<https://segmentfault.com/a/1190000008626070>

Angular 2 的指令分为以下三种：

* 组件(Component directive)：用于构建UI组件，继承于 Directive 类
* 属性指令(Attribute directive): 用于改变组件的外观或行为
* 结构指令(Structural directive): 用于动态添加或删除DOM元素来改变DOM布局

**Angular 2 组件**

组件示例：

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

@Component({

selector: 'my-app', // 定义组件在HTML代码中匹配的标签

template: `<h1>Hello {{name}}</h1>`, // 指定组件关联的内联模板

})

export class AppComponent {

name = 'Angular';

}

**Angular 2 内置属性指令**

**1.ngStyle指令： 用于设定给定 DOM 元素的 style 属性**

绑定常量

<div [ngStyle]="{'background-color': 'green'}"></div>

绑定表达式

<div [ngStyle]="{'background-color': person.country === 'UK' ? 'green' : 'red'}">

具体示例：

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

@Component({

selector: 'ngstyle-example',

template: `<h4>NgStyle</h4>

<ul \*ngFor="let person of people">

<li [ngStyle]="{'color': getColor(person.country)}">

{{ person.name }} ({{person.country}})

</li>

</ul>

`

})

export class NgStyleExampleComponent {

getColor(country: string) {

switch (country) {

case 'CN':

return 'red';

case 'USA':

return 'blue';

case 'UK':

return 'green';

}

}

people: any[] = [

{

name: "Semlinker",

country: 'CN'

},

{

name: "Donald John Trump",

country: 'USA'

},

{

name: "Daniel Manson",

country: 'UK'

}

];

}

上面的例子，除了使用 ngStyle 指令，我们还可以使用 [style.<property>] 的语法：

<ul \*ngFor="let person of people">

<li [style.color]="getColor(person.country)">

{{ person.name }} ({{person.country}})

</li>

</ul>

**2.ngClass指令：用于动态的设定 DOM 元素的 CSS class**

绑定常量

<div [ngClass]="{'text-success': true }"></div>

绑定表达式

<div [ngClass]="{'text-success': person.country === 'CN'}"></div>

具体示例：

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

@Component({

selector: 'ngclass-example',

template: `

<style>

.text-success {

color: green

}

.text-primary {

color: red

}

.text-secondary {

color: blue

}

</style>

<h4>NgClass</h4>

<ul \*ngFor="let person of people">

<li [ngClass]="{

'text-success': person.country === 'UK',

'text-primary': person.country === 'CN',

'text-secondary': person.country === 'USA'

}"> {{ person.name }} ({{person.country}})

</li>

</ul>

`,

})

export class NgClassExampleComponent {

people: any[] = [

{

name: "Semlinker",

country: 'CN'

},

{

name: "Donald John Trump",

country: 'USA'

},

{

name: "Daniel Manson",

country: 'UK'

}

];

}

**Angular 2 内置结构指令**

1.ngIf指令：根据表达式的值，显示或移除元素

<div \*ngIf="person.country === 'CN'">{{ person.name }} ({{person.country}})</div>

2.ngFor指令：使用可迭代的每个项作为模板的上下文来重复模板，类似于 Ng 1.x 中的 ng-repeat 指令

<div \*ngFor="let person of people">{{person.name}}</div>

3.ngSwitch指令：它包括两个指令，一个属性指令和一个结构指令。它类似于 JavaScript 中的 switch 语句

<ul [ngSwitch]='person.country'>

<li \*ngSwitchCase="'UK'" class='text-success'>

{{ person.name }} ({{person.country}})

</li>

<li \*ngSwitchCase="'USA'" class='text-secondary'>

{{ person.name }} ({{person.country}})

</li>

<li \*ngSwitchDefault class='text-primary'>

{{ person.name }} ({{person.country}})

</li>

</ul>

通过上面的例子，可以看出结构指令和属性指令的区别。结构指令是以 \* 作为前缀，这个星号其实是一个语法糖。它是 ngIf 和 ngFor 语法的一种简写形式。Angular 引擎在解析时会自动转换成 <template> 标准语法。

**Angular 2 内置结构指令标准形式**

1.ngIf指令：

<template [ngIf]='condition'>

<p>I am the content to show</p>

</template>

2.ngFor指令：

<template ngFor [ngForOf]="people" let-person>

<div> {{ person.name }} ({{person.country}}) </div>

</template>

3.ngSwitch指令：

<ul [ngSwitch]='person.country'>

<template [ngSwitchCase]="'UK'">

<li class='text-success'>

{{ person.name }} ({{person.country}})

</li>

</template>

<template [ngSwitchCase]="'USA'">

<li class='text-secondary'>

{{ person.name }} ({{person.country}})

</li>

</template>

<template [ngSwitchDefault]>

<li class='text-primary'>

{{ person.name }} ({{person.country}})

</li>

</template>

</ul>

**Angular 2 内置结构指令定义**

1.ngIf指令定义：

@Directive({selector: '[ngIf]'})

export class NgIf {}

2.ngFor指令定义：

@Directive({selector: '[ngFor][ngForOf]'})

export class NgForOf<T> implements DoCheck, OnChanges {}

3.ngSwitch指令定义：

@Directive({selector: '[ngSwitch]'})

export class NgSwitch {}

@Directive({selector: '[ngSwitchCase]'})

export class NgSwitchCase implements DoCheck {}

@Directive({selector: '[ngSwitchDefault]'})

export class NgSwitchDefault {}

**自定义属性指令**

指令功能描述：该指令用于在用户点击宿主元素时，根据输入的背景颜色，更新宿主元素的背景颜色。宿主元素的默认颜色是黄色。

**1.指令实现**

import {Directive, Input, ElementRef, Renderer, HostListener} from "@angular/core";

@Directive({

selector: '[exeBackground]'

