动态组件

1、组件的模板不会永远是固定的。

2、应用可能会需要在运行期间加载一些新的组件。

3、展示如何使用 ComponentFactoryResolver 来动态添加组件。

有两个场景 ——

eg1：假设正在计划一个广告活动，要在广告条中显示一系列不同的广告。几个不同的小组可能会频繁加入新的广告组件。

eg2：用户拖动相关的块到特定区域，区域中便会生成相应的UI控件，此UI控件有自己的模板、行为等等。生成UI 后便会在区域中显示出来。将UI控件封为一个小组件，再动态加载是个不错的方案。

**！！！**此时再用只支持静态组件结构的模板显然是不现实的。需要一种新的组件加载方式，它不需要在广告条组件的模板中引用固定的组件。Angular 自带的 API 就能支持动态加载组件。

指令

在添加组件之前，先要定义一个锚点来告诉 Angular 要把组件插入到什么地方。

广告条使用一个名叫 AdDirective 的辅助指令来在模板中标记出有效的插入点。



AdDirective 注入了 ViewContainerRef 来获取对容器视图的访问权，这个容器就是那些动态加入的组件的宿主。

在 @Directive 装饰器中，要注意选择器的名称：appAd，它就是将应用到元素上的指令。

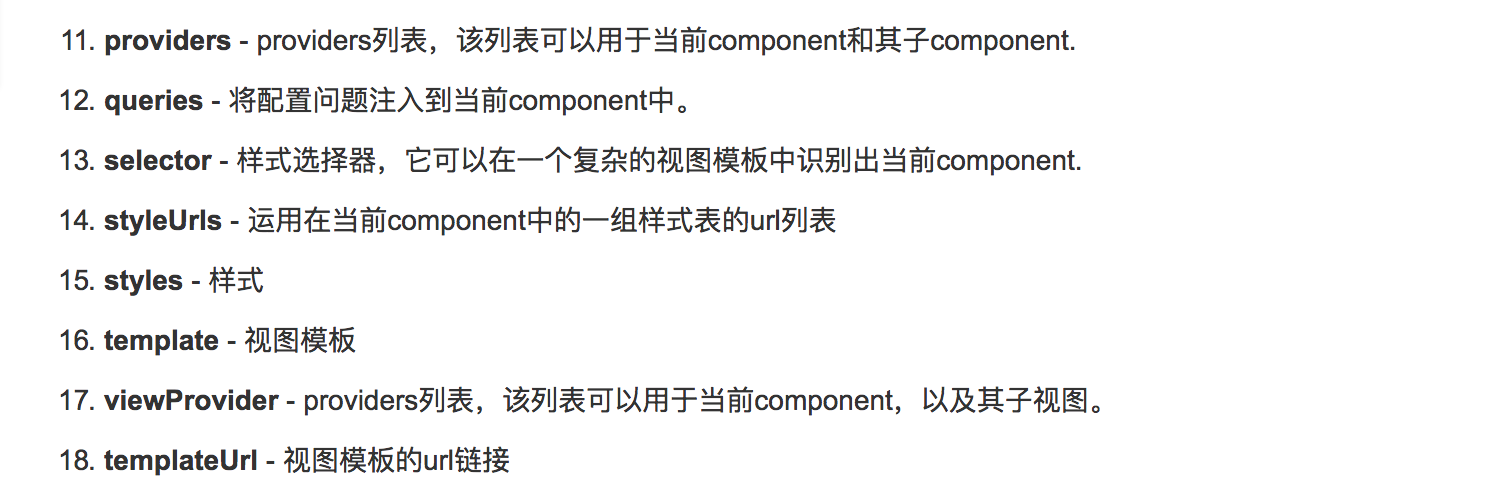
加载组件

广告条的大部分实现代码都在 ad-banner.component.ts 中。 这里为了方便HTML 被直接放在了 @[Component](https://www.angular.cn/api/core/Component) 装饰器的 template 属性中。（拓展：@Component 元数据的属性）

<ng-template> 元素就是刚才制作的指令将应用到的地方。 要应用 AdDirective，回忆一下来自 ad.directive.ts 的选择器 appAd。把它应用到 <ng-template>（不用带方括号）。 这下，Angular 就知道该把组件动态加载到哪里了。





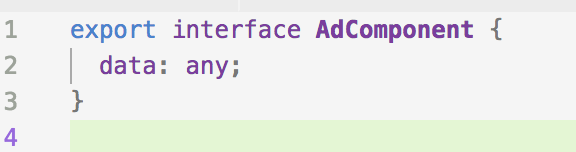


<ng-template> 用于定义模板，使用**\***语法糖的结构指令，最终都会转换为 <ng-template> 模板指令，ng-template元素是动态加载组件的最佳选择，因为它不会渲染任何额外的输出。模板内的内容如果不进行处理，是不会在页面中显示。

