第3章

指　令

3.1　指令概述

3.1.1　指令含义

Angular的模板是动态的，当Angular渲染它们时，会根据模板中的指令给出的指示对DOM进行转换。指令是为Angular应用中的元素添加额外行为的类，是一个带有@Directive()装饰器的类。从技术角度上说，组件也是指令。由于组件对Angular应用来说非常重要和独特，因此Angular专门定义了@Component()装饰器，它使用一些面向模板的特性扩展了@Directive()装饰器，即组件是带有模板特性的指令。

Angular本身定义了一系列结构型指令、属性型指令等内置指令，可以管理表单、列表、样式以及用户界面上要让用户看到的其他内容；也可以使用@Directive()装饰器定义自定义指令。

3.1.2　指令类型

指令可以分为组件、属性型指令、结构型指令等不同类型。为了区分，本书将组件和指令分开介绍。属性型指令可以更改元素、组件或其他指令的外观或行为；结构型指令可以通过添加和删除或替换DOM元素来更改DOM布局。

3.1.3　指令和模板的关系

指令的元数据把它所装饰的指令类和一个选择器(selector)关联起来，selector用来把关联的指令插入HTML模板中。在模板中，指令通常作为特性出现在模板元素(HTML标签)上，可能仅仅作为名字出现，也可能作为赋值目标或绑定目标出现。在模板中，它们看起来就像普通的HTML特性一样。

3.2　内置属性型指令

3.2.1　内置属性型指令说明

属性型指令会监听并修改HTML元素和组件的行为、attribute和property。许多

NgModule(如 RouterModule 和 FormsModule)都定义了自己的属性型指令。最常见的属性型指令包括添加和删除一组 CSS 类的指令 NgClass、添加和删除一组 HTML 样式的指令 NgStyle、将数据双向绑定添加到 HTML 表单元素的指令 NgModel 等指令。内置指令只会使用公开 API。它们不会访问任何不能被其他指令访问的私有 API。

3.2.2 NgClass 说明

要添加或删除单个类,优先使用类绑定而不是 NgClass。用 NgClass 可以同时添加或删除多个 CSS 类。可以将 NgClass 与表达式一起使用,即在要设置样式的元素上添加[ngClass]并将其设置为等于某个表达式。如代码< div [ngClass]="isSpecial ? 'special' : ''"> This div is special 中将 isSpecial 设置为布尔值 true 后,NgClass 就会把 special 类应用于< div >上。

要想将 NgClass 与方法一起使用,需要将方法添加到组件中。如例 3-1 中的 setCurrentClasses()方法使用一个对象来设置属性 currentClasses,该对象根据 canSave、isUnchanged、isSpecial 等组件属性为 true 或 false 来添加或删除三个类。该对象的每个键(key)(如 saveable)都是一个类名。如果键(类名)的取值为 true,则 NgClass 添加该类。如果键(类名)的取值为 false,则 NgClass 删除该类。于是,在模板中把 NgClass 属性绑定到 currentClasses,根据它来设置此元素的 CSS 类。在例 3-1 中,Angular 会在初始化以及发生更改的情况下应用这些类,即在 ngOnInit()方法中进行初始化以及通过单击按钮更改相关属性时调用 setCurrentClasses()方法。

