angular-university 课程

[https://blog.](https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/)[angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/](https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/)

源代码库

https://github.com/angular-university/reactive-patterns-course

新代码库

https://github.com/angular-university/reactive-angular-course

智能组件和演示组件之间的典型交互

我们在这里看到的示例非常频繁，我们让智能组件通过将数据注入到演示组件@Input，并通过接收演示组件可能触发的任何操作@Output。

在这种情况下，我们使用自定义lesson事件来表明我们已经在列表中选择了给定的课程。

@Output通过清晰定义的界面，使用演示组件仍与智能组件保持隔离：

课程列表演示组件仅知道它发出了一个事件，但不知道该事件的接收者是什么，或者接收者将如何响应该事件

主屏幕智能组件订阅了lesson自定义事件并对事件做出反应，但是它不知道是什么触发了该事件。用户是双击课程列表还是单击视图按钮？这对于智能组件是透明的。

这样就很简单了，这里可能出什么问题了？

拆分智能组件与演示组件的清晰方法？

这样，我们可能会得出结论，构建我们的应用程序就像使所有顶级组件成为智能组件一样简单，并通过使用本地表示组件树来构建它们。

但事实是，有时并不是那么简单，因为自定义事件lesson不会冒出来。因此，例如，如果您有一棵深的组件树，并且希望多个级别的组件知道该事件，则该事件将不会冒泡。

什么问题导致自定义事件不会冒泡呢？

可以说，我们不但在课程列表和home组件之间仅嵌套了一层，还具有以下几层：课程列表位于选项卡面板中的可折叠面板内部。

课程列表仍希望通知home组件已通过lesson事件选择了课程。但是中间的两个组件TabPanel和CollapsiblePanel非特定于应用程序的表示组件。

想象它们是Angular Material库的组件！

这些仅用于Presentation的组件不知道lesson事件，因此它们无法冒泡。那么，我们该如何实现呢？为什么自定义事件不能简单地冒出来呢？

为什么自定义事件不冒出来，就像点击之类的DOM事件那样?

这不是偶然的，这是为了避免类似于服务总线的解决方案，比如AngularJs中的$scope.$emit()和$scope.$broadcast()意外创建的事件soup场景而设计的。

这些类型的机制往往会在应用程序的不同位置之间创建紧密的依赖关系，而这些位置不应该相互意识到，而且事件会被多次触发，或者以只查看一个文件时不明显的顺序触发。

因此，表示组件的自定义事件将只对其父组件可见，而不会在树的更上层可见。

如果由于某种原因我们确实需要冒泡行为，我们仍然可以使用element.dispatchEvent()。但是大多数时候这不是我们想要实现的。

那么，如何解决选项卡面板方案中可折叠面板中课程列表的情况？

我们仍然应该为课程列表创建一个表示组件。显示课程的功能是独立的，所以LessonsListComponent的版本仍然适用，它只是在应用程序中随处可见的东西。但是这个列表如何通知home组件呢?

对此有几个解决方案。我们应该研究的一个解决方案，特别是在构建一个大规模应用程序时，是研究像ngrx/store这样的解决方案。

但是，即使使用存储解决方案，我们也可能不希望将存储注入表示组件中。因为选择课程的结果可能并不总是将事件分派到存储。

为了保持示例简单，让我们从创建一个专门的类似于商店的服务开始，以仅解决这个课程选择问题:

@Injectable()

export class LessonSelectedService {

private \_selected: BehaviorSubject<Lesson> = new BehaviorSubject(null);

public selected$ = this.\_selected.asObservable().filter(lesson => !!lesson);

select(lesson:Lesson) {

this.\_selected.next(lesson);

}

}

正如我们所看到的，LessonSelectedService公开了一个可观察的selected$，它将在每次选择新课程时发出一个值。

请注意，我们在服务内部创建了一个主题(BehaviorSubject)，但没有将其公开给外部。这是因为主题(BehaviorSubject)本质上是事件总线，所以我们希望保持对谁可以在服务中发出事件的控制。

如果我们公开主题，应用程序的任何其他部分都可以代表服务发出事件，这是要避免的。

如何使用这个服务，因为我们不能将它注入LessonsListComponent，对吗?我们将讨论这一部分，现在让我们首先看看如何在Home组件中使用这个新服务。

在Home组件中使用新服务

Home组件将做的是，它将在构造函数中注入新组件:

@Component({...})

export class HomeComponent implements OnInit {

lessons: Lesson[];

constructor(

private lessonsService: LessonsService,

private lessonSelectedService: LessonSelectedService) {

}

ngOnInit() {

this.lessonSelectedService.selected$.subscribe(

lesson => this.selectLesson(lesson));

}

selectLesson(lesson) { }

}

如我们所见，我们已经订阅了selected$ Observable，它会发出新的lesson，我们触发组件的特定逻辑来处理选择。

但是请注意，Home组件并不知道课程列表，它只知道应用程序的其他部分触发了课程选择。应用程序的两个部分仍然是隔离的:

1 selected$的发射器不知道Home组件，

2 Home组件不知道课程，

3 两个参与者都只知道LessonSelectedService

所以我们已经解决了这个问题，对吧?还没有，因为我们仍然不想在LessonListComponent中注入新服务，这将使它成为一个智能组件，而我们希望它保持一个表示组件。那么如何解决这个问题呢?

如何保持LessonsListComponent是一个演示组件?

实际上，解决这个问题的一种方法是使它成为一个智能组件;-)我们可能会得出这样的结论:在应用程序中存在该表的任何地方，我们总是希望触发对LessonSelectedService的调用。

这将使课程列表组件成为特定于应用程序的组件，而它可能已经是这样了。例如，我们可能不会交付该组件并在多个应用程序中使用它。

因此，这将解决问题，这意味着像Home组件这样的顶级应用程序组件可能由一棵组件树组成，而不仅仅是表示组件。

智能组件不仅仅是顶级组件

一个智能组件不需要仅仅是一个顶级路由器组件。我们可以看到，树的下面可能还有其他组件，它们也被注入了像LessonSelectedService这样的服务，并不一定只从@Input()获得它们的数据。

另一种将LessonsListComponent保持为表示组件的解决方案

解决这个问题的另一种方法是保持课程列表组件不变，并在任何需要的地方使用它。但在这种情况下，我们可以将其封装在一个智能组件中，它被注入LessonSelectedService:

@Component({

selector: 'custom-lessons-list',

template: `

<lessons-list [lessons]="lessons" (lesson)="selectLesson($event)"></lessons-list> `

})

export class CustomLessonsListComponent {

constructor(private lessonSelectedService: LessonSelectedService) { }

selectLesson(lesson) { this.lessonSelectedService.select(lesson); }

}

在这里，我们创建了一个包装器智能组件，并将其称为CustomLessonsListComponent。在本例中，我们包装了自己的表示组件，但也可以包装来自第三方库的组件。

假设有一个MyCustomCountrySelectDropdown，它包装了一个通用的下拉列表，并将来自具体服务的数据注入其中。

如何决定构建什么组件?

在开始构建我们的应用程序时，并不总是很清楚什么是组件，以及什么是智能组件vs .表示组件。

那么，如何将应用程序分割成许多组件呢?一个页面的标题应该是一个组件吗，即使它只被使用了一次?

组织和可读性是创建组件的唯一原因，即使组件只在一个地方使用。在小文件中分离内容有助于保持代码基础的可维护性，并且使用Angular CLI，在创建新组件时没有任何开销:只需一个命令，我们就有了一个工作组件，可以在几秒钟内粘贴头部。

如何进行组件设计

解决这个问题的一种方法是避免从一开始就定义什么是组件以及组件的类型:我们可以从只使用纯HTML和第三方组件构建顶级组件开始。

只有当模板开始变大时，我们才开始把它分解成组件。如果在屏幕的几个部分中使用了某些东西，并且总是触发一个给定的动作，比如调用存储分派，那么我们可以考虑将其重构为一个更小的智能组件。

如果后来我们意识到需要像刚才创建的智能组件那样表示相同的数据，我们可以将其表示部分提取到表示组件中。

获得一组设计良好的组件的最佳方法是通过持续重构，使用Angular CLI可以轻松完成重构。

结论  
在构建应用程序时，我们可以寻找机会将纯粹的表示逻辑提取到表示组件中:这些组件只使用@Input和Output，在我们需要隔离表示逻辑并重用它时非常有用。

如果我们想让两个组件解耦并且不知道彼此，那么可以使用共享服务或存储来实现组件树中不同级别的智能组件或兄弟组件之间的通信。

但我们也可能希望将组件完全注入彼此并创建紧密耦合，有时这是最好的解决方案。在这种情况下，通过@ViewChild将组件相互注入可能是最好的方法。

智能组件与表示组件是一个有用的区别

一般来说，记住智能组件和表示组件之间的区别是非常有用的，但它可能并不适用于应用程序的所有组件。

我们可以使用一个小组件，它既能识别服务，又能在树的更深处显示一些数据，比如在课程选择时调用存储的课程列表。

将该组件进一步拆分为智能组件和表示组件可能并不总是必要的。

智能组件vs演示组件更多的是一种心态，我们会问自己:

1 这种表示逻辑在应用程序的其他地方有用吗?

2 把事情进一步分开会有用吗?

