppModule中注册实体配置时使用的实体配置。

Registering the entity store

创建实体配置后，需要将其放入NgRx的根存储中。这是通过导入entityConfig，然后将其传递给EntityDataModule.forRoot()函数来完成的。

@NgModule({

imports: [

HttpClientModule,

StoreModule.forRoot({}),

EffectsModule.forRoot([]),

*EntityDataModule.forRoot(entityConfig)*

]

})

export class AppModule {}

Creating entity data services

NgRx数据通过在服务类中扩展EntityCollectionServiceBase来处理服务器上的数据创建、检索、更新和删除。

@Injectable({ providedIn: 'root' })

export class HeroService extends EntityCollectionServiceBase<Hero> {

constructor(serviceElementsFactory: EntityCollectionServiceElementsFactory) {

super('Hero', serviceElementsFactory);

}

}

Using NgRx Data in components

要访问实体数据，组件应该注入实体数据服务。

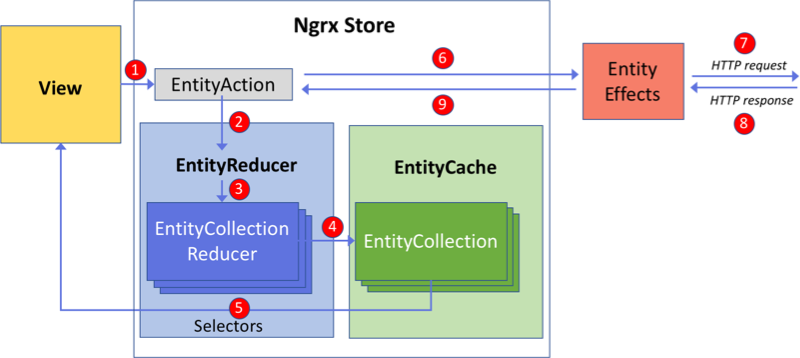
***架构总述***

您可以用几行实体元数据向NgRx数据描述您的实体模型，然后让库来完成其余的工作。

您的组件注入一个NgRx Data EntityCollectionService，并调用一个或多个标准命令方法集来调度操作。

您的组件还订阅了服务的一个或多个Observable选择器，以反应地处理和显示由这些命令产生的实体状态更改。

NgRx data实际上只是NgRx的底层。数据以典型的NgRx方式流动。下图演示了持久化EntityAction(比如Hero实体类型的QUERY\_ALL)的过程。



1 - 视图/组件调用EntityCollectionService.getAll()，它将英雄的QUERY\_ALL EntityAction分派到存储。

2 - NgRx启动…

1. NgRx数据EntityReducer读取操作的entityName属性(本例中为Hero)，并将操作和现有实体集合状态转发给英雄的EntityCollectionReducer。

(2)集合reducer根据操作的entityOp (operation)属性选择开关用例。这种情况下将操作和集合状态处理为一个新的(更新的)英雄集合。

(3)存储区使用更新后的集合更新状态树中的实体缓存。

(4)NgRx observable选择器检测并向视图中的订阅者报告变化(如果有的话)。

3 - 初始的EntityAction然后进入到EntityEffects。

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

5 - 其 effec是将HTTP响应转换为带有英雄的新的成功操作(或者请求失败时的错误操作)。

6 - NgRx效果将该操作分派到存储，该存储重申第2步，即使用英雄更新集合并刷新视图。

***Entity*** ***Metadata***

NgRx数据库在NgRx存储中维护实体收集数据的缓存。

您可以通过Entity Metadat告诉NgRx数据库关于这些集合及其包含的实体的信息。

集合中的实体属于相同的实体类型。每个实体类型都显示为NgRx Data EntityMetadata<T>接口的命名实例。

您可以在EntityMetadataMap中同时为多个实体指定元数据。

下面是一个EntityMetadataMap示例，它类似于演示应用程序中定义了两个实体(Hero和Villain)的元数据的示例。

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*

必须向NgRx Data EntityDefinitionService注册元数据。

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

EntityDataModule.forRoot({

...

entityMetadata: appEntityMetadata,

...

})

如果你在几个不同的、快速加载的Angular模块中定义了实体，你就可以通过多提供程序为每个模块添加元数据。

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

这种技术对延迟加载的模块不起作用。在延迟加载模块到达时，ENTITY\_METADATA\_TOKEN提供程序已经设置并使用了。

模块应该注入EntityDefinitionService，并直接使用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，这些方面告诉NgRx数据库如何管理类型为T的实体数据集合。

类型T是应用程序对该实体的TypeScript表示;它可以是接口或类。

entityName

类型的entityName是惟一必需的元数据属性（metadata property）。它是缓存中实体类型元数据（metadata）的唯一键。

必须为单个EntityMetadata实例指定它。如果您在EntityMetadataMap中省略了它，则映射键将成为本例中所示的entityName。

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”，默认的数据服务将发布到一个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。  
它不擅长多元化。它会为URL生成“api/heros”来获取所有的英雄，因为它盲目地在小写实体名的末尾添加了一个“s”。当然，“hero”的复数形式是“heroes”，而不是“heros”。您将在下面看到如何纠正这个问题。

filterFn

许多应用程序允许用户筛选缓存的实体集合。  
在附带的演示应用程序中，用户可以按名字过滤英雄，也可以按名字或恶棍的话语过滤恶棍。

我们认为这种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功能。  
每个集合的filteredEntities选择器根据用户的筛选标准(保存在存储的实体集合的筛选属性中)将筛选函数应用到集合。