解析组件

认识interface：

声明接口



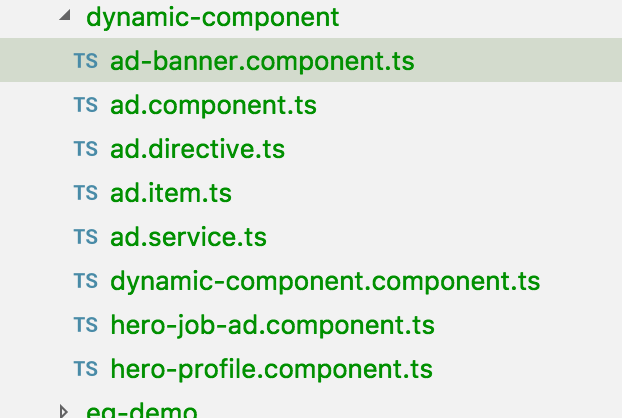
typescript声明合并简单介绍：<https://www.tslang.cn/docs/handbook/declaration-merging.html>

先简单了解angular2 中相关的api：

* ViewChild：一个属性装饰器，用来从模板视图中获取对应的元素，可以通过模板变量获取，获取 时可以通过 read 属性设置查询的条件，就是说可以把此视图转为不同的实例
* ViewContainerRef ：一个视图容器，可以在此上面创建、插入、删除组件等等
* ComponentFactoryResolve：一个服务，动态加载组件的核心，这个服务可以将一个组件实例呈现到另一个组件视图上

有了这三个，一个简单的思路便连贯了：特定区域就是一个视图容器，可以通过 ViewChild 来实现获取和查询，然后使用ComponentFactoryResolve将已声明未实例化的组件解析成为可以动态加载的 component，再将此component 呈现到此前的视图容器中。

测试模块结构图：



深入看看 ad-banner.component.ts 中的代码：

AdBannerComponent 接收一个 AdItem 对象的数组作为输入，它最终来自 AdService。

AdItem 对象指定要加载的组件类，以及绑定到该组件上的任意数据。

AdService 可以返回广告活动中的那些广告。

给 AdBannerComponent 传入一个组件数组可以在模板中放入一个广告的动态列表，而不用写死在模板中。

通过 getAds() 方法，AdBannerComponent 可以循环遍历 AdItems 的数组，并且每三秒调用一次 loadComponent() 来加载新组件。



这里的 loadComponent() 方法很重要。首先，它选取了一个广告。

loadComponent() 如何选择广告？

loadComponent() 方法使用什么算法选择了的广告？

（循环选取算法——首先，它把 currentAdIndex 递增一，然后用它除以 AdItem 数组长度的余数作为新的 currentAdIndex 的值，最后用这个值来从数组中选取一个 adItem。）

接下来，你要把 viewContainerRef 指向这个组件的现有实例。但你怎么才能找到这个实例呢？很简单，因为它指向了 adHost，而这个adHost 就是你以前设置过的指令，用来告诉 Angular 该把动态组件插入到什么位置。

AdDirective 曾在它的构造函数中注入了一个 ViewContainerRef。 因此这个指令可以访问到这个你打算用作动态组件宿主的元素。

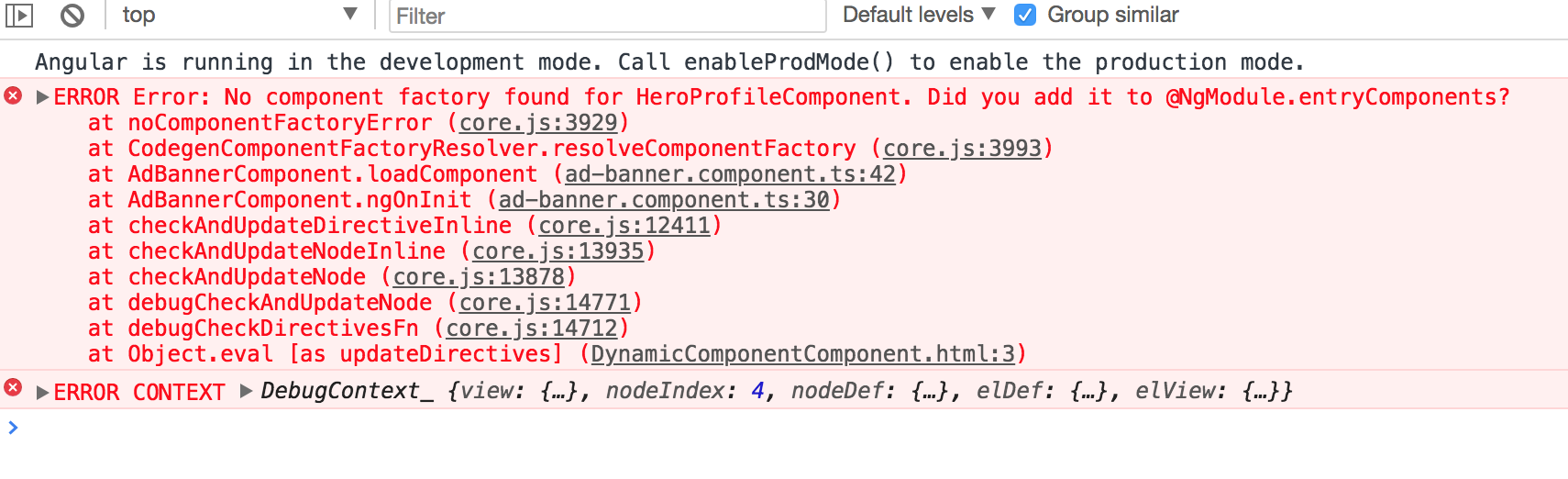
要把这个组件添加到模板中，你可以调用 ViewContainerRef 的 createComponent()。