})

export class ExeBackgroundDirective {

private \_defaultColor = 'yellow';

@Input('exeBackground')

backgroundColor: string; // 输入属性，用于设置元素的背景颜色

constructor(private elementRef: ElementRef, private renderer: Renderer) {

this.setStyle(this.\_defaultColor);

}

@HostListener('click')

onClick() { // 监听宿主元素的点击事件，设置元素背景色

this.setStyle(this.backgroundColor || this.\_defaultColor);

}

private setStyle(color: string) { // 调用renderer对象提供的API设置元素的背景颜色

this.renderer.setElementStyle(this.elementRef.nativeElement,

'backgroundColor', color);

}

}

**2.指令应用：**

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

@Component({

selector: 'my-app',

template: `<h1 [exeBackground]="'red'">Hello {{name}}</h1>`,

})

export class AppComponent {

name = 'Angular';

}

**自定义结构指令**

指令功能描述：该指令实现 ngIf 指令相反的效果，当指令的输入条件为 Falsy 值时，显示DOM元素。

**1.指令实现**

@Directive({

selector: '[exeUnless]'

})

export class UnlessDirective {

@Input('exeUnless')

set condition(newCondition: boolean) {

if (!newCondition) { // 创建模板对应的内嵌视图

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear();

}

}

constructor(private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) {

}

}

**2.指令应用**

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

@Component({

selector: 'my-app',

template: `<h1 [exeBackground]="'red'" \*exeUnless="condition">Hello {{name}}</h1>`,

})

export class AppComponent {

name = 'Angular';

condition: boolean = false;

}

**我有话说**

1.自定义属性指令中的 ElementRef 与 Renderer 的作用

为了能够支持跨平台，Angular 2 通过抽象层封装了不同平台的差异，统一了 API 接口。如定义了抽象类 Renderer 、抽象类 RootRenderer 等。此外还定义了以下引用类型：ElementRef、TemplateRef、ViewRef 、ComponentRef 和 ViewContainerRef 等。

详细内容请参考 - [Angular 2 ElementRef](https://segmentfault.com/a/1190000008653690)

2.自定义结构指令中的 TemplateRef 与 ViewContainerRef 的作用

TemplateRef：用于表示内嵌的 template 模板元素，通过 TemplateRef 实例，我们可以方便创建内嵌视图(Embedded Views)，且可以轻松地访问到通过 ElementRef 封装后的 nativeElement。需要注意的是组件视图中的 template 模板元素，经过渲染后会被替换成 comment 元素。

ViewContainerRef：用于表示一个视图容器，可添加一个或多个视图。通ViewContainerRef 实例，我们可以基于 TemplateRef 实例创建内嵌视图，并能指定内嵌视图的插入位置，也可以方便对视图容器中已有的视图进行管理。简而言之，ViewContainerRef 的主要作用是创建和管理内嵌视图或组件视图。

详细内容请参考 - [Angular 2 TemplateRef & ViewContainerRef](https://segmentfault.com/a/1190000008672478)

3.Angular 2 中指令与组件的关系

组件继承于指令，并扩展了与 UI 视图相关的属性，如 template、styles、animations、encapsulation 等。

详细内容请参考 - [Angular 2 Directive Lifecycle](https://segmentfault.com/a/1190000008716308)

**总结**

本文主要介绍了 Angular 2 中的指令，通过具体示例介绍了 Angular 2 常见内建指令的使用方式和区别。最终，我们通过自定义属性指令和自定义结构指令两个示例，展示了如何开发自定义指令。

<http://ghmagical.com/article/page/id/RgIvn9Pai3ks>

# Angular4 开发实战：(6) 创建指令(Directive)

阅读 2845 [评论 0](http://ghmagical.com/article/page/id/RgIvn9Pai3ks#comment) 喜欢 0

指令(Directive)也是我们在开发中常用的。

**创建指令** 我们创建一个button指令：

ng g directive button

对于指令，会生成两个文件：

button.directive.ts

button.directive.spec.ts

基础模板如下：

// button.directive.ts

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

@Directive({

  selector: '[appButton]'

})

export class ButtonDirective {

  constructor() { }

}

对于指令，selector一般使用**方括号**属性方式，如上。

使用方法：

<button appButton>指令按钮</button>

指令有两种：

* 属性型指令：修改宿主元素的样式或行为等
* 结构型指令，比如`\*ngIf`，`\*ngFor`：修改DOM结构。

**属性型指令** 上面的button指令还没有什么作用，下面让我们来修改一下，先往style.css中添加下列样式：

button.btn {

  padding: 10px;

  background: blue;

  color: #fff;

  border-radius: 4px;

  border: none;

}

修改button.directive.ts：

import { Directive, ElementRef, AfterViewInit, Renderer2 } from '@angular/core';

@Directive({

  selector: '[appButton]'

})

export class ButtonDirective implements AfterViewInit{

  constructor(private er: ElementRef, private renderer2: Renderer2) { }

  ngAfterViewInit() {

    this.renderer2.addClass(this.er.nativeElement, 'btn');

  }

}

在上面的代码中，我们将ElementRef和Renderer2注入构造函数，前者会得到添加了appButton属性的元素，后者是Angular提供的DOM API操作。 注：ElementRef和Renderer2后续会详解。 然后在demo-directive.component.html中添加：

<button appButton>指令按钮/button>

当你打开网页时，会发现此button添加上了btn样式类(背景色变蓝，字体颜色变白色)。 当然，还有一些快捷的方式来给使用了指令的宿主元素添加属性和事件： **(1) @HostBinding()** 再来修改一下button.directive.ts：

export class ButtonDirective {

  @HostBinding('style.font-size') fontSize = '20px';

  ...