如果没有筛选函数，则filteredEntities选择器与selectAll选择器相同，后者返回集合中的所有实体。  
筛选函数(参见EntityFilterFn)接受实体集合和用户的筛选标准(筛选模式)，并返回所选实体的数组。

下面的示例筛选具有name属性的实体，该属性的值包含搜索字符串。

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

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

}

NgRx数据库包括一个助手函数PropsFilterFnFactory<T>，它创建了一个实体筛选函数，该函数将用户的输入作为不区分大小写的正则表达式处理，并将其应用于实体的一个或多个属性。

演示使用这个助手创建英雄和恶棍过滤器。下面是应用程序如何为恶人创建nameAndSayingFilter函数。

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

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

}

selectId

每个实体类型都必须有一个主键，它的值是整数或字符串。  
NgRx数据库假设实体有一个id属性，它的值是主键。  
不是每个实体都有一个名为id的主键属性。对于某些实体，主键可能是两个或多个属性的组合值。

在这些情况下，指定selectId函数，给定实体实例，该函数返回一个整数或字符串主键值。  
在EntityCollectionReducer测试中，恶棍类型具有一个名为key的字符串主键属性。选择id函数是这样的:

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

sortComparer

NgRx数据库按照特定的顺序保存集合实体。

他实际上是底层NgRx Entity library的一个特性。

默认的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。  
您可能更喜欢以其他顺序维护集合。当您提供sortComparer函数时，NgRx-lib将按照比较器规定的顺序保存集合。  
在演示应用程序中，恶人元数据没有比较器，因此其实体按默认顺序排列。  
英雄元数据有一个sortByName比较器，它按照名字的字母顺序保存集合。

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

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

}

运行演示程序并尝试更改现有英雄的名字或添加新的英雄。  
您的应用程序可以调用selectKey选择器来查看集合的id属性，该属性将返回一个按顺序排列的集合主键值数组。

entityDispatcherOptions

这些选项决定集合的dispatcher的默认行为，该dispatcher将操作发送给 reducers and effects。

dispatcher save命令将在发送相应的HTTP请求(乐观请求)之前或之后(悲观请求)添加、删除或更新集合。调用者可以在可选的isOptimistic参数中指定。如果调用者没有指定，则dispatcher根据默认选项进行选择。

默认值是安全的:乐观值表示删除，悲观值表示添加和更新。你可以在这里推翻这些选择。

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，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来演示。

additionalCollectionState: {

foo: 'Foo',

bar: 3.14

}

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

NgRx数据库为这些属性生成选择器，但无法更新它们。你必须自己创建或扩展现有的reducers。

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，扩展DefaultPersistenceResultHandler并覆盖handleSuccess方法，其目的是解析从DataService接收到的数据，检索额外的属性，然后将其保存到action.payload中。注意，成功操作的默认reducer需要action.payload。数据是实体的数组或实体。这需要在检索附加属性之后设置，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);

// return a factory to get a data handler to

// parse data from DataService and save to 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以从操作中保存附加属性。EntityCollection实例的有效负载

按照前面的步骤，我们已经向action.payload添加了额外的属性。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AdditionalEntityCollectionReducerMethods，它扩展了EntityCollectionReducerMethods。此外，我们需要覆盖方法以匹配您的操作。例如，如果附加属性foo只在queryMany操作中可用(由EntityCollectionService.getWithQuery触发)，我们可以采用这种方法。

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:注册自定义的EntityCollectionReducerMethods和AdditionalPersistenceResultHandler

最后，我们需要注册AdditionalPersistenceResultHandler和

AdditionalEntityCollectionReducerMethods来替换默认实现。

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，有关详细信息，请参阅实体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;

}

}

将AdditionalEntityCollectionReducerMethodsFactory注册到NgModule，

@NgModule({

{

provide: EntityCollectionReducerMethodsFactory,

useClass: AdditionalEntityCollectionReducerMethodsFactory

},

})

现在您可以像另一个EntityCollection级别的属性一样从后端获取foo。

Pluralizing the entity name

NgRx数据DefaultDataService依赖于HttpUrlGenerator为每种实体类型创建传统的HTTP资源名称(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”，默认的数据服务将发布到一个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”的东西。  
HttpUrlGenerator自己不能对实体类型名称进行复数。它委托给一个注入的多元化类，称为Pluralizer。

Pluralizer类有一个pluralize()方法，该方法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，如附加一个's'。这对于恶棍(变成了“恶棍”)，甚至对于公司(变成了“公司”)来说都很好。  
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还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()方法在映射中查找单一实体名称，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。

如果这个方案适用于你，创建一个映射的单复数实体名称的例外情况:

export const pluralNames = {

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

Hero: 'Heroes'

};

然后在配置NgRx数据库时指定此映射。

EntityDataModule.forRoot({

...