createComponent() 方法返回一个引用，指向这个刚刚加载的组件。 使用这个引用就可以与该组件进行交互，比如设置它的属性或调用它的方法。

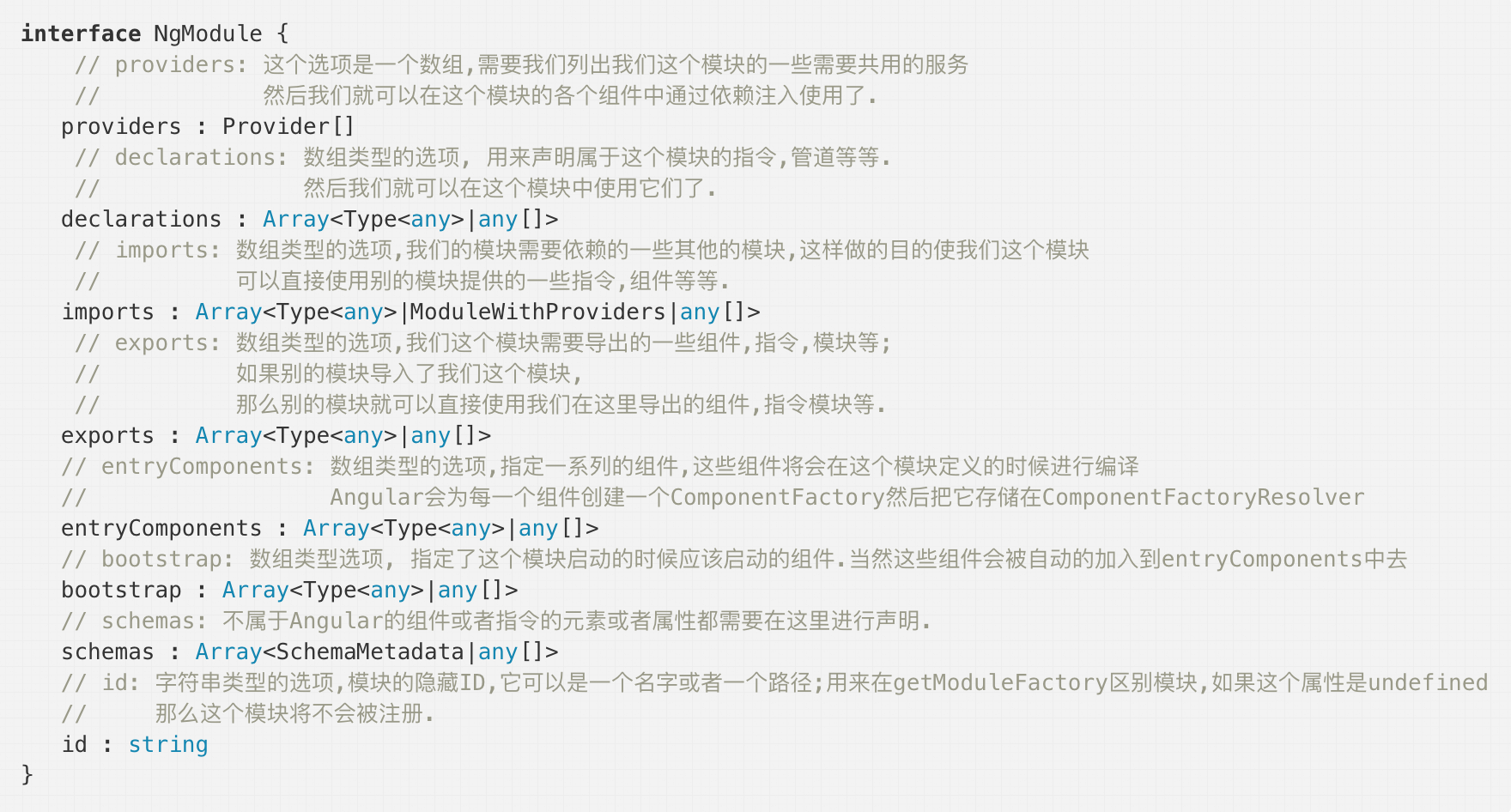
(<AdComponent>componentRef.instance).data => typescript 类型保护拓展

（还可以在模板变量名获取时，通过 read 选项设置为一个 ViewContainerRef ，最终在生命钩期 ngAfterViewInit 过后便会获取此区域的一个 ViewContainerRef 实例。）

此时有个报错哟——

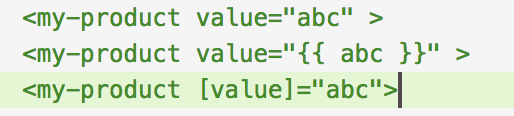


解决bug之前先认识下ngModule：



原来动态加载的组件必须声明在特性模块的 entryComponents 中，也就是动态加载必需的步骤。所以将其加到特性模块 entryComponents中。

根据instance拓展两个关于数据绑定的小知识点：

1. 