【例 3-1】 创建文件 inside-directive.component.ts 的代码,演示本章的基础知识点。

```
import {Component} from '@angular/core';
import {Item} from "./item";
@Component({
  selector: 'root',//'app-inside-directive',
  template: `
    <div id="ngClass">NgClass 绑定</div>
    <p>currentClasses is {{currentClasses | json}}</p>
    <div [ngClass]="currentClasses">This div is initially saveable, unchanged, and special.
</div>
    <ul>
      <li>
        <label for="saveable">saveable</label>
        <input type="checkbox" [(ngModel)]="canSave" id="saveable">
      </li>
      <li>
        <label for="modified">modified:</label>
<input type="checkbox" [value]="!isUnchanged" (change)="isUnchanged=!isUnchanged" id=
"modified"></li>
      <li>
<label for="special">special: <input type="checkbox" [(ngModel)]="isSpecial" id=
"special"></label>
      </li>
    </ul>
    <button (click)="setCurrentClasses()">Refresh currentClasses</button>
    <div [ngClass]="currentClasses">
      This div should be {{ canSave ? "": "not"}} saveable,
      {{isUnchanged ? "unchanged" : "modified"}} and
```

```
      {{isSpecial ? "": "not"}} special after clicking "Refresh".</div>
    <br><br>
    <!-- 使用属性切换 special 类开/关(on/off ) -->
    <div [ngClass] = "isSpecial ? 'special' : ''">This div is special</div>
    <div class = "helpful study course">Helpful study course</div>
    <div [ngClass] = "{'helpful':false, 'study':true, 'course':true}">Study course</div>
    <hr>
    <div>NgStyle 绑定</div>
    <div [style.font-size] = "isSpecial ? 'x-large' : 'smaller'">
      This div is x-large or smaller.
    </div>
    <h4>[ngStyle] binding to currentStyles - CSS property names</h4>
    <p>currentStyles is {{currentStyles | json}}</p>
    <div [ngStyle] = "currentStyles">
      This div is initially italic, normal weight, and extra large (24px).
    </div>
    <br>
    <label>italic: <input type = "checkbox" [(ngModel)] = "canSave"></label> |
    <label>normal: <input type = "checkbox" [(ngModel)] = "isUnchanged"></label> |
    <label>xlarge: <input type = "checkbox" [(ngModel)] = "isSpecial"></label>
    <button (click) = "setCurrentStyles()">Refresh currentStyles</button>
    <br><br>
    <div [ngStyle] = "currentStyles">
      This div should be {{ canSave ? "italic": "plain"}},
      {{isUnchanged ? "normal weight" : "bold"}} and,
      {{isSpecial ? "extra large": "normal size"}} after clicking "Refresh".</div>
    <hr>
    <div id = "ngModel">NgModel 双向绑定</div>
    <fieldset><h4>NgModel examples</h4>
      <p>Current item name: {{currentItem.name}}</p>
      <p>
        <label for = "without">without NgModel:</label>
        <input [value] = "currentItem.name" (input) = "currentItem.name = getValue( $event)"
id = "without">
      </p>
      <p>
        <label for = "example-ngModel">[(ngModel)]:</label>
        <input [(ngModel)] = "currentItem.name" id = "example-ngModel">
      </p>
      <p>
        <label for = "example-change">(ngModelChange) = "...name = $event":</label>
<input [ngModel] = "currentItem.name" (ngModelChange) = "currentItem.name = $event" id =
"example-change">
      </p>
      <p>
        <label for = "example-uppercase">(ngModelChange) = "setUppercaseName( $event)"
<input [ngModel] = "currentItem.name" (ngModelChange) = "setUppercaseName( $event)" id =
"example-uppercase">
        </label>
      </p>
    </fieldset>
    <hr>
    <div *ngIf = "nullCustomer">Hello, <span>{{nullCustomer.id}}</span></div>
    <div *ngIf = "nullCustomer2">Hello, <span>{{nullCustomer2}}</span></div>
```

```
    <div *ngFor="let item of items; let i = index">{{i + 1}} - {{item.name}}</div>
    <div *ngFor="let item of items; trackBy: trackByItems">
