模型-视图-表示器的历史

<https://indepth.dev/posts/1071/the-history-of-model-view-presenter>

Smalltalk是模型-视图-控制器的孵化器，也在模型-视图-表示器中发挥了作用。

说前端软件架构有意义吗?还是说这是白日梦?事实证明，维护复杂前端的挑战是一个至少早在20世纪70年代就被发现的问题。

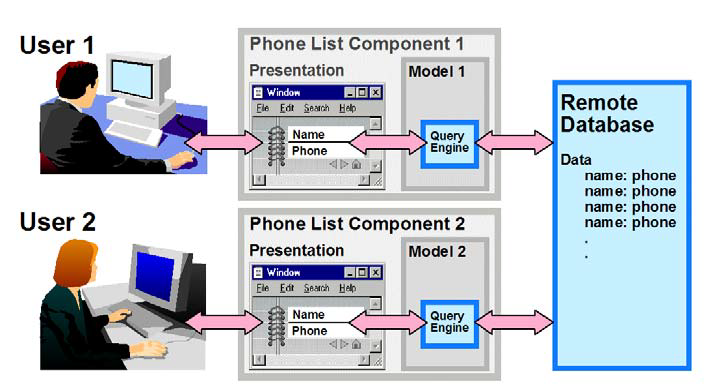
和我一起重温历史书籍，从现代web的角度探索模型-视图-表示器设计模式的起源。进入翻译页面

Model-View-Controller

模型-视图-控制器(Model-View-Controller, MVC)是模型-视图-表示器的哥哥，在20世纪70年代由Trygve Reenskaug首次为编程语言Smalltalk引入。当时，仅仅捕获用户输入就需要大量的逻辑。这是由控制器处理的，屏幕上的每个用户控件都存在一个模型-视图-控制器三元组。

随着20世纪90年代web的出现，由于NeXT和Java的webbobjects，对模型-视图-控制器的兴趣增加了。21世纪初，Struts和Spring (Java)、Django (Python)、Rails (Ruby)和ASP等服务器端web框架进一步普及了它。NET MVC (c#和朋友)。通常，框架通过使用所谓的前端控制器将请求路由到相关的控制器。

Model-View-Presenter



在20世纪90年代早期，Taligent首次在c++中描述并使用了模型-视图-演示器(MVP)。在20世纪90年代末，Taligent将其引入Java平台，Dolphin将其应用于Smalltalk UI框架。

在21世纪初的后期，微软开始提倡使用模型-视图-展示器(Model-View-Presenter)来使用。net开发丰富的应用程序，如Windows Forms、Silverlight、SharePoint和ASP.NET。

在模型-视图-表示器模式被使用的最初几十年里，出现了几种变体。要了解它们之间的差异，请阅读Derek“有抱负的工匠”Greer的文章“交互式应用程序架构模式”中有关talented和Dolphin Smalltalk变体的材料。 “[Interactive Application Architecture Patterns](http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/" \l "the-model-view-presenter-pattern)”

在2000年代末和2010年代初，第一代UI框架——如AngularJS、Backbone、Dojo Toolkit、Ember、JavaScriptMVC和knockout——将模型-视图-控制器和模型-视图-视图模型(MVVM)模式引入了客户端web开发。这允许我们在保持水平软件层同步的同时，将表现逻辑与应用程序状态分离开来。

当前一代的UI框架，如Angular、Aurelia、Dojo、Inferno、Preact、React、苗条和Vue都是基于组件的。它们专注于UI小部件，但将表示层与应用程序的其余部分分开的细节留给我们。

Model-View-Presenter in modern web applications

由于流行的UI框架没有将应用程序划分为水平软件层的设计模式可供选择，所以如果我们想要实施关注点分离，就必须选择一个并自己应用它。

借鉴React社区的智慧，我描述了一种非常适合Angular平台的模型-视图-演示者模式的变体。阅读《Model-View-Presenter with Angular》了解更多。"[Model-View-Presenter with Angular](https://indepth.dev/model-view-presenter-with-angular/)"

\*\*\*\*

D:\下载\ngx-tour-of-heroes-mvp-develop\src\app\heroes\heroes.container.html

<app-heroes-ui

[heroes]="heroes$ | async"

title="My Heroes"

(add)="add($event)"

(remove)="delete($event)"></app-heroes-ui>

@Component({

changeDetection: ChangeDetectionStrategy.OnPush,

selector: 'app-heroes',

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

})

export class HeroesContainerComponent {

private heroAdd: Subject<Hero> = new Subject();

private heroRemove: Subject<Hero> = new Subject();

heroes$: Observable<Hero[]> = multiScan(

this.heroService.getHeroes(),

(heroes, loadedHeroes) => [...heroes, ...loadedHeroes],

this.heroAdd,

(heroes, hero) => [...heroes, hero],

this.heroRemove,

(heroes, hero) => heroes.filter(h => h !== hero),

[]);

constructor(private heroService: HeroService) {}

add(name: string): void {

this.heroService.addHero({ name } as Hero)

.subscribe({ next: h => this.heroAdd.next(h),error: noop,});

}

delete(hero: Hero): void {

this.heroRemove.next(hero);

this.heroService.deleteHero(hero)

.subscribe({ error: () => this.heroAdd.next(hero), });

}

}

\*\*\*\*\*\*\*

D:\下载\ngx-tour-of-heroes-mvp-develop\src\app\heroes\heroes.component.ts

<h2>{{title}}</h2>

<div>

<label>Hero name:

<input [formControl]="nameControl" />

</label>

<button (click)="addHero()">

add

</button>

</div>

<ul class="heroes">

<li \*ngFor="let hero of heroes">

<a routerLink="/detail/{{hero.id}}">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</a>

<button class="delete" title="delete hero"

(click)="remove.emit(hero)">x</button>

</li>

</ul>

selector: 'app-heroes-ui',

export class HeroesComponent implements OnInit {

@Input() heroes: Hero[];

@Input() title: string;

@Output() add: EventEmitter<string> = new EventEmitter();

@Output() remove: EventEmitter<Hero> = new EventEmitter();

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();

}

}

\*\*\*\*

D:\下载\ngx-tour-of-heroes-mvp-develop\src\app\heroes\heroes.presenter.ts

export class HeroesPresenter implements OnDestroy {

private add: Subject<string> = new Subject();

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

nameControl = new FormControl('');

ngOnDestroy(): void {

this.add.complete();

}

public addHero(): void {

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

this.nameControl.setValue('');

if (!name) { return; }

this.add.next(name);

}

}

<https://indepth.dev/posts/1072/lean-angular-components>

Lean Angular components

这几乎没有告诉我们任何东西。我甚至认为一个应用程序不是一个组件集合，而是一个组件组合。这与应用程序在任何时候都有一个由所有激活组件(即当前页面上处于当前状态的激活组件)组成的组件树的事实相一致。

组件是用来表示的

组件是应用程序表示层的一部分:用户交互和表示层。组件位于整个应用程序架构的最外层。它们是向用户显示信息并允许他们与应用程序交互的方法。

考虑到图1中的web应用程序的水平层，组件通常会直接解决来自多个层的问题。这是一个错误。

Model-View-Presenter—技术集合

我之前已经收集了一些常见的技术，并在“Angular模型-视图-演示器”及其相关文章中对它们进行了描述。为什么?来获得关于如何在Angular应用中分离关注点的详细指南。

遵循我为Angular设计的模型-视图-表示器变体，你几乎肯定会拥有一个可维护、可测试、可扩展和高性能的应用。

不相信吗?这很好。让我们看一看官方入门指南中的一个例子。

// cart.component.ts

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

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

import { CartService } from '../cart.service';