1）左边不使用 [括弧] :

这样的写法右边的值 "abc" 只是一个很普通的string, 它并不是引用我们component instance的属性。正如此,之后就不可能改变它的值了。算是 one time binding and only for string. (html 就是这样)。

2）不适用括弧但用插值:

这样的话右边的 "abc" 会引用 component instance, 而且会一直保持同步, 也就是说不是 one time binding 了。但值依然只能是 string( abc must be string)。

3）左边是 [括弧] :

这样的话 abc 就可以传入任何类型了, 也会一直保持同步。这也是我们最常用的表达模式。

2. ng是通过数据绑定加上事件绑定来完成双向绑定，[(value)] 只是语法糖。把它拆开是[]、()。当我们绑定到原生组件时，dom Attribute 和 dom Property？

1）dom Attribute 和 dom Property 是同一个东西？

2）<img src="123.jpg" /> 写在 element 上的是 Attribute, 它的值只用来初始化 property 的值

3）imgDomObject.src = "456.jpg" 这个才是 property。

4）ng 的绑定是在 property 上的。

Attribute和Property 不总是一对一匹配的, 有些 Attribute 是没有 property 的，这时我们要这样写<some-elem [attr.attributeName]="..." > 要在前面加 ‘attr.’ 。

Wait! Wait! Wait!

你是否发现了，以上属于 动态加载已经声明的组件？！

是否可以 用动态创建模块的方式来加载动态创建的组件咧？

--Jit 的情况下你可以动态的写组件模板, 最后 append 出去, 类似 ng1 的 $compile 。

--Aot 的话, 模板是固定的, 我们只是可以动态创建 component 然后 append 出去。

另外一个项目场景：

我拖动生成了UI 控件，只是为了展示对应的样式，UI控件是需要第三方环境支持的，所以我需要加工拖动的数据，抛给后台处理，后台返回包含表示模板、组件的字符串的 JSON 数据回来，然后我通知另外一个网站，此网站包含了第三方环境，让他们去动态创建这些组件。一句话概括，就是我要动态创建不存在的组件而不是已经声明的组件。

（详见demo）

指令

在 Angular 中有三种类型的指令：

组件 — 拥有模板的指令

结构型指令 — 通过添加和移除 DOM 元素改变 DOM 布局的指令

属性型指令 — 改变元素、组件或其它指令的外观和行为的指令。

组件是这三种指令中最常用的。 你在快速上手例子中第一次见到组件。

结构型指令修改视图的结构。例如，NgFor 和 NgIf。 要了解更多，参见结构型指令 guide。

属性型指令改变一个元素的外观或行为。例如，内置的 NgStyle 指令可以同时修改元素的多个样式。

属性型指令

比如一个高亮效果的自定义指令



在指令的构造函数中注入 ElementRef，来引用宿主 DOM 元素，ElementRef 通过其 nativeElement 属性给你了直接访问宿主 DOM 元素的能力。

使用 HostListener 装饰器添加两个事件处理器，它们会在鼠标进入或离开时进行响应。

@HostListener 装饰器引用属性型指令的宿主元素，在这个例子中就是 <span>

当然也你可以通过标准的 JavaScript 方式手动给宿主 DOM 元素附加一个事件监听器。 但这样至少有三个问题：

1）必须正确的书写事件监听器。

2）当指令被销毁的时候，必须拆卸事件监听器，否则会导致[内存泄露](https://blog.csdn.net/ruiruihahaha/article/details/70270574)。

3）必须直接和 DOM API 打交道，应该避免这样做。

使用 @Input 数据绑定向指令传递值

实现目标：让指令的使用者可以指定要用哪种颜色进行高亮。

@Input 装饰器。它往类上添加了一些元数据，从而让该指令的 highlightColor 能用于绑定。它之所以称为输入属性，是因为数据流是从绑定表达式流向指令内部的。 如果没有这个元数据，Angular 就会拒绝绑定。



很不错，但如果可以在应用该指令时在同一个属性中设置颜色就更好了，就像这样：



[appHighlight] 属性同时做了两件事：把这个高亮指令应用到了 <span> 元素上，并且通过属性绑定设置了该指令的高亮颜色。你复用了该指令的属性选择器 [appHighlight] 来同时完成它们。这是清爽、简约的语法。