}

当你添加了上面的代码时，你会发现demo-directive.component.html中的按钮字体变大了。 注：@HostBinding()里是的属性和直接属性绑定是一样的效果，主要给宿主元素设置属性。 **(2) @HostListener** 我们还可以为宿主元素绑定事件：

// button.directive.ts

@HostListener('click') onClick() {

  alert('你点了我!');

}

添加如上代码后，你可以试试点击按钮。 注：@HostListener()里是事件名称，后面跟着一个监听函数，名称可以是任意合法的字符。 有些时候我们需要给这个指令传递一些值，如何实现呢？ 还记得前面《组件通讯(@Input和@Output)》一章中我们讲过@Input()可以设置别名。 修改button.directive.ts：

export class ButtonDirective implements AfterViewInit {

  ...

  @Input('appButton') name: string; // 定义别名

  constructor(private er: ElementRef, private renderer2: Renderer2) { }

  ngAfterViewInit() {

    ...

    if (this.name) {

      const text = this.renderer2.createText(this.name);

      this.renderer2.appendChild(this.er.nativeElement, text);

    }

  }

}

通过@Input()定义与指令同名的输入属性，我们就可以给指令传递参数了：

// demo-directive.component.html

<button appButton="额外名称">指令按钮</button>

**结构型指令** 最常用的内置结构型指令有ngIf, ngFor, ngSwitch。 **(1) ngIf** 在demo-directive.component.html中添加：

<div>

  <button (click)="isShow = !isShow">点击试试</button>

  <div \*ngIf="isShow">ngIf结构型指令</div>

</div>

随着点击按钮，isShow属性值会在true和false之间切换，而div也会跟着显示或隐藏，其实是插入或移除。 而在Angular4中，新增else效果(使用模板变量#name)：

<button (click)="isShowElse = !isShowElse">点击试试ngIf else</button>

<div \*ngIf="isShowElse else next">ngIf结构型指令中的else</div>

<ng-template #next>else效果</ng-template>

有些时候我们还可以这样：

<ng-container \*ngIf="isShow"></ng-container>

ng-container元素会将其里面的内容插入或移除，但ng-container元素不会出现在页面里。 **（2） ngFor** ngFor一般用来显示数据列表：

// demo-directive.component.html

<li \*ngFor="let book of books">{{book}}</li>

// demo-directive.component.ts

this.books = ['HTML', 'Javascript'];

带索引：

<h4>带索引</h4>

<ul>

  <li \*ngFor="let book of books; index as i">{{i + ':' + book}}</li>

</ul>

**(3) ngSwitch** ngSwitch与JavaScript中的switch类似：

<div [ngSwitch]="animal">

  <div \*ngSwitchCase="'dog'">汪</div>

  <div \*ngSwitchCase="'cat'">喵</div>

  <div \*ngSwitchDefault>哼</div>

</div>

**自定义结构型指令** 我们可以创建自定义结构型指令my-if.directive.ts：

import { Directive, Input, ViewContainerRef, TemplateRef } from '@angular/core';

@Directive({

  selector: '[appMyIf]'

})

export class MyIfDirective {

  constructor(private templateRef: TemplateRef<any>,

      private viewContainer: ViewContainerRef) { }

  @Input() set appMyIf(condition: boolean) {

    if (condition) {

      this.viewContainer.createEmbeddedView(this.templateRef);

    } else {

      this.viewContainer.clear();

    }

  }

}

在上面的代码中，我们使用@Input()属性装饰器的setter方法来定义appMyIf方法，里面的condition参数就是其属性值。 使用demo-directive.component.html：

 <button (click)="isMyShow = !isMyShow">点击试试</button>

<div \*appMyIf="isMyShow">自定义结构型指令appMyIf</div>

在上面的代码中，当isMyShow变为true时，底部的内容显示出来；当为false时，底部内容消失了。 如发现任何问题或有好的建议，欢迎在下方评论留言。

著作权归作者所有。  
商业转载请联系作者获得授权,非商业转载请注明出处。  
原文: <http://ghmagical.com/article/page/id/RgIvn9Pai3ks> © [ghmagical.com](http://www.w3cplus.com)

import { Directive, Input, ElementRef, Renderer, HostListener, OnInit } from '@angular/core';

@Directive({

selector: '.GBC' //以class=”GBC”

})

export class BgDirective {

//@Input("GBC") public color:string;

public color:string = "red";

constructor(public elRef: ElementRef, public render: Renderer) {

}

ngOnInit() {

this.elRef.nativeElement.style.backgroundColor = this.color;

}

}

<p class="GBC">Highlight me!</p>



import { Directive, Input, ElementRef, Renderer, HostListener, OnInit } from '@angular/core';

@Directive({

selector: '[gBC]' // 用属性

})

export class BgDirective {

//@Input("GBC") public color:string;

public color:string = "red";

constructor(public elRef: ElementRef, public render: Renderer) {

}

ngOnInit() {

this.elRef.nativeElement.style.backgroundColor = this.color;

}

}

<p gBC>Highlight me!</p>



import { Directive, Input, ElementRef, Renderer, HostListener, OnInit } from '@angular/core';

@Directive({

selector: '[gBC]'

})

export class BgDirective {

@Input("gBC") public color:string;

constructor(public elRef: ElementRef, public render: Renderer) {

}

ngOnInit() {

this.elRef.nativeElement.style.backgroundColor = this.color;

}

}

<p [gBC]="'orange'">Highlight me!</p>



可以添加事件

import { Directive, Input, ElementRef, Renderer, HostListener, OnInit } from '@angular/core';

@Directive({

selector: '[gBC]'

})

export class BgDirective {

@Input("gBC") public color:string;

constructor(public elRef: ElementRef, public render: Renderer) {

}

ngOnInit() {

this.elRef.nativeElement.style.backgroundColor = this.color;

}

@HostListener("mouseenter")

onmm (){

this.elRef.nativeElement.style.backgroundColor = "blue";

