**<https://blog.strongbrew.io/Redux-best-practices/>**

基本的最佳实践  
虽然下面的列表对于有经验的Redux开发人员来说可能是常识，但是为了完整起见，让我们对它们进行总结。

1 - 我们的应用程序只能计算一个store，否则它会变得太复杂

2 - Reducers必须是纯的，这是函数编程的原则，它使函数可预测并避免副作用

3 - 不可变的数据结构都是非常重要的优化变化检测周期,避免意想不到的行为,因此reducers应该以一个不可变的方式处理数据

4 - reducers总是返回一个值!因此，不要忘记实现switch语句的默认情况，以返回原始状态

## Don’t add models to the store

模型可以看作是一个具有功能的javascript对象，如下面的例子:

class User{

constructor(private firstName: string, private lastName:string){

}

get fullName(): string {

return `${this.firstName} ${this.lastName}`;

}

}

虽然由Dan Abramov编写的Redux包禁止将这些原型对象作为有效负载发送，但@ngrx/store还没有禁止它。然而，这是一种糟糕的做法，因为它给存储增加了很多复杂性，而且由于处理数据的不可变方式，模型损坏的可能性很大。检查这个例子的实例:

const user = new User('Brecht', 'Billiet');

console.log(user.fullName); // Brecht Billiet

const updatedUser = {...user, lastName: 'Doe'};

console.log(updatedUser.fullName); // undefined

由于我们以不可变的方式更新了用户，它创建了一个新的引用，因此它的所有功能都丢失了。这正是我们的简化者对流入他们的数据所做的。所以在动作中发送有效负载时，总是发送普通对象。

模型的另一种方法是使用接口。接口很棒，因为它只在编译时进行解释，在运行时不使用内存，而且非常有表现力。如果您的模型不需要计算(如前面示例中的get fullName())，那么建议使用接口。

使用接口可以让我们利用一些TypeScript特性，比如Pick<T>。使用Pick<T>，我们可以从接口创建类型，这在某些情况下非常方便，比如创建一个模型和一个拥有更少属性的记录:

export interface User {

id: number;

name: string;

middleName: string;

lastName: string;

fullName: string;

address: string;

city: string;

state: string;

zip: string;

}

export type UserRecord = Pick<User, 'id' | 'fullName'>;

如果获取用户列表的端点返回的是部分对象而不是完整对象，那么我们可以创建一个新类型，而不是两个类或接口。如果您的模型为UI做好了准备，那么这是一种更干净、更容易维护的方法。

我们在store里放什么?

我们不应该仅仅因为我们有能力就把东西放在store里。我们需要思考，这里需要什么状态，为什么。例如，组件之间共享的状态有时可以保存在父组件中。我们称之为内部状态:组件保持自己的状态，组件本身负责。如果组件状态不影响应用程序状态中的任何内容，则不需要在应用程序状态或触摸redux上。

但是，当需要在不同根组件之间共享状态时(在路由器-outlet中呈现)，我们可能希望将该状态保存在存储中。

当我们在浏览应用程序时需要记住一个值时，我们也可以把它放到存储中。这里的一个例子是:记住侧边栏是否被折叠，这样当我们导航到带有侧边栏的页面时，它仍然会被折叠。

我们也可能希望将复杂状态放到存储中，因为Redux可以以一种优雅的方式处理复杂状态管理。这里的一般经验法则是，只保留共享状态、我们想要记住的值和存储中的复杂状态。如果不需要，就不要在存储中添加状态，这会导致不必要的样板和复杂性。

话虽如此，还有两个原因，我们可能想要添加额外的状态到store:

1 当我们想让我们的应用程序实时。看看我们是如何用6行代码实现我们的应用实时化的。

2 当我们想进行乐观更新时。检查Angular2和Redux中可取消的乐观更新

不要忘记路由器参数

一个常见的错误是将可以很容易添加到url中的东西放在存储中。在url中保存状态的好处是:

我们可以使用浏览器导航按钮  
我们可以将url添加到书签中  
我们可以和其他人分享这个url

如果我们能把简单的东西放到url中，我们至少应该考虑它。

避免巨大的列表

Redux不能被视为本地内存数据库，因此出于性能原因，我们不能将所有数据放入存储中。Redux可以看作是应用程序在特定时间需要的状态和数据的抽象。

例如，如果我们有一个10000名用户的列表，我们不想把他们都放在存储中。我们可以做的是跟踪存储中包含500个用户的列表，用户可以在特定时间看到该列表，并在后台加载更多用户并更新缓冲的列表。