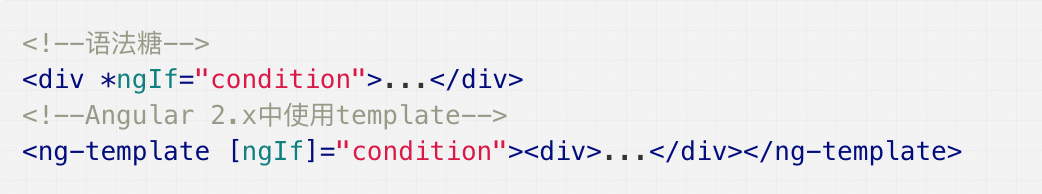
属性型指令

像其它指令一样，你可以把结构型指令应用到一个宿主元素上。 然后它就可以对宿主元素及其子元素做点什么。

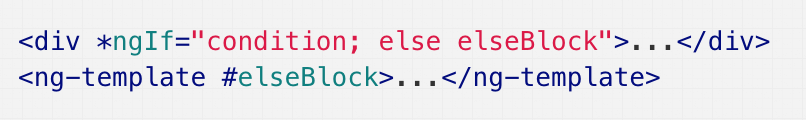
1. ngIf 指令并不是使用 CSS 来隐藏元素的，它会把这些元素从 DOM 中物理删除。当条件为假时，NgIf 会从 DOM 中移除它的宿主元素，取消它监听过的那些 DOM 事件，从 Angular 变更检测中移除该组件，并销毁它。 这些组件和 DOM 节点可以被当做垃圾收集起来，并且释放它们占用的内存。

NgIf 指令语法

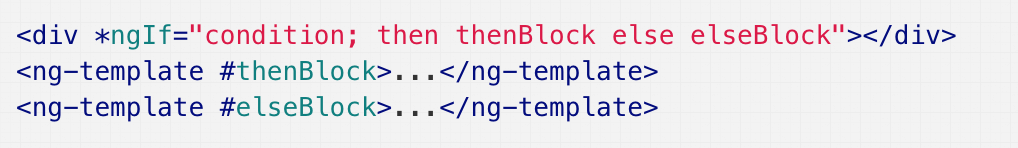
简单形式



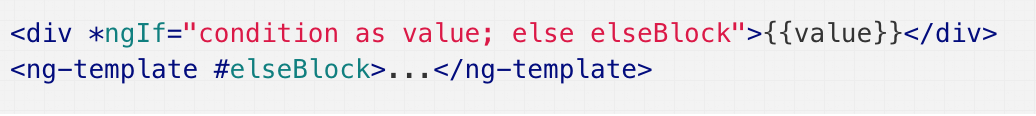
使用else块



使用then和else块



使用as语法

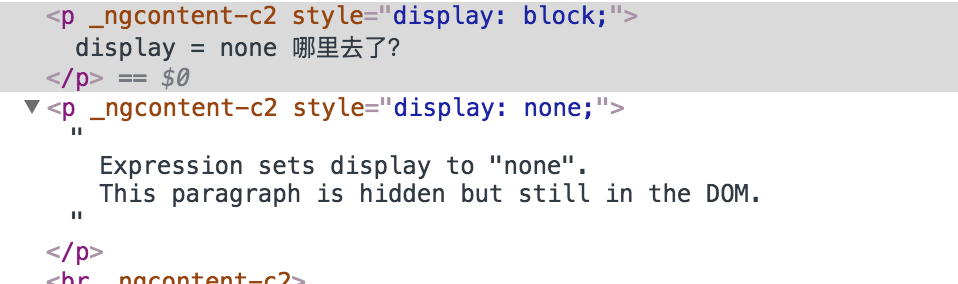


as可以理解为带有隐形转换的作用！！！

ngIf 指令用于根据表达式的值，在指定位置渲染 then 或 else 模板的内容。

* then 模板除非绑定到不同的值，否则默认是 ngIf 指令关联的内联模板。
* else 模板除非绑定对应的值，否则默认是 null。

指令也可以通过把它的 display 风格设置为 none 而隐藏不需要的段落。



对于简单的段落，隐藏和移除之间的差异影响不大，但对于资源占用较多的组件是不一样的。 当隐藏掉一个元素时，组件的行为还在继续 —— 它仍然附加在它所属的 DOM 元素上， 它也仍在监听事件。Angular 会继续检查哪些能影响数据绑定的变更。 组件原本要做的那些事情仍在继续。

虽然不可见，组件及其各级子组件仍然占用着资源，而这些资源如果分配给别人可能会更有用。 在性能和内存方面的负担相当可观，响应度会降低，而用户却可能无法从中受益。

当然，从积极的一面看，重新显示这个元素会非常快。 组件以前的状态被保留着，并随时可以显示。 组件不用重新初始化 —— 该操作可能会比较昂贵。 这时候隐藏和显示就成了正确的选择。

但是，除非有非常强烈的理由来保留它们，否则你会更倾向于移除用户看不见的那些 DOM 元素，并且使用 NgIf 这样的结构型指令来收回用不到的资源。

ngFor ngIf 不能作用在同个dom上 ！

ngFor循环内嵌套的 ngIf 如何加索引变量 index (i) ！

ngFor as 后面不能接表达式和filter !