}

@HostListener("mouseleave")

mmll(){

this.elRef.nativeElement.style.backgroundColor = "grey";

}

@HostListener("click")

clickme() {

this.elRef.nativeElement.style.backgroundColor = "yellow";

}

}

mouseleave



注意：

Directive 只能在本模块里使用，不可以传递到所包括的子模块里

import { BrowserModule } from '@angular/platform-browser';

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

import { FormsModule } from "@angular/forms";

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { MyrouterRoutingModule } from "./myrouter/myrouter-routing.module";

import { HelpModule } from './help/help.module';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MyrouterRoutingModule,

HelpModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

如果放在上面的模块里， EmployeeComponent， LoginComponent， 仍然不可用， 出错

只能放在这里， 才可以让两个Component 使用

----------------------------------------------------------------

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

import { BrowserModule } from '@angular/platform-browser';

import { Routes, RouterModule } from '@angular/router';

import { EmployeeComponent } from "./employee/employee.component";

import { LoginComponent } from "./login/login.component";

import { HttpClientModule } from '@angular/common/http';

import { FormsModule } from "@angular/forms";

import { BgDirective } from './highlight.directive';

const routes: Routes = [

{ path: "employee", component: EmployeeComponent,

loadChildren: './hisrouter/hisrouter-routing.module#HisrouterRoutingModule'

},

{ path: "login", component: LoginComponent }

];

@NgModule({

declarations: [EmployeeComponent, LoginComponent, BgDirective],

imports: [RouterModule.forRoot(routes), HttpClientModule, FormsModule, BrowserModule],

providers: [],

exports: [RouterModule, EmployeeComponent, LoginComponent]

})

export class AppRoutingModule { }

<p>employee works!</p>

Employee List:<br>

<ul>

<li [gBC]="'orange'" \*ngFor="let e of emps; index as i">

Summary: <a [routerLink]="['/employee/summary', e.Id]" [queryParams]="{sname: wmname, dname:wmname}" routerLinkActive="router-link-active">{{i}} - {{e.Id}} - {{e.Name}} - {{e.Age}}</a><br>

Depart: <a [routerLink]="['/employee/depart', e.Id]" [queryParams]="" routerLinkActive="router-link-active">{{i}} - {{e.Id}} - {{e.Name}} - {{e.Age}}</a><br>

</li>

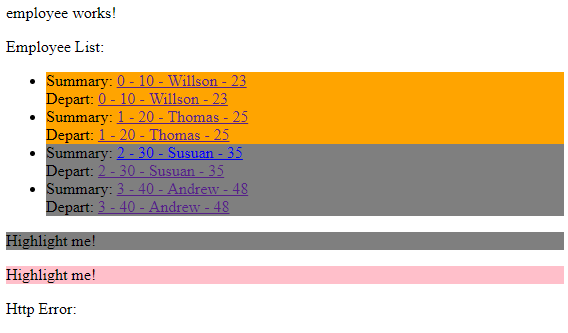
</ul>

<p [gBC]="'orange'">Highlight me!</p>

<p [gBC]="'pink'">Highlight me!</p>

Http Error: {{this.err|json}}

<router-outlet></router-outlet>



Login Employee List:<br>

<ul>

<li \*ngFor="let e of emps; index as i">

Summary: <a [routerLink]="['/employee/summary', e.Id]" [queryParams]="{sname: wmname, dname:wmname}" routerLinkActive="router-link-active">{{i}} - {{e.Id}} - {{e.Name}} - {{e.Age}}</a><br>

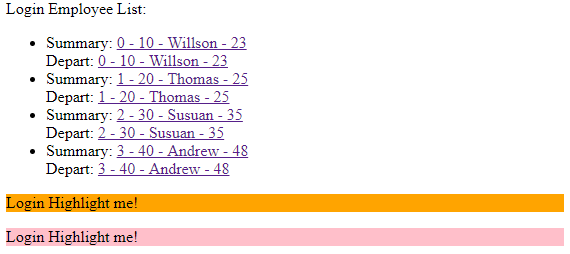
Depart: <a [routerLink]="['/employee/depart', e.Id]" [queryParams]="" routerLinkActive="router-link-active">{{i}} - {{e.Id}} - {{e.Name}} - {{e.Age}}</a><br>

</li>

</ul>

<p [gBC]="'orange'">Login Highlight me!</p>

<p [gBC]="'pink'">Login Highlight me!</p>



可以直接使用 属性

import { Directive, Input, ElementRef, Renderer, HostListener, OnInit } from '@angular/core';

@Directive({

selector: '[gBC]'

})

export class BgDirective {

@Input("gBC") public color:string;

@Input() public **defColor**:string;

constructor(public elRef: ElementRef, public render: Renderer) {

}

ngOnInit() {

this.elRef.nativeElement.style.backgroundColor = this.defColor;

}

@HostListener("mouseenter")

onmm (){

this.elRef.nativeElement.style.backgroundColor = "blue";

}

@HostListener("mouseleave")

mmll(){

this.elRef.nativeElement.style.backgroundColor = "grey";

}

@HostListener("click")

clickme() {

this.elRef.nativeElement.style.backgroundColor = "yellow";

}

}

<p [gBC]="'orange'" defColor="green">Highlight me!</p>

<p [gBC]="'pink'" defColor="green">Highlight me!</p>



结构指令： structural directive

自定义结构指令主要操作 TemplateRef 与 ViewContainerRef ：

TemplateRef：用于表示内嵌的 template 模板元素，通过 TemplateRef 实例，我们可以方便创建内嵌视图(Embedded Views)，且可以轻松地访问到通过 ElementRef 封装后的 nativeElement。需要注意的是组件视图中的 template 模板元素，经过渲染后会被替换成 comment 元素。