@Component({

selector: 'app-cart',

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

styleUrls: ['./cart.component.css']

})

export class CartComponent {

items;

checkoutForm;

constructor(

private cartService: CartService,

private formBuilder: FormBuilder,

) {

this.items = this.cartService.getItems();

this.checkoutForm = this.formBuilder.group({

name: '',

address: ''

});

}

onSubmit(customerData) {

// Process checkout data here

console.warn('Your order has been submitted', customerData);

this.items = this.cartService.clearCart();

this.checkoutForm.reset();

}

}

清单1中的cart组件发生了什么?在它的UI属性中，我们看到一个项目列表和一个签出表单。在使用表单生成器初始化签出表单时，条目从购物车服务初始化。

当用户提交表单时，签出表单的值被记录到控制台(因为这是一个简单的示例)，使用cart服务清除项目列表，并重置签出表单。

*解决太多的问题*

这个组件有什么问题?首先，它解决了跨多个水平层的问题。其次，它包含不同抽象层次的逻辑——它包含低层实现细节和高层策略。

购物车组件使用两部分应用程序状态:结帐表单(本地UI状态)和购物车中包含的项目。

*状态管理是一个独立的问题*

关于如何管理应用程序状态，至少有两点是错误的。购物车条目列表是从购物车服务初始化的。至少我们没有在组件中直接执行HTTP请求。我们在更高的抽象层次上说，我们想要购物车中的物品，而不是如何获得它们。

然后，我们从购物车服务中提取商品。为了分离关注点，我们需要一个表示购物车组件，它只关心显示购物车条目并允许用户提交结帐表单。不多不少。

什么这很重要?官方建筑指南如下

Angular把组件和服务区分开来，以提高模块化和可重用性。通过将组件的视图相关功能与其他类型的处理分离，可以使组件类精简和高效。

这正是我想说服你的。我们应该努力让组件只关心表示。架构指南提到了视图相关的功能。我们可以讨论这个术语的含义。我的观点是展示和用户交互。

我想更进一步说，甚至表示或用户交互都不应该成为组件模型的一部分。除了简单逻辑之外的任何东西都应该提取到服务和其他依赖项中。

*Designing presentational components*

呈现性组件显示通过输入属性传递给它们的应用程序状态。如果数据转换或计算属性是必需的，应用程序状态将通过一个呈现器(组件级服务)传递。

架构指南在下一段中对此进行了描述:

理想情况下，组件的工作是支持用户体验，仅此而已。组件应该为数据绑定提供属性和方法，以便在视图(由模板呈现)和应用程序逻辑(通常包含一些模型的概念)之间充当中介。

*管理控制流程*

表示组件的另一项职责是将由用户交互发起的控制流传递给行为封装服务——我称之为presenters。presenters的副作用会根据需要映射到输出属性。在简单的用例中，用户交互直接映射到输出属性。

这是架构指南中提到的另一个原则:

可以将某些任务委托给服务，例如从服务器获取数据、验证用户输入或直接在控制台进行日志记录。

本指南中提到的三个示例几乎完全对应于cart组件中的业务逻辑。它告诉我们委托给服务。

*我们需要纪律*

《指南》导言的最后一段如下:

Angular没有强制执行这些原则。Angular确实帮助你遵循了这些原则，它让你很容易地将应用逻辑分解为服务，并通过依赖注入让这些服务对组件可用。进入翻译页面

这真是一针见血。该指南鼓励我们遵循这些原则，但在设计组件时，这取决于我们是否实践它们。我们需要规范我们的组件架构。

*容器组件是不够的*

即使将组件分为容器组件和表示组件，我们仍然应该添加另一层抽象来防止组件做太多的事情。应该将业务逻辑提取到组件级服务(如facade)中，甚至提取到特定于组件的服务(presenters.)中。

*对于长期存在的项目和软件产品*

过度设计吗?对于小型或简单的应用程序，当然可以。对于长期存在的项目和软件产品，我们最好尽早考虑我们的组件架构。

通过将关注点分离成非常具体的软件工件，每一个都很容易推理和测试。一旦不可避免地需要新的需求，我们就可以扩展封装特定关注点的软件工件，并在正确的抽象级别上处理它。

*案例研究:精益购物车组件*

那么入门指南中的cart组件发生了什么情况呢?

<!-- cart.component.html -->

<h3>Cart</h3>

<p>

<a routerLink="/shipping">Shipping Prices</a>

</p>

<div class="cart-item" \*ngFor="let item of items">

<span>{{ item.name }} </span>

<span>{{ item.price | currency }}</span>

</div>

<form [formGroup]="checkoutForm" (ngSubmit)="onSubmit(checkoutForm.value)">

<div>

<label for="name">

Name

</label>

<input id="name" type="text" formControlName="name">

</div>

<div>

<label for="address">

Address

</label>

<input id="address" type="text" formControlName="address">

</div>

<button class="button" type="submit">Purchase</button>

</form>

清单2.1。初始混合组件模板。

// cart.component.ts

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

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

import { CartService } from '../cart.service';

@Component({

selector: 'app-cart',

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

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

})

export class CartComponent {

items;

checkoutForm;

constructor(

private cartService: CartService,

private formBuilder: FormBuilder,

) {

this.items = this.cartService.getItems();

this.checkoutForm = this.formBuilder.group({

name: '',

address: '',

});

}

onSubmit(customerData) {

// Process checkout data here

console.warn('Your order has been submitted', customerData);

this.items = this.cartService.clearCart();

this.checkoutForm.reset();

}

}

清单2.2。初始混合组件模型。

清单2.1和2.2是我们的起点——它是一个混合的组件，涉及多个水平层。它还具有不同抽象级别的逻辑。

混合cart组件包含表示逻辑、表示实现细节、高级表示逻辑、低级表示逻辑和低级用户交互逻辑。这些类别的逻辑可以添加到表示组件中，但是我们肯定应该考虑重构它。

它还在状态管理实现细节的类别中包含非表示逻辑，以及低级业务逻辑。状态管理是我们应该首先关注的问题。本地UI状态是一个异常，它被归类为用户交互关注点——UI行为的一部分。

<!-- cart.container.html -->

<app-cart-ui

[items]="items"

[shippingPath]="shippingPath"

(checkout)="onCheckout($event)"></app-cart-ui>

Listing 3.1. Cart: Container component template.

// cart.container.ts

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

import { Customer } from '../customer';

import { pathPrefix, routes } from '../root-routes';

import { CartService } from './cart.service';

@Component({

selector: 'app-cart',

templateUrl: './cart.container.html',

})

export class CartContainerComponent {

items = this.cartService.getItems();

shippingPath = pathPrefix + routes.shipping.path;

constructor(

private cartService: CartService,

) {}

onCheckout(customerData: Customer) {

// Process checkout data here

console.warn('Your order has been submitted', customerData);

this.items = this.cartService.clearCart();

}

}

Listing 3.2. Cart: Container component model.