3 我们是否在应用程序中创建了意外的紧密耦合?

我们不需要将构建的每个组件的所有呈现逻辑提取到单独的表示组件中。它更多地是关于在任何给定时间构建对我们的应用程序最有意义的组件，并在CLI简化的连续迭代过程中进行必要的重构。

通用组件设计(以及它的潜在问题)

Angular应用开发中一个非常重要的方面就是应用组件设计:如何将不同类型的组件组合在一起，何时使用组件vs .使用指令，如何以及何时将组件中的功能提取到指令中，等等。

Angular组件提供了许多种方式使用和组合的功能。这允许我们根据情况采用各种各样的应用程序设计。

在这篇文章中，我们将讨论一种我们经常听说的特殊类型的设计。

容器组件vs表示组件

我们将要讨论的设计场景是容器组件与表示组件之间的组件分离。这是一个很流行的设计，现在在Angular生态系统中被越来越多的人使用，因为现在Angular支持组件模型了。丹·阿布拉莫夫(Dan Abramov @dan\_abramov)在博客中展示了该设计:

表示组件和容器组件 ：   
这篇文章是写给React的，但其理念同样适用于任何允许基于组件的设计模型的生态系统，比如Angular。

容器与展示设计的一个例子

让我们举一个设计的例子，请记住不同的术语被用来命名不同类型的组件。  
设计的核心思想是有不同类型的组件。使用同样的术语，在上面的博客文章中，我们有:

1 容器组件——这些组件知道如何从服务层检索数据。请注意，路由的顶级组件通常是一个容器组件，这就是为什么这种类型的组件最初如此命名的原因

2 表示组件——这些组件只是将数据作为输入，并知道如何在屏幕上显示。它们还可以发出自定义事件

让我们给出一个简单的设计示例，它实际上已经包含了一个潜在的设计问题。为了让它更有趣，我建议如下:试着在我展示示例时发现设计问题，我们将在稍后的文章中讨论这个问题。  
如果您已经尝试使用这种设计，您很可能遇到过这个问题。

以反应式风格编写的顶级组件  
让我们从路由的顶级组件开始。让我们来看看一个简单的路由顶层组件，以反应式风格编写: 这是一个显示课程详细信息的简单组件:in包含一个标题，其中包含课程摘要(加上时事通讯框)和课程列表。 让我们来分解一下这个顶级组件中有什么以及它目前是如何设计的:

1 该组件注入了路由器依赖项，但也注入了一些特定于应用程序的服务

2 组件没有任何直接引用数据的变量，比如课程或课程

3 相反，组件在ngOnInit上声明了一些可观察的东西，这些可观察的东西是从服务层获得的其他可观察的东西派生出来的

顶层组件设计概述  
这个顶级组件将根据路由标识符参数定义如何从服务层获取数据。

这是反应式应用程序中典型的顶级组件，它不使用路由器数据预取(稍后详细介绍)。该组件最初将在没有任何数据的情况下显示，并且它将对服务层执行一次或多次调用以获取数据。

*注意，组件只是定义了一组可观察对象，但在组件类中没有进行订阅:那么如何显示数据呢?*

顶级组件的模板  
现在让我们看看这个组件的模板，看看这些可观察的组件是如何被使用的:

正如我们所看到的，我们获取了可观察的对象，并通过异步管道订阅了它们。然后，数据被应用到本地组件树中，这些组件位于该路由的顶级组件之下:

1 多种类型的数据(包括用户、课程和课程)被传递到课程-细节-header组件以及课程列表。

2 这些组件负责显示顶级组件检索到的数据

**关于多重订阅的说明**一个重要的事情是:lessons$ observable订阅了两倍。在这种情况下，它不会造成问题，因为来自服务层的可观察对象被设计为防止对后端的多个请求，例如使用publishLast(). refcount()。

请注意，这只是确保多个订阅不成为问题的一种可能的解决方案。现在让我们来看看顶级组件模板中使用的其中一个本地组件。我们将看到他们有一个非常不同的设计。

研究表示组件的设计  
所以顶层组件是一个容器组件，但是模板中使用的其他组件呢?

表示组件将负责获取输入数据并将其呈现给用户。例如，course-detail-header就是一个表示组件。让我们看看这个组件看起来是什么样的:.

*提醒:试着找出这个设计的问题*

正如我们所看到的，组件接受一些数据作为输入，然后将这些数据呈现在屏幕上。此组件与应用程序的服务层没有依赖关系，相反，它通过输入接收数据。

它还发出输出，例如订阅输出事件。但这个事件从何而来?我们可以看到，它是为响应来自newsletter组件的同名事件而触发的。

newsletter组件是什么样子的，它是如何设计的?让我们看一看。

表示组件比组件树深一层  
newsletter组件也是一个表示组件，因为它接受输入、显示订阅表单并在订阅时发出事件:

这是我们目前的newsletter组件的设计，所以让我们更详细地回顾一下，看看会有什么问题。

这种设计的一个潜在问题  
你可能已经注意到newsletter组件与course-detail-header组件类似:

输入属性的名称

输出事件subscribe

这两个元素在两个组件中重复出现。因此，看起来我们在这个设计中做的一些事情在更大的组件树中可能无法很好地伸缩，因为涉及到很多重复。 让我们回顾一下这两个问题，看看我们如何设计它。

设计问题1 -中间组件的无关特性

看起来我们将firstName之类的输入传递到本地组件树上，以便让像newsletter组件这样的叶子组件使用它们。但是中间组件本身并不使用输入，它们只是将输入传递给它们的子组件。

通常，本地组件树要比这个示例大得多，因此这个问题可能会导致大量输入的重复。

更重要的是:如果我们使用第三方小部件库，并且使用其中的一些组件作为中间组件，我们可能会在通过组件树传递所有必要的数据时遇到麻烦，这取决于库的设计方式。

还有另一个与产出相关的类似问题。  
设计问题2 -自定义事件在本地组件树上冒泡

正如我们所看到的，订阅事件也在组件树的多个级别上重复，这是因为根据设计定制事件不会冒泡。因此，这里还有一个代码重复问题，它不能很好地扩展到更大的示例中，也不能与第三方库一起工作——在这种情况下，我们不能应用这种技术。

另外，订阅时事通讯(对时事通讯服务的调用)的逻辑位于顶层路由组件上，而不是在时事通讯组件上。这是因为只有顶级组件可以访问服务层，但这最终可能会导致大量逻辑被保留在该组件上。

那么我们如何在Angular中解决这些问题呢?让我们看看一个可能的解决方案。

防止自定义事件冒泡  
如果我们发现自己处于手动将事件向上冒泡的情况下，那么这种方法可能适用于某些更简单的情况。但是，如果事件冒泡/无关属性开始变得难以维护，这里有一个替代方法。

我们将通过一步一步的重构来展示另一种选择。让我们再次从顶层组件开始重构，看看新的解决方案如何避免我们已经确定的问题。

重构的顶级组件  
让我们更改顶层组件，使其不再从本地组件树传递尽可能多的数据或接收尽可能多的事件。让我们也删除newsletter订阅逻辑。

顶层组件的新版本现在有比以前少得多的代码:  
这看起来是个不错的开始。那么顶层组件模板呢?重构后的新模板基本上是相同的，除了course-detail-header组件:

这看起来比我们以前的版本更好:我们不再看到firstName的传递或订阅事件的冒泡。

那么，重构之后，course-detail-header中间组件现在是什么样子呢?

重构的中间组件  
我们可以看到，在重构之后，新版本的course-detail-header组件变得简单多了:  
这个新版本的组件仍然包含newsletter，但它不再冒泡事件和传递组件本身不需要的数据。

所以这个看起来比我们最初的版本好多了。但是现在订阅时事通讯的功能在哪里呢?

现在让我们看一下重构中的最后一个组件:叶组件。

重构的叶子组件  
正如我们所看到的，newsletter叶子组件现在以一种完全不同的方式设计:那么，这个新版本的newsletter组件在设计上最大的不同是什么呢?最大的不同实际上是这个新版本看起来很像一个容器组件!

因此，我们可以看到，有时最好的解决方案是将服务深入到组件树中。这确实简化了本例中涉及的所有多个组件。

但是叶子组件的这种实现还可以进一步改进，所以让我们进一步回顾这个设计，看看如何改进。

评审新的组件设计方案  
这个组件树的新设计似乎更容易维护。不再有自定义事件冒泡，也不再通过组件树传递无关的输入属性。

newsletter组件现在知道服务层，并从服务层获取所有数据。它有一个对时事通讯服务的引用，所以它可以直接调用它。请注意，如果需要，该组件仍然可以接收输入，稍后将详细介绍这一点。

利用Angular特性来实现更简单的设计  
我们可以看到，在这个新版本的组件树中，我们正在利用Angular依赖注入系统将服务注入到本地组件树的深处。

这允许深度嵌套的组件(如newsletter组件)直接从服务层接收数据，而不必通过输入接收。

这使得顶级组件和中间组件都更简单，并避免了代码重复。它还允许将与服务层交互的逻辑深入到组件树中，如果在组件树中使用该逻辑最有意义的话。

newsletter组件的当前实现有一个问题  
这个新版本的时事通讯组件只有一个问题:不像以前的版本是一个演示组件:  
*这个新版本不能与OnPush变更检测一起工作!*

使newsletter组件兼容OnPush

您可能已经注意到，有时当我们切换组件使用OnPush变更检测时，事情会停止工作——即使我们没有在组件级别上进行数据的局部突变。

其中一个例子就是newsletter组件的当前版本，它实际上不会反映模板中名字的新版本。  
但这里是一个版本的组件兼容OnPush也:

那么这个新实现有什么不同呢?在这个版本的组件中，我们定义了一个可观察的名字，并且在模板中使用异步管道使用它。

使用async管道将确保在发出第一个名称的新版本时重新呈现组件(例如,当用户登录),即使组件没有输入——因为async管道将检测到一个新值由可观察对象发出，因此它将标记组件以重新呈现。

结论  
所以我们可以看到，有很多可能的组件设计，取决于情况。如果我们需要的话，使用Angular的依赖注入系统可以让我们很容易地将服务注入到组件树的深处。

因此，我们不必通过组件树的多个级别传递数据和事件，因为这可能会由于代码重复(以及其他问题)而导致可维护性问题。  
但是为什么在尝试应用容器+表现设计时，这种情况会经常发生呢?