设计状态

设计应用程序的状态是一个重要的步骤，我们建议首先在白板上绘制该状态。这里最重要的规则是:尽可能保持状态平坦Keep the state as flat as possible

最常见的一种不良做法是将state深深嵌在一些变得相当复杂的事情中:

/ this is an example of how not to design state

export interface ApplicationState {

moduleA: {

data: {

foo: {

bar: {

users: User[],

cars: Car[]

}

}

}

}

}

// keeping it flat makes the application way easier

export interface ApplicationState {

users: User[],

cars: Car[]

}

我并不是说你不能改变状态，我是说我们在这样做的时候必须非常小心。这里的经验法则是:尽可能保持状态平坦，如果我们想在@ngrx/store中组合状态，我们可以使用特性模块简化器并延迟加载它们，就像我们在特性模块状态组合中看到的那样。

Make everything readonly

我们已经讨论了为什么我们需要不可变的工作，但是我们如何实现这一点呢?Typescript附带一个readonly关键字，我们可以使用它来使属性为readonly

type User = {

readonly firstName: string;

readonly lastName: string;

}

const user: User = {firstName: 'Brecht', lastName: 'Billiet'};

user.lastName = 'Doe';//cannot assign to 'lastName'

// because it is a constant of read-only property

肯定会确保我们不会意外地更新减少器中的属性。我们必须为每个属性写只读，这确实很糟糕。很酷的是typescript为我们提供了一种叫做“高级类型”的东西，我们可以这样做:

// ：通过使用只读<>高级类型，可以对类型内的所有属性进行类型划分

// 默认情况下是只读的

type User = Readonly<{

firstName: string;

lastName: string;

}>;

Action design

Actiontypes

操作类型应该是一个字符串，它解释该操作在存储中应该更改什么。保持这些字符串一致。不要让actiontype太长，保持简短和清晰。

Payload design

// This is bad (inconsistent with the rest of the actions)

class UpdateUserAction implements Action {

type = UPDATE\_USER;

payload: User;

constructor(user: User){

*this.payload = user;*

}

}

// This is better

class UpdateUserAction implements Action{

type = UPDATE\_USER;

payload: {user: User};

constructor(user: User){

*this.payload = {user}*;

}

}

Reducer design

Destructuring the payload

如果我们想让这个reducer代码更容易读，更短，我们可以使用javascript析构。这可能是个人偏好，但它确实让我们的减速器更容易阅读。以这个例子为例:

function usersReducer

(state: User[], action: UserActions): User []{

switch(action.type) {

case 'SET\_USER\_ADDRESS':

return state.map(v =>

v.id === action.payload.user\_id ?

*{...user, address: action.payload.address}* :

v

)

}

}

action.payload。代码返回几次，导致更长的代码线。下面的代码可能更容易读:

function usersReducer

(state: User[], action: UserActions): User []{

switch(action.type) {

case 'SET\_USER\_ADDRESS': {

*const {user\_id, address} = action.payload;*

return state.map(v =>

v.id === user\_id ?

*{...user, address}* :

v

)

}

}

}

如我们所见，已经使用析构将有效负载的属性提取到变量中。清洁对吧?假设我们的动作在有效载荷上有5个甚至更多的属性。在这种情况下，这肯定会有帮助。这里需要注意的是，case实现包装在一个block语句中。这一点很重要，因为我们的reducer可以为不同的操作使用相同的有效负载属性。

这意味着user\_id和address在其他case语句中不可用，这正是我们想要的。

不要在Reducer 中编写业务逻辑

reducer不应该包含业务逻辑，它们是用来以不可变的方式处理状态的。我们不会在缩减器中编写业务逻辑，因为:

它会变得非常复杂  
业务逻辑与状态管理无关  
我们有这样的服务

Child reducers

当reducer需要更新树中几个级别的状态时，它会变得很复杂。以这个例子为例:

type User = {

id: string;

contracts: Contract[];

}

type Contract = {

id: string;

assignees: Assignee[];

}

type ApplicationState = {

users: User[];

}

如果我们将所有逻辑添加到特定用户的特定契约中，代码将难以阅读。检验以下代码:

// This is bad

function usersReducer

(state: User[], action: UserActions): User []{

switch(action.type) {

case 'ADD\_USER\_CONTRACT\_ASSIGNEE': {

const {user\_id, contract\_id, assignee} = action.payload;

return state.map(v =>

v.id === user\_id ?

