**https://blog.strongbrew.io/managing-state-in-angular/**

介绍  
在本文中，我们将了解应用程序中的不同类型的状态，以及这些状态可能存在的位置。

状态是什么？  
状态基本上是定义用户将要使用的UI的所有东西。状态可以是按钮是否可见，它可以是按钮点击的结果也可以是来自API的用户数组。状态可以存在于整个应用程序的不同位置。某些状态是特定于某个组件的，而其他状态可能在应用程序的不同部分共享。一个状态块可以是一个单实例，其中另一个状态块可以共享可以随时销毁的组件有限的生命周期。

状态可能是什么样子，它会存在多久，它从哪里来，这些很大的变化导致了我们需要管理的复杂性。

什么是状态管理?  
状态管理是在应用程序中添加、更新、删除和读取状态的概念。当我们有深度嵌套的数据结构，并且希望更新树中的某个特定部分时，可能会变得复杂。在这种情况下，我们有包含存储的状态管理库，它可以帮助我们进行状态管理，从而摆脱复杂性。快速提醒一下，我们必须小心这些库不会因为过度使用而增加复杂性。

*状态反应*

将状态管理与响应式编程结合起来是开发单页面应用程序的一种非常好的方法。无论我们关注的是Angular、Vue还是React，将这两个原则结合起来将会产生更可预测的应用程序。

现在，状态与反应性编程有什么关系呢?一个状态可以随着时间而改变，所以在某种程度上我们正在等待新的状态的改变。这使得它是异步的。

让我们举个例子:

// false------true-----false---true...

sidebarCollapsed$ = this.state.sidebarCollapsed$

sidebarCollapsed$流开始是假的，后来它变成真，如此等等。这条小溪继续流淌。在Angular中，这种状态可以通过 [async pipe](https://angular.io/api/common/AsyncPipe)（异步管道）很容易地消耗:

<my-awesome-sidebar \*ngIf="sidebarCollapsed$|async">

</my-awesome-sidebar>

异步管道将订阅sidebarcollapse $，将其传递给组件，标记为检查，并在组件被销毁时自动取消订阅。将状态保持在observer模式中很好，因为我们可以订阅这些更改。哦，我提到过它和Angular一起玩得超棒吗?

我们可以使用支持可观察性的BehaviorSubject或状态管理框架。这里有一些真正伟大的支持观察: [Ngrx](https://ngrx.io/) ； [Akita](https://netbasal.gitbook.io/akita/) ； [Ngxs](https://github.com/ngxs/store)

*不变性和单向数据流*

在深入了解状态之前，有两个重要的原则是我们在管理状态时应该遵循的。第一个原则是不变性，这意味着我们不应该在不创建对象的新引用的情况下直接改变数据。如果我们直接改变数据，我们的应用程序将变得不可预测，并且很难跟踪错误。当我们以不可变的方式工作时，我们还可以利用像变更检测这样的性能策略。OnPush from Angular或React的PureComponent。

当我们使用typescript时，我们可以强制typescript编译器在我们改变数据时发出抱怨

type Foo = {

readonly bar: string;

readonly baz: number;

}

let first = {bar: 'test', baz: 1};

first.bar = 'test2'; // compilation error

first = {...first, bar: 'test2'}; // success

在上一个示例中，我们用一个具有更新过的bar属性的完整的新实例覆盖了第一个实例。

数组可以这样处理:

let arr = ['Brecht', 'Kwinten'];

arr.push('John'); // 坏的:arr发生突变

arr = [...arr, 'John']; //很好，arr得到了新的参考

数组原型还有一些很好的辅助函数，我们可以使用它们来增强不可变性，比如map()和filter()，但这不在本文的讨论范围之内。

第二个原则是单向数据流。简而言之，这意味着我们永远不应该对状态使用双向数据绑定。它是负责更新特定状态的绝对所有者(当然是不可变的)。

Redux模式高度执行这两个原则。

*有哪些状态?*

### *1 - Router state*

经常被遗忘，但这是web应用程序可能拥有的最重要的状态之一。将状路由到态放中，有以下优点:

我们可以使用浏览器导航按钮  
我们可以将状态添加到书签中  
我们可以将带有状态的url复制粘贴到其他用户  
我们不需要管理它，它总是在路由上

*提示:与其使用userDetailModalVisible属性来处理模式，为什么不享受上面提到的所有好处，并将其绑定到user /:userId路由呢?在Angular中使用子路由-outlet可以让这一切变得轻而易举，就像我们在这个代码片段中看到的那样。*