ViewContainerRef：用于表示一个视图容器，可添加一个或多个视图。通ViewContainerRef 实例，我们可以基于 TemplateRef 实例创建内嵌视图，并能指定内嵌视图的插入位置，也可以方便对视图容器中已有的视图进行管理。简而言之，ViewContainerRef 的主要作用是创建和管理内嵌视图或组件视图。

Angular 中指令与组件的关系

组件继承于指令，并扩展了与 UI 视图相关的属性，如 template、styles、animations、encapsulation 等。

指令专注于添加 Dom 的behavor 到已存在的 Dom 元素上。

组件则是生产我们需要重复使用的 HTML View



## Custom Directives in Angular 7

* [Adesh](https://www.zeptobook.com/author/admin/)
* November 24, 2018
* [Angular](https://www.zeptobook.com/category/angular/) / [Angular 7.0](https://www.zeptobook.com/category/angular-7-0/)
* [5 Comments](https://www.zeptobook.com/custom-directives-in-angular-7/#comments)

## Getting started with Directives

Directives are the most powerful feature of any Angular applications. In fact, most commonly used feature, which is component, is itself a directive. There are basically three types of directives:

* Component directive – a directive with templates.
* Attribute directive –  a directive which manipulate DOM by changing behaviour and appearance
* Structural directive – a directive which create and destroy DOM element

Component directive is what we are using it in our day to day programming since Angular 2. So, I am going to skip discussing about Component directive. In this post, we will briefly discuss the rest of the two directives: Attribute and Structural directives.

### Structural Directives

Structural directives are used to add, remove or manipulate the elements from the DOM. It is easy to identify structural directives. Structural directives are prefixed by asterisk (\*) with the directive name. Example of structural directives are : NgIf, NgSwitch & NgFor. These are inbuilt structural directives.

### Attribute Directives

Attribute directives are used to change the behavior and appearance of an DOM element. As the name suggests, they are applied as an attribute on the DOM element.



|  |  |
| --- | --- |
| 1  2 | <p [style.color]="'red'">attribute directive</p>  <p [hidden]="shouldHide">attribute directive</p> |

### Custom Directive

Let’s create our first custom directive. To generate the directive, run this command in the command terminal.



|  |  |
| --- | --- |
| 1 | ng generate directive color |

This command will generate a color.directive.ts file in the app folder.

#### color.directive.ts



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import { Directive } from '@angular/core';    @Directive({    selector: '[appColor]'  })  export class ColorDirective {      constructor() { }    } |

In the first line, Angular CLI imported Directive from @angular/core package. This will provide the @Directive decorator.

The name of our directive is appColor in the [] selector. Angular will look for this attribute on the html element, and applies the directive logic to that element. This directive has a class name ColorDirective.

This command will also add an entry in our app.module.ts file as well. See the line 7 and 13 in app.module.ts file.

#### app.module.ts



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';  import { StructuralDirectiveComponent } from './structural-directive/structural-directive.component';  import { ColorDirective } from './color.directive';    @NgModule({    declarations: [      AppComponent,      StructuralDirectiveComponent,      ColorDirective    ],    imports: [      BrowserModule,      AppRoutingModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

Let’s put some code logic in our appColor directive. We are going to make the background color of an element Red. So, our directive code looks like this:

#### color.directive.ts



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import { Directive, ElementRef } from '@angular/core';    @Directive({    selector: '[appColor]'  })  export class ColorDirective {      constructor(el: ElementRef) {      el.nativeElement.style.backgroundColor = 'red';    }  } |

If you notice in line first, we also imported ElementRef



|  |  |
| --- | --- |
| 1 | import { Directive, ElementRef } from '@angular/core'; |

and injected this in our directive class constructor in order to access the element, on which this directive is applied.



|  |  |
| --- | --- |
| 1 | constructor(el: ElementRef) |

After injecting ElementRef in our class constructor, we can now access the element here. So, let’s access the element and set its background color to Red.



|  |  |
| --- | --- |
| 1  2  3 | constructor(el: ElementRef) {      el.nativeElement.style.backgroundColor = 'red';    } |

This is how, we can create our basic directive. But how can we use it on any element?

To apply this directive, we have added appColor to <p></p> element.



|  |  |
| --- | --- |
| 1 | <p appColor>Show me my color</p> |

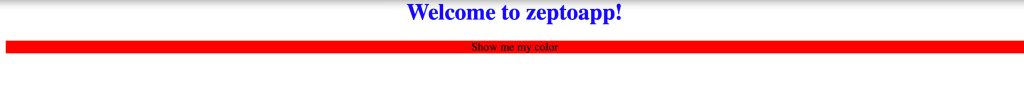
See the complete code of app.component.html file.

#### app.component.html



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <!--The content below is only a placeholder and can be replaced.-->  <div style="text-align:center">    <h1>      Welcome to {{ title }}!    </h1>  <p appColor>Show me my color</p>  <router-outlet></router-outlet>  </div> |

Open the browse and see the page.



Here, we can see, our directive changed the red background color of the text.

So, in this way, we have modified the DOM through our custom directive. We have changed the color of **p element**through angular attribute directive.

### Passing value to Custom Directive

In the above directive, we have hard coded the color code Red. It would be great, if we can make it dynamic, i.e, our directive can accept any color name from the element and render the background in same color. In order to make our custom directive dynamic, let’s tweak our code again.

#### color.directive.ts



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import { Directive, ElementRef, Input, OnInit } from '@angular/core';    @Directive({    selector: '[appColor]'  })  export class ColorDirective implements OnInit {    @Input() appColor: string;      constructor(private el: ElementRef) { }      ngOnInit(){      this.el.nativeElement.style.backgroundColor = this.appColor;     }  } |

In the line first, we have further imported OnInit from @angular/core.



|  |  |
| --- | --- |
| 1 | import { Directive, ElementRef, Input, OnInit } from '@angular/core'; |

**OnInit**

A lifecycle hook that is called after Angular has initialized all data-bound properties of a directive. Define an ngOnInit() method to handle any additional initialization tasks.