在清单3.1和3.2中，我们从混合的cart组件中提取了一个容器组件。所有状态管理集成逻辑现在都在这个组件中。

// root-routes.ts

export const pathPrefix = '/';

export const routes = {

shipping: {

path: 'shipping',

},

};

清单3.3从cart组件提取后的根路由。

在初始的cart组件中，配送路径硬编码在模板中。现在，我们已经将路由路径提取到一个单独的模块，如清单3.3所示，使其可重用并易于更改。

在模板和组件模型中硬编码路由url和路径是一种糟糕的实践

我们最好把路由路径和url存储在一个单独的模块中，这样我们就可以从组件、指令和服务中引用它们了。

Max Tarsis的Routeshub是一个路由管理库，可以很容易地与Angular的ro集成

容器组件创建了一个完整的路由URL，并将其传递给我们接下来要讨论的演示式购物车组件。

<!-- cart.component.html -->

<h3>Cart</h3>

<p>

<a [routerLink]="shippingPath">Shipping Prices</a>

</p>

<app-cart-item \*ngFor="let item of items"

[item]="item"></app-cart-item>

<app-checkout (checkout)="checkout.emit($event)"></app-checkout>

Listing 4.1. Cart: Presentational component template.

// cart.component.ts

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

import { Customer } from '../customer';

import { Products } from '../product';

@Component({

selector: 'app-cart-ui',

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

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

})

export class CartComponent {

@Input()

items: Products;

@Input()

shippingPath: string;

@Output()

checkout = new EventEmitter<Customer>();

}

Listing 4.2. Cart: Presentational component model.

在清单4.1和4.2中，我们看到现在表示cart组件中只剩下最小的逻辑。传送路径URL作为输入属性传递。这个组件不关心完整的路由URL是什么，也不关心如何访问它。

同样，该组件很乐意在产品上进行迭代，将每个产品传递给我们提取的另一个表示组件(cart item组件)的单独实例。

我不会详细介绍更简单的购物车项目组件的实现细节，但是完整的解决方案可以在StackBlitz工作区中获得。

我们提取了另一个表示组件，签出组件。

<!-- checkout.component.html -->

<form [formGroup]="checkoutForm" (ngSubmit)="onSubmit()">

<div>

<label for="name">

Name

</label>

<input id="name" type="text" formControlName="name">

</div>

<div>

<label for="address">

Address

</label>

<input id="address" type="text" formControlName="address">

</div>

<button class="button" type="submit">Purchase</button>

</form>

Listing 5.1. Cart: Checkout component template.

// checkout.component.ts

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

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

import { Customer } from '../customer';

import { CheckoutPresenter } from './checkout.presenter';

@Component({

selector: 'app-checkout',

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

viewProviders: [CheckoutPresenter],

})

export class CheckoutComponent {

@Output()

checkout = new EventEmitter<Customer>();

get checkoutForm(): FormGroup {

return this.presenter.form;

}

constructor(

private presenter: CheckoutPresenter,

) {}

onSubmit() {

const customerData = this.presenter.checkout();

this.checkout.emit(customerData);

}

}

Listing 5.2. Cart: Checkout component model.

清单5.1中的checkout组件模板将原生表单控件绑定到响应式Angular表单组和控件上。

清单5.2中的组件模型公开了来自checkout presenter的表单组，这是一个封装了用户交互逻辑的组件级服务。

这个表示组件通过将逻辑委托给签出演示者，将表单提交转换为输出属性事件发送。

// checkout.presenter.ts

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

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

import { Customer } from '../../customer';

@Injectable()

export class CheckoutPresenter {

form = this.formBuilder.group({

name: '',

address: '',

});

constructor(

private formBuilder: FormBuilder,

) {}

checkout(): Customer {

const customerData: Customer = this.form.value;

this.form.reset();

return customerData;

}

}

Listing 6. Cart: Checkout presenter.

构建反应式签出表单组的低级逻辑被封装在清单6中的签出呈现器中。表单通过公共属性公开

帐就是从响应式表单控件收集条目，重新设置表单组，并从结帐方法返回收集的表单条目。

我通常公开可观察对象，当执行签出之类的操作时，这些可观察对象会发出值。通过这种方式，表示组件可以将按钮直接映射到presenters方法，而不必关心处理返回值。相反，一个output属性连接到演示者的observable属性。

对于这个案例研究，我希望实现细节看起来与初始解决方案中类似，因此我在签出组件的表单提交处理程序中保留了一些命令式控制流逻辑。

*文件和目录结构*

分离关注点和提取接口之后，我们从图2中的文件开始

最初，许多不同的关注点位于单个混合组件中，如图4所示。下面部分中的关注点肯定不应该出现在表示组件中，即状态管理实现细节、高级业务逻辑和低级业务逻辑。

在将cart特性重构为多个组件和一个 presenter之后，关注点被合理地分离了，如图5所示。

每个软件工件都与单个水平层或最多两个层次的逻辑有关。在大多数情况下，它们还设法保持单一的抽象级别。

总有改进的空间，但对于这个应用程序特性，这是一个相当可靠的组件设计。

*风格指南值得再读一遍*

官方的Angular风格指南也推荐用这种方式来设计组件。让我们在另一篇阅读中给出一些推荐的实践。

*提取服务的非表示逻辑*

风格05-15:将复杂的组件逻辑委托给服务

这是整个风格指南中最重要的指导原则之一，尤其是在精益Angular组件的上下文中。

首先，它告诉我们提取服务的非表示性逻辑。接下来，它告诉我们保持组件的简单性，并专注于它们应该做什么。换句话说，我们应该尽量减少模板中的逻辑，将逻辑从组件模型中委托出去，保持组件小，这样组件就不会有1000行代码。

即使是一个组件的100行代码也应该让我们担心，或者至少考虑将组件拆分为多个组件或委托给服务。

将逻辑提取到服务的一个原因是重用。数据服务和其他与应用程序状态管理相关的服务很有可能被应用程序的多个部分重用，甚至可能被多个项目重用。

类似地，当从组件中提取非表示性逻辑时，我们最终得到的是表示性组件，这些组件可以使用来自应用程序不同部分或多个项目的状态进行重用。

将逻辑提取到服务的另一个好处是，隔离测试服务更容易、更快。此外，仅具有表示关注点的简单组件更容易进行隔离测试。这是因为我们将依赖关系转移到组件之外，并将实现细节封装在组件之外。

这条指导方针留给我们的最后一个动机是，从组件中提取逻辑会让它们变得苗条、整洁和集中。

*不要把表示逻辑放在模板中*

风格05-17:将表示逻辑放在组件类中

在同一抽象级别上的相同关注点的逻辑应该位于非常接近的位置。虽然组件模型、样式和模板是密切相关的，但每一个都有自己需要关注的问题。

组件模板应该关心声明式DOM操作和事件绑定，而不是实现细节。组件模型应该列出可以绑定到模板的属性。这些属性从模板中隐藏实现细节。

风格指南还告诉我们，将表示逻辑放在组件模型中而不是模板中，可以提高可重用性、可维护性和可测试性——所有这些特征都值得最大化。

*当指令能做你需要做的事情时，不要创建组件*

样式06-01:使用指令来增强元素

这个指导原则提醒我们，我们不应该总是直接跳到使用组件。事实上，如果不需要模板，或者DOM的变化可以反映到宿主元素本身，一个属性型指令将会为我们带来好处。

当我们用一个属性指令替换一个组件时，我们为每个指令实例保存一个DOM元素，这对于复杂的应用程序或者需要一个特定的DOM结构时可能是至关重要的。

如果我们想仅基于某种状态或在其他条件下显示内容，结构化指令是我们的朋友。

使用指令而不是组件的缺点是它们不能动态应用。将其与可以动态呈现的组件进行比较。

积极的一面是，准则提到一个模板元素可以应用多个指令。对于组件，只有一个组件可以应用于模板元素。

*做一件事并把它做好*

风格07-02:单一责任

这个指导方针立即让我们想到了单一责任原则。是否与此相关取决于我们如何定义责任。我现在不讨论这个问题，尽管它很重要。

我从这个样式建议中得到的是，我们应该创建从单个抽象级别的单个水平层封装逻辑的服务。

实际上，Angular风格指南在简介中给出了这个建议:

将单一职责原则应用于所有组件、服务和其他符号。这有助于使应用程序更干净，更容易阅读和维护，更易于测试。

不幸的是，它并没有教会我们如何去做。那是另一篇文章的主题了。

*组件级别的服务*

样式07-03:提供服务

样式指南建议使用根目录提供的可注入装饰器，使服务成为单例服务，从而能够在应用程序的多个部分之间共享业务逻辑和状态。

当两个不同的组件需要一个服务的不同实例时，这是不理想的。在此场景中，最好在需要新的单独实例的组件级别提供服务。

更有趣的是，该指南告诉我们，当需要在不同组件中单独实例时，应该在组件级提供服务。无论我们的组件是具有不同组件模型的，还是由相同组件模型构造的，我们都可以通过使用组件装饰器工厂的providers或viewProviders选项来实现这一点。

*提取服务的非表现性关注点*

风格08-01:通过服务与服务器通信

该指南建议我们提取数据操作和交互逻辑到服务。示例:HTTP通信、web存储读写和内存中的数据存储(如Flux-like stores)。

组件的职责是表示和收集视图的信息。它不应该关心如何获取数据，而应该知道谁需要这些数据。分离数据服务将如何获得数据服务的逻辑转移到数据服务，并让组件更简单，更专注于视图。

这在很大程度上是关于关注点的分离。我们不会在同一个地方工作于应用程序的不同水平层。相反，让我们设计只关心表示的组件。我甚至更喜欢提取表示和用户交互，将这些关注委托给组件级服务，比如presenters.。

样式指南提醒我们，从组件中提取逻辑到一个抽象接口中，可以更容易地测试组件。

我们让组件将检索或存储数据的任务委托给服务，这样它就不必知道或担心细节。这样我们可以在不破坏组件的情况下更容易地更改细节。

*简而言之，精益Angular组件*

对于简单的应用程序或生命周期较短的项目，混合组件是可以接受的，因为对代码库的可维护性、可伸缩性甚至可测试性都没有太大的需求。

对于大型、复杂的应用程序，我们更需要最大化刚才提到的有价值的特性。要做到这一点，我们需要只包含属于表示的水平应用层的逻辑的表示性组件。这些组件应该只处理单个抽象级别上的一个关注点。

精益表示组件只关心向用户显示信息，并允许他们与应用程序交互。实现细节留给属于不同水平应用程序层的其他软件工件。用于表示或用户交互的复杂逻辑被委托给组件级服务，比如presenters.。

我们也可以有精益容器组件。它们将应用程序状态投影到表示组件，并将应用程序特定的事件转换为影响应用程序状态的命令。复杂的逻辑被委托给应用程序级的服务(如facade)——甚至可能是组件级的特定于组件的服务(如数据映射器)

容器/表示性组件模式的另一种选择是使用块(业务逻辑组件)来封装非表示性逻辑。

请参阅稻富树的文章《用Angular设计集团模式》了解集团。“[BLoC design pattern with Angular](https://medium.com/lacolaco-blog/bloc-design-pattern-with-angular-1c2f0339f6a3)”.

请务必阅读我的后续文章，我将这种设计模式与容器组件、演示组件和演示者进行了比较，并对Suguru的想法提出了一些改进建议。

即使我们的代码库中有更多的软件构件，每个类都处理整个应用程序控制流的一个非常特定的部分。每个组件都很容易推理，每个依赖项都很容易在测试中存根或模拟

我们从官方入门指南的“表单”部分将这些指导原则应用到cart组件。因此，我们看到了一个更健壮的组件体系结构，它遵循最佳实践，提高了可维护性、可测试性和可伸缩性。

简而言之，我们最终得到了精益的Angular组件，在许多简单的类上实现了更合理的关注点平衡

最后，我们通过选择风格指南的推荐来结束我们的旅程。如果您只阅读一条准则，请阅读样式05-15:将复杂的组件逻辑委托给服务。

我们还了解到，Angular服务和依赖的架构指南鼓励使用精益的Angular组件

别相信我的话。Ward Bell是Angular架构指南的最初作者。这条推文记录了他对这个话题的评论。

让我们将组件中的逻辑最小化到几乎不值得测试的水平。我们来运行一台瘦小的，有角度的机器。让我们设计精益和高效的Angular组件。

<https://indepth.dev/posts/1050/container-components-with-angular>

*Container components with Angular*

容器组件位于表示层的边界，并将UI与应用程序状态集成在一起。它们有两个主要目的:

容器组件提供用于表示的数据流。

容器组件将特定于组件的事件转换为应用状态命令或动作，并将其放到Redux/NgRx存储术语中。

容器组件还可以将UI集成到其他非表示层，如I/O或消息传递层。

在本文中，我们将介绍从混合组件中提取容器组件的过程。

本文中使用的大多数术语在介绍性文章“Angular的模型-视图-演示器”中有解释。

*容器组件*

我们称它们为容器组件，因为它们包含了视图中子组件所需的所有状态。此外，它们在它们的视图中只包含子组件——没有表示内容。容器组件的模板完全由子组件和数据绑定组成。

考虑容器组件的另一种有用方式是，它们——就像运输容器一样——是完全自包含的，并且可以在组件模板中任意移动，因为它们没有输入或输出属性。

容器组件解决了通过组件树的几个层来传递桶事件和属性的问题——在React社区中，这种现象被称为“道具钻取”。

简单的例子

我们从《英雄指南》教程中的DashboardComponent开始。

// dashboard.component.ts

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

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

import { HeroService } from '../hero.service';

@Component({

selector: 'app-dashboard',

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

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

})

export class DashboardComponent implements OnInit {

heroes: Hero[] = [];

constructor(private heroService: HeroService) {}

ngOnInit() {

this.getHeroes();

}

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes.slice(1, 5));

}

}

*确定混合问题*

正如介绍性文章中所描述的那样，我们看到这个组件在应用程序中有跨多个水平层的混合关注点。

web应用程序的水平层。在新标签页中打开。

首先，它与演示有关。它有一个英雄数组，显示在它的模板中。

<!-- dashboard.component.html -->

<h3>Top Heroes</h3>

<div class="grid grid-pad">

<a \*ngFor="let hero of heroes" class="col-1-4"

routerLink="/detail/{{hero.id}}">

<div class="module hero">

<h4>{{hero.name}}</h4>

</div>

</a>

</div>

<app-hero-search></app-hero-search>

虽然表示是UI组件的一个有效关注点，但这个混合组件也与状态管理紧密耦合。在一个NgRx应用程序中，这个组件可以注入一个Store，并使用状态选择器查询应用程序的状态。在《英雄指南》中，它会注入一个HeroService，并通过一个可观察对象查询英雄的状态，然后切片数组的一个子集，并在它的Heroes属性中存储一个引用。

*生命周期钩*

值得指出的是，我们的混合仪表板组件挂钩到它生命周期的OnInit时刻。它就是在这里订阅由HeroService#getHeroes返回的可观察对象的。这是一个合适的位置，因为订阅一个可观察对象会触发一个副作用，而我们不希望在构造函数或属性初始化器中出现这种副作用。

特别地，当我们订阅由HeroService#getHeroes返回的observable时，HTTP请求就会被发送。通过将异步代码排除在构造函数和属性初始化器之外，我们使组件更容易测试和推理。

