Model-View-Presenter with Angular

<https://indepth.dev/posts/1070/model-view-presenter-with-angular>

随着应用程序的增长，它变得越来越难维护。复杂性增加，而可重用模块的价值增加。我们知道我们必须在冒失败的风险之前采取行动。

设计模式来拯救!

复杂应用

一个复杂的应用程序至少具有以下特征之一:

1 组件树中的多个组件显示应用程序状态的相同部分

2 几个更新应用状态的来源，例如:

多个用户同时交互

实时向浏览器推送更新状态的后端系统

将后台任务

接近传感器或其他设备传感器

3 非常频繁地更新应用程序状态

4 大量的组件

5 用很多行代码构建的组件，让人回想起过去AngularJS的控制器

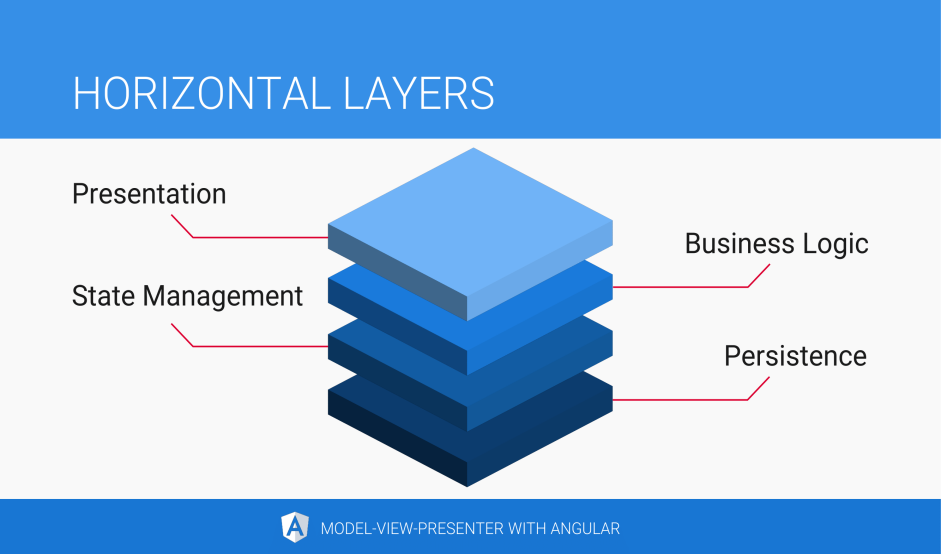
6 组件的高度圈复杂度——高度集中的逻辑分支或异步控制流

同时，我们需要一个可维护、可测试、可扩展和高性能的应用程序。

复杂的应用程序很少具备所有有价值的特性。我们不能避免所有复杂的特性，同时仍然满足高级项目的要求，但我们可以设计我们的应用程序，使其有价值的特性最大化

关注点分离(Separation of concerns)

我们可以将关注点分离(SoC)看作应用程序的划分。我们将逻辑按系统关注点进行分组，以便能够一次关注单个关注点。在最顶层，关注点分离是一个体系结构规程。在日复一日的发展中，它几乎是熟记什么要往哪里去。



我们可以垂直地、水平地或同时对应用程序进行分割。当垂直分割时，我们按特性对软件工件进行分组。水平切片时，我们按软件层分组。在我们的应用程序中，我们可以将软件构件分为这些水平层，或者系统关注点:

web应用程序的水平层。

同样的规则也适用于我们的Angular组件。它们应该只关注表示层和用户交互层。结果是我们放松了系统中运动部件之间的耦合

当然，这个过程需要很多规则，因为我们要添加额外的抽象层，但最终结果的有价值的特性弥补了这一点。请记住，我们只是在创建一开始就应该存在的抽象。

The Model-View-Presenter pattern

模型-视图-演示者(通常缩写为MVP)是一种用于实现应用程序用户界面(UI)的体系结构软件设计模式。我们使用它来最小化难以测试的类、函数和模块(软件工件)中的复杂逻辑。特别地，我们要避免使用复杂的ui特定软件构件，比如Angular组件。

类似于模型-视图-控制器-它衍生的模式-模型-视图-表示器将表示与域模型分离。表示层通过应用观察者模式来响应领域中的变化，正如Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides(也被称为“四人帮”)在他们的经典著作《设计模式:可重用的面向对象软件的元素》中所描述的那样

在Observer模式中，subject维护一个观察者列表，当状态发生变化时它会通知该列表。这听起来熟悉吗?你猜对了，RxJS是基于观察者模式的。

除了以数据绑定和小部件组合的形式，视图不包含任何逻辑或行为。当用户交互发生时，它将控制权委托给演示者。

演示者批量修改状态，这样用户填写表单时就会导致一个大的状态变化，而不是许多小的变化，例如，每个表单更新一次应用程序状态，而不是每个字段更新一次。这使得撤消或重做状态更改变得很容易。演示者通过向模型发出命令来更新状态。由于观察者同步，状态更改反映在视图中。

The Angular variation(变化)

模型-视图-表示器可以与Angular结合使用。图来自我的演讲“使用Angular的模型-视图-演示者”(幻灯片)。

受原始的模型-视图-展示器模式和变体的启发，我们将创建适合Angular平台及其关键UI构件组件的软件构件。

理想情况下，Angular组件只关注表示和用户交互。实际上，我们必须严格遵守规则，以确保我们的组件只关心向用户呈现应用程序状态的一部分，并使它们能够影响该状态。

本文中引入的模型-视图-表示器变体是封装表示器风格的一种体现。然而，我们的演讲者将不会提及他们的观点。相反，我们将使用可观察对象将演示者连接到模型和视图，从而使演示者可以从其视图中独立地进行测试。

在应用模型-视图-表示器模式时，我们倾向于使用监督控制器方法。我们的视图(Angular组件)只是简单地依赖于它们的呈现器来进行用户交互。由于我们的演示者被他们的视图封装，数据和事件都在某个点流经组件模型。

在组件模型的帮助下，演示者将用户交互转换为特定于组件的事件。这个事件被转换成一个命令，发送到模型。最终的转换由所谓的容器组件来处理，我们很快就会介绍这些组件。

我们的演示器将具有表示模型的一些特征，因为它将包含表示逻辑，如布尔或可观察属性，以指示是否应该禁用DOM元素。另一个例子是指示DOM元素应该呈现哪种颜色的属性。

我们的视图绑定到演示者的属性上，只需要投影它所表示的状态，而不需要任何附加逻辑。其结果是一个具有非常简单的组件模板的瘦组件模型。

Model-View-Presenter concepts for Angular

要把模型-视图-展示器模式应用到Angular应用中，我们将引入一些概念，这些概念很大程度上是受到React社区的启发。就本文而言，我们的组件将属于以下三个类别之一:

Presentational components

Container components

Mixed components

React开发人员已经从混合组件中提取表示组件和容器组件好几年了。我们也可以在Angular应用中使用同样的概念。此外，我们将介绍presenters(主持人)的概念

Presentational components

表示性组件是纯粹的表示性和交互式视图。它们向用户呈现应用程序状态的一部分，并使它们能够影响应用程序的状态。

除了演示程序之外，演示组件完全不知道应用程序的任何其他部分。它们有一个数据绑定API，用来描述它们处理的用户交互和它们需要的数据。

为了消除对UI进行单元测试的大多数原因，我们将表示组件的复杂性保持在绝对最小的程度，无论是组件模型还是组件模板。

Container components

容器组件向表示组件公开应用程序状态的片段。它们通过将特定于组件的事件转换为非表示层的命令和查询，将表示层与应用程序的其余部分集成在一起。

通常，我们在容器组件和表示组件之间有一对一的关系。容器组件具有与其表示组件的输入属性相匹配的类属性，以及响应通过表示组件的输出属性发出的事件的方法。

Mixed components

如果一个组件不是容器组件或表示组件，它就是一个混合组件。给定一个现有的应用程序，它很有可能由混合的组件组成。我们称它们为混合组件，因为它们具有混合的系统关注点——它们包含属于多个水平层的逻辑。

如果您偶然发现一个组件——除了包含用于表示的域对象数组之外——直接访问设备摄像机、发送HTTP请求并使用WebStorage缓存应用程序状态，请不要感到惊讶。

虽然这种逻辑在应用程序中是预期的，但将其全部分组在一个单独的位置会使其难以测试、难以推理、难以重用和紧密耦合。

Presenters

将行为逻辑和复杂的表示逻辑提取到演示程序中，得到一个简单的表示组件。演示者没有UI，而且通常没有或者只有几个注入的依赖项，这使得测试和推理变得很容易。

演示者很少知道应用程序的其余部分。通常，只有一个表示组件引用一个表示器。进入翻译页面

The Model-View-Presenter triad

这三个软件工件被组合成我们所说的模型-视图-表示者三元组。由容器组件表示的模型是显示给用户以供他们浏览和修改的应用程序状态。

由表示组件表示的视图是一个瘦用户界面，它表示应用程序状态，并将用户交互转换为特定于组件的事件，通常将控制流重定向到表示程序。

演示者通常只是一个类的实例，它完全不知道应用程序的其余部分。

Data flow

Let us visualise how data and events flow through a Model-View-Presenter triad.

数据沿着组件树向下流动

图2。数据流开始于服务，结束于DOM。在新标签页中打开。

在图2中，服务中发生了应用程序状态更改。container组件会收到通知，因为它已经订阅了服务上的一个observable属性。

在图2中，服务中发生了应用程序状态更改。container组件会收到通知，因为它已经订阅了服务上的一个observable属性。

演示组件将更新后的数据传递给演示者，演示者重新计算演示组件模板中使用的附加属性。

现在，数据已经完成了组件树的向下流动，Angular将更新后的状态呈现给DOM，并以列表的形式显示给用户。

事件沿组件树向上流动

图3。以用户交互开始并以服务结束的事件流。在新标签页中打开。

在图3中，用户单击一个按钮。Angular把控制权交给了表示组件模型中的事件处理程序，因为它的模板中有一个事件绑定。

用户交互被演示程序截获，它将用户交互转换成数据结构，并通过一个可观察属性发出。表示组件模型观察更改并通过输出属性发出值。

Angular会把特定于组件的事件中发出的值通知给容器组件，因为它的模板中有一个事件绑定。

现在事件已经完成了组件树的向上流动，容器组件将数据结构转换为参数，并将其传递给服务上的方法。

在更改应用状态的命令之后，服务通常会在其可观察属性中发出状态更改，数据再次向下流动到组件树中，如图2所示。

改进的Angular应用程序

有些人会认为我们的新UI架构是过度工程的一个过于复杂的结果，而实际上我们剩下的只是许多简单的、模块化的软件。模块化的软件架构使我们能够做到敏捷。不是敏捷过程和仪式意义上的敏捷，而是在改变成本方面的敏捷。

模块化的软件架构使我们更加敏捷。

我们主动处理客户需求的变化，而不是不断增加的技术债务。如果系统是紧密耦合的，而且很难测试，需要几个月的时间来重构，那么就很难达到这样的敏捷程度。

Maintainable（可维护的）

尽管最终的系统由许多运动部件组成，但每个部件都非常简单，只处理单个系统关注的问题。此外，我们有一个明确的系统来决定什么东西要去哪里。

Testable 可测试的

我们将特定于angular的软件工件中的逻辑最小化，因为它们通常很难测试，而且测试速度很慢。由于每个软件都只关注一个系统关注点，因此很容易对它们进行推理。我们可以很容易地在自动化测试中验证的假设。

UI的测试尤其困难和缓慢，Angular也不例外。使用模型-视图-表示器，我们将表示组件中的逻辑数量最小化到几乎不值得测试的程度。相反，我们可以选择完全跳过单元测试，并依赖于我们的开发工具、集成测试和端到端测试来捕获简单的错误，如拼写错误、语法错误和未初始化的属性。

Scalable可扩展的

特性可以在相互隔离的情况下开发。甚至可以在单独的水平层中开发和测试软件工件。我们知道每个逻辑的确切位置。

既然我们可以独立开发层，我们就可以区分技术前端开发和可视化前端开发了。有的开发人员擅长使用RxJS实现行为，有的开发人员喜欢后端集成，还有的开发人员喜欢完善设计，用CSS和HTML解决易访问性问题。

因为我们可以独立开发特性，所以任务可以在团队之间分开。一个团队关注产品目录，而另一个团队处理电子商务系统中购物车的问题和新特性。

Performant

适当的关注点分离通常会给我们带来高性能，尤其是在表示层。性能瓶颈很容易跟踪和隔离。

通过OnPush变更检测策略，我们将Angular变更检测周期对应用程序性能的影响降到最低。

案例分析:《英雄指南》

我们从角。io“英雄指南”教程结束。它之所以被用作我们的起点，是因为它是Angular开发者所熟知的一个教程。

在《英雄指南》最后的教程代码中，所有的组件都是混合组件。考虑到它们都没有输出属性，但其中一些会改变应用程序状态，这一事实是显而易见的。

在相关文章中，我们将把模型-视图-表示器模式应用到这些组件的选择中，并逐步提供大量实际操作的代码示例。我们还将讨论在模型-视图-表示器三元组中测试哪些行为。

你会注意到，我们并没有改变应用的任何特性或行为，而只是将它的Angular组件重构为更专业的软件构件。

虽然这些文章只讨论了《英雄指南》中的部分组件，但我已经将模型-视图-演示者模式应用到整个应用程序中，并在这个GitHub存储库中为容器组件和演示者添加了测试套件。in [this GitHub repository](https://github.com/LayZeeDK/ngx-tour-of-heroes-mvp).