*<table>*

*<!--contains users -->*

*</table>*

*<router-outlet>*

*<!--* 用户细节模态呈现在这里 *-->*

*</router-outlet>*

### *2 - Component state*

每个组件都可以包含状态。该状态可以与它的哑组件共享，或者可以在组件本身中使用。例如:当ItemComponent有一个属性selectedItems，它是一个id数组，并且该数组从未在其他组件(不是该组件的子组件)中使用，我们可以认为它是组件状态。它属于那个组件，因此组件应该对它负责。子组件可以使用该状态，但不应该改变它。这些组件可以通知负责它的父组件，父组件可以以不可变的方式更新它。有关智能组件和哑组件的更多信息，请查看这里。

就个人而言，我尽量避免使用状态管理框架来管理组件状态，因为管理该状态是组件的职责。然而，使用状态管理框架来管理组件状态也有很好的理由:

当状态管理非常复杂的时候  
如果我们想做乐观的更新  
如果我们想用它来做实时的事情

如果组件的状态管理变得有点过于复杂，而我们现在又不想使用状态管理框架，那么我们可以在组件本身中使用状态reducer。

### *3 - Persisted state*

持久化状态是用户在不同页面之间导航时记住的状态。这可以是侧边栏是否被折叠，或者当用户返回到一个有很多过滤器的网格时，他希望这些过滤器被记住并在返回时重新应用。另一个示例是具有不同步骤的向导，每个步骤都需要持久化，以便用户可以来回导航，最后一页是所有这些步骤的结果。

持久化状态是一种我们通常使用状态管理框架的状态类型，也就是说，如果我们不想依赖于外部依赖，我们也可以在Angular服务中管理它，它可以是一个单例，在整个应用程序中共享。如果该服务变得过于复杂，或者需要管理很多状态，我会考虑将该状态放入状态管理框架中。

### *4 - Shared state*

当我们讨论共享状态时，我们讨论的是需要在应用程序的不同部分之间共享的状态。在不同智能组件之间共享的状态。这意味着此状态块的实例应该位于比需要使用它的组件更高的级别上。

共享状态可以在状态管理框架(如Redux, Ngrx, Akita, Ngxs等)中管理，但如果状态很小很简单，我们也可以手动管理。假设我们想要一个国家的Observable（ 可观察对象），我们需要在整个应用程序中共享这些国家。在Angular中，我们可以有一个CountryService，它从API获取国家一次，然后在整个应用程序中共享它。为此，我们可以使用RxJS中的shareReplay操作符。

export class CountryService {

...

countries$ = this.httpClient.get('countries').pipe(shareReplay(1));

}

简单吧，一行代码?!为此，我们不需要状态管理框架，尽管它也有它的好处。有些开发人员喜欢将他们所有的主数据保存在一个Redux存储中，这很好。只要知道我们不必这么做。我喜欢尽可能地根据KISS原则(保持简单愚蠢)进行开发，所以我很多次都喜欢这个方法。考虑一下这种方法节省的代码行数。注意，我们编写的每一行代码，不仅需要编写，而且需要维护。

*Which state needs to be managed?*

现在我们知道了什么是状态，我们必须问自己需要管理哪个状态，以及在哪里管理该状态?在组件、单例服务还是框架(存储)中?

这是强烈观点浮现的部分。我建议使用对你和你的团队有效的方法，并认真思考，但这里是我个人的意见:

我尽量避免使用状态管理框架。RxJS已经在很多方面利用了我们，我喜欢KISS。  
我尽量避免使用状态管理框架来与应用程序中的不同部分通信，我认为状态与通信无关。  
当我的组件可以处理状态且不太复杂时，我让组件负责管理该状态。

国家等主数据在使用shareReplay操作符的服务中公开。  
如果没有人使用getById API调用的结果，除了请求它的组件之外，我不会将该结果放入存储中  
我在我的智能组件和我的商店/服务之间使用虚包，以使将来的重构更容易。

然而，也有一种流行的观点，把所有的东西都放在store里，这有以下优点:

我们可以在devtools中看到代码流  
一致的模式  
我们可以通过记忆利用选择器  
更容易的实时应用程序  
乐观的更新更容易

然而，也有一些缺点:

大量的膨胀代码:更大的包大小，更多的维护和开发时间。如果我们要使用完整的Ngrx模式的国家$的例子，我们将不得不写一个:行动，行动类型，效果和一个减缩。

与未来难以摆脱的强烈依赖紧密相连

通常通常更复杂

用户的屏幕可以脱离后端同步

缓存失效:如果我们在存储中添加了currentUserToEdit，我们必须在导航离开时将其取出

我们不能使用异步管道取消挂起的XHR请求

我们创造了某种分布式的庞然大物

*结束*  
国家管理是一个充满讨论和意见的话题。没有对错之分，使用对你和你的团队有效的方法。有一些很棒的库，如果对你有好处就使用它们，但至少在使用之前要考虑一下。这毕竟是本文的目标，让每个人开始考虑状态管理，而不是跳到第一个解决方案。