如果你对RxJS可观察对象的基本概念不确定，可以阅读Gerard Sans的《Angular -响应式扩展(RxJS)简介》

*分裂混合组分*

为了分离混合组件的多层关注点，我们将其分为两个组件——一个容器组件和一个表示组件。

容器组件负责将UI与应用程序的非表示层(如应用程序状态管理层和持久层)集成在一起。

一旦我们确定了混合组件中的非表示性逻辑，我们就可以通过从混合组件模型中剪切源代码并将其粘贴到容器组件模型中来隔离和提取该逻辑来创建容器组件。

// dashboard.component.ts

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

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

import { HeroService } from '../hero.service';

@Component({

selector: 'app-dashboard',

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

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

})

export class DashboardComponent implements OnInit {

heroes: Hero[] = [];

constructor(private heroService: HeroService) {}

ngOnInit() {

this.getHeroes();

}

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes.slice(1, 5));

}

}

// dashboard.component.ts

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

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

@Component({

selector: 'app-dashboard',

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

styleUrls: ['./dashboard.component.css']

})

export class DashboardComponent {

heroes: Hero[] = [];

}

在将逻辑移动到容器组件之后，还有几个步骤要将混合组件转换为表示组件。这些步骤将在后面的一篇文章中详细解释，包括重命名标记名称，以及将数据绑定API与我们希望在容器组件模板中使用的API进行匹配。

*隔离和提取层集成*

// dashboard.container.ts

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

import { Observable } from 'rxjs';

import { map } from 'rxjs/operators';

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

import { HeroService } from '../hero.service';

@Component({

changeDetection: ChangeDetectionStrategy.OnPush,

selector: 'app-dashboard',

templateUrl: './dashboard.container.html',

})

export class DashboardContainerComponent {

topHeroes$: Observable<Hero[]> = this.heroService.getHeroes().pipe(

map(heroes => heroes.slice(1, 5)),

);

constructor(private heroService: HeroService) {}

}

我们提取了HeroService依赖，并创建了一个与混合型仪表板组件中的数据流相匹配的数据流。这是topHeroes$ observable属性，它会在HeroService#getHeroes返回的observable上添加一个操作管道。

我们的top heroes流在英雄服务的observable之后发出一个值，但只有当它被观察到的时候——当订阅已经创建的时候。我们在触发的英雄数组上进行映射，得到要呈现给用户的英雄子集。

*使用数据绑定连接表示组件*

在提取了应用状态集成逻辑之后，我们现在可以把仪表板组件看作是一个表示组件，并假设它有一个heroes input属性，就像仪表板容器组件的模板中看到的那样。

提取容器组件的最后一步是通过数据绑定(即容器组件模板中的属性绑定和事件绑定)将其连接到生成的表示组件

<!-- dashboard.container.html -->

<app-dashboard-ui

[heroes]="topHeroes$ | async"

title="Top Heroes"></app-dashboard-ui>

app-dashboard-ui是仪表板组件被转换成表示组件后的标签名。我们通过async管道将topHeroes$ observable连接到它的heroes输入属性。

我还从混合组件中提取了标题文本，并在容器组件模板中将其定义为标题。在下一篇关于表示组件的文章中，我将解释我们何时以及为什么要这样做。

现在，让我们满足于眼前的好处吧:presentational dashboard组件有可能在应用程序的不同部分被重新使用，并有一个标题来描述我们提供给它的英雄的不同子集。

谁来管理订阅?

*谁来管理订阅?*

有趣的是，我们去掉了ngOnInit生命周期钩子。我们的容器组件模型通过从一个已经存在的可观察对象中管道来准备top heroes数据流，这不会产生副作用，也就是说没有订阅。

现在订阅在哪里初始化?答案是，Angular替我们管理订阅。我们声明性地告诉Angular，通过在容器组件模板中使用async管道来订阅top heroes observable。

其结果是一个遵循presentationdashboard组件生命周期的订阅，并将英雄列表发送到heroes input属性中。

我们很高兴能够摆脱手动订阅管理，因为它既繁琐又容易出错。如果我们忘记取消订阅一个从未完成的可观察对象，我们可能会得到多个在应用程序会话剩余时间运行的订阅，从而导致内存泄漏。

*数据从容器组件向下流动*

把仪表板特性放入图1的流程图中，我们可以看到容器组件是如何通过一个可观察对象向英雄服务请求英雄的。

容器组件计算顶级英雄，并把它们传递给presentational组件的input属性。英雄数组可以在最终在DOM中显示给用户之前通过一个演示器传递，但容器组件不知道这一点，因为它只知道演示组件的数据绑定API。

*先进的例子*

让我们继续看《英雄指南》中的HeroesComponent，看一个更高级的例子。

// heroes.component.ts

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

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

import { HeroService } from '../hero.service';

@Component({

selector: 'app-heroes',

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

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

})

export class HeroesComponent implements OnInit {

heroes: Hero[];

constructor(private heroService: HeroService) {}

ngOnInit() {

this.getHeroes();

}

add(name: string): void {

name = name.trim();

if (!name) { return; }

this.heroService.addHero({ name } as Hero)

.subscribe(hero => {

this.heroes.push(hero);

});

}

delete(hero: Hero): void {

this.heroes = this.heroes.filter(h => h !== hero);

this.heroService.deleteHero(hero).subscribe();

}

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes);

}

}

*隔离层的集成*

乍一看，这个组件可能看起来很小、简单而天真。仔细观察，这个组件似乎有很多问题(双关语)。像前面的例子一样，ngOnInit生命周期钩子和getHeroes方法是用来查询应用程序状态的。

web应用程序的水平层(或系统关注点)。在新标签页中打开。

delete方法处理持久化状态，它用一个数组替换heroes属性，删除的英雄会被过滤掉。这个方法还与持久性有关，因为它通过hero服务从服务器状态中删除了一个英雄。

最后，add方法处理用户交互，在创建英雄之前验证英雄名，这是持久化层和应用状态层关心的问题。

*提取层集成*

我们的工作有困难吗?让我们通过将这些多层系统提取到容器组件中来摆脱它们。

// heroes.component.ts

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

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

import { HeroService } from '../hero.service';

@Component({

selector: 'app-heroes',

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

})

export class HeroesContainerComponent implements OnInit {

heroes: Hero[];

constructor(private heroService: HeroService) {}

ngOnInit() {

this.getHeroes();

}

add(name: string): void {

this.heroService.addHero({ name } as Hero)

.subscribe(hero => {

this.heroes.push(hero);

});

}

delete(hero: Hero): void {

this.heroes = this.heroes.filter(h => h !== hero);

this.heroService.deleteHero(hero).subscribe();

}

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes);

}

}

像这个简单的例子一样，我们把HeroService依赖提取到一个容器组件中。我们在可变英雄属性中维护英雄状态。

这将与默认的变更检测策略一起工作，但是我们希望通过使用OnPush变更检测策略来提高性能。我们需要一个可观察对象来管理英雄状态。

hero服务返回一个可观察对象，它会发出一个英雄数组，但我们还需要支持添加和删除英雄。一种解决方案是使用BehaviorSubject创建一个有状态的observable。

然而，要使用subject，我们需要订阅hero service observable，这会产生副作用。如果可观察对象在发出一个值后还没有完成，我们还必须自己管理订阅，以防止内存泄漏。

此外，当添加或删除一个英雄时，我们必须减少英雄状态。这很快就会变得复杂起来。

*状态管理*

为了以响应方式跟踪应用程序状态，我创建了一个名为rxjs-multi-scan的微库。多扫描组合操作符通过一次扫描操作合并多个可观测对象以计算当前状态，但每个可观测源通常有一个小减速函数。将初始状态作为最后一个参数传递给操作符。