Prerequisites

除了本文介绍的概念之外，我希望您只熟悉Angular的几个关键概念。模型-视图-表示器的概念在相关的文章中有详细的解释。

我希望你能很好地理解Angular组件，比如数据绑定语法以及输入和输出属性。我还假设您具有基本的RxJS知识——您对可观察对象、主题、操作符和订阅有些熟悉

我们将构建独立的单元测试，在其中使用Jasmine间谍来存根服务依赖项。存根和其他测试双精度对理解测试来说不是关键。请将您的注意力集中在测试用例上，并尝试理解为什么我们要测试由测试执行的行为。

Resources

[Browse the final Tour of Heroes tutorial code on StackBlitz](https://angular.io/generated/live-examples/toh-pt6/stackblitz.html).

[Download the final Tour of Heroes tutorial code](https://angular.io/generated/zips/toh-pt6/toh-pt6.zip) (zip archive, 30 KB)

[Browse the Tour of Heroes—Model-View-Presenter style repository on GitHub](https://github.com/LayZeeDK/ngx-tour-of-heroes-mvp).

[Watch my talk “Model-View-Presenter with Angular” from ngAarhus May 2018](https://youtu.be/D_ytOCPQrI0) meetup:

相关文章

解模型-视图-表示器模式的历史，以及它的兄弟模式模型-视图-控制器是如何引入到web的客户端UI框架中的。读“[The history of Model-View-Presenter](https://indepth.dev/the-history-of-model-view-presenter/)”.

你是否厌倦了担心Angular组件中的状态管理和后端内容?将所有讨厌的非表示逻辑提取到容器组件中。读到的 “[Container components with Angular](https://indepth.dev/container-components-with-angular/)”.

学习如何使用快速单元测试测试容器组件逻辑“[Testing Angular container components](https://indepth.dev/testing-angular-container-components)”.

"[Presentational components with Angular](https://indepth.dev/presentational-components-with-angular/)" 讨论纯粹的、确定性的和潜在可重用的组件，这些组件仅依赖于输入属性和用户交互触发的事件来确定其内部状态。

In “[Lean Angular components](https://indepth.dev/lean-angular-components/)”, 我们将讨论健壮的组件体系结构的重要性。模型-视图-表示器封装了几个帮助我们实现这一目标的模式。

实现高度的关注点分离是受到Robert“Bob叔叔”Martin作品的启发，特别是他的书“清洁架构:软件结构和设计的工匠指南”。

将模型-视图-演示器模式应用到Angular应用中，灵感来自于Dave M. Bush的文章《模型视图演示器，Angular和测试》。 “[Model View Presenter, Angular, and Testing](https://davembush.github.io/model-view-presenter-angular-and-testing/)” by [Dave M. Bush](https://www.linkedin.com/in/davembush/).

在我最初的研究中，我检查了Roy Peled的文章“JavaScript的MVP指南-模型-视图-演示者”中描述的普通JavaScript的模型-视图-演示者模式。 “[An MVP guide to JavaScript — Model-View-Presenter](http://www.roypeled.com/an-mvp-guide-to-javascript-model-view-presenter/)” by [Roy Peled](https://www.linkedin.com/in/roypeled/).

Presenters with Angular

<https://indepth.dev/posts/1201/presenters-with-angular>

Presenters （演示程序）是组件级服务，它封装了复杂的表示逻辑和用户交互。它们可以与平台或框架无关，支持跨应用程序、平台和设备的一致UI行为。我们在表示组件和混合组件中提供并使用它们。

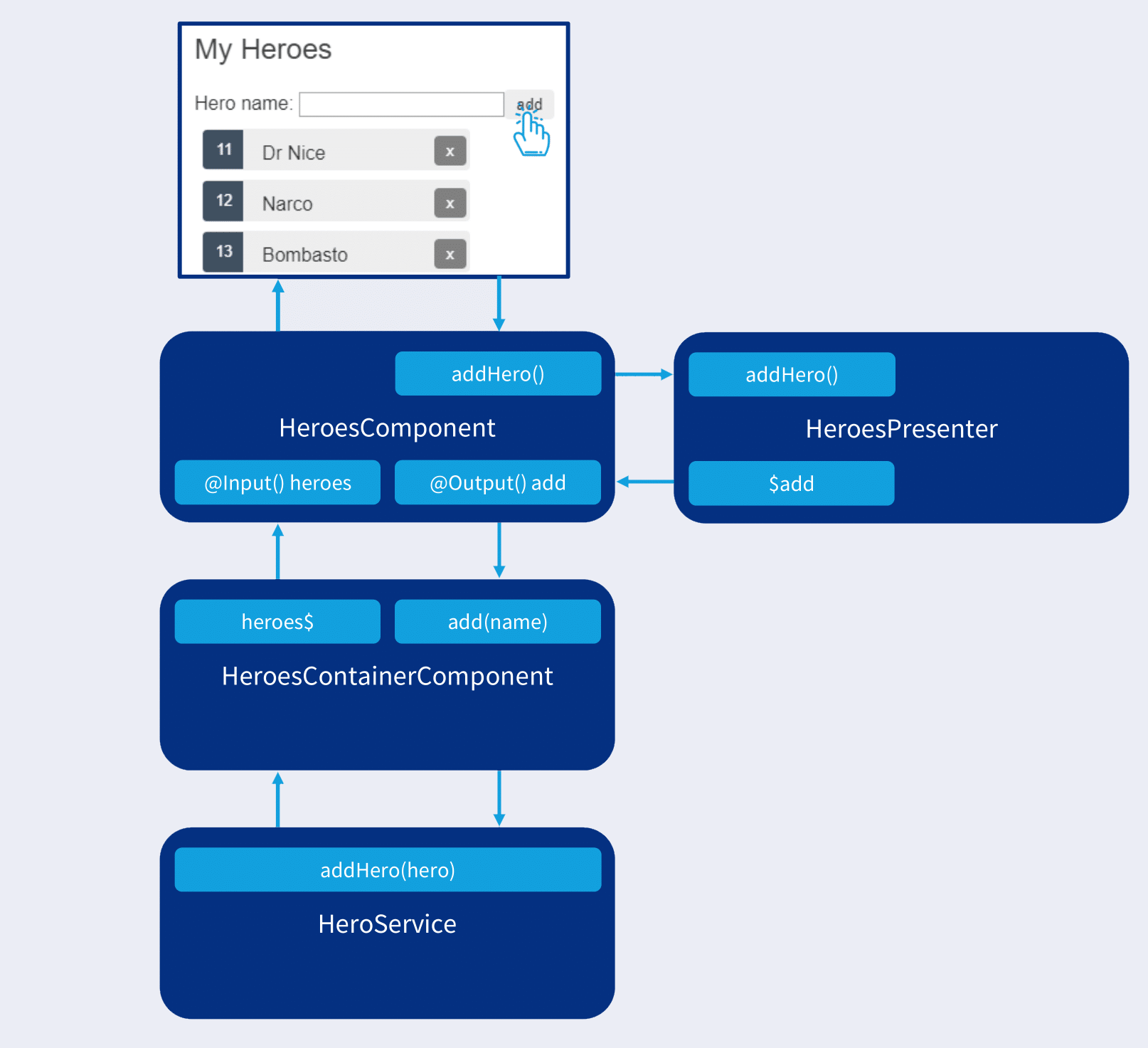
演示程序实际上与应用程序的其他部分隔离开来。它们通常根本没有依赖关系，除了复合演示程序中的其他演示程序。这使得它们在没有DOM或UI的情况下，甚至在没有Angular的情况下，都很容易被单独测试，只要我们仔细设计它们。