管道

每个应用开始的时候差不多都是一些简单任务：获取数据、转换它们，然后把它们显示给用户。 获取数据可能简单到创建一个局部变量就行，也可能复杂到从 WebSocket 中获取数据流。

一旦取到数据，你就可以把它们原始值的 toString 结果直接推入视图中。 但这种做法很少能具备良好的用户体验。

Angular 内置了一些管道，比如 DatePipe、UpperCasePipe、LowerCasePipe、CurrencyPipe 和 PercentPipe。 它们全都可以直接用在任何模板中。

对管道进行参数化



链式管道­­­­

表达式从左至右求值，date先通过Pipe1处理后将处理后的数据作为输入数据再进行Pipe2的处理，以此类推，直到所有管道都已执行完毕。

这种链式调用的方式可以展示更丰富更复杂的数据。

管道与变更检测

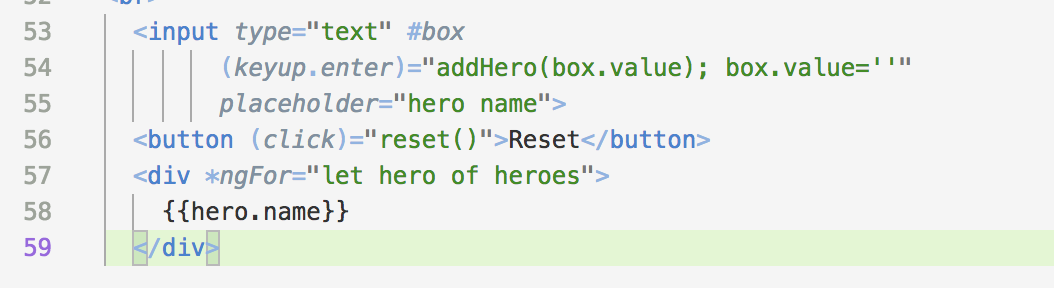
Angular 通过变更检测过程来查找绑定值的更改，并在每一次异步事件之后运行：每次按键、鼠标移动、定时器以及服务器的响应。 这可能会让变更检测显得很昂贵，但是 Angular 会尽可能降低变更检测的成本。

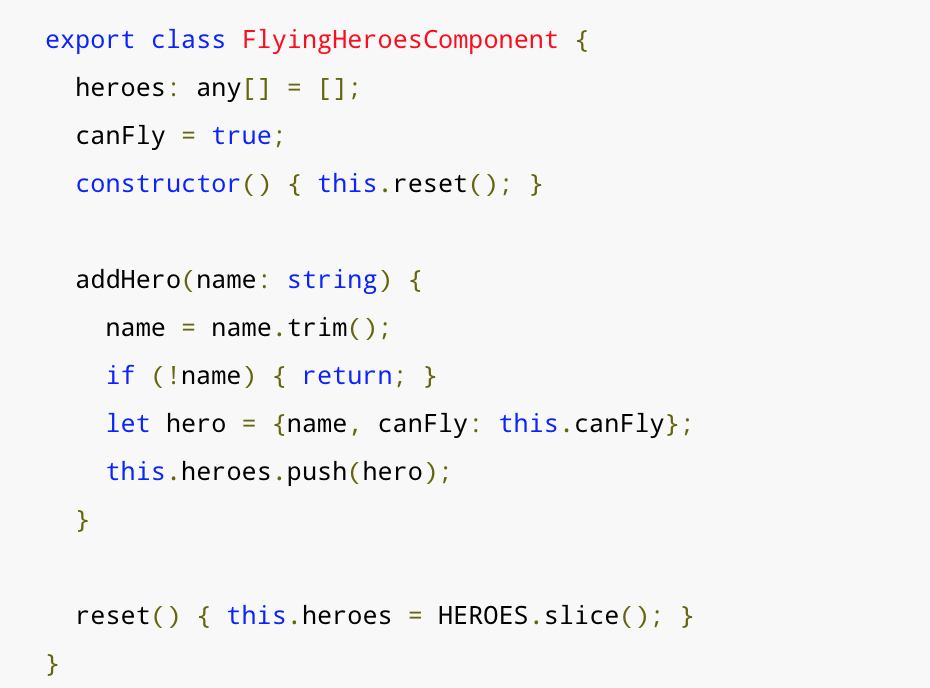
当使用管道时，Angular 会选用一种更简单、更快速的变更检测算法。

无管道

在下列例子中，组件使用默认的、激进(昂贵)的变更检测策略来检测和更新 heroes 数组中的每个英雄。和模板相伴的组件类可以提供英雄数组，能把新的英雄添加到数组中，还能重置英雄数组。

你可以添加新的英雄，加完之后，Angular 就会更新显示。 reset 按钮会把 heroes 替换成一个由原来的英雄组成的新数组，重置完之后，Angular 就会更新显示。 如果你提供了删除或修改英雄的能力，Angular 也会检测到那些更改，并更新显示。