每个奇参数-除了初始状态参数-是一个源可观察对象，它的偶数参数是其扫描状态的减速函数。

// heroes.container.ts

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

import { noop, Observable, Subject } from 'rxjs';

import { multiScan } from 'rxjs-multi-scan';

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

import { HeroService } from '../hero.service';

@Component({

changeDetection: ChangeDetectionStrategy.OnPush,

selector: 'app-heroes',

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

})

export class HeroesContainerComponent {

private heroAdd: Subject<Hero> = new Subject();

private heroRemove: Subject<Hero> = new Subject();

heroes$: Observable<Hero[]> = multiScan(

this.heroService.getHeroes(),

(heroes, loadedHeroes) => [...heroes, ...loadedHeroes],

this.heroAdd,

(heroes, hero) => [...heroes, hero],

this.heroRemove,

(heroes, hero) => heroes.filter(h => h !== hero),

[]);

constructor(private heroService: HeroService) {}

add(name: string): void {

this.heroService.addHero({ name } as Hero)

.subscribe({

next: h => this.heroAdd.next(h),

error: noop,

});

}

delete(hero: Hero): void {

this.heroRemove.next(hero);

this.heroService.deleteHero(hero)

.subscribe({

error: () => this.heroAdd.next(hero),

});

}

}

在我们的用例中，初始状态是一个空数组。当由HeroService#getHeroes返回的可观察对象发出一个英雄数组时，它会把它们连接到当前状态。

我为每个用户交互创建了一个RxJS主题——一个用于添加英雄，另一个用于删除英雄。当一个英雄通过私有的heroAdd属性被触发时，在多扫描操作中相应的reducer函数会把它追加到当前状态。

当一个英雄被移除时，这个英雄会通过heroRemove主题被释放出来，它会触发当前英雄状态的过滤器来过滤指定的英雄。

*坚持更新策略*

我们允许在公共方法add和delete中添加或删除英雄。当添加了一个英雄时，我们使用悲观更新策略，首先通过hero服务将该英雄持久化到服务器状态中，只有成功了，我们才会在heroes$中更新持久化状态。

目前，我们不处理更新服务器状态时的错误。这可以从subscribe observer参数中的错误处理程序为noop中看出。假设我们希望向用户显示toast或重试操作，我们将在错误处理程序中这样做。

当删除一个英雄时，我们应用乐观更新策略，首先从持久状态中删除这个英雄，然后再从服务器状态中删除它。如果删除失败，我们就回滚这个持久化状态，通过heroadd主题把这个英雄添加回heroes$中。

这是对最初实现的一个改进，在最初的实现中，删除英雄时不会处理服务器错误。

*事件向上流动到容器组件*

让我们在心里把英雄特性放入图2的流程图中。想象一下用户是如何输入英雄的名字，然后点击Add按钮的。

表示组件模型中的一个方法会以新英雄的名字来调用。在通过输出属性将英雄名作为事件发出之前，演示组件可能会将用户交互逻辑委托给演示者。

容器组件会收到发出的英雄名的通知，它会把它传递给英雄服务，并最终更新容器组件模型中的持久状态。

更新后的英雄状态会通知表示组件，数据流继续进行，如图1所示。

*应用程序状态是另一个问题*

需要注意的是，虽然应用的状态可以特定于应用的某个特性，但《英雄指南》中的多个区域都使用了英雄状态。如前所述，它是镜像服务器状态的部分的持久状态。理想情况下，我们的英雄容器组件不应该自己管理持久化状态，而应该依赖于hero服务来做这件事——或者是使用NgRx store的应用程序中的store。

尽管heroes的状态是在特定特性的容器组件中管理的，但它在应用中是一致的。这是因为仪表板每次初始化英雄服务器时都会向英雄服务请求英雄服务器的状态，这会导致一个HTTP请求来水合物(初始化)持久化状态。

在这些相关文章中，我们将重点关注Angular组件。为了做到这一点，我们不会修改服务。如果你想把英雄状态放到它所属的英雄服务中，你可以从这个容器组件中提取状态管理。

看到了吗?一旦我们分离了关注点，就很容易隔离特定类型的逻辑，并将其放在它所属的应用层中。

*使用不可变数据*

在mixed heroes组件中，使用数组#push方法将一个英雄添加到英雄状态中。这会改变数组，这意味着不会创建新的引用。虽然Angular的默认变更检测策略支持这一点，但我们在所有组件中都使用OnPush变更检测策略来选择性能。

要让这个策略生效，我们需要在添加英雄时发出一个新的数组引用。为此，我们在一个新的数组字面量中使用spread操作符(…)，从英雄的快照(当前)值中复制英雄，并包含额外的英雄。这个新数组会被发送给heroes$属性的观察者。

*剩下的逻辑*

如果你在编辑器中继续看下去，你可能会注意到，我们把验证逻辑留在了mixed heroes组件中。这是有意为之的，因为它既不涉及应用程序状态，也不涉及持久性。

// heroes.component.ts

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

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

@Component({

selector: 'app-heroes',

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

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

})

export class HeroesComponent {

heroes: Hero[];

add(name: string): void {

name = name.trim();

if (!name) { return; }

}

delete(hero: Hero): void {}

}

*使用表示组件的数据绑定API连接它*

最后一步是将容器组件连接到容器组件模板中的表示组件的数据绑定API。

<!-- heroes.container.html -->

<app-heroes-ui

[heroes]="heroes$ | async"

title="My Heroes"

(add)="add($event)"

(remove)="delete($event)"></app-heroes-ui>

和这个简单的例子一样，我们通过异步管道将英雄的input属性连接到我们的observable属性。每当英雄的状态发生变化时，它就会向presentational组件传递一个新的数组引用。

记住，当我们使用异步管道时，Angular会替我们管理heroes$ observable的订阅，这样它就会遵循呈现组件的生命周期。

*事件绑定*

在presentational heroes组件中，用户可以通过添加或删除英雄来改变应用的状态。我们希望，当用户每次添加或删除一个英雄时，演示组件都会通过输出属性发出一个英雄，所以我们把容器组件的add方法连接到演示组件的add事件中。

同样，我们将delete方法连接到remove事件。我之所以将这个方法命名为delete，是因为它的目的是从服务器状态中删除英雄，同时保持持久化状态的同步。

虽然容器组件可以处理删除意图，但表示组件不应该关注应用程序状态，只关注本地UI状态。它只能在用户请求删除英雄时触发特定于组件的事件。移除事件会被heroes容器组件转换成一个持久化命令，从而改变应用的状态。新的状态以新的数组引用的形式流向表示组件的输入属性。

*应用OnPush变更检测策略*

在构建容器组件时，我们要确保使用的是可观察对象来流化应用程序的状态。与此同时，我们只在可观察对象中使用不可变的数据结构。

这使我们能够在容器组件中使用OnPush变更检测策略，因为异步管道会在通过一个可观察对象发出值时触发变更检测。因为在处理不可变数据结构时，每个新值都会触发一个新的引用，所以我们也可以将OnPush变更检测策略应用到表示组件上。

请记住，容器组件有两个主要目的:

容器组件提供用于表示的数据流。

容器组件将特定于组件的事件转换为应用状态命令或动作，将其放到Redux/NgRx存储术语中。

使用容器组件的一大优点是增加了可测试性。继续学习“测试Angular容器组件”。

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

Presentational components with Angular

表示组件就是我们Angular应用的用户界面。它们有两个目的:

向用户显示应用程序状态

更改由用户交互触发的应用程序状态

为了与应用程序的其他部分进行通信，表示组件具有输入属性，为它们提供将格式化以便显示的数据。它们使用输出属性来通知由用户交互发起的应用程序状态更改。这是它们的数据绑定API。

表示组件一方面与用户交互，另一方面与应用程序状态和其他非表示层交互。它们使用容器组件作为粘合剂，将它们连接到非表示的应用层。

与应用程序的其他部分相比，数据进入，事件发出。表示组件不关心数据从哪里来，事件从哪里去。当涉及到用户时，数据通过组件模板呈现给他们。来自用户的事件通过事件处理程序(即使用事件绑定在组件模板中绑定的方法)进入组件。

表示组件也可以使用属性绑定吗?当然，它们可以在组件树中传递任何状态。同样，它们也可能从其他表示组件向组件树转发事件，类似于向组件树转发用户初始化的事件。

*无状态的表示组件*

表示组件可以是无状态的，这意味着它们的外观和行为总是相同的。无状态表示组件完全与呈现给用户的内容有关。

在Angular中，我们有另一个用于表示的构建块:指令。指令应该是我们演示的解决方案。如果创建一个组件来添加特定的样式(如粗体字体或背景颜色)，这将是一个糟糕的选择。我们可以使用纯CSS来实现这一点，也可以将表示封装在一个属性指令中。

无状态表示组件的一个很好的例子是工具栏组件，它只是投影内容，并围绕内容包装DOM结构以实现样式化和布局目的。

*有状态的表示组件*

表示组件可以有自己独立的状态。考虑一个复选框组件。它至少有两个状态:已检查和已清除。复选框状态是本地UI状态的一部分。

复选框组件是有状态的。如果这种状态不能持续下去，那又有什么用呢?下次我们返回到这个路径时，我们通常会期望复选框处于相同的状态。

*通知应用程序状态更改*

为了有用，这个本地UI状态必须与应用程序的其他状态同步。然而，如果我们添加持久化逻辑来在WebStorage中存储复选框状态，我们将处理一个混合组件，而不是表示组件。

为了保持纯粹的表示性，复选框组件通过输出属性发送状态事件，将其状态更改传达给应用程序的其余部分。

*获得状态更改的通知*

有状态表示组件可以具有初始状态，而与应用程序的其他部分无关。但是为了与应用程序的其他部分同步，它需要一种方法在组件被激活时对其状态进行水合物化。复选框状态将由输入属性控制。

用户可能不是唯一能够更改复选框状态的参与者。可能出于某种原因，计时器会在10秒后将状态设置为检查状态。checkbox组件将通过其input属性得到状态更改的通知。

*精益的组件*

当遵循模型-视图-表示器模式时，我们将保持表示组件的精简。我们将逻辑排除在组件模板之外，也排除了组件模型(组件类实例)。

组件模板应该只是为表示设置表达式绑定，为用户交互设置事件绑定。

行为应该委托给演示者，这些演示者是组件级的依赖关系，与应用程序的其他部分完全隔离。这确保了组件模型只协调输入属性、输出属性、UI属性和演示器的配置和绑定。

除了数据绑定API、UI属性、事件处理程序和演示程序之间的粘合代码之外，模型-视图-演示程序风格的演示组件的组件模型不包含任何业务逻辑。

*Presentational components*

们称它们为表示组件，因为它们表示应用程序的表示层，如表1所示的表示层和用户交互层

优选地，我们将用户交互提取到组件级服务，如“精益表示组件”一节中所述的演示程序。

*表示组件通常是可重用的*

表示组件通常是可重用的。它们的数据绑定API或渲染API允许在许多地方使用它们。

不过，我们可以很容易地拥有一次性的表示组件。例如，我们可以有一个标识组件来显示我们的标识图像。徽标组件通常只在主布局中使用。其他布局组件，比如主导航组件、顶部应用栏组件或侧边抽屉组件，都是只在父组件中使用的，但肯定至少有一个表示组件。根据它们的复杂性，还可以将它们分为容器组件和表示组件。

可重用表示组件的一个很好的例子是按钮组件。我们可以为我们的组织实现一个包含按钮的设计系统。每个团队中的所有开发人员都应该能够重用按钮组件，而不必担心设计系统在颜色、字体或间距方面的变化。或者我们从角材质切换到材质UI。当设计系统不可避免地发生变化时，按钮组件的实现细节将使我们能够在单个地方进行更改。

*表示组件是纯粹的*

表示组件是纯粹的，因为它们没有副作用。与状态管理、持久性、消息传递、I/O和其他非表示层的集成属于容器组件

因为它们是纯的，所以它们呈现DOM和通过输出属性发出事件的方式是确定的。

图1说明了当传递输入值valueX和valueY时，这个表示组件的DOM将始终呈现在组合AxBy中。

在图2中，valueX是输入，随后是作为事件y拦截的用户交互。输入值和事件序列的组合导致了DOM组合AxEy。当输入valueX时，事件Y发生时总是这样。

图3中的表示组件具有基于valueX作为输入传递的DOM组合Ax。作为事件Z拦截的用户交互导致eventZ的值通过输出属性发出。

当valueX是输入值时，事件Z总是发生这种情况。

我们必须能够在测试中演示图1、2和3中的示例。否则，我们的组件是不纯的，因为它们依赖于外部状态。如果是这种情况，我们需要创建另一个输入属性，并传入该外部状态，将组件转换为具有确定性行为和呈现的表示组件。

表示组件变得不干净，因为有两个原因之一需要进行不干净检查:

发生了外部事件(如用户交互)，并由组件模板中的事件绑定获取

新数据被传递给一个或多个输入属性

正因为如此，我们可以通过使用OnPush变更检测策略来优化变更检测周期中的性能。

*Simple example*

于涛

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

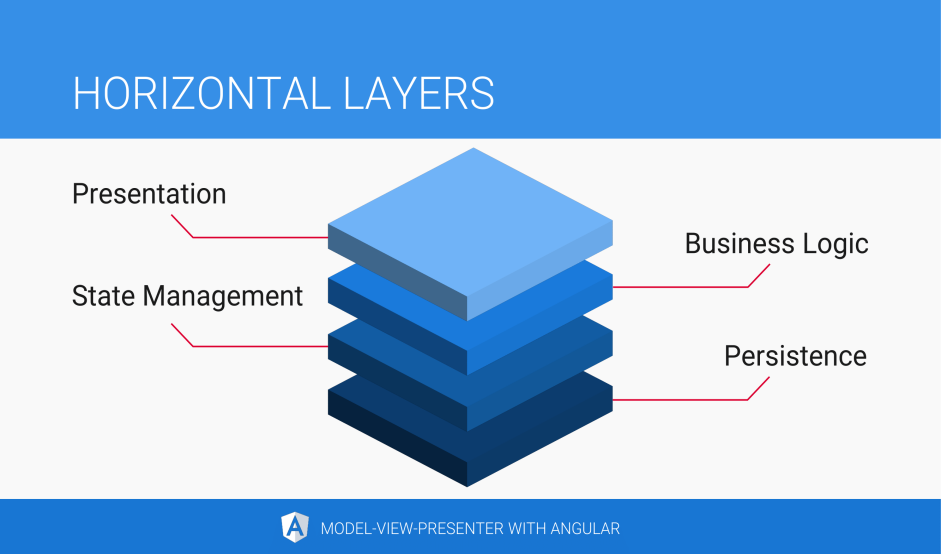
6 组件的高度圈复杂度——高度集中的逻辑分支或异步控制流

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-表示器(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，它将控制权委托给表示器。

表示器批量修改状态，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。表示器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-表示器”(幻灯片)。