**自定义事件冒泡问题的可能解释**这种设计最终会被使用的一个可能的主要原因是:在软件设计中，我们给东西起的名字会有很大的影响。

容器组件这个名称让我们认为，只有路由的顶级组件才应该具有这种设计类型，而它使用的所有其他组件都应该是表示型的，但事实并非如此。

名称容器不会让我们想到像newsletter组件那样的叶子组件。

为了避免这个问题,这里有一个建议:如果我们需要提供一个名称的组件意识到服务层,和拥有一个名称有助于应用程序设计的讨论,我们可以称之为相反,例如像智能组件和保留顶层的术语容器组件的路线。

在实践中，根据我们的需要混合和匹配多种类型的组件设计，并根据需要在树的不同级别上使用不同类型的组件——根据我们的需要混合不同的特性。

<https://blog.angular-university.io/how-to-build-angular2-apps-using-rxjs-observable-data-services-pitfalls-to-avoid/>

构建Angular应用程序的备选架构  
构建Angular应用程序有几种可能。最近有一种趋势，即以类似于Redux的风格，使用单个状态原子构建类似flux的应用程序。以下是几种以这种方式构建应用程序的备选方案:

1 如果您正在使用集中式存储模式以响应式风格构建应用程序，推荐的方法是使用@ngrx/store guide。

2 如果使用Redux本身构建应用程序，请参阅这篇文章了解更多细节和示例应用程序

3 如果使用Redux和单状态原子的概念构建应用程序，但在Rxjs中实现它。查看另一篇文章，找到一种方法和一个示例应用程序

这篇文章将提供一个替代方案，它并不意味着一个单一的原子状态，而是包含使用可观察数据服务。如果你刚开始学习观察和Angular，你可能会想看看这篇文章，在那里我们介绍了一些常见的问题场景。

什么是可观察数据服务  
可观察数据服务是Angular可注入的服务，可以为应用程序的多个部分提供数据。服务，可以被命名为商店，可以被注入到任何需要数据的地方:

在本例中，我们注入了两个服务，一个包含应用程序数据(即待办事项列表)，另一个服务包含UI的当前状态:例如，当前显示给用户的错误消息。

如何使用可观察数据服务  
数据服务公开一个可观察的，例如TodoStore公开可观察的todos。这个可观察到的每个值都是一个待办事项的新列表。  
数据服务然后可以直接在模板中使用异步管道:  
这个管道将订阅todos observable，并检索它的最后一个值。

如何修改服务的数据  
通过调用服务上的动作方法来修改服务中的数据，例如:  
然后，数据存储将根据动作方法调用为其数据发出一个新值，所有订阅者将收到新值并相应地进行更新。

关于可观察数据服务有几个有趣的事情  
注意，TodoStore的用户不知道是什么触发了正在发出的一个新的todo列表:添加todo、删除或切换todo。商店的消费者只知道有一个新值可用，视图将相应地进行调整。这有效地解耦了应用程序的多个部分，因为数据的使用者不知道修饰符。

还要注意，注入存储的应用程序的智能组件没有任何状态变量，这是一件好事，因为这些是编程错误的常见来源。  
还需要注意的是，智能组件中没有直接使用Http后端服务的地方，只有调用存储才能触发数据修改。

现在我们已经了解了如何使用可观察数据服务，接下来让我们看看如何使用RxJs构建一个可观察数据服务。

RxJs主题和如何使用它  
可观察数据服务的核心是RxJs主题。主题同时实现了观察者和可观察接口，这意味着我们可以使用它们来发出值和注册订阅者。

这个主题只不过是一个传统的事件总线，但是它更强大，因为它提供了所有RxJs函数操作符。但在其核心，我们只是使用它来订阅，就像一个常规的观察:  
但不像常规的观察，Subject也可以用来发射值给它的订阅者:

Subject有一种特殊性，它阻止我们使用它来构建可观察数据服务:如果我们订阅它，我们将无法获得最后一个值，我们将不得不等待，直到应用程序的某些部分调用next()。

这就带来了一个问题，尤其是在引导的情况下，应用程序仍然在初始化并且不是所有的订阅者都已经注册，例如不是所有的异步管道都有机会注册自己，因为不是所有的模板都还没有初始化

BehaviorSubject如何使用它

解决这个问题的方法是使用一个BehaviorSubject。这种类型的Subject做什么，它会在订阅时返回流的最后一个值，或者一个初始状态，如果没有值被发出:

BehaviorSubject还有另一个有趣的属性:我们可以在任何时候检索流的当前值:

这使得BehaviorSubject成为了可观察数据服务的核心，我们不需要更多来构建一个。让我们看一个具体的例子。

如何构建可观察数据服务

你可以在这里找到一个完整的商店例子，但这是服务的最重要的部分:  
https://github.com/jhades/angular2-rxjs-observable-data-services/blob/master/src/state/TodoStore.ts

我们可以看到存储包含一个私有成员变量\_todos，它只是一个初始状态为空的Todos列表的BehaviorSubject。

构造器被注入Http后端服务，这是应用程序中使用此服务的唯一位置，应用程序的其余部分注入了TodoStore。  
store在构建时被初始化，所以我们再次使用一个BehaviorSubject很重要，否则它将无法工作。  
但是这个额外的公共成员变量todos背后的原因是什么呢?

陷阱1:不要直接暴露对象

在本例中，我们没有直接向存储客户端公开主题，而是公开一个可观察对象。  
这是为了防止服务客户端自己直接发出存储值，而不是调用操作方法，从而绕过存储。

**避免事件汤**直接暴露主题可能导致事件soup应用程序，其中事件以难以推理的方式链接在一起。

直接访问主题的内部实现细节就像返回对对象的内部数据结构的内部引用:公开内部方法以获得对主题的控制，并允许第三方发出值。可能存在这样的有效用例，但这几乎从来都不是我们想要的。

**写一个动作方法**在这种类型的应用程序中，操作只是存储提供的方法。举例来说，看看addTodo动作是如何构建的:

这只是一种方法。我们调用后端服务，它会在成功或错误时返回一个可观察对象。  
我们订阅相同的可观察对象，成功之后，我们通过将新的todo添加到当前列表来计算新的todo列表。

陷阱#2 -避免重复的HTTP调用

在本例中需要记住的一点是，Http返回的可观察对象将有两个订阅者:一个在addTodo方法内部，另一个订阅者调用addTodo。

这将导致(由于可观察对象在默认情况下的工作方式)重复HTTP调用，因为设置了两个独立的处理链。请参阅这篇文章，了解更多关于这方面的细节，以及其他可能让我们惊讶的观察方式。

为了解决这个问题，我们可以做如下的例子，以确保没有重复的http调用可以发生:  
注意直接返回热可观察对象而不是HTTP冷可观察对象的权衡:没有重复的网络调用，但是saveTodo的调用者自己可能无法执行某些操作(如重试)。

学习Angular服务、HTTP和RxJs

结论  
可观察数据服务或存储是一种简单而直观的模式，它允许在不引入太多新概念的情况下利用Angular中的函数反应式编程的强大功能。

熟悉的概念，如主题(基本上是一个事件总线)是这个模式的基础，这使得它比其他需要其他几个RxJs构造的模式更容易学习。  
一些预防措施(如不直接公开主题)可能足以让应用程序简单地推理，但这取决于用例。  
正如我们在陷阱部分中看到的，需要熟悉RxJs和可观察对象的工作方式。查看前一篇文章了解更多细节。

Angular服务层——存储架构  
Angular的当前版本相对于上一个版本(AngularJs)是一个很大的改进。视图层比以往任何时候都更容易学习和使用。  
但是服务层(也称为数据层)是应用程序的真正功能核心，它提供了许多选项:

我们应该如何构造服务层?  
我们应该去商店吗?  
我们应该使用Redux吗?  
我们应该使用普通RxJs吗?  
那NgRx商店呢?

### 在Angular的世界里，有一件事非常流行，那就是存储解决方案。 它们起源于React世界，经历了通常的技术采用曲线:大量采用，意识到它并不是所有事情的最终解决方案，然后在某些情况下使用它，而不是其他情况。 为什么stores在React如此受欢迎?