演示者对所呈现的应用程序状态进行格式化，管理本地UI状态，并提供一个用户交互的接口。

显示的应用状态可以被建模为RxJS的可观察对象、常规属性或组件输入属性传递的简单方法。

在讨论了不同类型的演示器以及将它们与组件组合在一起的不同方法之后，我们回到《英雄指南》教程中的HeroesComponent重构。

我们最终得到了如图1所示的控制流。



有状态的Presenters

Presentational 组件，相应地，演示者并不关心应用程序的状态来自哪里。它们管理自己对其消费组件感兴趣的应用程序状态的任何部分的同步表示。

有状态的Presentational 可以对许多不同类型的应用程序状态建模，包括持久状态、客户端状态、临时客户端状态和本地UI状态。这要么表示为常规属性，要么表示为可观察属性。

无状态的Presenters

无状态Presenters不使用属性、主题或其他类型的可观察对象来实现本地UI状态。相反，它们只转换数据，这使得它们最适用于表示，而不是用户交互。  
因为我们更喜欢将本地UI状态委托给展示程序，所以单个无状态展示程序很难满足组件的所有需求。

Component-to-presenter ratios

每个组件需要多少presenter ?答案是，我们可以任意组合它们。让我们来讨论一下不同的组件-presenter 比率，以及何时使用它们是有意义的。

***每个组件一个presenter***

对于特定于用例的组件，我们通常从单个表示组件使用单个表示器开始。我们有一个1:1的组件与presenter 比例。

一旦组件的表示逻辑开始变得复杂，我们可以选择将该逻辑提取到特定于该组件的表示器中，作为第一步。随着组件的增长，我们可以选择将其分成多个组件，使用1:1的presenter 。

复合presenter 在底层使用其他presenter ，基本上是其他presenter 的外观。它要么是特定于组件的，要么是特定于行为的。在行为特定的情况下，它将可重用的、专门的presenter 组合成更复杂的UI行为。如果它绑定到一个特定的组件，我们通常会有一个1:1的组件与presenter 比例。

***每个组件有多个presenter***

随着项目的发展，我们应该找到越来越多的机会来增加特性和用例之间的代码重用。此时，单个组件将使用多个演示者，给出1:n组件与演示者的比例。

我们也可以有多个特定于同一用例的演示者，但是处理不同的关注点。例如，许多组件同时具有表示逻辑和行为逻辑。我们可以为每一个问题都安排一个presenter

也许我们的组件在它的模板中有一个特定的部分，对于这两个非常有内聚性的关注点具有复杂的操作。我们可以有一个演示者来处理这部分模板的两个关注点。但是要注意，这通常表明我们应该提取一个子组件，而不是仅仅在演示器中封装这个逻辑。

***一个演示者在多个组件之间共享。***

还有一种情况是，单个演示者分发应用程序状态并编排多个组件之间的用户交互。这有一个n:1组件与演示者的比率。

无状态表示器可以很容易地在多个组件之间共享，特别是同一个组件的多个实例。如果它们有独立的本地UI状态，它们将需要单独的演示者实例，这意味着1:1或1:n的比例。

共享演示程序的另一个例子是由多个组件组成的复杂数据表UI。单个容器组件提供应用程序状态，并将用户交互转换为转发给服务的命令。

每个行或单元格组件可以拥有一个或多个演示程序，特定于行或特定于列的演示程序处理UI行为、本地UI状态、表单验证和格式化。

我们可以有一个单独的、跨列的、表级的演示程序，它将应用程序状态从容器组件分发到单独的行级和单元级组件以及它们的演示程序。它还将编排源自行和单元级组件和演示程序的用户交互。

何时使用组件而不是presenter（表示器）

我们可以在一个组件中使用多个演示者来封装不同的关注点。如果是这样的话，为什么我们不将表示或用户交互逻辑封装在一个单独的组件中呢?

一个原因可能是我们不能引入更多的DOM元素，因为我们使用的第三方库或有效的HTML结构阻止我们这么做。每个DOM元素只能有一个Angular组件。解决这个问题的一种方法是让单个组件编排多个演示者。

解决刚性DOM结构问题的另一种方法是使用容器指令或提供者指令，我们将在另一篇文章中讨论。

如果可以引入额外的DOM元素，或者可以将组件应用于现有的DOM元素，那么什么时候应该使用组件而不是表示器来封装表示逻辑呢?

假设我们有一个如清单1所示的搜索演示程序。

// search.presenter.ts

import { OnDestroy } from '@angular/core';

import { Subject } from 'rxjs';

export class SearchPresenter implements OnDestroy {

private searchQuery = new Subject<string>();

searchQuery$ = this.searchQuery.asObservable();

ngOnDestroy(): void {

this.searchQuery.complete();

}

search(query: string): void {

this.searchQuery.next(query);

}

}

这是一个可重用的演示器，可以在有搜索框的多个组件中重用。

拥有一个可重用的演示者的好处是我们可以在一个地方改变搜索行为。假设我们希望取消搜索查询并消除连续的重复，因为用户将在物理键盘或软键盘上输入。在清单2所示的可重用表示器中很容易进行此更改。

// search.presenter.ts

import { OnDestroy } from '@angular/core';

import { Subject } from 'rxjs';

import { debounceTime, distinctUntilChanged } from 'rxjs/operators';

export class SearchPresenter implements OnDestroy {

private searchQuery = new Subject<string>();

searchQuery$ = this.searchQuery.pipe(

debounceTime(150), // 👈

distinctUntilChanged(), // 👈

);

ngOnDestroy(): void {

this.searchQuery.complete();

}

search(query: string): void {

this.searchQuery.next(query);

}

}

作为实验，让我们按照清单3将这个演示器绑定到一个搜索框组件。

// search-box.component.ts

import { Component, EventEmitter, OnInit, Output } from '@angular/core';

import { SearchPresenter } from './search.presenter';

@Component({

providers: [SearchPresenter],

selector: 'app-search-box',

template: `

<input

type="search"

placeholder="Search..."