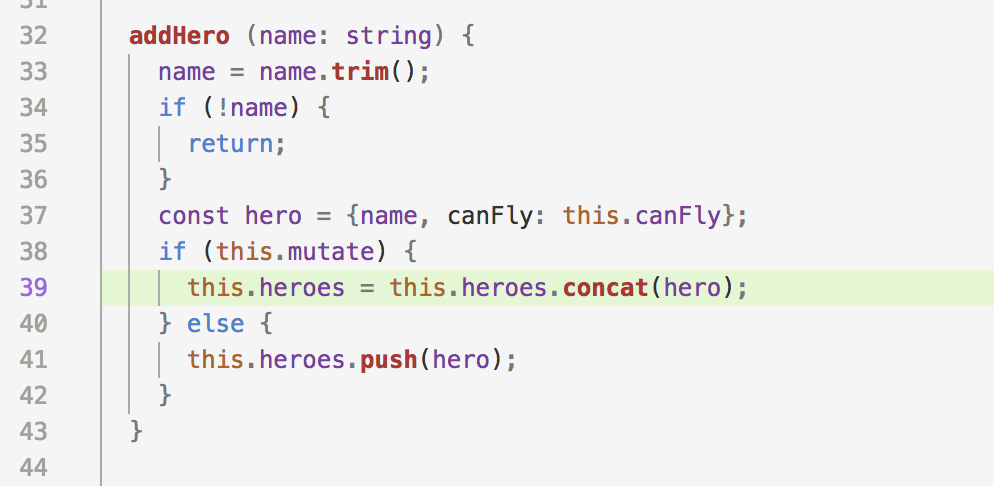




使用管道

往 \*ngFor 重复器中添加一个 flyingHeroes 管道，过滤出有个性的人物。







**！！！**添加的每个英雄都是会飞行的英雄，但是没有一个被显示出来。

虽然你没有得到期望的行为，但 Angular 也没有出错。 这里只是用了另一种变更检测算法 —— 它会忽略对列表及其子项所做的任何更改。

往 heroes 数组中添加一个新的人物时，这个数组的引用并没有改变。它还是那个数组。而引用却是 Angular 所关心的一切。 从 Angular 的角度来看，这是同一个数组，没有变化，也就不需要更新显示。

这是因为Angular优化了管道的监测机制——

**它会忽略对象内值的变化，只会监测指向对象的指针是否发生改变**

要修复它，就要创建一个新数组，把这个人物追加进去，并把它赋给 heroes。 这次，Angular 检测到数组的引用变化了。它执行了这个管道，并使用这个新数组更新显示，这次它就包括新的有个人性人了。

如果你修改了这个数组，没有管道被执行，也没有显示被更新。 如果你替换了这个数组，管道就会被执行，显示也更新了。这个例子用检查框和其它显示内容扩展了原有代码，实现这些效果。

直接替换这个数组是通知 Angular 更新显示的一种高效方式。

该什么时候替换这个数组呢？当数据变化的时候。 在这个例子中，这是一个简单的规则，因为这里修改数据的唯一途径就是添加新人物。

更多情况下，你不知道什么时候数据变化了，尤其是在那些有很多种途径改动数据的程序中 —— 可能在程序中很远的地方。 组件就是一个通常无法知道那些改动的例子。此外，它会导致削足适履 —— 扭曲组件的设计来适应管道。 要尽可能保持组件类独立于 HTML。组件不应该关心管道的存在。

这种管道称为纯管道，虽然纯管道优化了性能，但有时无法满足需求，就像上面的例子那样。

Angular管道有两种变化监测机制，一种是纯管道（默认），另一种即非纯管道。

纯管道 (Pure Pipe)

使用纯管道时，只有监测到输入值发生纯变更才会调用纯管道的transform方法来转换数据。

纯变更是指对基本数据类型输入值的变更，或对对象的引用放生改变。

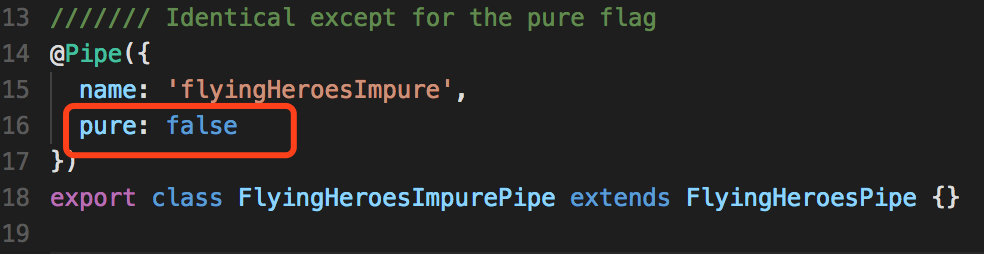
Angular 只有在它检测到输入值发生了纯变更时才会执行纯管道。 纯变更是指对原始类型值(String、Number、Boolean、Symbol)的更改， 或者对对象引用(Date、Array、Function、Object)的更改。

Angular 会忽略(复合)对象内部的更改。 如果你更改了输入日期(Date)中的月份、往一个输入数组(Array)中添加新值或者更新了一个输入对象(Object)的属性，Angular 都不会调用纯管道。

这可能看起来是一种限制，但它保证了速度。 对象引用的检查是非常快的(比递归的深检查要快得多)，所以 Angular 可以快速的决定是否应该跳过管道执行和视图更新。