      ({{item.id}}) {{item.name}}
    </div>
    <p>
      I turned the corner
      <ng-container *ngIf="item">
        and saw {{item.name}}. I waved
      </ng-container>
      and continued on my way.
    </p>
    <hr><h2>NgSwitch Binding</h2>
    <p>Pick your favorite item</p>
    <div>
      <label *ngFor="let i of items">
        <div><input type="radio" name="items" [(ngModel)]="currentItem" [value]="i">
{{i.name}}
        </div>
      </label>
    </div>
    <div [ngSwitch]="currentItem.feature">
      <app-stout-item      *ngSwitchCase="'stout'"     [item]="currentItem"></app-
stout-item>
      <app-device-item    *ngSwitchCase="'slim'"      [item]="currentItem"></app-
device-item>
      <app-lost-item      *ngSwitchCase="'vintage'"   [item]="currentItem"></app-lost-
item>
      <app-best-item      *ngSwitchCase="'bright'"    [item]="currentItem"></app-best-
item>
      <div *ngSwitchCase="'bright'">Are you as bright as {{currentItem.name}}?</div>
      <app-unknown-item    *ngSwitchDefault            [item]="currentItem"></app-
unknown-item>
    </div>
    <hr>
    <!--当指针悬停在 p 元素上时,背景颜色就会出现;而当指针移出时,背景颜色就会消失-->
    <p  [appHighlight]="color">Highlight me!</p>
    <h2>Pick a highlight color</h2>
    <div>
      <input type="radio" name="colors" (click)="color='lightgreen'">Green
      <input type="radio" name="colors" (click)="color='yellow'">Yellow
      <input type="radio" name="colors" (click)="color='cyan'">Cyan
    </div>
    <p [appHighlight]="color">Highlight me!</p>
    <p ngNonBindable>不会显示 2: {{ 1 + 1 }}</p>
    <div ngNonBindable [appHighlight]="'yellow'">
      This should not evaluate: {{ 1 + 1 }}, but will highlight yellow.
    </div>
    <hr>
    <p *appUnless="condition" class="unless a">
      (A) This paragraph is displayed because the condition is false.
    </p>
    <p *appUnless="!condition" class="unless b">
      (B) Although the condition is true,
      this paragraph is displayed because appUnless is set to false.
```

```
    </p>
    <p>
      The condition is currently
      <span [ngClass]="{ 'a': !condition, 'b': condition, 'unless': true }">{{condition}}</span>.
      <button
        (click)="condition = !condition"
        [ngClass] = "{ 'a': condition, 'b': !condition }" >
        Toggle condition to {{condition ? 'false' : 'true'}}
      </button>
    </p>
  `,
})
export class InsideDirectiveComponent {
  currentClasses: Record<string, boolean> = {};
  currentStyles: Record<string, string> = {};
  canSave: boolean = true;
  isUnchanged: boolean = true;
  isSpecial: boolean = true;
  item!: Item;
  items: Item[] = [];
  currentItem!: Item;
  nullCustomer: any = {
    id:21,
    name:'zsf'
  };
  nullCustomer2: any;
  color: any = 'red';
  condition: any = false;
  resetItems() {
    this.items = Item.items.map(item => item.clone());
    this.currentItem = this.items[0];
    this.item = this.currentItem;
  }
  ngOnInit() {
    this.resetItems();
    this.setCurrentClasses();
    this.setCurrentStyles();
  }
  setCurrentClasses() {
    this.currentClasses = {
      saveable: this.canSave,
      modified: !this.isUnchanged,
      special: this.isSpecial
    };
  }
  setCurrentStyles() {
    this.currentStyles = {
      'font-style':  this.canSave      ? 'italic' : 'normal',
      'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
      'font-size':   this.isSpecial    ? '24px'   : '12px'
    };
  }
  getValue(event: Event): string {
    return (event.target as HTMLInputElement).value;
```

```
  }
  setUppercaseName(name: string) {
    this.currentItem.name = name.toUpperCase();
  }
  trackByItems(index: number, item: Item): number { return item.id; }
}
```