(input)="onSearch($event.target.value)"> <!-- [1] -->

`,

})

export class SearchBoxComponent implements OnInit {

@Output()

search = new EventEmitter<string>();

constructor(

private presenter: SearchPresenter,

) {}

ngOnInit(): void {

this.presenter.searchQuery$.subscribe(searchQuery => // [4]

this.search.emit(searchQuery)); // [4]

}

onSearch(query: string): void { // [2]

this.presenter.search(query); // [3]

}

}

我们故意只让数据流指向一个方向。用户输入搜索查询(1)截获的组件的事件处理程序(2)。查询然后透过主持人(3)。最后,主持人的搜索查询的连接组件的输出属性(4),允许父组件使用事件绑定到用户搜索的通知。

我们有效地将搜索演示者与搜索框绑定在一起。如果这是我们使用这个用户交互逻辑的唯一地方，那么我们也可以重用搜索框组件，而不是搜索展示器。这样，我们的消费者(或父组件)只需要使用搜索框组件并绑定到它的搜索事件来添加搜索功能。

如果搜索功能在不同用例中的工作方式有一些不同，我们可能会发现重用搜索表示器比重用搜索框组件更好。

我们必须在每个使用常见搜索行为的组件中编写类似于清单3中的粘合代码。这样做的好处是，我们可以很容易地为搜索查询observable添加额外的响应式操作。它们可以添加到组件或另一个表示器中，例如特定于组件的表示器或复合表示器。

总而言之，当演示程序和DOM片段之间具有高度内聚性时，我们可以重用组件(带有演示程序)而不是演示程序。我们还需要非常确定我们总是会在我们的应用中使用准确的行为，没有任何变化。

何时使用管道而不是展示器

通常，我们通过转换的表示器方法传递UI属性或输入属性。其他时候，我们通过可观察对象的操作来管道它们，这些可观察对象的操作最终会连接到组件模板，比如使用异步管道或NgRx推送管道。

在转换方法的情况下，每当我们的组件被脏检查时，都会对其进行评估，这在代价高昂的转换情况下可能会降低性能。但是，我们可以记住它，以便在以后查找相同值的转换。

当呈现器与表示组件配对时，性能下降可以忽略不计，因为只有在其输入属性发生更改时才会对其进行脏检查。然而，一些输入值的变化非常频繁。

记忆化的Angular管道会缓存所有之前转换过的结果，以固定的时间返回它们。

纯Angular管道在脏检查中求模板表达式的当前值和参数是否与前一个相同时，会短路。基本上是一个带有单个值缓冲区的内存化管道。

在性能关键的用例中，我们可以使用纯Angular管道或记忆管道来代替演示器。

这样做的代价是，Angular管道只关心一个单独的值，这是一个非常细粒度的表示逻辑。很难将管道转换与它们所处的其他用例集成在一起进行测试。我们必须通过DOM进行测试才能做到这一点。

另一个代价是，Angular管道需要相对大量的设置，包括Angular模块声明、导出和导入的间接形式，以及组件模板中使用的管道名。

最后，Angular管道只封装了表示逻辑。它们不能用于任何形式的用户交互。

简单的例子

在《Angular的演示组件》中，我们把演示逻辑从heroes组件模板移到了heroes组件模型中，以最小化模板中的逻辑

// heroes.component.ts

import { Component, EventEmitter, Input, Output } from '@angular/core';

import { FormControl } from '@angular/forms';

import { Hero } from '../hero';

@Component({

selector: 'app-heroes-ui',

styleUrls: ['./heroes.component.css'],

templateUrl: './heroes.component.html',

})

export class HeroesComponent {

@Input()

heroes: Hero[];

@Input()

title: string;

@Output()

add = new EventEmitter<string>();

@Output()

remove = new EventEmitter<Hero>();

nameControl = new FormControl('');

addHero(): void {

let name = this.nameControl.value;

this.nameControl.setValue(''); // [2]

name = name.trim(); // [1]

if (!name) { // [1]

return;

}

this.add.emit(name);

}

}

清单4。英雄:带有表单验证和UI行为的表示组件模型

在清单4中，我们看到addHero方法中有用于表单验证(1)和UI行为(2)的复杂用户交互逻辑。

将复杂的表示逻辑提取到表示器中

让我们通过从presentational组件中提取复杂的表现逻辑来创建一个heroes演示器。

// heroes.presenter.ts

import { FormControl } from '@angular/forms';

export class HeroesPresenter {

nameControl = new FormControl(''); // [2]

addHero(): void { // [1]

const name = this.nameControl.value.trim();

this.nameControl.setValue(''); // [3]

if (!name) {

return;

}

this.add.emit(name); // [4]

}

}

清单5。Heroes:Presenter(演示者)，带有提取的表单控件和相关方法。

我们把addHero方法(1)提取到一个名为HeroesPresenter的特定组件的演示器中。

我们需要在演示者(2)中包含name form控件，因为addHero方法通过清除form控件(3)来控制UI行为。

该方法的最后一条语句以前被用来通过组件输出属性(4)发出一个值。

我们可以给这个演示器添加一个Angular的事件发射器，但我们更喜欢让演示器与框架无关，至少在有意义的地方是这样，所以我们决定使用RxJS主题来代替，如清单6所示。此外，一旦我们在事件发射器上添加了任何操作，它就会被更改为可观察类型。

// heroes.presenter.ts

import { FormControl } from '@angular/forms';

import { Observable, Subject } from 'rxjs';

export class HeroesPresenter {

private add = new Subject<string>(); // 👈

add$: Observable<string> = this.add.asObservable(); // 👈

nameControl = new FormControl('');

addHero(): void {

const name = this.nameControl.value.trim();

this.nameControl.setValue('');

if (!name) {

return;

}

this.add.next(name);

}

}

清单6。英雄:演示者和被观察对象暴露在一起。

演示者现在有了一个公开的可观察对象add$属性，我们的演示组件可以连接到它。

API设计提示:我们不应该公开主题或事件发射器，除非作为组件输出属性。

将presenter（演示程序）注入（presentational）演示组件

我们想把heroes演示器注入到presentational组件的构造函数中。为此，我们将其作为组件级服务提供，如清单7所示。

// heroes.component.ts

import { Component, EventEmitter, Input, Output } from '@angular/core';

import { Hero } from '../hero';

import { HeroesPresenter } from './heroes.presenter';