After importing OnInit, we implemented this directive in our class in line 6.



|  |  |
| --- | --- |
| 1 | export class ColorDirective implements OnInit |

Let’s create an input property in our directive named same as our directive name.



|  |  |
| --- | --- |
| 1 | @Input() appColor: string; |

Then implements the ngOnInit() method. In this method, we wrote the code of setting the background color of an element to the color, passed through element to our directive.



|  |  |
| --- | --- |
| 1  2  3 | ngOnInit(){      this.el.nativeElement.style.backgroundColor = this.appColor;    } |

Now, let’s see the code in our html template file, how to use this directive now.

#### app.component.html



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <!--The content below is only a placeholder and can be replaced.-->  <div style="text-align:center">    <h1>      Welcome to {{ title }}!    </h1>  <p [appColor]= "'green'">Show me Green Color</p>  <p [appColor]= "'red'">Show me Red Color</p>  <router-outlet></router-outlet>  </div> |

See these two lines in our template file. We have passed green & red color to <p></p> elements.



|  |  |
| --- | --- |
| 1  2 | <p [appColor]= "'green'">Show me Green Color</p>  <p [appColor]= "'red'">Show me Red Color</p> |

Note: String values must be passed with single quotes (‘) with double quotes (“).

Now, open the browser and see the page.



### Passing value to Custom Directive by Component Class

To make our above code more simpler, we can declare two model property in our component class app.component.ts file.

#### app.component.ts



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'zeptoapp';    greenColor = 'green';    redColor = 'red';  } |

Now, you can see in line 10 & 11, we have created two model properties: greenColor & redColor. Then, we passed it to our custom directive through html template file.

#### app.component.html



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <!--The content below is only a placeholder and can be replaced.-->  <div style="text-align:center">    <h1>      Welcome to {{ title }}!    </h1>  <p [appColor]= "greenColor">Show me Green Color</p>  <p [appColor]= "redColor">Show me Red Color</p>  <router-outlet></router-outlet>  </div> |

In this way, we can pass custom directive values from the component class as well.

### Summary

In this blog, we looked into different kinds of Angular Directives. And then we have created our custom directive. We also learned about how to pass value from html template file, and then from component class as well.

# [Angular 2 TemplateRef & ViewContainerRef](https://segmentfault.com/a/1190000008672478)

* [angularjs](https://segmentfault.com/t/angularjs/blogs)
* [angular2](https://segmentfault.com/t/angular2/blogs)
* [typescript](https://segmentfault.com/t/typescript/blogs)

19.2k 次阅读  ·  读完需要 32 分钟

阅读 Angular 6/RxJS 最新教程，请访问[前端修仙之路](http://www.semlinker.com/)

### TemplateRef

在介绍 TemplateRef 前，我们先来了解一下 HTML [模板元素](https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/template) - <template> 。模板元素是一种机制，允许包含加载页面时不渲染，但又可以随后通过 JavaScript 进行实例化的客户端内容。我们可以将模板视作为存储在页面上稍后使用的一小段内容。

在 HTML5 标准引入 template 模板元素之前，我们都是使用 <script> 标签进行客户端模板的定义，具体如下：

<script id="tpl-mock" type="text/template">

<span>I am span in mock template</span>

</script>

对于支持 HTML5 template 模板元素的浏览器，我们可以这样创建客户端模板：

<template id="tpl">

<span>I am span in template</span>

</template>

下面我们来看一下 HTML5 template 模板元素的使用示例：

<!DOCTYPE html>

<html lang="en">

<head><meta charset="UTF-8"> <title>HTML5 Template Element Demo</title></head>

<body>

<h4>HTML5 Template Element Demo</h4>

<!-- Template Container -->

<div class="tpl-container"></div>

<!-- Template -->

<template id="tpl">

<span>I am span in template</span>

</template>

<!-- Script -->

<script type="text/javascript">

(function renderTpl() {

if ('content' in document.createElement('template')) {

var tpl = document.querySelector('#tpl');

var tplContainer = document.querySelector('.tpl-container');

var tplNode = document.importNode(tpl.content, true);

tplContainer.appendChild(tplNode);

} else {

throw new Error("Current browser doesn't support template element");

}

})();

</script>

</body>

</html>

以上代码运行后，在浏览器中我们会看到以下内容：

HTML5 Template Element Demo

I am span in template

而当我们注释掉 tplContainer.appendChild(tplNode) 语句时，刷新浏览器后看到的是：

HTML5 Template Element Demo

这说明页面中 <template> 模板元素中的内容，如果没有进行处理对用户来说是不可见的。Angular 2 中，<template> 模板元素主要应用在结构指令中，此外在 [Angular 2 属性指令 vs 结构指令](https://segmentfault.com/a/1190000008626070) 文章中我们也介绍了 <template> 模板元素和自定义结构指令，接下来我们先来介绍一下本文中的第一个主角 - TemplateRef：

import {Component, TemplateRef, ViewChild, AfterViewInit} from '@angular/core';

@Component({

selector: 'my-app',

template: `

<h1>Welcome to Angular World</h1>

<template #tpl>

<span>I am span in template</span>

</template>

`,

})

export class AppComponent {

name: string = 'Semlinker';

@ViewChild('tpl')

tpl: TemplateRef<any>;

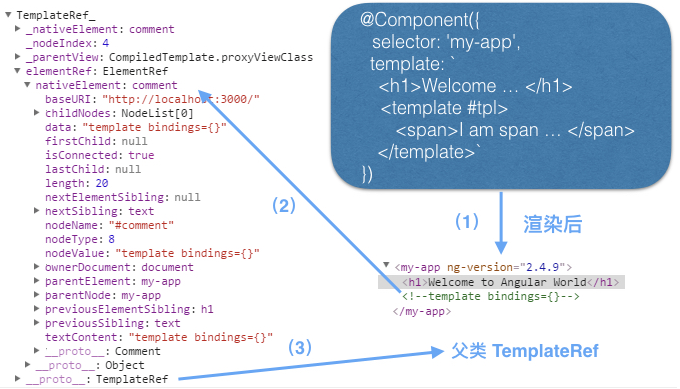
ngAfterViewInit() {

console.dir(this.tpl);