pluralNames: pluralNames

})

如果你在单独的Angular模块中定义你的实体模型，你就可以增量地添加一个复数名称映射到多提供者。

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

如果此方案不适用于您，则用您自己的发明替换Pluralizer类。

{ provide: [Pluralizer](https://ngrx.io/api/data/Pluralizer), useClass: MyPluralizer }

Entity Actions

当您调用EntityCollectionService的命令来查询或更新缓存集合中的实体时，EntityAction会向NgRx存储分派一个EntityAction。

Action and EntityAction

一个vanilla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。它的可选负载携带执行操作所需的消息数据。  
EntityAction是NgRx操作的超集。它有额外的属性来指导NgRx数据对操作的处理。这是完整的界面。

EntityAction

export interface EntityAction<P = any> extends Action {

readonly type: string;

readonly payload: EntityActionPayload<P>;

}

EntityActionPayload

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)，则为真.

mergeStrategy - 如何将实体合并到缓存中。看到更改跟踪.

tag? - 要在生成的类型中使用的标记。如果未指定，则entityName是标记.

error? -意外操作处理错误.

skip? - 如果下游使用者应该跳过处理操作，则

type是NgRx唯一需要的属性。它是一个字符串，在可分派到存储的所有操作类型中唯一标识操作。  
NgRx数据不关心type。它关注entityName和entityOp属性。

entityName是实体类型的名称。它标识该操作应用于的NgRx数据缓存中的实体集合。此名称对应于该集合的 [NgRx Data metadata](https://ngrx.io/guide/data/entity-metadata)。实体接口或类名，如“Hero”，是典型的entityName。

entityOp标识要在实体集合上执行的操作，这是entityOp枚举的一个，它对应于NgRx数据可以在集合上执行的近60种不同操作之一。

数据在概念上是消息的主体。它的类型和内容应该符合要执行的操作的要求

可选correlationId吗?是一个可选的序列化对象(通常是GUID)，它关联两个或多个操作，比如启动服务器操作(“获取所有英雄”)的操作，以及服务器操作完成后的后续操作(“got heroes successfully”或“error while getting heroes”)。

可选的合并策略告诉NgRx Data如何将操作的结果“merge”到缓存中。大多数情况下，这是对变更跟踪子系统的指令。

当EntityActionFactory创建这个EntityAction时，可选标记将出现在生成的类型文本中。

entityName是出现在格式化类型中括号之间的默认标签，例如'[Hero] NgRx Data/query-all。您可以设置此标记来标识操作的目的以及分发操作的“who”。Ngrx数据将把您的标记放在格式化类型的括号之间

error属性指示在处理操作时发生了错误。请参阅下面的更多。

skip属性告诉下游操作接收方，它们应该跳过通常的操作处理。这个标志通常是缺失的，并且是隐式假的。请参阅下面的更多。

EntityAction consumers

NgRx数据库忽略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数据EntityReducer根据entityName将操作重定向到EntityCollectionReducer，并根据entityOp处理操作。

如果一个操作的entityOp位于持久化EntityAction的小集合中，则EntityEffects会拦截该操作。entityOp名字。效果为该操作的entityName选择正确的数据服务，然后告诉该服务发出适当的HTTP请求并处理响应。

Creating an EntityAction

如果您愿意，您可以手动创建EntityAction。NgRx数据库认为任何具有entityName和entityOp属性的操作都是EntityAction。

[EntityActionFactory.create()](https://ngrx.io/api/data/EntityActionFactory" \l "create)方法帮助您创建一贯格式良好的EntityAction实例，其类型是由标记(默认为entityName)和entityOp组成的字符串。

例如，查询服务器上所有英雄的默认 [Action.type](https://ngrx.io/api/store/Action" \l "type)是'[Hero] NgRx Data/query-all'。

EntityActionFactory.create()方法调用工厂的formatActionType()方法来生成Action.type字符串。

因为NgRx数据忽略了类型，如果您喜欢另一种格式，您可以用自己的方法替换formatActionType()，或者提供并注入您自己的EntityActionFactory。

*请注意，每种实体类型对于每个操作\_都有自己的\_unique操作，就好像您是手工单独创建它们一样。*

Tagging the EntityAction

一个格式良好的动作类型可以告诉读者什么改变了，谁改变了它。

NgRx数据库不查看EntityAction的类型，只查看其entityName和entityOp。因此，您可以从几个不同的操作中获得相同的行为，每个操作都具有自己的信息类型，只要它们共享相同的entityName和entityOp。

EntityActionFactory.create()方法的可选标记参数使生成有意义的EntityActions变得容易。

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

下面是一个使用injectable EntityActionFactory构造默认的“query all heroes”动作的示例。

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

'Hero',

EntityOp.QUERY\_ALL

);

store.dispatch(action);

由于NgRx数据Effects，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。但是您不能从这些日志条目理解操作的上下文。你不知道是谁策划了这次行动，也不知道为什么。操作类型太泛型了。

您可以通过提供一个标记来创建信息更丰富的操作，该标记可以更好地描述正在发生的事情，还可以更容易地查找代码在何处分派该操作。  
例如,

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

'Hero',

EntityOp.QUERY\_ALL,

null,

{ tag: 'Load Heroes On Start' }

);

store.dispatch(action);

动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。任何具有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”操作。

const action = {

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

entityName: 'Hero',

entityOp: EntityOp.QUERY\_ALL

};

store.dispatch(action);

它通过NgRxdata Effect触发HTTP请求，如前面的示例所示。  
请注意，NgRx data Effects使用EntityActionFactory创建第二个成功操作。如果没有tag属性，它将生成一个通用的成功操作。  
这两种动作类型的日志如下所示:

some/arbitrary/action/type

[Hero] NgRx Data/query-all-success

EntityActions在哪里?