为什么商店在React世界如此受欢迎，是有特定的原因还是有多种原因?这些原因是否也适用于Angular世界，或者有没有其他的解决方案?商店解决了什么问题?

你是否注意到有很多关于商店解决方案的信息，但是关于我们应该什么时候使用它们以及为什么使用它们的信息却很有限?我们来复习一下这些问题。

内容  
在这篇文章中，我们将涵盖以下主题:

什么时候使用Redux或一般商店?

我们通常需要一个商店吗?

为什么Redux在React世界如此受欢迎?

在Angular的世界里，Redux解决的问题也存在吗?  
商店能解决什么问题?  
什么类型的应用程序从存储解决方案中获益?  
与存储解决方案相关联的工具类型是什么?  
单向数据流在React和Angular中  
商店和可测试性  
商店和性能  
商店和工具

Redux vs Mobx

与Mobx和CycleJs的工具比较  
方法建议  
结论和建议

注意:下面有一个演示Ngrx开发工具的视频。另外，在这篇关于集中式存储模式和Ngrx存储的文章中——Angular Ngrx速成课程第1部分:Ngrx存储——通过理解最初的Facebook Counter Bug来学习它。

什么时候使用Redux或一般商店?

商店起源于Redux世界，所以那将是最好的地方之一，首先看看，然后从那里开始。  
让我们看看如何引导React生态系统，有什么建议?这里有一句很重要的话:

*你可能听说过Flux。关于Flux有很多错误的信息。很多人坐下来构建一个应用程序，并想要定义他们的数据模型，他们认为他们需要使用Flux来做这件事。这是采用Flux的错误方法。*

还有一个著名的帖子是Redux的创建者写的——你可能不需要Redux(https://zhuanlan.zhihu.com/p/22597353)，它可以应用于任何商店解决方案。

然后在React How-To中还有另一个声明，似乎同样适用于原始Flux, Redux, Ngrx Store或任何一般的Store解决方案:

当你需要Flux的时候，你就会知道。如果你不确定是否需要它，你就不需要它。  
基于此，一些原创者似乎不推荐系统使用商店。在这些帖子中，我们得到的印象是，创造者似乎担心商店被视为一刀切的解决方案。

但是当我们遇到像这样的帖子时——我似乎总是需要Redux

不知何故，即使商店被自己的创造者谨慎地推荐，它们仍然被React世界大规模地采用。  
为什么会这样呢?我们来试着回答这个问题。

什么时候推荐使用Redux?

如果我们深入探究React How-To的文档，我们会发现几个迹象表明我们何时能从Flux获益:

*React组件按层次结构排列。大多数时候，数据模型也遵循层次结构。在这些情况下，变化并不能给你带来什么。但是，有时您的数据模型不是分层的。当你的反应组件开始收到感觉无关的道具，或者你有少量的组件开始变得非常复杂，然后你可能想看看Flux。*

如果我们深入研究这些问题，我们也会得到这些建议。如果:  
你有一段数据需要在应用程序的多个地方使用，通过道具传递它会破坏组件的单一责任原则(例如，使它们的界面变得不那么有意义)

但也有这样一种情况:  
有多个独立的参与者(通常是服务器和终端用户)可能会改变数据  
所以有几种情况下我们建议使用store溶液和React。让我们看看这是如何融入Angular的。

具有并发更新的存储和应用程序

如果我们只基于最后一部分，那么只有一小部分应用程序，即具有服务器推送需求的应用程序将从Flux中受益。因为这通常是当我们有多个参与者更新相同的数据时，这就是最初的Facebook反问题的情况，它产生了Flux。

注意，我们不需要使用服务器push来陷入这种情况，使用setInterval进行长轮询或修改setTimeout中的数据都会导致相同的场景:多个参与者同时编辑相同的数据。

我们可以放心地说，很多应用程序都没有这个问题，对吧?这是一个重要的问题，我们需要针对它进行设计，但大多数应用程序都有吗?可能不是，只是某一类应用程序。  
但是为什么Redux在React的世界里如此普遍地被采用呢?这就留下了另一个原因。

Redux解决的最常见的问题是什么?

Redux还解决了“额外的 props”问题。这也是《Redux》在React中如此受欢迎的主要原因之一。

从角度来看，“额外的 props”是什么意思呢?props相当于Angular组件的@Input()成员变量。

因此，这意味着Redux可以帮助我们处理这样的情况:我们使用@Input()将输入传递到组件树上的组件，但是这些输入感觉上是无关的，不是应用程序的一部分。

例如，我们在组件树上传递了5或10个级别的内容。树的叶子知道如何使用它，但是对于中间的所有组件来说，输入感觉是多余的，并且使该组件的可重用性更低，与应用程序的联系更紧密。但这只是一个例子。

额外的props，它还能意味着什么?

无关的props具问题似乎是一个组件间的沟通问题。  
有些情况下，组件在组件树中完全依赖于其他组件，将输入向上传递10层，将回调函数向下传递10层，然后向上传递5层，另一个分支在复杂性上是不可伸缩的。

以下是发生这种情况的其他情况:

1 将数据传递到树的深处，并对组件树的几个级别上的事件做出反应

2 另一个问题是，树中有相互依赖的兄弟组件，它们代表屏幕上相同数据的不同视图，比如带有未读消息的文件夹列表，以及页头上的总未读消息计数器。

还有更多的例子。如果我们只有props或@Input()作为组件的通信机制，我们会很快陷入麻烦。只向组件传递输入不会增加复杂性。  
这些情况实际上很常见，所以这就是我们的答案。

为什么Redux在React世界如此受欢迎?  
可能是因为它还解决了无关的props问题:这意味着它为更复杂的组件交互场景提供了解决方案。

这是一个基本问题，没有它我们就无法构建更大的应用程序，Redux可以解决这个问题。  
几乎所有重要的应用程序都有这些场景，它实际上不需要大型应用程序，大多数典型的企业应用程序都有某种复杂的组件间通信场景。

为什么Redux在这些情况下工作得很好?  
如果我们尝试使用像AngularJs $scope.broadcast()这样的事件总线来解决这些场景，我们将很容易地以事件soup场景结束，在这种场景中，事件以意想不到的方式链起来，并且很难对应用程序进行推理

这是因为可以很容易地将事件转换为命令，从而使发射器了解接收方的内部情况。此外，还有可能意外地将事件链接在一起。

Redux看起来像事件总线，但它不是。实际上，Redux存储是命令模式和可观察模式的组合。我们对存储所做的是，我们给它发送一个命令对象称为一个动作:

我们将一个操作分派到存储中，存储将对存储中的数据进行操作。但是动作的发射器并不知道存储将对它做什么。

我们还可以从应用程序的完全不同的部分分派另一个动作:  
存储将对其进行处理并更新消息列表。然后将消息发送到应用程序中需要它的任何部分。但接收端并不知道是什么触发了新数据的生成:

1 来自后端的新消息 2 请求刷新 3 消息被标记为已读

那么这与复杂性的解耦和缩放有什么关系呢?

存储如何允许解耦组件交互  
组件使用新版本的数据(可能是消息列表和一个计数器)不知道什么导致了数据的变化,就像当我们订阅RxJs可见我们不知道触发发射,我们只知道,我们有一个新值。

消费组件已经订阅了自己到存储，就像它们订阅了一个RxJs Observable。此模式工作得很好，因为我们必须竭尽所能地将发出的数据转换为命令，而对于事件总线，这是非常容易做到的。

服务器推送呢?  
现在假设服务器也在不断推送新数据，新消息。数据也通过一个分派动作推送:

在所有情况下，都会接收并呈现一个新的消息列表，或者是一个消息列表，或者是一个未读消息计数器。呈现的结果将是一致的:我们不会有一个全部读取的消息列表和一个计数器，说有3条未读取的消息。

这种情况是a store shines

对于这个可编辑数据和多个参与者的问题，存储是一个理想的解决方案，但是让我们假设数据不是从服务器推出的。在这种情况下，我们只有组件交互和协调问题，但不存在竞争条件的可能性。

在这种情况下，我们试图解决的问题只是在组件树的多个断开连接的地方进行组件交互，对吧?  
我们不再需要一个由多个并发参与者编辑相同数据的解决方案。这就引出了Redux和一般商店的一个重要特性。

store是针对多个问题的复合解决方案，而不仅仅是一个问题

通过这个例子我们可以看到，store是一个多责任的解决方案:

1 它们通过可观察模式解决了组件交互的问题

2 如果需要，它们提供客户端缓存，以避免重复执行Ajax请求

3 它们提供了放置临时UI状态的地方，当我们填写一个大表单或希望在搜索表单中存储搜索条件时在路由器视图之间导航

4 解决了允许多个参与者修改客户端瞬时数据的问题

store不仅解决了其中一个问题，而且解决了所有的问题。

多责任解决方案的问题是什么?