受原始的模型-视图-表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表示器连接到模型和视图，从而使表示器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。由于我们的表示器被他们的视图封装，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，表示器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。

我们的视图绑定到表示器的属性上，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-表示器模式应用到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。

除了表示器之外，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表示器中，得到一个简单的表示组件。表示器没有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。

表示器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。

表示器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

《看图》

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。

显示组件（presentational component）将更新后的数据传递给表示器(presenter)，表示器(presenter)重新计算显示组件（presentational component）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

《看图》

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。

用户交互被表示器截获，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-表示器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表示器添加了测试套件。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

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-表示器”中描述的普通JavaScript的模型-视图-表示器模式。 “[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 （表示器）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。

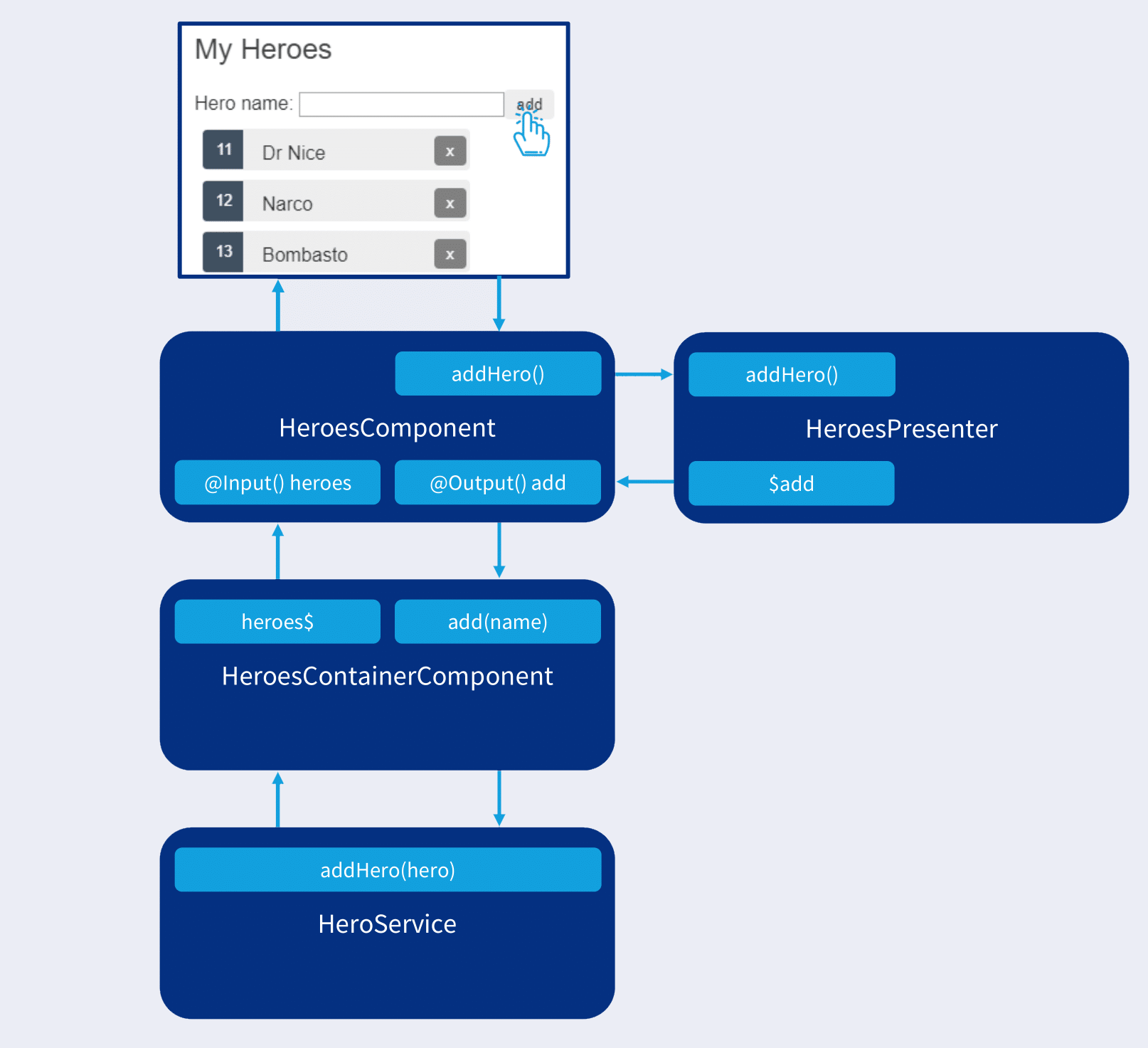
表示器实际上与应用程序的其他部分隔离开来。它们通常根本没有依赖关系，除了复合表示器中的其他表示器。这使得它们在没有DOM或UI的情况下，甚至在没有Angular的情况下，都很容易被单独测试，只要我们仔细设计它们。

表示器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，表示器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表示器，给出1:n组件与表示器的比例。

我们也可以有多个特定于同一用例的表示器，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。我们可以有一个表示器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。

***一个表示器在多个组件之间共享。***

还有一种情况是，单个表示器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组件与表示器的比率。

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，它们将需要单独的表示器实例，这意味着1:1或1:n的比例。

共享表示器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表示器，特定于行或特定于列的表示器处理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、表级的表示器，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表示器。它还将编排源自行和单元级组件和表示器的用户交互。

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表示器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表示器。

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所示的搜索表示器。

// 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。

拥有一个可重用的表示器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。

我们有效地将搜索表示器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，而不是搜索表示器。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，当表示器和DOM片段之间具有高度内聚性时，我们可以重用组件(带有表示器)而不是表示器。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表示器

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(表示器)，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。

我们需要在表示器(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:表示器和被观察对象暴露在一起。

表示器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（表示器）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。

**：**表示器被添加到providers组件选项中，该选项将其范围扩展到组件级别，这意味着表示器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表示器（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表示器。

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

将表示器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，让我们连接表示器的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连接到它的表示器。

在清单9A中，我们订阅了表示器的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连接到它的表示器。

另外，我们还可以通过将输出属性订阅到可观察对象add$ property，将表示器连接到输出属性，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委托给表示器的可观察对象属性。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。

如果我们的表示器包含表示转换方法(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表示器。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

如果我们使用了表示器的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，让表示器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。

使用由表示器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

表示器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:表示器，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“组件与表示器的比率”一节所讨论的，我们可能想要共享一个允许我们共享的表示器。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。

表示器注入提示:默认使用组件的viewProviders选项提供表示器，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表示器本身来防止表示器的意外共享。

// 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:表示器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:表示器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。使用表示器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表示器实例，这样就不可能意外地共享表示器。

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，在将英雄表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表示器注入。

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表示器。

表示器注入提示:使用Self decorator工厂注入表示器，除非它是一个共享的表示器。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。我们的表示器公开了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表示器。

// 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表示器。

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，我们应该让我们的表示器与框架无关。

// 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表示器。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特有的表示器，包装了与框架无关的英雄表示器。

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的表示器保留了相同的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——在复合的表示器中，依赖关系可能是其他表示器。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。

表示器可以是有状态的，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。

状态的表示器关心的是表示，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-表示器比率。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表示器，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。无状态的表示器很容易以这种方式共享。

我们了解到，当表示器和DOM片段之间的内聚性很高时，我们可以重用组件而不是表示器。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，表示器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。

使用表示器的好处之一是，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表示器，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 将表示器注入演示组件。

3 将表示器连接到演示组件的数据绑定API。

4 管理可观测的订阅。

在提取表示器时，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。