在NgRx Data应用程序中，NgRx Data库为您创建并分派EntityActions。

在调用EntityCollectionService API时，EntityActions在很大程度上是不可见的。您可以看到它们与NgRx store [dev-tools](https://ngrx.io/guide/store-devtools).工具一起工作。

为什么这很重要

在一个普通的NgRx应用程序中，您为存储中的每个状态以及处理这些操作的reducers手工编码每个actions。

它需要许多actions、一个复杂的reducer以及NgRx Effects的帮助来管理查询和保存单个实体类型。

NgRx Data library大大简化了这项工作

*NgRx Data library在内部将大部分繁重的工作委托给了NgRx Entity.。*

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

这些工件只针对缓存的实体集合。

您可以编写多达18个附加动作来支持异步CRUD(创建、检索、更新、删除)操作的典型补充。您必须将它们分派到Store，在那里您将使用更多的reducer方法和Effect来处理它们，您还必须手工编写这些方法和效果。

对于普通的NgRx，您将针对每种实体类型完成这个练习。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 Data创建操作dispatchers,以及响应这些actions、reducers和effects。

EntityAction.error

EntityAction.error属性的出现表明在处理操作时发生了错误。

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

错误的主要用例是捕获reducer异常。如果某个reducers抛出异常，NgRx将停止订阅这些reducers。捕获reducer异常允许应用程序继续操作。

NgRx Data 捕捉由EntityCollectionReducer抛出的错误，并设置EntityAction。捕获的错误对象的错误属性。

当错误action是持久性操作(比如SAVE\_ADD\_ONE)时，error属性非常重要。EntityEffects将看到这样的操作有一个错误，并将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错误操作通过影响触发HTTP请求的唯一方法。*

EntityAction.skip

skip属性告诉下游操作接收方，它们应该跳过通常的操作处理。这个标志通常是缺失的，并且是隐式假的。

当您试图删除未保存的新实体时，NgRx数据集skip=true。当EntityEffects.persist$方法看到EntityAction信封上的这个标志设置为true，它跳过HTTP请求并使用原始请求有效负载分派适当的\_SUCCESS动作。

当您试图删除已添加到集合但未保存的实体时，该特性允许NgRx数据避免发出删除请求。这样的请求在服务器上会失败，因为没有这样的实体可以删除。

*有关变更跟踪的更多信息，请参见EntityChangeTracker页面。*

EntityCache-level actions

有一些操作总体上针对实体缓存。

SET\_ENTITY\_CACHE使用操作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为已知状态。

MERGE\_ENTITY\_CACHE使用操作有效负载中存在的集合替换当前实体缓存中的特定实体集合。它不影响当前的其他集合。

在“EntityReducer”文档中了解它们。

Entity Collection

NgRx数据库为NgRx存储中的每个实体类型维护一个实体集合缓存(EntityCache)。

entity\_collection实现了实体类型的EntityCollection接口。

ids 主键值按默认排序顺序排列

entities 主键到实体数据值的映射

filter 用户的筛选标准

loaded 集合是否被QueryAll填充;强制假后清楚

loading 当前是否在等待来自服务器的查询结果

changeState 启用更改跟踪时，未保存实体的更改属性

EntityCollectionService

EntityCollectionService是NgRx数据分配器和选择器$上的一个facade，用于管理缓存在NgRx store中的实体T集合。

Dispatcher提供将实体操作分派到NgRx存储的命令方法。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数据库使用响应数据分派新操作，这些操作更新实体集合。

EntityCommands接口列出了所有命令以及它们的作用。

您的应用程序调用这些命令方法来更新NgRx存储中缓存的实体集合。

**Selectors**$是返回可观察选择器的属性。每个可观察对象会监视缓存的实体集合中的特定更改，并发出更改后的值。

EntitySelectors$ interface列出了所有预定义的选择器可观察属性，并解释了它们观察的集合属性。

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

下面是来自演示应用程序的HeroesComponent的简化摘录，显示了组件调用命令方法和订阅选择器观察项。

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EntityCollectionServiceFactory并为英雄实体创建一个EntityCollectionService。

*在本指南的后面我们将进入工厂。*

创建EntityCollectionService作为一个类

或者，您可以在其他地方(比如在AppModule中)创建一个HeroEntityService，并将其注入组件的构造函数中。

有两种创建服务类的基本方法。

来自EntityCollectionServiceBase < T >

用您需要的API编写一个HeroEntityService。

当HeroEntityService从EntityCollectionServiceBase<T>派生时，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示例。

