**<https://blog.strongbrew.io/A-scalable-angular-architecture-part2/>**

介绍  
不久前，我发布了一篇名为“可伸缩的angular 2架构”的博文，主要是关于创建大型企业网络应用的。标题实际上应该是“一个可伸缩的angular架构”(它只是angular的缩写)，甚至应该是“一个可伸缩的SPA架构”，因为我们可以在React或vuv .js中使用相同的架构。

这篇文章中有很大一部分是关于解释所谓的“沙箱”的，这是一种非常特殊的外观的个人术语。既然写那篇文章时提出了一大堆关于这些“沙箱”的问题，那么写一篇后续文章似乎是个好主意。

在本文中，我们将介绍沙箱的真正含义，以及为什么首先引入它们。

*注意:此体系结构只是构建应用程序的一种方法，本文中的陈述完全是个人的，可能是自己的看法。*

简单地说，沙盒是什么  
在现实生活中，这是一个我们可以把孩子和有限的玩具放在一起的地方。这样，他们就只能玩那些玩具，玩得开心，而我们作为父母就不用担心他们在花园里跑来跑去了。

在这个体系结构中，沙箱是一个受控的环境，也是组件可以使用应用程序片段的唯一位置。它是智能组件与应用程序的其余部分通信的唯一接口。它是一种从组件中抽象出逻辑的外观。但它确实有非常具体的逻辑。

下面是我引入沙盒的原因  
创建大型客户端javascript应用程序最具挑战性的部分之一是封装和划分职责。有一个像样的封装结构，其中每个组件、服务、类都有明确的存在理由，这使得我们作为开发人员可以很容易地维护代码和开发新特性。  
在编写单页应用程序时，可能会面临以下挑战:

*挑战1:我们如何构造组件?*  
由于这个主题在前一篇文章中已经讨论过，所以我们将不再深入讨论这个主题。我们使用智能/哑组件方法来构造我们的组件。哑组件大多具有表示目的，而智能组件与应用程序的其余部分交互。

*挑战2:我们如何处理状态管理?*上一篇文章讨论的另一个主题是如何处理状态管理。简而言之:我们使用@ngrx/store，或者另一个包含响应式编程的redux实现。

*挑战3*我们遇到了前两个挑战:我们有一个清晰的、结构化的表示层，而且由于我们使用了@ngrx/store(或任何状态管理工具)，我们有一个清晰的方式来管理状态。我们可以优化不可变数据的性能，并且仍然有明确的单向数据流。

然而，仍然有一些拼图缺失，这可能会引发以下问题:

问题1 *如何将表示层与应用程序的其余部分分离?*

如果我们在智能组件中注入任何我们想要的东西，那么它们的构造函数可能很快就会变得很大。这些智能组件真的需要了解应用程序的所有信息吗?他们是否需要知道每件东西的位置以及它应该从应用程序的哪个模块调用什么服务?如果他们有一个可以交谈的接口，为他们处理事情，不是更容易吗?当然，这有助于将angular模块彼此解耦，并保持那些智能组件的职责清晰紧凑。这是沙盒也许能帮助我们的东西。

问题2 *我们的组件需要知道redux吗?*

Redux是一个很棒的库/原则，它帮助我们管理状态，但如果你不小心管理它，它是一个非常严重的依赖。如果您非常具体的redux操作在组件和服务中随处可见，该怎么办?在这种情况下，您的代码基将完全受到它的影响，并且在将来使用其他状态管理工具将非常困难。在非redux应用程序中共享这些代码是不可能的。如果你想切换到firebase，或者MOBX，或者甚至编写自己的状态管理工具，那会是一个巨大的依赖性。您需要重构整个应用程序。redux的唯一原则是管理状态，我更喜欢这样使用它。这也是为什么我不使用effects, thunk or saga。(这是个人的偏好)对我来说，这意味着:使用redux只管理状态，而不是触发后端操作等。有人甚至认为，动作分派和状态选择应该在应用程序的中心位置结合。让我们把这种逻辑也添加到沙箱中。

我们的HTTP服务需要知道redux实现吗?