这样做的一个潜在问题是，这些问题并不总是同时出现:你可能想解决一个问题，但不想解决另一个。并不是每个应用程序都有像Facebook一样的限制:它是世界上最大的网络应用程序，拥有18亿用户。

假设您的应用程序是典型的用户少于100个的企业应用程序:您对客户端缓存的使用有限，并且可能没有服务器推送需求。您可以使用服务器推送，但数据大多是只读的。

在这种情况下，您可能无法从存储体系结构中获益(稍后将对此进行详细介绍)。  
另外，您可能需要处理一个复杂的组件交互场景，而不需要为此在内存中存储数据。这里重要的一点是，这些问题并不总是同时出现:它们只出现在特定的应用程序中，而不是其他应用程序中。

### Redux是否避免了与state-related的问题? 重要的是要知道，通常情况下，它和其他全局store解决方案都不是这样，因为使用Redux，我们创建了一个大型的全局应用程序级状态:store是应用程序范围的单例服务。

全局应用程序状态的问题不在于它创建的方式，而在于它存在的事实。由于我们忘记清理它，很容易产生微妙的错误。它实际上并没有改变使用纯的reducer函数创建状态的事实，或者全局状态是不可变的。

所有这些都有帮助，但我们仍然创建了全局应用程序状态，而主要问题仍然存在:它存在，我们需要在所有正确的地方清理它，而这不能很好地扩展复杂性。  
但是如果需要的话，全局状态并没有什么问题:一些用户数据到处都需要，为什么不加载一次，然后把它放到一个单独的服务中呢?

应对全局state的最佳方式是什么?  
避免全局应用程序状态的最佳方法是，除非必要，否则不要创建它，但很多时候都不是这样。现代应用程序确实比以前需要更多的状态:例如，当我们在应用程序中导航时，我们应该将给定搜索表单的最后搜索结果保存在哪里?

我们不希望每次从一个细节返回到主表时重复一次搜索，即使我们触发了一个路由器导航。

我们可以使用临时的本地状态吗?  
对于这些情况，理想的情况是能够创建一个只针对与特定的主-细节设置交互的本地状态，并使其在使用后自动清理。

这就是Angular所能做到的我们一会儿就会看到。

在Angular的世界里，除了存储，还有别的解决方案吗?

在Angular中，我们有一套完整的内置解决方案来处理复杂的组件交互场景。所有这些解决方案的核心是Angular的依赖注入系统:

1 如果我们愿意，我们可以将服务注入到组件树的深处，看看Angular智能组件和表示组件的区别:什么时候使用它们，为什么?)

2 我们甚至可以将组件或服务注入到彼此中，如果我们认为它们天生是紧密耦合的

3 我们可以创建共享数据服务，这些服务可以存储数据，也可以不存储数据

但这仅仅是开始。让我们回到主-细节场景:我们可以创建一个非全局服务，并使用分层注入器将其仅关联到页面的某个部分。这意味着服务及其最终状态将透明地清理自己。

创建一个自我清理的地方政府  
假设我们已经浏览了应用程序中包含消息列表的部分，然后单击该列表，就可以看到消息的详细信息。

这是该路线的顶级组件:

注意provider属性中的MessagesService。这是什么意思?这意味着服务不是应用程序范围内的单例。因此，如果我们希望在打开和关闭多个细节时将主服务器的搜索结果保存在内存中，MessagesService将是放置它的理想位置，而不是全局存储。为什么?

因为MessagesService的这个实例对于MessagesContainerComponent及其兄弟组件是本地的。它只能注入到这里，而不能注入到应用程序的其他任何地方。

您还可以创建一个MessagesTableService并在表级别注入它，使用它来加载和分页数据，并同时拥有多个表，每个表都有自己的MessagesTableService实例。  
这些只对组件树的子集可见的本地服务的好处是，当我们导航离开其路由时，它们与相关组件一起清理自己。

例如，可以将本地有状态服务实现为可观察数据服务。  
Angular和Stores——一个常见的选择?

正如我们所看到的，在Angular中，我们有许多组件间的通信机制可用，不仅仅是@Input()，我们还有一个自动创建和处理本地状态的机制。  
在Angular中，我们不必从存储中获益来解决这些问题，有很多其他的内置解决方案。

很多时候，存储被添加到应用程序中，以获得类似可观察的API，以允许某些组件交互:为什么不简单地使用可观察对象呢?  
添加存储是对应用程序总体架构的一个重要约束，它意味着创建大量的全局应用程序状态。如果有更好的替代方案不暗示这一点，为什么不考虑它们呢?

使用商店是有成本的  
存储确实解决了组件交互的问题，但它也创建了在应用程序中管理状态的需求，否则在使用其他解决方案时可能不存在这种需求。

这可能意味着，在Angular中存储解决方案比在React中有用得多。实际上，在反应初期，我们会看到其他的解决办法。  
为了支持存储解决方案的选择，通常还会提到其他一些参数:性能、可测试性、工具以及保持应用程序的可预测性和简单性的能力。让我们从最后一个开始，一个一个地讲。

单向数据流  
单向数据流是我们在React和Angular中都听说过的一个重要属性:它指的是在应用程序中寻找的一个属性，确保它们是可预测的，并且易于推理。

单向数据流在React

在最初关于Flux的讨论中，单向数据流的描述如下:用户触发一个动作，它被分派到商店，商店生成一个新的模型并将其发送到视图。

但是视图本身在呈现时不能分派进一步的操作，如果某个操作的分派已经在进行中，也不能分派另一个操作。  
根据最初的演示，避免这种情况似乎是Flux的主要目标之一，请看这里。这里有另一个参考。  
另外，看看这里的原始流量调度器代码，在讨论中提到了在这里进行检查。

React和Flux中的UI可预测性似乎主要是通过在数据层上施加有益的约束来实现的:防止链式分派。  
Redux和单向数据流

一定要注意，Redux不能防止在最初讨论Flux时提到的链式分派场景。使用Redux，我们可以在原始通量中从订阅方法触发另一个分派，如果一个动作已经被分派，我们不能触发另一个。

因此，强制执行单向数据流的需要似乎并不是Redux被如此广泛采用的主要原因之一，因为从设计上来说，至少根据原始Flux对话中提供的定义，它并不能防止链式分派问题。  
也许是因为它太受约束了，而且在实践中，它并不经常发生?

单向数据流的Angular  
在Angular中，我们也看到了单向数据流的属性，它允许我们以一种可预测的方式推理应用程序。

但这里的关注点似乎有些不同，尽管相关:它不是关于在数据层上施加约束，数据层可以有任何形式。  
在Angular中，单向数据流被描述为确保视图不能更新自己。这是什么意思?

开发模式下的单向数据流和呈现  
当呈现开始时，我们在一次扫描中遍历组件树，并且在呈现期间，组件不能在第二次扫描中给出不同的结果或修改父组件。

基本上，对模板中的表达式求值或触发某些组件生命周期方法的行为本身不能触发视图中的进一步更改，从而创建类似于AngularJs的多步骤摘要循环的情况，有时会导致不可预测的结果。

打破Angular单向的数据流  
假设你正在打印一个随机数到屏幕:如果你试图通过组件getter方法计算它，并将它传递给一个模板表达式，我们将破坏应用程序在开发模式，因为你不能得到同样的结果在第二次自上而下的扫描:

Try it, you should get:

Expression has changed after it was checked

因此，为了确保UI中有一个可预测的呈现行为，并防止视图更新自身，我们不必采用类似于商店的架构。  
让我们看一下使用存储的另一个常见原因:改进的性能，接下来让我们看一下可测试性和工具。

商店和性能  
有时，商店被认为是一种使应用程序更高效的方法，因为我们可以使用像不可变js或Deep Freeze这样的东西使状态不可变，然后我们可以在任何地方使用OnPush变化检测。

角变化检测机制是开箱即用的，非常快，表现得非常直观。默认情况下，只有我们在模板中使用的表达式用于检测更改，其余的都被忽略(看看这篇文章https://blog.angular-university.io/how-does-angular-2-change-detection-really-work/)。

OnPush实际上是一种优化，只有少数应用程序可能从中受益，比如加载大量数据的应用程序(以及我们能加载多少对用户仍然有用的数据)，或者运行在非常有限的设备上的应用程序。

可以肯定地说，大多数应用程序不属于这些类别(考虑到目前的智能手机)。但是如果我们仍然需要OnPush，我们可以简单地使用它而不需要存储，特别是如果我们的数据大部分是只读的。

如果应用程序是类似于图表指示板的实时指示板，那么最好控制数据或其他解决方案。我们甚至可以将UI的一个分支从变更检测中分离出来，并限制其呈现。

这里的要点是,添加一个商店确实意味着我们将应用程序性能更高或更容易优化,因为我们可以优化改变探测系统完全独立地从商店——这两件事可以一起使用,但没有内在联系。

采用存储体系结构的另一个共同点是可测试性，让我们来研究一下，这是在演示工具之前的最后一点。

商店和可测试性

引入存储的主要好处之一是它将提高应用程序的可测试性。  
的确，reducer函数很容易测试，但是通过引入存储并不能使应用程序本身变得更加可测试，因为我们通过依赖注入系统注入依赖，而不是直接在组件内部创建它们。

假设应用程序没有大量的数据修改或服务器和用户并发的数据修改:该应用程序可能不需要存储，引入存储也不会使其更具可测试性。  
但最后，当然也不是最不重要的，我们得到了一个巨大的好处——工具。

商店和工具  
使用存储的最大原因之一是它提供的工具生态系统。该工具非常棒，可以进行时间旅行的调试，能够将存储状态附加到错误报告中，以及热重新加载，这些都是巨大的特性。