当使用有限的API定义HeroEntityService时，您还可以注入EntityCollectionServiceFactory作为您选择公开的功能的源。

让您喜欢的风格和应用程序需要决定您选择的创建技术。

Set component selector$ properties

[selector](https://ngrx.io/api/store/testing/MockSelector" \l "selector)$属性是一个可观察对象，当所选状态属性发生变化时发出该对象。有些人将这些属性称为状态流。

示例组件有两个这样的属性，公开了两个EntityCollectionService [selector](https://ngrx.io/api/store/testing/MockSelector" \l "selector)可观察对象:filteredEntities$和loading$。

filteredEntities$ observable生成一个当前缓存的满足用户筛选条件的英雄实体数组。如果用户改变了过滤器，或者某些操作改变了缓存集合中的英雄，这个观察就会产生一个新的英雄数组。

当数据服务等待来自服务器的英雄时，加载$ observable生成true。当服务器响应时，它会生成false。这个演示程序订阅了loading$，这样它就可以打开或关闭一个可视化的加载指示

*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。*

可观察选择器与选择器函数

[selector](https://ngrx.io/api/store/testing/MockSelector" \l "selector)$ observable(以“$”结尾)不同于同名的、紧密相关的[selector](https://ngrx.io/api/store/testing/MockSelector" \l "selector)函数(没有“$”后缀)

[selector](https://ngrx.io/api/store/testing/MockSelector" \l "selector)是一个从实体集合中选择状态片的函数。当状态发生变化时，[selector](https://ngrx.io/api/store/testing/MockSelector" \l "selector)$ observable会发出那片状态。

通过将选择器函数传递给NgRx select操作符并将其传送到NgRx存储中，NgRx Data创建了一个选择器$ observable，如下例所示:

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

Using selectors$

组件类不订阅这些选择器$属性，但组件模板订阅。模板绑定到它们，并将它们的可观察对象转发给订阅它们的Angular AsyncPipe。下面是filteredHeroes$ binding的一段摘录。

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

...

</div>

调用命令方法

大多数HeroesComponent方法委托给EntityCollectionService命令方法，如getAll()和add()。有两种命令:

触发对服务器的请求的命令。  
仅缓存命令，用于更新被缓存的实体集合。

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。

仅缓存的命令方法是更长的动词，如“addManyToCache”和“removeOneFromCache”，它们的名称都包含“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，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，对实体集合的直接更新将丢失。*

许多EntityCollectionService命令方法都取一个值。该值是键入的(通常为Hero)，因此您不会因传入错误类型的值而犯错误。

在内部，实体服务方法创建一个实体操作，该操作对应于该方法的意图。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。

不变性是redux模式的核心原则。有几个命令方法采用实体参数，比如Hero。实体参数绝对不能是缓存的实体对象。它可以是缓存的实体对象的副本，通常也是如此。演示应用程序总是使用实体数据的副本调用这些命令方法

所有命令方法都返回void。redux模式的一个核心原则是命令永远不会返回值。他们只是做一些有副作用的事情。

与其期望从命令中得到结果，不如订阅反映命令效果的selector$属性。如果命令做了您关心的事情，那么selector$属性应该能够告诉您。

EntityCollectionServiceFactory

NgRx数据EntityCollectionServiceFactory的create<T>()方法生成EntityCollectionServiceBase<T>类的一个新实例，该实例实现了实体类型T的EntityCollectionService接口。

Entity DataService

NgRx数据库希望通过调用类似rest的web api来持久化实体数据，并为每种实体类型提供端点。

EntityDataService维护一个服务类注册表，专门用于持久化特定实体类型的数据。

当NgRx数据库看到针对实体持久性操作的操作时，它向EntityDataService询问注册的数据服务，该数据服务对该实体类型进行HTTP调用，并调用适当的服务方法。

数据服务是实现EntityCollectionDataService的类的实例。这个接口支持实体的一组基本CRUD操作。每个返回可观测的:

add(entity: T) Add a new entity POST /api/hero/

delete(id: any) Delete an entity by primary key value DELETE /api/hero/5

getAll() Get all instances of this entity type GET /api/heroes/

getById(id: any) Get an entity by its primary key GET /api/hero/5

getWithQuery( Get entities that satisfy the query GET /api/heroes/?name=bombasto

queryParams: QueryParams | string)

update(update: Update<T>) Update an existing entity PUT /api/hero/5

*QueryParams是一个参数-名称/值映射，您还可以提供查询字符串本身。HttpClient安全地将两者编码为已编码的查询字符串。  
Update<T>是一个具有严格的实体属性子集的对象。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返回相应的Angular HttpClient方法返回的可观察对象。如果您创建自己的数据服务备选方案，它们应该返回类似的观察结果。

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，同时注册几个数据服务。

*可以创建和导入注册自定义数据服务的模块，如EntityDataService测试所示*

如果您决定注册一个实体数据服务，请确保在要求NgRx data为该实体执行持久性操作之前完成注册。否则，NgRx数据库将为该实体类型创建并注册默认数据服务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DefaultDataService，由注入的DefaultDataServiceFactory为每种实体类型创建。