3.2.3 NgStyle 说明

可以用 NgStyle 根据组件的状态同时设置多个内部样式。要使用 NgStyle,就要向组件添加一个方法。如例 3-1 中,setCurrentStyles()方法基于该组件 canSave、isUnchanged、isSpecial 等属性的状态,用一个定义了 font-style、font-weight、font-size 三个样式的对象设置了 currentStyles 属性。设置元素的样式,需要将 ngStyle 属性绑定到 currentStyles。在例 3-1 中,Angular 会在初始化以及发生更改的情况下应用这些类。完整的示例会在 ngOnInit()方法中进行初始化以及通过单击按钮更改相关属性时调用 setCurrentClasses()方法。

3.2.4 NgModel 说明

可以用 NgModel 指令显示数据属性,并在用户进行更改时更新该属性。导入 FormsModule,并将其添加到 NgModule 的 imports 列表中,如例 3-2 所示。在 HTML 的 < form >标签上添加[(ngModel)]绑定并将其设置为等于属性,如代码< input [(ngModel)]= "currentItem.name" id="example-ngModel">中设置数据绑定属性。要自定义配置,可以编写可展开的表单,该表单将属性绑定和事件绑定分开。使用属性绑定来设置属性,并使用事件绑定来响应更改。NgModel 指令适用于值访问器接口 ControlValueAccessor 支持的元素。Angular 为 HTML 表单所有基本元素提供了值访问器接口。要将[(ngModel)]应用于非表单型内置元素或第三方自定义组件,必须编写一个值访问器接口。编写 Angular 组件时,如果根据 Angular 的双向绑定语法命名 value 和 event 属性,则不需要用值访问器接口或 NgModel。

【例 3-2】 创建文件 app-directiveexample.module.ts 的代码,定义路由并声明组件和指令。

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule} from "@angular/router";
import {DirectiveexamplesComponent} from "./directiveexamples.component";
import {FormsModule} from "@angular/forms";
import {InsideDirectiveComponent} from "./inside-directive.component";
import {
  BestItemComponent,
  DeviceItemComponent,
  LostItemComponent,
  StoutItemComponent,
  UnknownItemComponent
} from "./item-switch.component";
import {HighlightDirective} from "./highlight.directive";
import {UnlessDirective} from "./unless.directive";
```