}

}

上述代码运行后的控制台的输出结果如下：



从上图中，我们发现 @Component template 中定义的 <template> 模板元素，渲染后被替换成 comment 元素，其内容为 "template bindings={}" 。此外我们通过 @ViewChild 获取的模板元素，是 TemplateRef\_ 类的实例，接下来我们来研究一下 TemplateRef\_ 类：

**TemplateRef\_**

// @angular/core/src/linker/template\_ref.d.ts

export declare class TemplateRef\_<C> extends TemplateRef<C> {

private \_parentView;

private \_nodeIndex;

private \_nativeElement;

constructor(\_parentView: AppView<any>, \_nodeIndex: number, \_nativeElement: any);

createEmbeddedView(context: C): EmbeddedViewRef<C>;

elementRef: ElementRef;

}

**TemplateRef**

// @angular/core/src/linker/template\_ref.d.ts

// 用于表示内嵌的template模板，能够用于创建内嵌视图(Embedded Views)

export declare abstract class TemplateRef<C> {

elementRef: ElementRef;

abstract createEmbeddedView(context: C): EmbeddedViewRef<C>;

}

(备注：抽象类与普通类的区别是抽象类有包含抽象方法，不能直接实例化抽象类，只能实例化该抽象类的子类)

我们已经知道 <template> 模板元素，渲染后被替换成 comment 元素，那么应该如何显示我们模板中定义的内容呢 ？我们注意到了 TemplateRef 抽象类中定义的 createEmbeddedView  
抽象方法，该方法的返回值是 EmbeddedViewRef 对象。那好我们马上来试一下：

import {Component, TemplateRef, ViewChild, AfterViewInit} from '@angular/core';

@Component({

selector: 'my-app',

template: `

<h1>Welcome to Angular World</h1>

<template #tpl>

<span>I am span in template</span>

</template>

`,

})

export class AppComponent {

name: string = 'Semlinker';

@ViewChild('tpl')

tpl: TemplateRef<any>;

ngAfterViewInit() {

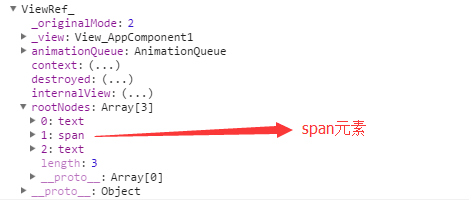
let embeddedView = this.tpl.createEmbeddedView(null);

console.dir(embeddedView);

}

}

上述代码运行后的控制台的输出结果如下：



从图中我们可以知道，当调用 createEmbeddedView 方法后返回了 ViewRef\_ 视图对象。该视图对象的 rootNodes 属性包含了 <template> 模板中的内容。在上面的例子中，我们知道了 TemplateRef 实例对象中的 elementRef 属性封装了我们的 comment 元素，那么我们可以通过 insertBefore 方法来创建我们模板中定义的内容。

import { Component, TemplateRef, ViewChild, AfterViewInit } from '@angular/core';

@Component({

selector: 'my-app',

template: `

<h1>Welcome to Angular World</h1>

<template #tpl>

<span>I am span in template {{name}}</span>

</template>

`,

})

export class AppComponent {

name: string = 'Semlinker';

@ViewChild('tpl')

tpl: TemplateRef<any>;

ngAfterViewInit() {

// 页面中的<!--template bindings={}-->元素

let commentElement = this.tpl.elementRef.nativeElement;

// 创建内嵌视图

let embeddedView = this.tpl.createEmbeddedView(null);

// 动态添加子节点

embeddedView.rootNodes.forEach((node) => {

commentElement.parentNode

.insertBefore(node, commentElement.nextSibling);

});

}

}

成功运行上面的代码后，在浏览器中我们会看到以下内容：

Welcome to Angular World

I am span in template

现在我们来回顾一下，上面的处理步骤：

* 创建内嵌视图(embedded view)
* 遍历内嵌视图中的 rootNodes，动态的插入 node

虽然我们已经成功的显示出 template 模板元素中的内容，但发现整个流程还是太复杂了，那有没有简单地方式呢 ？是时候介绍本文中第二个主角 - ViewContainerRef。

### ViewContainerRef

我们先来检验一下它的能力，然后再来好好地分析它。具体示例如下：

import { Component, TemplateRef, ViewChild, ViewContainerRef, AfterViewInit } from '@angular/core';

@Component({

selector: 'my-app',

template: `

<h1>Welcome to Angular World</h1>

<template #tpl>

<span>I am span in template</span>

</template>

`,

})

export class AppComponent {

name: string = 'Semlinker';

@ViewChild('tpl')

tplRef: TemplateRef<any>;

@ViewChild('tpl', { read: ViewContainerRef })

tplVcRef: ViewContainerRef;

ngAfterViewInit() {

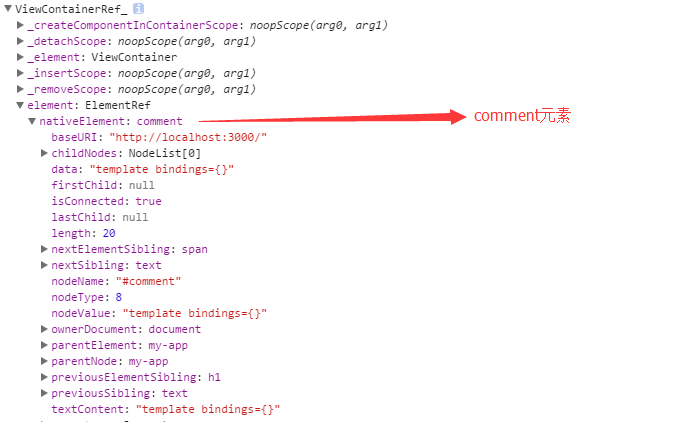
// console.dir(this.tplVcRef); (1)

this.tplVcRef.createEmbeddedView(this.tplRef);

