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

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。  
为什么商店在反应如此受欢迎?