```
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {path: 'insidedirective', component: InsideDirectiveComponent},
      {path: 'directiveexample', component: DirectiveexamplesComponent},
    ]),
    FormsModule,
  ],
  declarations: [
    InsideDirectiveComponent,
    StoutItemComponent,
    BestItemComponent,
    DeviceItemComponent,
    LostItemComponent,
    UnknownItemComponent,
    HighlightDirective,
    UnlessDirective,
    DirectiveexampleComponent
  ],
})
export class AppDirectiveexampleModule { }
```

3.3 内置结构型指令

3.3.1 内置结构型指令说明

结构型指令的职责是进行 HTML 布局，并通过添加、移除和操纵它们所附加的宿主元素(DOM 节点)来塑造或重塑 DOM 结构。由于结构型指令会在 DOM 中添加和删除节点，因此每个元素只能应用一个结构型指令。常见的内置结构型指令包括从模板中创建或销毁子视图的 NgIf 指令、为列表中的每个条目重复渲染一个节点的 NgFor 指令和一组在备用视图之间切换的 NgSwitch 指令。

3.3.2 NgIf 说明

在宿主元素上用 NgIf 指令可以通过条件决定是否添加或删除宿主元素。如果 NgIf 指令为 false，则 Angular 将从 DOM 中移除对应的元素及其后代。然后，Angular 会销毁其组件，从而释放内存和资源。要添加或删除元素，需要将 * ngIf 绑定到条件表达式。如代码 < app-item-detail * ngIf＝"isActive" [item]＝"item"> </app-item-detail >中将 * ngIf 绑定到 isActive，当 isActive 表达式返回 true 值时，NgIf 指令会把 app-item-detail 所在组件添加到 DOM 中。在默认情况下，NgIf 指令会阻止显示已绑定到空值的元素。如要使用 NgIf 指令保护< div >，就需要将代码“ * ngIf＝"yourProperty""添加到< div >，修改之后的结果可能为< div * ngIf＝"currentCustomer"> Hello, {{currentCustomer. name}} 。如果属性 currentCustomer 为 null，则 Angular 不会显示< div >(注意，是整个< div >都不会显示)。

3.3.3 NgFor 说明

NgFor 指令用于显示条目列表。定义一个 HTML 块，该块用于决定 Angular 如何渲

染单个条目,如代码< div * ngFor="let item of items">{{item.name}} 中要列出的条目 item,把字符串"let item of items"赋给 * ngFor。字符串"let item of items"用于指示 Angular 执行将 items 中的每个条目存储在局部循环变量 item 中、让每个条目在每次迭代时的模板 HTML 中都可用、将"let item of items"转换为环绕宿主元素的< ng-template >、对列表中的每个 item 重复< ng-template >等操作。Angular 会将指令转换为< ng-template >,然后反复使用此模板为列表中的每个 item 创建一组新的元素和绑定。

要复写某个组件元素,可以将 * ngFor 应用于其选择器,如代码< app-item-detail * ngFor="let item of items" [item]="item"> </app-item-detail >中的选择器为< app-item-detail >。在宿主元素的后代中,该选择器用以访问条目的属性,将 item 通过绑定传递给< app-item-detail >组件的 item 属性。

可以在模板输入变量中获取 * ngFor 的 index 索引并在模板中使用它。在 * ngFor 中,添加一个分号(;)和 let i=index 的简写形式,如代码< div * ngFor="let item of items; let i=index">{{i + 1}} - {{item.name}} 即把 index 赋予一个名为 i 的变量中,并将其与条目名称一起显示。NgFor 指令上下文的 index 属性在每次迭代中都会返回该条目的从零开始的索引号。

3.3.4 NgIf、NgFor 和容器

若要在特定条件为 true 时重复某个 HTML 块,则可以将 * ngIf 放在 * ngFor 元素的容器元素上。它们之一或两者都可以是< ng-container >,这样就不必引入额外的 HTML 层次了。

通过跟踪对条目列表的更改,可以减少应用对服务器的调用次数。使用 * ngFor 的 trackBy 属性,Angular 只能更改和重新渲染已更改的条目,而不必重新加载整个条目列表。如向某组件添加一个 trackByItems()方法,该方法返回 NgFor 指令应该跟踪的值(item.id);如果浏览器已经渲染过某个 id,Angular 就会跟踪它而不会重新向服务器查询相同的 id。如果没有 trackBy 属性,就会由触发完全的 DOM 元素替换。有了 trackBy 属性,只有修改了 id 的按钮才会触发元素替换。

Angular 的< ng-container >是一个分组元素,它不会干扰样式或布局,因为 Angular 不会将其放置在 DOM 中。当没有单个元素承载指令时,可以使用< ng-container >。同时,要有条件地排除< option >,需要将< option >包裹在< ng-container >中。