DefaultDataService<T>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,  
如果持久化操作是删除id=42的英雄，根路径为“api”，实体名称为“hero”，则删除请求URL为“api/hero/42”。

当持久性操作涉及多个实体时，DefaultDataService将用实体类型名称的复数形式替换资源名称。获取所有英雄的QUERY\_ALL操作将导致对URL 'api/heroes'的HTTP get请求。

DefaultDataService不知道如何以复数形式呈现实体类型名称。它甚至不知道如何创建基本资源名。

它依赖注入的HttpUrlGenerator服务来生成适当的端点。HttpUrlGenerator的默认实现依赖于Pluralizer服务来生成复数集合资源名。

[Entity Metadata](https://ngrx.io/guide/data/entity-metadata" \l "plurals)指南解释了如何配置默认的Pluralizer。

配置DefaultDataService

集合级别的数据服务为HTTP调用构造它们自己的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，用于将共享配置传输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，例如https://my-api-domain.com:8000/api/v1。

对于DefaultDataService<T>，默认值是'api'，这将导致url如api/heroes。

timeout属性设置了ng-lib持久性操作放弃接收服务器应答并取消操作之前的最大时间(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，不找到要删除的资源是无害的，您可以通过将这个标志设置为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，创建一个自定义配置对象的类型DefaultDataServiceConfig:

const defaultDataServiceConfig: DefaultDataServiceConfig = {

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

timeout: 3000, // request timeout

}

在eagerly-loaded 加载的NgModule中提供它，例如示例应用程序中的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，但是您可能希望进一步定制在为特定集合保存或检索数据时发生的情况。

例如，您可能需要修改已获取的实体，以将字符串转换为日期，或向实体添加其他属性。  
您可以通过创建一个自定义数据服务并将该服务注册到EntityDataService来实现这一点。

为了说明这一点，示例应用程序向Hero实体添加了一个dateLoaded属性，以便在英雄从服务器加载到NgRx-store实体缓存时进行记录。

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。  
在示例应用程序中，HeroDataService派生自NgRx数据DefaultDataService<T>，以便利用它的基本功能。它只是凌驾于它真正需要的之上。

@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与get操作挂钩来设置英雄。对获取的英雄实体进行数据加载。它还会告诉记录器创建它的时间(参见运行示例的控制台输出)。  
最后，我们必须告诉NgRx Data关于这个新的数据服务的信息。

示例应用程序提供了HeroDataService，通过调用应用程序实体存储模块中EntityDataService上的registerService()方法来注册:

@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，用您选择的任何机制查询和保存实体。  
您可以以同样的方式注册它，只要它符合接口。

// Register custom data service

entityDataService.registerService('Hero', peculiarHeroDataService);

Entity Effects

****Work in Progress****

Effects效果是引发副作用的一种方式的行动。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。您可以在NgRx Effects库的帮助下实现一个或多个Effects。

可以通过您的effect方法检测和处理分配给NgRx store的操作。在处理之后，不管是同步的还是异步的，您的方法都可以将新的操作分派到store

NgRx数据库在其EntityEffects类中实现了名为persist$的效果。

persist$ 方法筛选某些EntityAction.op的值。这些值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(无论响应良好还是出现错误)，persist$方法将使用适当的响应数据向存储分派新的EntityActions。

****Cancellation****

您可以通过调度带有要取消的持久性操作的相关id的CANCEL\_PERSIST EntityAction来尝试取消保存。EntityCache.cancel$监视此操作，并通过管道连接到EntityCache.persist$，在这里它可以尝试取消实体收集查询或保存操作，或至少阻止服务器响应更新缓存。

*对于保存操作来说，这并不是一个好主意。您无法告诉服务器取消这种方式，也无法知道服务器是否保存了该文件。也不能指望在客户端接收到服务器响应并在服务器或缓存上应用更改之前处理取消请求。*

*如果在服务器结果到达之前取消，EntityEffect将不会尝试用延迟到达的服务器结果更新缓存的集合。该Effects将发出CANCELED\_PERSIST操作。EntityCollection reducer会忽略此操作，但您可以在存储操作中侦听该操作，从而知道取消操作在客户机上生效。*

Entity Reducer

Entity Reducer是存储的实体缓存中所有实体集合的主reducer。

该库没有命名实体reducer类型。相反，它依赖EntityCacheReducerFactory.create()方法来生成该reducer，该reducer是一个NgRx ActionReducer<EntityCache, EntityAction>。

这样的reducer函数接受EntityCache状态和EntityAction动作，并返回EntityCache状态。

该reducer响应entitycache级操作(罕见)或针对实体集合的EntityAction(通常情况)。所有其他类型的动作都被忽略，并且reducer简单地返回给定的状态。

*该reducer专门筛选操作的entityType属性。它将任何具有entityType属性的操作视为EntityAction。*

实体reducer的主要工作是

从状态中提取操作的实体类型的EntityCollection。  
如果需要，创建一个新的初始化的实体集合。  
获取或创建该实体类型的EntityCollectionReducer。  
使用集合和动作调用实体收集reducer。  
用实体收集reducer返回的新收集替换EntityCache中的实体收集。