@Component({

providers: [HeroesPresenter], // 👈

selector: 'app-heroes-ui',

styleUrls: ['./heroes.component.css'],

templateUrl: './heroes.component.html',

})

export class HeroesComponent {

@Input()

heroes: Hero[];

@Input()

title: string;

@Output()

add = new EventEmitter<string>();

@Output()

remove = new EventEmitter<Hero>();

constructor(

private presenter: HeroesPresenter,

) {}

addHero(): void {}

}

清单7。英雄:带有演示器的演示组件。

**：**演示者被添加到providers组件选项中，该选项将其范围扩展到组件级别，这意味着演示者的生命周期遵循组件的生命周期。它在演示组件之前被实例化它在组件出现之前被销毁。

将UI属性和事件处理程序委托给演示者（presenter）

现在，presentational heroes组件已经可以访问演示器了，我们可以把UI属性和事件处理程序委托给它。

// heroes.component.ts

import { Component, EventEmitter, Input, Output } from '@angular/core';

import { FormControl } from '@angular/forms';

import { Hero } from '../hero';

import { HeroesPresenter } from './heroes.presenter';

@Component({

providers: [HeroesPresenter],

selector: 'app-heroes-ui',

styleUrls: ['./heroes.component.css'],

templateUrl: './heroes.component.html',

})

export class HeroesComponent {

@Input()

heroes: Hero[];

@Input()

title: string;

@Output()

add = new EventEmitter<string>();

@Output()

remove = new EventEmitter<Hero>();

get nameControl(): FormControl {

return this.presenter.nameControl; // 👈

}

constructor(

private presenter: HeroesPresenter,

) {}

addHero(): void {

this.presenter.addHero(); // 👈

}

}

清单8。Heroes:演示组件将UI属性和事件处理程序委托给它的演示者。

如清单8所示，heroes组件创建了一个nameControl getter，并将其委托给演示者。它还将其addHero事件处理程序的控制权转发给演示者的addHero方法。

将演示者连接到演示组件的数据绑定API

为了完成这个重构，我们还有一些事情要做。首先，让我们连接演示者的observable属性和组件的output属性。

// heroes.component.ts

import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

import { FormControl } from '@angular/forms';

import { Hero } from '../hero';

import { HeroesPresenter } from './heroes.presenter';

@Component({

providers: [HeroesPresenter],

selector: 'app-heroes-ui',

styleUrls: ['./heroes.component.css'],

templateUrl: './heroes.component.html',

})

export class HeroesComponent implements OnInit {

@Input()

heroes: Hero[];

@Input()

title: string;

@Output()

add = new EventEmitter<string>();

@Output()

remove = new EventEmitter<Hero>();

get nameControl(): FormControl {

return this.presenter.nameControl;

}

constructor(

private presenter: HeroesPresenter,

) {}

ngOnInit(): void {

this.presenter.add$.subscribe(name => this.add.emit(name)); // 👈

}

addHero(): void {

this.presenter.addHero();

}

}

清单9。Heroes:表示组件，它的数据绑定API连接到它的演示者。

在清单9A中，我们订阅了演示者的add$ observable，并把发出的值转发给了heroes组件的add output属性。

// heroes.component.ts

import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

import { FormControl } from '@angular/forms';

import { Hero } from '../hero';

import { HeroesPresenter } from './heroes.presenter';

@Component({

providers: [HeroesPresenter],

selector: 'app-heroes-ui',

styleUrls: ['./heroes.component.css'],

templateUrl: './heroes.component.html',

})

export class HeroesComponent implements OnInit {

@Input()

heroes: Hero[];

@Input()

title: string;

@Output()

add = new EventEmitter<string>();

@Output()

remove = new EventEmitter<Hero>();

get nameControl(): FormControl {

return this.presenter.nameControl;

}

constructor(

private presenter: HeroesPresenter,

) {}

ngOnInit(): void {

this.presenter.add$.subscribe(this.add); // 👈

}

addHero(): void {

this.presenter.addHero();

}

}

清单9 b。Heroes:表示组件，它的数据绑定API连接到它的演示者。

另外，我们还可以通过将输出属性订阅到可观察对象add$ property，将演示者连接到输出属性，如清单9B所示。

们可以不使用事件发射器，而是将一个标记为output属性的组件getter委托给演示者的可观察对象属性。这样做会很好，因为输出属性只需要有一个订阅方法，比如observable或subject。然而，让我们继续在组件中使用Angular自己的构建块。

如果我们的演示者包含表示转换方法(例如用于格式化)，那么我们将添加向它们传递输入属性的组件方法或getter。我们还可能有组件输入属性，它们的设置器将值传递给演示者。这些值将在组件UI属性中用于委托给呈现器上的getter或方法。

我们忘了什么吗?heroes组件的连接订阅是如何管理的?

管理可观测的订阅

如果我们使用了演示者的observable作为组件的输出属性，Angular就会替我们管理订阅了。

我们有三种选择来自己管理订阅。

// heroes.component.ts

import { Component, EventEmitter, OnDestroy, OnInit } from '@angular/core';

import { Subject } from 'rxjs';

import { takeUntil } from 'rxjs/operators';

import { HeroesPresenter } from './heroes.presenter';

@Component({

selector: 'app-heroes-ui',

})

export class HeroesComponent implements OnDestroy, OnInit {

private destroy = new Subject<void>(); // 👈

@Output()

add = new EventEmitter<string>();

constructor(

private presenter: HeroesPresenter,

) {}

ngOnInit(): void {

this.presenter.add$.pipe(

takeUntil(this.destroy), // 👈

).subscribe(name => this.add.emit(name));

}

ngOnDestroy(): void { // 👈

this.destroy.next();

this.destroy.complete();

}

}

清单10。Heroes:使用生命周期主题管理订阅的组件。

我们的第一个选项是在OnDestroy生命周期时刻调用的组件中添加一个私有destroy主题，并将其与takeUntil操作符结合起来，如清单10A所示。你可能以前见过这个技巧。

// heroes.component.ts

import { Component, EventEmitter, OnDestroy, OnInit } from '@angular/core';

import { Subscription } from 'rxjs';

import { HeroesPresenter } from './heroes.presenter';

@Component({

selector: 'app-heroes-ui',

})

export class HeroesComponent implements OnDestroy, OnInit {

private subscription: Subscription; // 👈

@Output()

add = new EventEmitter<string>();

constructor(

private presenter: HeroesPresenter,

) {}

ngOnInit(): void {

this.subscription = this.presenter.add$.subscribe(name =>

this.add.emit(name));