3.3.5 NgSwitch 说明

NgSwitch 指令会根据条件显示几个可能的元素中的一个;而 Angular 只将选定的元素放入 DOM。NgSwitch 指令包括属性型指令 NgSwitch、结构型指令 NgSwitchCase 和 NgSwitchDefault。NgSwitch 指令可以更改其伴生指令的行为。当把[ngSwitch]绑定到一个返回开关值(开关值可以是任何类型)的表达式。当[* ngSwitchCase]绑定值等于开关值时,可将其元素添加到 DOM 中,而在其不等于开关值时将其绑定值移除。当 NgSwitchCase 没有被选中时,将[* ngSwitchDefault]宿主元素添加到 DOM 中。NgSwitch 指令也同样适用于内置 HTML 元素和 Web Component。

3.4 自定义属性型指令

3.4.1 创建

创建属性型指令的步骤如下。

(1) 先创建指令文件 *.directive.ts 并在模块文件中声明指令类。

(2) @Directive()装饰器的配置属性会指定指令的 CSS 属性选择器(如 appHighlight)。

(3) 从@angular/core 导入 ElementRef,ElementRef 的 nativeElement 属性会提供对宿主元素的直接访问权限。

(4) 在指令的 constructor()方法中添加 ElementRef 以注入对宿主元素的引用,该元素就是指令 CSS 属性选择器的作用目标。向指令类中添加逻辑。

要注意的是,指令不支持名称空间。

3.4.2 应用

要应用自定义属性型指令,可以将元素(如标签)添加到 HTML 模板中,并以伪指令(或称为属性选择器,如 appHighlight)作为属性,如<p appHighlight>Highlight me! ,Angualr 会创建指令类的实例,并将元素的引用注入该指令的构造函数中。

属性型指令可以用于处理用户事件,需要添加事件处理程序,每个事件处理程序都带有@HostListener()装饰器。

@Input()装饰器会将元数据添加到指令类用于绑定的属性,将值传递给属性型指令。

要防止在浏览器中进行表达式求值,可以将 ngNonBindable 添加到宿主元素。如果将 ngNonBindable 应用于父元素,则 Angular 会禁用该元素的子元素的任何插值和绑定。ngNonBindable 会停用模板中的插值、指令和绑定。但是,ngNonBindable 仍然允许指令在应用 ngNonBindable 的元素上工作。如代码<div ngNonBindable [appHighlight]="'yellow'">This should not evaluate: {{1+1}}, but will highlight yellow. 中,appHighlight 指令仍处于活跃状态,但 Angular 不会对表达式{{1+1}}求值。

3.5 自定义结构型指令

3.5.1 创建

结构型指令(如 *ngIf)中的星号(*)语法是 Angular 解释并转换为较长形式的简写形式。Angular 将结构型指令前面的星号转换为围绕宿主元素及其后代的<ng-template>。以代码<div *ngIf="hero" class="name">{{hero.name}} 为例,它等价于"<ng-template [ngIf]="hero"> {{hero.name}} </ng-template>"。Angular 不会创建真正的<ng-template>元素,只会将 和占位符{{hero.name}}的内容渲染到 DOM 中。

解析器会将帕斯卡命名(PascalCase)法应用于所有指令(即指令名的每个单词首字母大写),并为它们加上指令的属性名称(例如 ngFor),如 ngFor 的输入特性 of 和 trackBy 会

映射为 ngForOf 和 ngForTrackBy。当 NgFor 指令遍历列表时,它会设置和重置上下文对象的属性。

Angular 的< ng-template >元素定义了一个默认情况下不渲染任何内容的模板。使用< ng-template >可以手动渲染内容以完全控制内容的显示方式。如果没有结构型指令,并且将某些元素包装在< ng-template >中,则这些元素会消失。

3.5.2 应用

模板中的结构型指令会根据输入的表达式来控制是否要在运行时渲染该模板。为了帮助编译器捕获模板类型中的错误,应该尽可能详细地指定模板内指令的输入表达式所期待的类型。类型保护函数会将输入表达式的预期类型缩小为可能在运行时传递给模板内指令的类型的子集;可以提供这样的功能(函数)来帮助类型检查器在编译时推断出表达式的正确类型。

为模板中指令的输入表达式提供更具体的类型,要在指令中添加 ngTemplateGuard_xx 属性,其中静态属性名称 xx 就是@Input()装饰器修饰的属性(字段)名字。该属性的值可以是基于其返回类型的常规类型窄化函数,也可以是字符串,例如 NgIf 中的"binding"。例如,NgIf 的实现使用类型窄化来确保只有当 * ngIf 的输入表达式为真时,模板才会被实例化。为了提供具体的类型要求,NgIf 指令定义了一个静态属性 ngTemplateGuard_ngIf: 'binding'。这里的 binding 值是一种常见的类型窄化的例子,它会对输入表达式进行求值,以满足类型要求。


微课视频

3.6 指令的基础应用

3.6.1 基础代码

在项目 src\examples 目录下创建子目录 directiveexamples,在 src\examples\directiveexamples 目录下创建文件 inside-directive.component.ts,代码如例 3-1 所示。在 src\examples\directiveexamples 目录下创建文件 app-directiveexample.module.ts,代码如例 3-2 所示。在 src\examples\directiveexamples 目录下创建文件 item.ts,代码如例 3-3 所示。

【例 3-3】 创建文件 item.ts 的代码,定义并导出条目类 Item。