}

}

移除上面代码中的注释，即可在控制台看到以下的输出信息：



而在浏览器中我们会看到以下内容：

Welcome to Angular World

I am span in template

接下来我们来看一下 ViewContainerRef\_ 类：

// @angular/core/src/linker/view\_container\_ref.d.ts

// 用于表示一个视图容器，可添加一个或多个视图

export declare class ViewContainerRef\_ implements ViewContainerRef {

...

length: number; // 返回视图容器中已存在的视图个数

element: ElementRef;

injector: Injector;

parentInjector: Injector;

// 基于TemplateRef创建内嵌视图，并自动添加到视图容器中，可通过index设置

// 视图添加的位置

createEmbeddedView<C>(templateRef: TemplateRef<C>, context?: C,

index?: number): EmbeddedViewRef<C>;

// 基 ComponentFactory创建组件视图

createComponent<C>(componentFactory: ComponentFactory<C>,

index?: number, injector?: Injector, projectableNodes?: any[][]): ComponentRef<C>;

insert(viewRef: ViewRef, index?: number): ViewRef;

move(viewRef: ViewRef, currentIndex: number): ViewRef;

indexOf(viewRef: ViewRef): number;

remove(index?: number): void;

detach(index?: number): ViewRef;

clear(): void;

}

通过源码我们可以知道通过 ViewContainerRef\_ 实例，我们可以方便地操作视图，也可以方便地基于 TemplateRef 创建视图。现在我们来总结一下 TemplateRef 与 ViewContainerRef。

TemplateRef：用于表示内嵌的 template 模板元素，通过 TemplateRef 实例，我们可以方便创建内嵌视图(Embedded Views)，且可以轻松地访问到通过 ElementRef 封装后的 nativeElement。需要注意的是组件视图中的 template 模板元素，经过渲染后会被替换成 comment 元素。

ViewContainerRef：用于表示一个视图容器，可添加一个或多个视图。通过 ViewContainer  
Ref 实例，我们可以基于 TemplateRef 实例创建内嵌视图，并能指定内嵌视图的插入位置，也可以方便对视图容器中已有的视图进行管理。简而言之，ViewContainerRef 的主要作用是创建和管理内嵌视图或组件视图。

### 我有话说

1.Angular 2 支持的 View(视图) 类型有哪几种 ？

* Embedded Views - Template 模板元素
* Host Views - Component 组件

1.1 如何创建 Embedded View

ngAfterViewInit() {

let view = this.tpl.createEmbeddedView(null);

}

1.2 如何创建 Host View

constructor(private injector: Injector,

private r: ComponentFactoryResolver) {

let factory = this.r.resolveComponentFactory(AppComponent);

let componentRef = factory.create(injector);

let view = componentRef.hostView;

}

2.Angular 2 Component 组件中定义的 <template> 模板元素为什么渲染后会被移除 ？

因为 <template> 模板元素，已经被 Angular 2 解析并封装成 TemplateRef 实例，通过 TemplateRef 实例，我们可以方便地创建内嵌视图(Embedded View)，我们不需要像开篇中的例子那样，手动操作 <template> 模板元素。

3.ViewRef 与 EmbeddedViewRef 之间有什么关系 ？

ViewRef 用于表示 Angular View(视图)，视图是可视化的 UI 界面。EmbeddedViewRef 继承于 ViewRef，用于表示 <template> 模板元素中定义的 UI 元素。

ViewRef

// @angular/core/src/linker/view\_ref.d.ts

export declare abstract class ViewRef {

destroyed: boolean;

abstract onDestroy(callback: Function): any;

}

EmbeddedViewRef

// @angular/core/src/linker/view\_ref.d.ts

export declare abstract class EmbeddedViewRef<C> extends ViewRef {

context: C;

rootNodes: any[]; // 保存<template>模板中定义的元素

abstract destroy(): void; // 用于销毁视图

}

### 总结

Angular 2 中 TemplateRef 与 ViewContainerRef 的概念对于初学者来说会比较羞涩难懂，本文从基本的 HTML 5 <template> 模板元素开始，介绍了如何操作和应用页面中定义的模板。然后通过实例介绍了 Angular 2 中 TemplateRef 和 ViewContainerRef 的定义和作用。希望通过这篇文章，读者能更好的理解 TemplateRef 与 ViewContainerRef。

How can i get the ViewRef of my current component -

I am trying to get ViewRef from a service. Here is the code -

*component.service.ts*

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

@Injectable()

export class CheckboxService{

constructor(private viewRef: ViewRef){

}

getViewRef(){

return this.viewRef;

}

}

*component.ts*

import { Component, EventEmitter, Output, ViewChild, ViewRef } from

'@angular/core';

import { AddFormDirective } from '../../add-form/add-form.directives';

import { CheckboxService } from './checkbox.service';

@Component({

selector: 'ang-checkbox',

templateUrl: './checkbox.component.html'

})

export class CheckboxViewComponent {

@ViewChild(AddFormDirective) addFormDirective: AddFormDirective;

constructor(private checkboxService: CheckboxService){

}

remove\_element(){

let viewContainerRef = this.addFormDirective.viewContainerRef;

let currentComponentIndex =

viewContainerRef.indexOf(this.checkboxService.getViewRef());

viewContainerRef.remove(currentComponentIndex);

}

}