}

ngOnDestroy(): void {

this.subscription.unsubscribe(); // 👈

}

}

清单10 b。Heroes:使用订阅对象管理订阅的组件。

第二种方法是将结果订阅存储在一个私有属性中，并在组件的OnDestroy生命周期钩子中取消订阅，如清单10B所示。这是传统的RxJS技术。

最后一个选项是通过在OnDestroy钩子中完成add主题，让演示者管理依赖于它的订阅。与其他选项相比，这是更少的代码。

// heroes.presenter.ts

import { OnDestroy } from '@angular/core';

import { FormControl } from '@angular/forms';

import { Observable, Subject } from 'rxjs';

export class HeroesPresenter implements OnDestroy {

private add = new Subject<string>();

add$: Observable<string> = this.add.asObservable();

nameControl = new FormControl('');

ngOnDestroy(): void {

this.add.complete(); // 👈

}

addHero(): void {

const name = this.nameControl.value.trim();

this.nameControl.setValue('');

if (!name) {

return;

}

this.add.next(name);

}

}

清单10 c。英雄:主讲人管理订阅者。

清单10C显示了我们添加了一个ngOnDestroy生命周期钩子，在这个钩子中我们完成了私有的add主题。完成一个subject或任何其他可观察对象会导致所有订阅者触发它们的完整钩子(如果它们有一个的话)并最终取消订阅。

但是，对于共享的有状态表示器，我们必须小心。如果组件有不同的生命周期，即它们在不同的时间被激活和销毁，我们就可以为已经销毁的组件运行订阅。

订阅管理规则:在一个或多个组件同时被激活和销毁的情况下，我们应该只依赖于提交者管理订阅。

当在路由组件、使用动态呈现或结构指令的组件之间共享呈现器时，我们应该使用订阅管理的传统选项之一。

使用由演示者和订阅组件共同结束订阅的组合可能是更好的选择。这有助于我们清理单元测试中的资源，并降低内存泄漏的风险。

可选的改进

总有改进的空间。以下是一些需要考虑的选项。

重构heroes演示器

演示者的一个很好的特点是，我们可以重构它的实现，或者添加行为或演示逻辑，而不会破坏它的公共API。

从演示组件中提取出来的UI行为和表单验证，很明显，这个封装的逻辑只关心add hero表单，其他什么都不关心。

我们可以把它重命名为HeroForm，它仍然是一个演示器。这将表明它不是特定于组件的表示器，而是潜在的可重用表示器，以及潜在的多个具有不同关注点的表示器之一。

addHero代码中的命令式代码似乎可以用更具声明性和响应性的方法来表示。

// heroes.presenter.ts

import { FormControl } from '@angular/forms';

import { Observable, Subject } from 'rxjs';

import { filter, map } from 'rxjs/operators';

export class HeroesPresenter {

private add = new Subject<string>();

add$: Observable<string> = this.add.pipe(

map(name => name.trim()), // 👈

filter(name => !!name), // 👈

);

nameControl = new FormControl('');

addHero(): void {

const name = this.nameControl.value;

this.nameControl.setValue('');

this.add.next(name);

}

}

清单11。英雄:演示者，在可观察对象管道中进行输入清理和验证。

清单11展示了如何使用RxJS操作符来表示清理和验证逻辑。响应式表单有一种更少命令式的方式来创建这个数据流，但这是另一种练习。

强制严格的依赖注入规则

Angular的依赖注入系统非常强大，但如果不小心，我们可能会把私有依赖泄露给其他可声明对象。

还记得我们把heroes演示器添加到providers数组选项中，以便能够将它注入到presentational heroes组件中吗?

当我们这么做的时候，我们就能让每一个视图子节点、内容子节点以及它们的后代都注入HeroesPresenter。正如“组件与演示者的比率”一节所讨论的，我们可能想要共享一个允许我们共享的演示者。但是，我们可能不希望将此服务提供给投影内容。

您的简单示例并没有投射内容。如果是这样，我们可以选择使用viewProviders选项来提供heroes演示器，以防止服务泄漏到我们不能立即控制的可声明对象。出于这个原因，使用viewprovider甚至可以被认为是一个很好的默认设置。

演示者注入提示:默认使用组件的viewProviders选项提供演示者，除非你计划将其与内容子组件共享。

我们可以通过只允许注入服务工厂而不是注入演示者本身来防止演示者的意外共享。

// heroes-presenter-factory.token.ts

import { InjectionToken } from '@angular/core';

import { HeroesPresenter } from './heroes.presenter';

export const heroesPresenterFactoryToken = new InjectionToken(

'Heroes presenter factory', {

factory: (): (() => HeroesPresenter) =>

() => new HeroesPresenter(),

});

清单12。英雄:演示者服务工厂的依赖注入令牌。

// heroes.presenter.ts

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root',

useFactory: (): never => {

throw new Error('Use heroesPresenterFactoryToken to create a hero presenter.'); },

})

export class HeroesPresenter {}

清单12 b。英雄:演示者提供者保护直接注入。

// heroes.component.ts

import { Component, Inject, OnDestroy } from '@angular/core';

import { HeroesPresenter } from './heroes.presenter';

import { heroesPresenterFactoryToken } from './heroes-presenter-factory.token';

@Component({

providers: [

{

deps: [

[new Inject(heroesPresenterFactoryToken)],

],

provide: HeroesPresenter,

useFactory:

(createHeroesPresenter: () => HeroesPresenter): HeroesPresenter =>

createHeroesPresenter(),

},

],

selector: 'app-heroes-ui',

})

export class HeroesComponent implements OnDestroy {

constructor(

private presenter: HeroesPresenter,

) {}

}

清单12 c。使用演示者服务工厂的演示性组件。

清单12A、12B和12C展示了如何使用服务工厂来创建英雄演示器。演示器服务提供商会抛出一个错误，以阻止其他可声明对象直接注入英雄演示器。

即使其他可声明对象注入了服务工厂，它们也会创建一个单独的英雄演示者实例，这样就不可能意外地共享演示者。

我们可以重用清单12C中使用的提供程序，例如从具有依赖项注入令牌的模块中导出它。

最后一种可以用来强制执行严格依赖注入规则的技术是，在将英雄展示器注入到演示组件中时，使用Self decorator工厂。如果没有服务工厂，它将如清单13所示。

// heroes.component.ts

import { Component, Self } from '@angular/core';

import { HeroesPresenter } from './heroes.presenter';

@Component({

selector: 'app-heroes-ui',

})

export class HeroesComponent {

constructor(

@Self() private presenter: HeroesPresenter,

) {}

}

清单13。Heroes:从自己的节点注入器强制演示者注入。

当我们使用Self装饰器工厂时，我们会告诉Angular，只允许通过组件自己的节点注入器来注入英雄展示器。

演示者注入提示:使用Self decorator工厂注入演示者，除非它是一个共享的演示者。这可以防止祖先组件的呈现程序意外注入。

使用可观察对象呈现器属性作为组件输出属性

纯粹主义者会专门使用eventemitter作为输出属性。从技术上讲，output属性要集成到Angular中，只需要有一个接受观察者的

这意味着我们可以使用observable作为输出属性。我们的演示者公开了observable，因此我们可以从组件中将其委托给它们，如清单14A和14B所示。

// heroes.component.ts

import { Component, Output } from '@angular/core';

import { HeroesPresenter } from './heroes.presenter';