因此，如果要和变更检测策略打交道，就会更喜欢用纯管道。

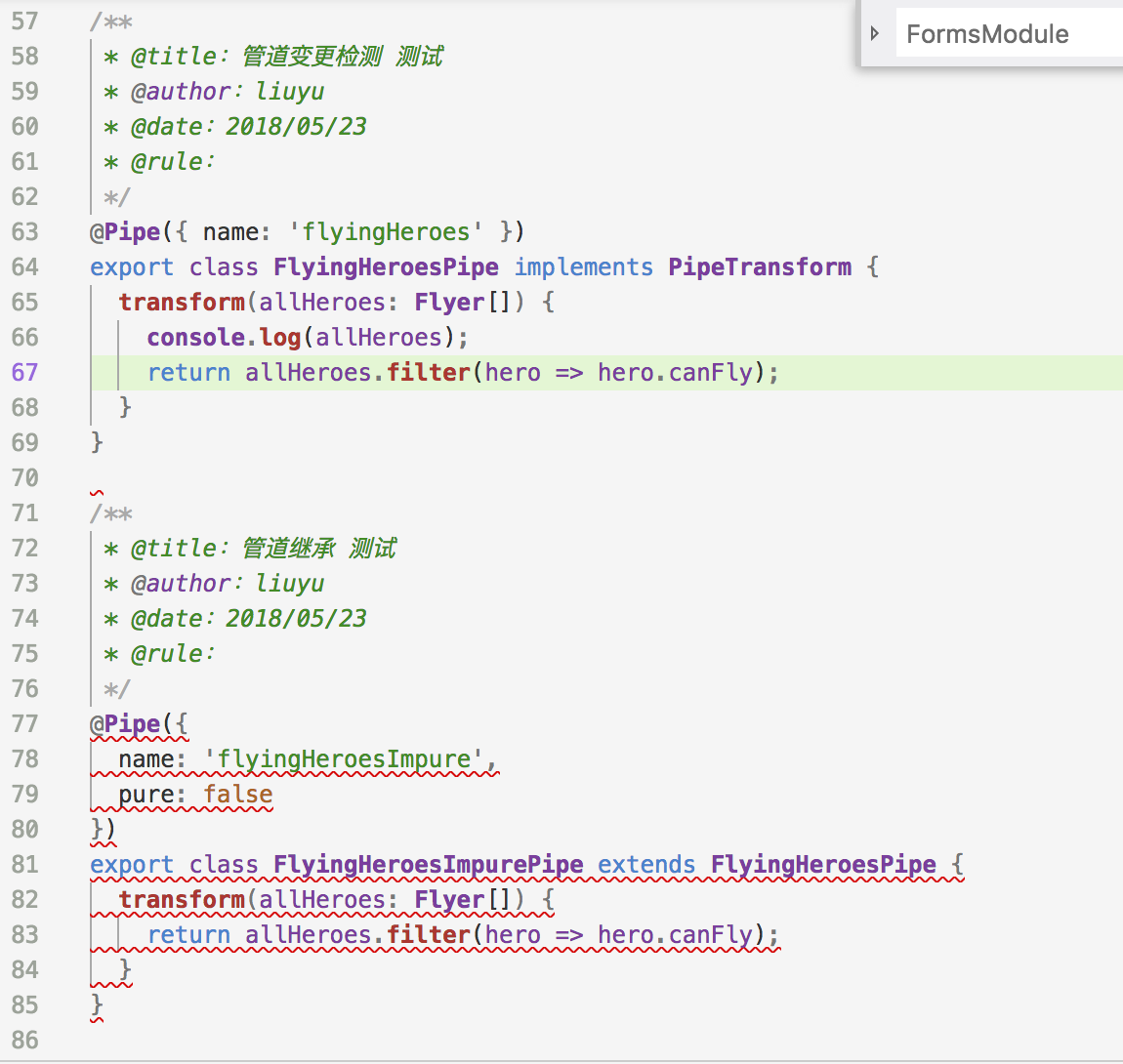
非纯管道 (Impure Pipe)



Angular 会在每个组件的变更检测周期中执行非纯管道。 非纯管道可能会被调用很多次，和每个按键或每次鼠标移动一样频繁。

要在脑子里绷着这根弦，必须小心翼翼的实现非纯管道。 一个昂贵、迟钝的管道将摧毁用户体验。

管道继承



**可以不注入FlyingHeroesPipe，只注入FlyingHeroesImpurePipe就可以使用！**

参考资料：

1. 推荐两个论坛网站：

1）<https://segmentfault.com/>

2）<https://stackoverflow.com/questions/>

2. 官方文档：<https://www.angular.cn/guide/dynamic-component-loader>

3. angular4动态创建组件的方案：<https://www.jianshu.com/p/10feab2c3102>

4. 掌握angular2的ngModule：<https://segmentfault.com/a/1190000007187393>

5. angular2 component: <https://www.cnblogs.com/keatkeat/p/5816368.html>

6. angular2-Interface：<http://www.coin163.com/it/758134614782080785>

7. ts文档-声明合并：<https://www.tslang.cn/docs/handbook/declaration-merging.html>

8. ts文档-高级类型：<https://www.tslang.cn/docs/handbook/advanced-types.html>

9. websocket教程：<http://www.ruanyifeng.com/blog/2017/05/websocket.html>

10. input绑定ngModel：<https://cloud.tencent.com/developer/ask/80559>

11. angular2管道：<https://blog.csdn.net/qq451354/article/details/58588252?locationNum=15&fps=1>

12. 内存泄漏、内存溢出：<https://blog.csdn.net/ruiruihahaha/article/details/70270574>

13. ngIf指令：<https://segmentfault.com/a/1190000009499160>