这些工具非常棒，但是现在看来Redux在新的React应用程序中并不是必须使用的，那么从工具的角度来看，Redux是如何工作的呢?最初采用Redux一段时间后，许多React应用程序都是使用MobX构建的，这是可观察模式的一种变体。我们可以在文档中看到这样的描述:MobX为现有的数据结构(如对象、数组和类实例)添加了可观察的功能。这可以通过使用@observable装饰器(. next)注释类属性来完成。下面是它的一个小代码示例:如果您看过上面关于NgRx开发工具的视频，您是否熟悉它?看一看，还有一些[开发人员工具]

(https://github.com/zalmoxisus/mobx-remotedev) like the redux dev tools for Mobx:

实际上，Mobx开发工具也使用相同的浏览器插件。基于这个示例，似乎我们没有必要采用这种高级类型的工具来实现存储架构。我们所需要做的就是使用具有良好或不断发展的工具生态系统的可观察库来编写应用程序。其他相关生态系统的工具另一个生态系统与流的概念和可观测的模式及其变化有关，这就是CycleJs生态系统下面是关于Flux, Redux和一些CycleJs生态系统中的开发工具的演示。就在前面，我们讨论了如何添加工具而不附带预定义的架构(通过[@andrestaltz](https://twitter.com/andrestaltz))。  
整个演讲非常精彩，但如果您正在为您的应用程序选择一种架构，那么在13:03的5分钟讨论中会提出一些非常有趣的观点:

然后在最后演示了工具，请注意，许多这种高级工具通常是跨生态系统进行的工作。  
这里需要记住的最重要的事情是，在不采用存储架构的情况下，似乎还有其他方法可以获得优秀的工具。

结论  
很有可能，存储架构最初在React世界中流行是因为它们解决了一些基本问题，这些问题仅仅是视图并没有(通过设计)提供开箱即用的解决方案:

为解耦的组件交互提供一个可观察的模式

为临时UI状态提供一个客户端容器

为避免过多的HTTP请求提供一个缓存

为多个参与者并发数据修改

提供一个工具挂钩

然后半年到一年后，这个生态系统发展到只在某些应用程序类型中采用商店，而不是其他应用程序类型。同样的事情可能正在这个有棱角的世界里发生，结果也可能是一样的。让我们关注一下工具，RxJs 5的主要特性之一是改进的可调试性。

建议  
那么，这一切意味着什么呢?如果您正在为您的应用程序选择一种架构，那么应该做什么呢?React How-To的原话似乎仍然是个不错的建议:

*当你需要Flux的时候，你就会知道。如果你不确定是否需要它，你就不需要它*

这里有一个建议:除非您有一个并发的数据修改场景，否则可以考虑使用一些普通的RxJs服务来构建您的应用程序，利用本地服务和依赖注入系统。  
然后，如果需要，如果出现用例，我们总是可以将应用程序的一部分重构到一个商店中。

**另一方面，如果我们的应用程序中有一个并发的数据修改场景，我们也可以从一开始就使用存储，因为这是一个很好的解决方案。**

<https://github.com/angular-university/ngrx-course-v4> 旧

<https://github.com/angular-university/ngrx-course> 新

Angular储架构  
你是否想过，使用集中式存储解决方案构建应用程序的好处是什么，无论是在Angular还是在一般的框架中?

我们经常直接深入到Actions, Reducers和存储体系结构的相关术语中，但是这些概念(尽管很重要)只是达到目的的手段。

集中存储体系结构实际上是一组应用程序设计模式，因此最好的出发点可能是了解设计意图本身，以及该体系结构帮助解决哪些问题以及如何解决这些问题。

通过首先理解这一点，我们就不太可能冒误用架构或在没有充分利用架构的情况下使用它的风险。

**Store架构的转变**使用商店架构代表了一个巨大的架构转变:随着单页面应用程序的出现，我们将模型转移到查看从服务器到客户端的转换。

存储体系结构在逻辑上更进一步:我们现在还将模型从服务器转移到视图模型映射到客户端。  
但是这种架构上的转变带来了什么好处呢?它是如何工作的?我们如何将所有这些东西结合在一起:Ngrx Store, Actions, Reducers, Selectors, Effects ?

这就是我们将在这一系列文章中所涵盖的内容。

内容  
这篇文章将涵盖以下主题:

1 商店解决方案的起源，最初的Flux Facebook聊天问题  
2 Facebook反问题的根源——模型vs视图模型

3 Facebook聊天问题1 -同一业务数据的多视图模型  
4 Facebook聊天问题2 -共享数据所有权/数据封装问题

5 Facebook聊天问题3 -避免使用Ajax“微调器”应用程序  
6 解决Facebook的反问题——简单地说就是商店模式  
7 Ngrx Store, Actions, Reducers, Selectors，以及它们如何解决Facebook的反问题

8 结论

从哪里入手最好的存储解决方案?  
让我们回到最初，回到商店架构的起源:我们将回顾最初的Facebook柜台问题，它产生了商店架构，看看问题是什么，以及如何解决它。

引发Flux的Facebook聊天Bug  
开始了解Flux架构和存储解决方案的一个好地方是Flux最初提出的地方，因为它讨论了最初的问题本身。

这背后有一个很好的故事:所有的一切都源自一个对Facebook用户非常明显的大错误:未读消息计数器的问题。实际上，如果你用过Facebook一段时间，你可能还记得它。  
这与其说是一个bug，不如说是在Flux之前架构中的一个限制，正如我们将要看到的。

什么是Facebook的反漏洞?  
问题是未读消息计数器系统地显示不正确的结果:例如，用户会看到一条未读消息，然后点击计数器，所有消息都已读过:

只是一个简单的错误?  
对于开发人员以外的任何人来说，这似乎是一个需要解决的简单问题。但我们可能都遇到过这样的问题:我们一直在修复一个明显的小错误，但问题总是在多个变体中出现。

这就是发生在Facebook团队身上的事情:他们几个月来一直在修复这个问题，但不知怎么的，同一个错误又出现了。要么是发现了错误的新变体，要么是引入了新特性导致错误重新出现。

他们是如何解决这个问题的?  
事实证明，为了解决这个问题，Facebook团队必须对应用程序的架构进行重大修改。他们在早期的Flux谈话中分享了他们的新设计。

Flux架构的介绍从10:19开始，对于任何寻找存储架构信息的人来说，它都是一个必须关注的话题。  
最初的讨论讨论了stores、dispatchers、actions和Flux架构的所有元素。但同样有趣的是，演讲主要涉及了问题是如何解决的，而不是问题本身的原因。

我们知道计数器有一个反复出现的bug, Flux解决了这个问题，Flux基于几个主要的概念。但是问题的根源是什么呢?

那么问题是什么呢?

只是柜台里的一个窃听器，对吗?那么，为什么实施起来如此困难呢?演讲给出了几点提示。其中一个关键的句子是，要解决这个问题，他们需要一个解决方案:

*…“为客户端带来更多的真实数据，更少的派生数据”。*

Flux的引入解决了这个问题和计数器没有返回的问题。这意味着什么，最初的问题是什么，Flux是怎样解决的?

Facebook的计数器问题在许多早期单页应用程序中很常见

从根本上说，Facebook的反问题与我们将创建的任何用户界面都存在的根本区别密切相关。  
每个UI应用程序都有隐式或显式两种类型的数据:

1 模型，或领域模型

2 一个视图模型

例如，这是一个简化版的Facebook聊天应用程序(两个用户在不同的机器上使用这个应用程序):

通过查看用户界面，我们可以看到在这个应用程序中有三种类型的数据:

Messages

Threads

Participants

如果我们在Typescript中构建我们的程序，这些数据类型对应以下三种自定义类型的定义:  
注意，Message和Participant都是纯pojo，而Thread包含一个属性Participant，它是一个映射:键是一个Participant Id，值是该线程中该Participant的未读消息的数量。

我们还可以用普通的Javascript构建程序，而不需要定义强类型。但是即使类型不会直接存在于程序中，这3个领域模型概念仍然存在:

1 如果在开发团队用来谈论程序的术语中没有其他的

2 在编写的项目功能规范文件中

能够使用Typescript在语言本身的层次上定义这三种类型并使用它们来构建程序是很有帮助的。

但这些类型并不是我们在屏幕上看到的，不是吗?  
这些类型与我们通常在应用程序中定义数据库数据所使用的类型密切相关。如果我们使用SQL数据库，这几乎可以是3个数据库表的定义。

另外，如果我们使用像Firebase这样的NoSQL数据库，并选择将非规范化视图模型存储在数据库中，即使这样，这3个领域模型概念仍然会直接存在于程序中或隐式地存在于项目文档中。

围绕我们用来讨论它的领域模型概念来编写我们的程序是非常方便的。只有一个问题:这些类型并不直接对应于我们在屏幕上看到的内容。

模型vs视图模型  
我们在屏幕上看到的不是模型，而是视图模型——这是一个重要的区别。有时视图模型与模型有直接的对应关系——但大多数情况下它们是两个不同的模型。  
例如，在上面的应用程序中，这将是一对视图模型的例子:

*(注意VM后缀代表视图模型)*

ThreadSummaryVM是一个很好的例子，它说明了视图模型是如何非常接近模型本身的，但它并不是完全一样的东西。这个视图模型对应于屏幕左侧的线程列表:

但是列表中的每个条目并不对应于一个Thread模型实例。实际上，它对应的组合是:

1 一个Thread实例

2 几个参与者实例,因为我们可以看到他们的名单在逗号分隔的列表

3 他当前的参与者,是否他读线程的信息(未读线程列表中出现在红)

4 一个消息实例(线程)的最后一条消息

还有一个额外的状态位纯粹是ui独有的:当前选择了哪个线程。在本例中，用蓝色突出显示的列表中的第一个。  
右侧的messages列表组件也使用此ui状态(当前选择的线程Id)，以知道要显示哪些消息。稍后将详细介绍此UI状态，现在让我们关注域数据。

视图模型类似于什么?  
继续与SQL数据库的类比，上面定义为接口的3种自定义类型将对应于连接3个表的SQL查询的输出。  
我们不需要使用Typescript自定义类型来定义视图模型，但是这样做确实使它与模型的不同变得非常清楚。

一个模型到视图模型区别的例子  
为了更好地理解这种区别，下面是我们在屏幕上看到的第一个线程的模型数据:  
这是与第一个线程相关的所有数据——包括它的所有消息和参与者。

视图模型是什么样子的?  
但我们在屏幕左侧的线程列表中实际看到的是一个视图模型(ThreadSummaryVM)，它是线程的一个特定视图:

正如我们所看到的，尽管这两个数据结构相似，但它们并不完全相同:视图模型是模型的转换，是对模型的查询。  
例如，我们现在将read标志设置为true，因为我们查询了模型数据dbMessagesQueuePerUser，并发现当前登录的id为1的用户没有要读取的消息。

那么这和Facebook的计数器有什么关系呢?  
这可能不是立即明显，但在Facebook聊天的情况下，我们面临的主要问题之一，在实现它是有多个视图模型的同一模型在屏幕上显示在同一时间，包括:

1 线程摘要列表

2 当前所选线程的消息列表

3 还有未读消息计数器!

计数器本身是一个视图模型，它也是模型的一个转换。计数器是一个聚合视图，反映了从参与者获得的信息以及参与者能够访问的所有多线程和消息。

counter是模型的概要视图，需要与应用程序中的其他视图模型保持同步，这就引出了第一个不引入商店就很难解决的问题。

Facebook聊天问题1 -同一业务数据的多视图模型  
在用户界面中，我们需要显示相同领域数据(模型)的多个视图模型，我们如何保持一切同步?

每个视图层顶层组件(线程列表、消息列表或计数器)需要显示给定的数据视图模型，并且这些多个视图需要始终保持一致。

不是每个用户界面都有这个问题，也不是应用程序的所有屏幕都可能有这个问题。但是Facebook主页上打开的聊天窗口和上面的未读信息计数器显然存在这个问题。

但这是Facebook对抗bug的唯一基本设计限制吗?

**如果这是一个只读应用程序呢?**

在这种情况下，我们只需要加载数据，构建它的多个视图模型并显示它们，这也将解决问题。  
但另一个问题是，Facebook聊天是一个不断修改数据的应用程序，例如通过以下操作:

1 如果用户发送一条新消息，则该消息将被添加到当前列表中，

2 UI通过服务器推送不断从后端接收新消息

3 到达的新消息反映在未读消息计数器和线程列表中(最后一条消息被更新，未读线程被标记为红色)

4 当用户单击线程时，线程中的所有消息都被标记为红色，这会影响消息计数器

这意味着应用程序的多个部分是:  
1 不断修改相同的领域模型数据，2 同时在屏幕上显示不同的数据视图模型

这就引出了Facebook的第二个问题

Facebook聊天问题2 -共享数据所有权/数据封装问题  
如果应用程序的多个部分需要修改相同的数据，那么应用程序的哪个部分拥有数据?是共享所有权的情况，还是多个组件保留数据的本地副本，并在每次修改时通知彼此?

它们能共享对相同数据的引用并直接对其进行修改吗?让我们逐一看看这些选项。

**保留数据的本地副本?**

在需要生成视图模型的每个组件的级别上保留数据副本是行不通的，因为它不能很好地扩展复杂性。  
我们很快就会以意大利面条式的事件结束，应用程序的多个部分都需要通知彼此他们修改了一些东西。

**共享对相同数据的引用并改变它?**

这也不行，因为它在应用程序的多个部分之间引入了间接耦合。

我们再也不能仅仅通过查看组件和它的模板来判断一个组件——我们是松散的局部性。

这是因为组件数据是由应用程序中完全不相关的部分直接修改的，而当我们添加更多改变数据的特性时，将会出现各种无法正常工作的边缘情况。

在Facebook的反问题中，这也是一个明显的问题。

**这是一个追溯到OO编程的古老问题**问题是:谁拥有可修改的数据，如何封装可修改的数据，并保持它的私有，只对程序的一部分，而仍然使它对应用程序的其他部分可见?

或者我们应该公开它并使其不可变?这也将防止共享所有权错误的情况。但是有三分之一的问题与构建可维护的程序无关，而是与用户体验有关。

Facebook聊天问题3 -避免使用Ajax“微调器”应用程序  
对于上述问题，有一个简单的解决方案:我们可以直接从后端获取视图模型，这实际上是目前编写的大多数应用程序的情况。

因此，线程部分、消息列表和计数器组件可以通过单独调用或对后端进行一次大型公共调用来获得它们的视图模型。

然后，如果数据发生了一些修改:发送一个请求到后端，并简单地刷新一切——这样所有的数据将是一致的。

**这种方法的问题是什么?**  
这将导致早期类型的Ajax“微调器”单页面应用程序的出现，在这些应用程序中，数据会随着用户的每个动作不断地重新加载，这不是一种良好的用户体验，尤其是在移动设备上。

对于像Facebook这样的大型用户应用程序来说尤其如此，因为用户体验是非常重要的。

Facebook柜台问题的总结

至此，我们对Facebook应用程序中的问题有了一个很好的认识:Facebook主屏幕+聊天需要在同一屏幕上同时显示同一模型数据的多个视图模型。

这些数据不属于应用程序的任何特定部分，并且由多个独立的参与者(通过服务器推送的服务器和用户)不断修改。

每个视图模型都需要保持同步和一致，而不需要一直刷新。

Facebook反bug的主要问题

主要问题是，通常的预存储解决方案(将模型转换为服务器上的视图模型，并让后端返回视图模型给客户机)在这种情况下不能很好地工作。

现在让我们将Store应用程序体系结构模式作为这些多重问题的解决方案——它工作得非常好。该商店提供了一个优雅的、可维护的解决方案，以解决Facebook柜台问题所面临的所有问题。

集中式Store设计模式概括

如果我们回到Flux的话题，让我们来看看关键的一句话“把更多的真实数据带到前沿”。这是什么意思?这实际上是商店解决方案的核心。

这意味着,而不是将该模型转换为服务器上的视图模型作为其通常做的,让我们将模式复制到客户端,而是保持一个内存中的去年第二客户端数据库和派生视图模型在客户端而不是服务器上这样做前期。

让我们更详细地分析这个设计，并展示每个部分的一些代码。

如何处理数据?