@Component({

providers: [HeroesPresenter],

selector: 'app-heroes-ui',

styleUrls: ['./heroes.component.css'],

templateUrl: './heroes.component.html',

})

export class HeroesComponent {

@Output('add')

get add$(): Observable<string> { // 👈

return this.presenter.add$;

}

constructor(

private presenter: HeroesPresenter,

) {}

addHero(): void {

this.presenter.addHero();

}

}

：清单14。Heroes:演示组件使用getter将输出属性委托给它的演示者。

// heroes.component.ts

import { Component, Output } from '@angular/core';

import { HeroesPresenter } from './heroes.presenter';

@Component({

providers: [HeroesPresenter],

selector: 'app-heroes-ui',

styleUrls: ['./heroes.component.css'],

templateUrl: './heroes.component.html',

})

export class HeroesComponent {

@Output('add')

add$ = this.presenter.add$; // 👈

constructor(

private presenter: HeroesPresenter,

) {}

addHero(): void {

this.presenter.addHero();

}

}

清单14 b。Heroes:演示组件使用属性引用将输出属性委托给它的演示者。

在清单13A和13B中的两个备选方案中，我们都不需要自己管理订阅来连接提交者的可观察对象和组件的事件发射器，所以我们删除了OnInit生命周期钩子。

Framework-agnostic presenters

如果我们想要在多个框架和平台之间启用代码共享，或者保留这样做的选项，我们应该让我们的演示者与框架无关。

// heroes.presenter.ts

import { Observable, Subject } from 'rxjs';

import { filter, map } from 'rxjs/operators';

export class HeroesPresenter {

private add = new Subject<string>();

add$: Observable<string> = this.add.pipe(

map(name => name.trim()), // [2]

filter(name => !!name), // [2]

);

destroy(): void { // [1]

this.add.complete();

}

addHero(name: string): void {

this.add.next(name);

}

}

清单15。主持人Framework-agnostic英雄。

清单15A显示了一个与框架无关的英雄展示器。我们删除了angular特有的生命周期钩子ngOnDestroy，并用一个简单的方法destroy(1)替换了它。

我们删除了FormControl。尽管响应式Angular表单可以与其他框架一起使用，而且是一个非常好的库，但我们把输入清理和验证逻辑移到了我们的可观察对象管道中(2)。

// app-heroes.presenter.ts

import { Injectable, OnDestroy } from '@angular/core';

import { FormControl } from '@angular/forms';

import { HeroesPresenter } from './heroes.presenter';

@Injectable()

export class AppHeroesPresenter implements OnDestroy {

add$ = this.presenter.add$; // [3]

nameControl = new FormControl('');

constructor(

private presenter: HeroesPresenter, // [1]

) {}

ngOnDestroy(): void {

this.presenter.destroy(); // [2]

}

addHero(): void {

const name = this.nameControl.value;

this.nameControl.setValue(''); // [5]

this.presenter.addHero(name); // [4]

}

}

清单15 b。angular特有的展示器，包装了与框架无关的英雄展示器。

清单15B展示了特定于angular的演示器，它包装了清单15A中的框架无关的英雄演示器。它会注入heroes的演示器(1)，并在ngOnDestroy生命周期钩子(2)中调用它的destroy方法。

angular特有的演示器委托给了heroes演示器(3)的add$ observable属性，并添加了一个FormControl，它将每个值转发给heroes演示器(4)的addHero方法，但保留了重置表单控件(5)的UI行为逻辑。

因为特定于angular的展示器保留了相同的API，所以我们在组件中以完全相同的方式使用它。

presenters的特点

演讲者可能是可重复使用的。它们几乎与应用程序的其他部分隔离开来，没有或只有很少的依赖关系——在复合的演示程序中，依赖关系可能是其他演示程序。我们将它们添加到组件级提供程序中，并在表示组件和混合组件中使用它们。

我们可以使用RxJS的可观察对象、常规属性或方法来对应用程序状态进行建模，这些属性或方法由组件传递输入属性，在通过组件的DOM将其呈现给用户之前进行格式化。

演示者可以是有状态的，如果他们管理自己的应用程序状态块的同步表示。大多数情况下，这是建模为常规属性或可观察属性的本地UI状态。

状态的演示者关心的是表示，因为他们只转换数据，这有意地阻止了他们管理用户交互。

我们可以使用一个符合我们用例的组件-演示者比率。我们可以为每个组件引入一个呈现器，比如特定于组件的复合呈现器

每个组件可以有多个演示者，例如一个用于用户交互，一个用于演示关注点。最后，我们可以在多个组件之间共享一个表示器。无状态的演示程序很容易以这种方式共享。

我们了解到，当演示程序和DOM片段之间的内聚性很高时，我们可以重用组件而不是演示程序。然而，如果UI行为在特定条件下发生变化，这就不是一个好方法。

对于性能关键型用例的演示关注点，我们可以用纯Angular管道或记忆管道来替换演示器。Angular管道有这样的权衡:它们非常细粒度，需要大量的设置。

除了这些权衡之外，Angular管道也不能用于用户交互问题。

值得重复的是，演示者可以是平台或框架无关的，它支持跨应用程序、平台和设备的一致UI行为。

使用演示程序的好处之一是，它们非常容易在测试中被隔离，而且可以在没有任何UI的情况下进行测试，或者——如果我们设计得好——不需要任何框架或特定于平台的代码。

从表示组件中提取表示器

要从演示组件中提取演示者，我们遵循以下步骤:  
1 将复杂的表示逻辑提取到表示器中。

2 将演示程序注入演示组件。

3 将演示者连接到演示组件的数据绑定API。

4 管理可观测的订阅。

在提取演示程序时，演示组件的模板和数据绑定API应该没有什么需要更改的理由。当我们提取复杂的表示逻辑时，UI属性可能会改变。

我们最后会有一个或多个演讲者来讨论这些问题:

显示/转换应用程序状态(格式化)

UI行为(本地UI状态)  
表单验证(本地UI状态)  
特定于应用程序的事件

相关文章

阅读介绍性文章“模型-视图-演示器与Angular”。您还可以在这里找到到配套GitHub存储库、相关文章和其他有用资源的链接。

在“使用Angular的演示组件”中学习如何将一个混合组件转换成一个演示组件。

<https://indepth.dev/posts/1066/presentational-components-with-angular>