EntityCollectionReducers

EntityCollectionReducer将操作应用于NgRx存储区中保存的EntityCache中的EntityCollection。

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减速器，它就会在注入的EntityCollectionReducerFactory的帮助下创建一个，并注册这个减速器，以便下次可以再次使用它。

****Register custom reducers****

您可以为实体类型创建定制的reducer并直接向entitycollectionreducerregister . registerreducer()注册它。

通过调用EntityCollectionReducerRegistry.registerReducers(reducerMap)，你可以同时注册几个自定义的reducers，其中的reducerMap是一个由reducers散列组成的，以实体-类型-名称为键值。

默认EntityCollectionReducer

EntityCollectionReducerFactory根据应用程序的[entity metadata](https://ngrx.io/guide/data/entity-metadata).创建一个默认的reducer，它利用NgRx EntityAdapter的功能

默认的reducer根据EntityAction.op决定要做什么。它期望该属性的字符串值是EntityOp枚举的一个成员。

许多EntityOp值被忽略;该reducer只返回给定的实体集合。

例如，某些面向持久化的操作应该由NgRx [persist$ effect](https://ngrx.io/guide/data/entity-effects)来处理。它们不更新收集数据(除了翻转加载标志之外)。另一些则从集合中添加、更新和删除实体。

*请记住，不可变对象是redux/NgRx模式的核心原则。这些还原器实际上不会改变原始集合或其中的任何对象。它们对集合进行复制，并且只更新集合内对象的副本。*

请参阅NgRx实体EntityAdapter收集方法，了解由默认实体收集reducer执行的缓存更改操作的基本指南。

EntityCollectionReducerFactory类及其测试是关于默认的reducer实际如何工作的权威。

Initializing collection state

NgRxDataModule向NgRx数据存储添加一个空的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，它将在注入到EntityCacheReducerFactory中的EntityCollectionCreator的帮助下创建一个新的初始化的集合。

创建器从实体的EntityDefinition中的initialState返回一个初始化的集合。如果实体类型没有定义，或者定义没有initialState属性值，创建者将返回EntityCollection。

然后，实体回收程序在实体回收程序的state参数中传递新收集。

自定义实体reducer行为

通过注册自定义替代，可以替换任何实体集合reducers程序。

通过提供EntityCollectionReducerFactory的自定义替代，可以替换默认的实体reducer。

您甚至可以通过提供EntityCacheReducerFactory的自定义替代方案来替换主实体reducer。

但通常您会希望使用一些在此之前或之后运行的附加的reducer逻辑来扩展集合reducer。

EntityCache-level actions

有一些操作总体上针对实体缓存。

SET\_ENTITY\_CACHE使用操作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为已知状态。

MERGE\_ENTITY\_CACHE使用操作有效负载中存在的集合替换当前实体缓存中的特定实体集合。它不影响当前的其他集合。

*看到entity-**reducer.spec.ts是这些动作的例子。*

这些操作可能是支持离线场景或同时回滚对多个集合的更改计划的一部分。

您可以订阅EntityServices.entityCache$选择器。当缓存更改时，可以将缓存序列化到浏览器本地存储。你可能想要崩溃几秒钟，以减少搅动。

稍后，在重新启动应用程序时，可以调度SET\_ENTITY\_CACHE操作来初始化实体缓存，即使在断开连接的情况下也是如此。或者您可以分派MERGE\_ENTITY\_CACHE将选择的集合回滚到已知状态，就像错误恢复或“假设”场景中那样。

*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。它并不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。可能会重新考虑并在将来这么做。*

如果您想创建和减少额外的缓存范围的操作，请考虑下一节中描述的EntityCache MetaReducer。

MetaReducers

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

MetaReducers是接受一个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MetaReducers返回的reducer，就像它调用任何reducer一样。它使用状态对象和动作来调用它。

MetaReducers接受者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、对更新状态进行后处理，或者执行上述所有操作。

*记住，动作本身是不可变的。不要改变动作!*

与每个reducer一样，传递给MetaReducer的reducer的状态仅是该reducer作用域内的存储区部分。

NgRx数据支持两个层次的元转换器  
EntityCache MetaReducer，作用域为整个实体缓存  
EntityCollection元areducer，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元ducer来同时更新多个集合。

EntityCache MetaReducer reducer必须满足三个要求:  
总是返回整个实体缓存。  
同步返回(不等待服务器响应)。  
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进行。目前，请参阅NgRx数据 entity-data.module.spec.ts为例子。

Entity Collection MetaReducers

实体集合MetaReducer接受实体集合reducer作为其reducer参数，并返回一个新的实体集合reducer。

新的reducer接收原本会转到原始reducer的EntityCollection和EntityAction参数。

它可以用这些参数做它想做的事情，例如:  
记录操作，将操作转换为不同的操作(对于相同的实体集合)，调用原始的reducer，对原始reducer的结果进行后处理。

新的实体收集reducer必须满足三个要求:

总是为同一实体返回EntityCollection。  
同步返回(不等待服务器响应)。  
不要改变原来的动作;克隆它来改变它。

**与存储元转换器相比(Compared to Store MetaReducers)**

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

该store MetaReducer主要针对的是reducers。它包装store reducers，查看所有操作，并可以更新其作用域内的任何状态。

但是Store MetaReducer 既不看到也不包装实体收集reducer。这些实体集合reducers是注册到NgRx数据特性的EntityCache reducers的内部。

entity collection MetaReducer只专注于单个目标实体集合的操作。它封装了所有实体集合reducer。

*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。如果您需要一个交叉收集的元数据转换器，请尝试上面描述的EntityCache元数据转换器。*

**Provide Entity****MetaReducers to the factory**

创建一个或多个实体集合MetaReducers，并将其添加到数组中。

这个数组提供ENTITY\_COLLECTION\_META\_REDUCERS注入标记，在其中导入NgRxDataModule。

EntityCollectionReducerRegistry将注入元元转换器并将MetaReducers数组组合到单个MetaReducers中。前面的MetaReducers将后面的MetaReducers封装在数组中。

当工厂注册EntityCollectionReducer(包括它创建的简化器)时，它在将该简化器添加到其注册表之前将其包装在meta-MetaReducer 中。

分派到存储的所有EntityActions在进出特定于实体的简化器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元转换器。现在，请参阅 entity-reducer.spec.ts 为例子。*

EntityServices

EntityServices是NgRx数据服务和NgRx数据EntityCache上的外观。

Registry of EntityCollectionServices

它主要是EntityCollectionServices的注册中心。

调用它的EntityServices.getEntityCollectionService(entityName)方法来获取该实体类型的单例EntityCollectionService。 这是一个组件。

@Component({...})

export class HeroesComponent implements OnInit {

heroesService: EntityCollectionService<Hero>;

constructor(entityServices: EntityServices) {

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

}

}

如果EntityCollectionService服务还不存在，则EntityServices创建该服务的默认实例，并注册该实例以供将来引用。

Create a custom EntityCollectionService

您经常会创建带有附加功能和便利成员的自定义EntityCollectionService类，正如在EntityCollectionService文档中解释的那样。  
这是一个例子。

@Injectable()

export class HeroesService extends EntityCollectionServiceBase<Hero> {

constructor(elementsFactory: EntityCollectionServiceElementsFactory) {

super('Hero', elementsFactory);

}

// ... your special sauce here

}

当然，你必须在使用它之前提供定制服务，通常在Angular NgModule中。

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

@NgModule({

imports: [...],

declarations: [...],

providers: [HeroesService]

})

export class HeroesModule {}

下面的替代示例使用首选的“****tree-shakable****”Injectable()在根模块中提供服务。

@Injectable({ providedIn: 'root' })

export class HeroesService extends EntityCollectionServiceBase<Hero> {

...

}

可以将该自定义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$;

}

...

}

到目前为止还没有什么新东西。但是我们希望能够从EntityServices.getEntityCollectionService()中获得HeroesService，就像我们获得默认的收集服务一样。  
当应用程序有很多收集服务时，这种一致性就会得到回报

Register the custom EntityCollectionService

当您向EntityServices注册自定义EntityCollectionService的实例时，EntityServices. getentitycollectionservice()的其他调用者将获得该自定义服务实例。

您希望在任何东西试图通过EntityServices获取它之前完成此操作。  
一种解决方案是将自定义集合服务注入提供它们的模块的构造函数中，并在那里注册它们。  
下面的示例进行了演示。

@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]);

}

}

为了方便应用程序类，子类EntityServices

另一个有用的解决方案是创建EntityServices的子类，它注入自定义集合服务并为应用程序添加便利成员。  
下面的AppEntityServices进行了演示。

@Injectable()

export class AppEntityServices extends EntityServicesBase {

constructor(

elements: EntityServicesElements,

// Inject custom services, register them with the EntityServices, and expose in API.

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，并将其直接传递给基类构造函数。“elements”包含基类创建和管理元数据中描述的实体所需的成分。

然后，它注入两个自定义集合服务HeroesService和VillainsService，并将它们直接暴露给消费者，作为访问这些服务的方便属性

在本例中，我们不需要为SideKick实体定制集合服务。默认服务就可以了。

尽管如此，我们还是添加了一个sideKicksService属性来获取或创建SideKick的默认服务。使用者会发现这比getEntityCollectionService()更容易发现和调用。

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

接下来，在Angular NgModule中提供AppEntityServices作为它自己(AppEntityServices)和EntityServices的别名。

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

在示例应用程序中可以看到它。

@NgModule({

imports: [ ... ],

providers: [

AppEntityServices,

{ provide: EntityServices, useExisting: AppEntityServices },

...

]

})

export class EntityStoreModule { ... }

访问多个EntityCollectionServices

复杂组件可能需要访问多个实体集合。即使EntityCollectionServices是针对每种实体类型定制的，EntityServices注册中心也可以简化这一点。  
您只需要一个注入的构造函数参数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的特定于应用程序的子类，比如上面的AppEntityServices，使这一点更好一些。

@Component({...})

export class CharacterContainerComponent implements OnInit {

heroes$: Observable<Hero>;

...

constructor(private appEntityServices: AppEntityServices) {

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

...

}

...

}