嗯，正如名字所揭示的。HTTP服务主要是进行HTTP通信并将异步对象返回给组件。HTTP服务根本不应该有关于redux的概念，也不应该有任何其他状态管理工具。它的唯一目的是获取和返回数据。让我们看看下面的例子。我们从后台获取一个葡萄酒数组，并希望将其保存在一个redux商店中。这是一个例子，一个HTTP服务被滥用，以实现:

class WineService {

// bad

fetchWines(): void {

this.api.get('url')

.subscribe(wines =>

this.store.dispatch({type: 'SET\_WINES', payload: {wines}});

}

}

此代码不属于这里，它不是该服务的职责。问题是它属于哪里?它属于组件吗?他们会很快变得脏兮兮的……\*再一次，这将属于沙盒!

http服务应该是这样的:

class WineService {

// good

fetchWines(): Observable<any> {

return this.api.get('url')

}

}

问题3 *我们在什么位置向存储分派动作**?*

*Err…, the sandbox =)*

问题4 *我们在哪里处理乐观的更新*同样，这就是沙盒

问题5 *如何不让你的整个应用程序闻起来像REDUX?*  
组件和服务不应该知道redux，因此存储交互也应该放在一个非常具体的位置。

我打赌你预见到了，但是，是的!这就是沙盒的作用!它是关于关注点的分离和使用一个非常具体的接口来处理这个非常具体的逻辑。

*沙箱是做什么的?*

沙盒有一个非常具体的职责:  
它让您的模块与其他模块进行通信，而无需该模块了解应用程序的其余部分  
它发送Redux动作  
它暴露了数据流(来自Redux、Firebase或其他技术)  
它处理乐观更新  
它使容器足够愚蠢

*沙盒上有什么特别的东西?*

他们处理容器告诉他们要处理的东西，而不需要容器知道如何…  
它们在模块之间相互作用  
它们处理大量的redux(或类似的技术)逻辑  
这些函数大多有一个void返回类型，除非我们想要能够取消HTTP调用

他们公开可见/流  
它主要包含冗余的代码，但没有冗余的逻辑(全部是KISS而不是DRY)  
几乎每个模块都有一个沙箱，除非它包含大量逻辑(在这种情况下，我们可以为每个容器实现一个沙箱)  
它不包含业务逻辑，永远……  
它大多不包含if语句，那将是业务逻辑，对吗?!  
它让我们对模块的响应能力有了一个清晰的概述。

*沙箱不能做什么?*它不包含业务逻辑。它包含沙箱逻辑(见上面)。  
它不包含表示逻辑，比如路由等。  
它不直接进行HTTP调用，而是委托给HTTP服务。  
它不会让组件做任何它们想做的事情=)

*一个例子*

@Injectable()

export class StockSandbox {

//这些是store select语句

//把它们放在组件上是很脏的

// 我们的组件只想获得数据流，不管它们来自哪里

wines$ = this.store.select(state => state.wines);

isAuthenticated$ = this.store.select(state => state.authentication.isAuthenticated);

constructor(

private store: Store<ApplicationState>,

private stockService: StockService,

private fooService: FooService) {

}

addWine(wine: Wine): void {

//使用stockservice添加葡萄酒

// 完成后，处理statemanmanagementt

this.stockService.add(wine).subscribe((wine: Wine) => {

this.store.dispatch(new AddWine(wine));

}, () => this.handleError());

}

removeWine(wine: Wine): void {

//移除一瓶酒可以乐观地完成!!

// 这肯定会提高我们的应用程序的性能和snappyness

// 这就是沙盒逻辑

let action = new RemoveWine(wine.\_id);

this.store.dispatch(action);

this.stockService.remove(wine).subscribe(

() => {},

// 如果调用失败，我们必须撤销一个操作。这也是沙盒逻辑!

() => this.store.dispatch({type: UNDO\_ACTION, payload: action}););

}

fetchWine(id: string): Observable<Wine> {

// simple delegation of fetching something: Sandbox logic

return this.stockService.fetchWine(id).share();

}

notifyAnotherModule(): void {

// 我们的组件不应该知道fooService在哪里，或者它做什么

// 它应该告诉它的沙箱来处理一个特定的动作

this.fooService.doSomething();

}

}

看看这个智能组件变得多干净:

export class StockPageContainer {

wines$ = this.sb.wines$; // 这个流来自Redux吗?或者Firebase(我不需要知道) numberOfWines$ = this.wines$.map(

wines => sumBy(wines, (wine: Wine) => wine.inStock));

// 干净的构造函数使测试变得容易.

// Util依赖或表示依赖，如路由器

// d不属于沙盒。

// 它们属于这里的组件，在表示层中

constructor(private sb: StockSandbox, private router: Router) {

}

onRemove(wine: Wine): void {

// 嘿沙箱!请把酒拿掉，我不管你怎么做

//我甚至不在乎你是否乐观更新，这不关我的事

this.sb.removeWine(wine);

}

notifyAnotherModule(): void {

//我不知道会发生什么，但没关系……这不是我的责任

this.sb.notifyAnotherModule();

}

}