1 我们将为应用程序数据创建一个客户端内存数据库

2 我们会把真实领域的数据，模型放到前端，而不是UI需要的多视图模型

3 我们将模型数据保留在内存数据库中，这实际上是数据库的用户特定切片（通常是分页的）

4 我们将把客户端内存数据库放在一个集中服务中，我们将其称为商店

5 我们将通过确保对其进行封装或将其公开为不可变的来确保集中式服务拥有数据

6 该集中式服务将具有反应性，我们可以订阅该服务以在模型数据更改时得到通知

视图模型是如何产生的？

这是此商店设计中视图层的结构方式：

1 每个需要产生数据视图模型的组件都将订阅以获取新版本的内存数据库数据

2 每个组件在收到模型数据后都会生成自己的视图模型，最后一秒在组件本身的级别

3这将确保所有视图模型始终与集中式数据模型同步

4 从模型到视图模型的转换最后一次是在视图根本身上完成的，而不是在服务器上完成的

5 从视图到视图模型的转换是通过称为选择器的函数完成的-选择器的输入是模型，而输出是视图模型

数据如何修改？

这是我们在保持所有视图同步的同时可以修改此商店设计中的数据的方式：

1 只能由商店本身在集中式服务中修改数据

2 如果组件想要触发数据修改，则它以命令有效载荷的形式将消息发送到集中式服务，这称为操作对象

3 动作的发出者不知道该动作将影响应用程序的哪些部分，涉及的多个View组件之间没有紧密的耦合

4 该动作包含触发一系列操作所需的所有信息，这些操作将导致修改模型数据

5 通过获取先前的状态和操作并将其应用为称为Reducer的纯函数，可以生成应用程序Model状态的新版本。

6 Reducer函数可以分为多个较小的函数，每个函数都修改状态的一部分

7 当新的状态模型可用时，它将广播到所有感兴趣的组件，然后将其转换为他们的View模型

8新的状态模型在广播之前已冻结，因此订户无法对其进行修改

商店内保存哪些数据？

存储不仅包含模型数据，还包含仅在客户端存在的任何特定于UI的全局状态，例如：

当前选择的线程ID

当前用户的数据

这是Facebook使用的确切解决方案吗？

Facebook最初使用的解决方案是使用多个商店，这些商店可以互相等待以得到通知，并包含一个数据子集。

上面的设计仅使用一个集中式商店（该设计由Redux流行），因此它已经是Facebook使用的解决方案的演进。但是解决方案的本质是相同的。

因此，让我们逐步分解该设计，并查看每个设计元素的外观。

安装商店解决方案

让我们从在应用程序中安装ngrx store开始：

npm install --save @ngrx/store @ngrx/core

这将使我们能够为我们的应用程序定义一个客户端内存数据库。我们要定义的第一件事是商店将保留哪种数据类型。

定义应用程序状态的形状

安装商店后，最好定义一个自定义类型ApplicationState：

如我们所见，内存数据库的内容可以分为两部分：一些状态与UI纯粹相关，我们将其定义为名为的自定义类型UiState：

此状态包括当前在我们的聊天应用程序中登录的用户，线程列表中当前选择的线程以及显示的当前错误消息（如果有）。

如我们所见，此状态与线程的数据完全不同，后者是在StoreData自定义类型中定义的：

请注意，数据不是按数组组织的，而是按地图组织的。对于这些映射，键是一个数字，对应于存储在映射中的元素的ID字段。例如，参与者的数据如下所示：

乍一看，这种存储数据的方式可能看起来像重复，因为我们重复了两次Id：一次在映射键中，另一次在对象本身中。

优化了这种存储数据的方式，以使数据尽可能浅，并避免了深层嵌套，并针对“通过ID查找”查询进行了优化。这种存储数据的方式与SQL数据库中的数据库表非常相似，其中主键是对象的ID字段。

这些数据类型定义了存储区中包含的数据的外观，但是数据的初始值又如何呢？

定义存储状态的初始值

除了定义内存数据的结构外，我们还可以通过以下方式定义每种数据类型的初始状态：

这将是内存中数据的初始值，而我们不会使用后端数据加载存储。在聊天应用程序的情况下，我们将使用初始请求的内容向后端加载商店。

如何填充内存数据库？

在这一点上，我们要做的是通过后端的纯HTTP请求获取数据并将其保存在存储中。

我们与商店互动的方式是通过向商店发送命令以某种预定义的方式修改其内部数据。该命令对象称为Action。该操作将以同步和立即的方式触发存储状态的修改。

那么异步动作呢？这也将在第2部分中介绍。现在，我们已经准备好将数据加载到存储中，那么如何加载它呢？

我们首先定义一个Action：

该动作包含类型和有效负载。在这种情况下，有效负载是一种传输对象数据结构，该结构与在初始后端请求中从后端获取的数据相对应：

如果我们有一个后端服务，可以让我们从后端返回数据，则可以通过以下方法将其分发到商店：

如我们所见，我们正在将命令对象和所有需要的数据发送到商店集中式服务。但是我们不知道此更改以及如何影响应用程序的其他部分。

但是存储将如何执行此操作，它将如何保存数据？

商店如何处理每个动作？

请记住，该操作的目标是以某种方式修改内存数据库内部的状态。

每次商店收到新操作时，它将采用当前状态加上该操作，并使用这两个输入来生成应用程序状态的新版本。

这个新状态是通过reducer函数产生的。reducer函数是一个纯函数（意味着它没有副作用），其签名与此类似：

此功能称为reducer，因为它与reduce功能编程操作共享相同的签名，并且它只是响应操作的分派而产生新存储状态的一种方式。

那么这个功能是什么样的呢？

reducer功能

请记住，Reducer函数只是我们试图实现的设计难题的一小部分。我们在这里所做的是将Model数据带到前端并保存在内存中，而reducer是一种以可维护的方式修改该数据的方法。

编写reducer函数的方法有很多，一种常见的方法是将其分解为较小的函数，一种会影响存储在存储中的每种状态。例如，这可能是有效的store reducer函数，它将把新状态的创建委派给较小的函数：

正如预期的那样，此对象的输出是的实例ApplicationState，该实例通过将其计算委托给较小的函数来构建uiState和storeData属性。

让我们看一下这些较小的reducer函数之一是什么样的：

reducer函数通常具有一个switch语句，该语句可以是action类型的。在其中，我们将添加特定于给定操作类型的处理逻辑：在这种情况下，我们通过保存每个ID的对应映射，将所有数据加载到存储中。

为了保存每个Id从后端收到的数据，我们使用LodashkeyBy实用程序功能。

该函数将从后端获取数据并初始化内存数据库。但是，多个View层组件如何使用此数据？

如何使用内存数据库中的数据？

如果只要有新数据可用就希望通知应用程序中的给定组件，最简单的方法是通过构造函数注入商店集中式服务。

视图层将存储视为应用程序状态的可观察对象。让我们回顾一下以反应式方式使用商店数据的组件的实现：

那么，该ThreadSection组件中发生了什么？让我们分解一下：

1 这是一个顶级组件，通过其构造函数获取服务

2 这种类型的组件也称为智能组件，因为它知道应用程序的服务层以及如何使用它

3 注入此智能组件的唯一服务是商店服务本身，无需将其他服务注入到View层

4 该组件的目标是进行存储并定义一系列从应用程序状态派生的数据流

5 所有成员变量都是可观察的，它们是使用select运算符从商店本身派生的

6 所有可观察的成员变量，例如threadSummaries$发出视图模型，而不是模型数据

7 该组件不会直接访问应用程序数据，也没有直接引用应用程序数据

8 组件通过分派存储操作来与应用程序的其余部分进行交互，因此存储实质上充当了View层外观服务，以确保ThreadSection组件与应用程序的其余部分保持分离

9 配置为成员变量的数据流将传递到模板，并使用async管道进行使用

如您所见，存储被视为应用程序状态的可观察状态，并用于派生视图模型流。在将数据注入视图层之前的最后一秒钟，将从存储中的模型数据派生出视图模型。

如何从应用程序状态生成视图模型？

通过应用选择器功能，将商店模型数据（保存在storeData里面的属性下ApplicationState）转换为视图模型。

Selector函数将应用程序状态作为输入，并从中返回特定的视图模型。例如，此函数返回未读消息计数器：

如我们所见，该函数只是获取应用程序状态，检索当前用户并遍历内存数据库中的所有线程。

然后，它对当前参与者的所有具有未读消息的线程进行求和，并生成未读消息的总数。

表现成分

该ThreadSection组件也是容器组件的一个很好的例子：它设置了一系列数据流，并使用async管道将实际数据传递给本地组件树。

组件本地树的组件之一是thread-list组件，让我们看一下它：

如我们所见，该组件与智能组件的构建方式非常不同。这种类型的组件称为表示性组件，让我们在这里细分该组件的工作方式：

1 该组件的主要设计目标是显示一些数据

2 数据通过 @Input()

3 该组件可以使用以下命令向其父组件发出事件 Output()

4 该组件更可重用，因为它可以在应用程序的其他位置使用以显示不同的线程列表

5该组件仅知道它接收到一个线程列表以及如何显示它，而不知道如何从后端检索数据

因此，这些都是集中式商店设计的基本元素，以及它们如何与整体设计目标融合在一起。

此解决方案如何解决Facebook柜台问题？

让我们记住，这完全是关于创建内存中的客户端数据库（该数据库是用户特定的一部分），并使用该数据在客户端上从中获取视图模型。

这很好地解决了Facebook柜台问题：

数据的多个视图始终根据设计保持同步

共享数据的所有权移至集中服务

用户体验得到了改善，因为我们不必不断刷新应用程序

至此，我们已经涵盖了集中式存储模式的基本部分。

结论

Ngrx生态系统不仅仅是一组库，实际上是一组应用程序设计模式：存储，操作，Reducer，Selectors，这些都是实现这些模式的手段。

为了更好地理解这些概念并知道它们如何组合在一起，我们建议首先从学习集中式存储模式本身开始，然后从那里深入了解具体的实现细节。

如果我们将它们作为实现整体应用程序设计的手段，那么与商店相关的所有概念（如动作，Reducer等）都将更易于理解。

集中式商店设计对于解决许多本来很难解决的问题非常有效，并且可以通过引入一些架构上的折衷来解决。

要查看使用ngrx store和Ngrx生态系统的许多其他库的实际存储模式示例应用程序，请查看最新的Ngrx示例应用程序。https://github.com/ngrx/platform

#### 商店和后端设计

但是，存储不仅与客户端有关，使用存储真实Domain数据的内存数据库还可以建立更多可重用的后端，这些后端发出Model数据而不是View模型。

例如，商店是GraphQL后端的绝佳组合，我们可以在每个请求中定义所需的确切模型数据-端点不返回预定义的数据结构，这与传统的REST端点不同。

这意味着我们只能为系统构建一个单一的后端，供UI和第三方服务使用，这在当前并不是最常见的情况。

通常，我们最终会为第三方集成构建特定于UI的后端和单独的REST后端。

通过将映射从服务器的模型到视图模型从服务器移动到客户端，并将查询语言功能提供给我们的后端，我们现在可以构建一个由UI和第三方使用的单个可重用后端。