{

...user,

contracts: user.contracts.map(contract =>

contract.id === contract\_id ?

{

...contract,

assignees: [...contract.assignees, assignee]

} :

contract

)

} :

v

)

}

default:

return state;

}

}

当减速机变得复杂时，将减速机分成子减速机是一个好主意。检查前面例子的重构版本:

// This is better

function usersReducer

(state: User[], action: UserActions): User []{

switch(action.type) {

case 'ADD\_USER\_CONTRACT\_ASSIGNEE': {

const {user\_id, contract\_id, assignee} = action.payload;

return state.map(v =>

v.id === user\_id ?

{

...user,

contracts: contractsReducer(contracts, action.payload)

} :

v

)

}

default:

return state;

}

}

function contractsReducer

(state: Contract[], action: UserActions): Contract []{

switch(action.type) {

case 'ADD\_USER\_CONTRACT\_ASSIGNEE': {

const { contract\_id, assignee} = action.payload;

return state.map(v =>

v.id === contract\_id ?

{

...contract,

assignees: [...assignees, assignee]

} :

v

)

}

default:

return state;

}

}

正如我们所看到的，我们已经将契约的处理提取到它自己的减速器中，它遵循与常规减速器完全相同的原则。  
这个示例变得更容易阅读，更易于维护。在遍历复杂数据结构时，减速机嵌套是管理状态的一种真正优雅的方式

Testing

因为还原器是纯函数，所以对它们进行单元测试非常容易。我们不需要模拟任何依赖项，只需要测试reducer返回的值。  
我们还可以使用deepfreeze来冻结减速机意外突变数据时测试将抛出错误的状态。深度冻结只不过是一个递归对象

describe('reducer: usersReducer', () => {

describe('case UPDATE\_USER', () => {

it('should return a new instance with the correct state',

() => {

const initialState = [new User('1'), new User('2')];

// deepfreeze makes sure the reducer

// doesn't mutate anything by accident

deepfreeze(initialState);

const user = new User('2');

const action = new UpdateUserAction(user);

const newState = usersReducer(initialState, action);

// check if the result of the array is a new ref

expect(newState).not.toBe(initialState);

// check if the result of the user is a new ref

expect(newState[1]).not.toBe(initialState[1]);

// check if the user got updated automatically

expect(newState[1]).toEqual(user);

});

});

});

*注意:不要忘记测试默认操作*

将redux从表示层解耦

在我们的应用程序中到处注入store不是一个好主意。我们想要创建一个Angular、Vue或React应用程序。不是Redux申请。

因此，我们可以考虑以下最佳实践:

组件不需要知道我们在使用Redux，不需要将存储注入到它们中。

服务通常不需要知道我们在使用Redux，不需要将存储注入其中。

我们希望能够不费很大力气就从应用程序中重构Redux

因此，我们希望在表示层和状态管理层之间有某种抽象层。

如何抽象出状态管理层可以在以下两篇文章中读到:可伸缩的angular架构和可伸缩的angular架构第2部分。这是我们在客户中使用的一种架构，它对我们非常有效。

Redux作为消息总线，而Redux作为状态管理层

这可能是个人偏好，但我喜欢使用Redux作为纯状态管理层。是的，有一些像@ngrx/effects这样的工具，我们可以将操作发送到应用程序，这些操作不仅执行状态管理，还会执行XHR调用。

这种方法的好处是我们使用了某种消息传递总线。然而，我更喜欢尽可能地保持简单和抽象的回报。因此我不使用@ngrx/effects，只使用Redux来更新状态片段并消耗这些片段。我认为Redux不应该用来执行后端调用，也不应该用来决定何时进行乐观的更新。我通常用这种方式处理乐观的更新https://blog.strongbrew.io/Cancellable-optimistic-updates-in-Angular2-and-Redux/。

话虽如此，我不会把我的方法称为最佳实践，但这是真正考虑我们想要哪种方式的最佳实践。

结论  
RxJS给我们很大的控制HTTP调用!如果我们知道错误处理是如何工作的，就可以轻而易举地将HTTP调用提升到下一个层次。这不仅适用于typeahead搜索，也适用于我们将现有流与受错误影响的流(如HTTP)组合在一起的每个可观察对象。例如:这也可以发生在NgRx效果或angular路由器。  
在那里，我告诉过你它会很短，我希望你学到了一些东西。  
另外，请务必检查本文:使用指数后退时RxJS的强大功能