```
export class Item {
  static nextId = 0;
  static items: Item[] = [
    new Item(
      0,
      'Teapot',
      'stout'
    ),
    new Item(1, 'Lamp', 'bright'),
    new Item(2, 'Phone', 'slim'),
    new Item(3, 'Television', 'vintage'),
    new Item(4, 'Fishbowl')
```

```
  ];
  constructor(
    public id: number,
    public name?: string,
    public feature?: string,
    public url?: string,
    public rate = 100,
    ) {
    this.id = id ? id : Item.nextId++;
  }
  clone(): Item {
    return Object.assign(new Item(this.id), this);
  }
}
```

3.6.2 自定义指令

在 src\examples\directiveexamples 目录下创建文件 highlight.directive.ts，代码如例 3-4 所示。

【例 3-4】 创建文件 highlight.directive.ts 的代码，演示自定义属性型指令的方法。

```
import {Directive, ElementRef, HostListener, Input} from "@angular/core";
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  //@Input() 装饰器会将元数据添加到此类，以便让该指令的 appHighlight 属性可用于绑定
  @Input() appHighlight = '';
  private highlightColor = 'red';
  @Input() defaultColor = '';
  constructor(private el: ElementRef) {
    //向 HighlightDirective 类中添加逻辑，将背景设置为黄色
    el.nativeElement.style.backgroundColor = 'yellow';
  }
  //在鼠标进入时作出响应，事件处理程序都带有 @HostListener() 装饰器
//要订阅本属性型指令宿主元素(例 3-1 中标签<p>)的事件，可以使用 @HostListener() 装饰器
  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || this.defaultColor || 'red');
  }
  //在鼠标离开时作出响应，事件处理程序都带有 @HostListener() 装饰器
  @HostListener('mouseleave') onMouseLeave() {
    this.highlight('');
  }
  //辅助方法 highlight()，该方法会设置宿主元素 el 的颜色
  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

在 src\examples\directiveexamples 目录下创建文件 unless.directive.ts，代码如例 3-5 所示。

【例 3-5】 创建文件 unless.directive.ts 的代码,演示自定义结构型指令的方法。

```
import {Directive, Input, TemplateRef, ViewContainerRef} from '@angular/core';
@Directive({selector: '[appUnless]'})
export class UnlessDirective {
  private hasView = false;
  constructor(
    //在指令的构造函数中将 TemplateRef 和 ViewContainerRef 注入成私有变量
    //TemplateRef 可帮助获取 <ng-template> 的内容,而 ViewContainerRef 可以访问视图容器
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }
  //每当条件 condition 的值被更改时,Angular 都会设置 appUnless 属性
//若条件取假值且 Angular 尚未创建视图,则此 setter 方法会导致视图容器从模板创建出嵌入式视图
  //若条件取真值且当前正显示着视图,则此 setter 方法会清除容器,这会导致销毁该视图
    @Input() set appUnless(condition: boolean) {
    if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }
}
```

3.6.3 组件

在 src\examples\directiveexamples 目录下创建文件 item-switch.component.ts,代码如例 3-6 所示。

【例 3-6】 创建文件 item-switch.component.ts 的代码,定义组件。

```
import {Component, Input} from '@angular/core';
import {Item} from './item';
@Component({
  selector: 'app-stout-item',
  template: "I'm a little {{item.name}}, short and stout!"
})
export class StoutItemComponent {
  @Input() item!: Item;
}
@Component({
  selector: 'app-best-item',
  template: 'This is the brightest {{item.name}} in town.'
})
export class BestItemComponent {
  @Input() item!: Item;
}
@Component({
  selector: 'app-device-item',
  template: 'Which is the slimmest {{item.name}}?'
```

```
})
export class DeviceItemComponent {
  @Input() item!: Item;
}
@Component({
  selector: 'app-lost-item',
  template: 'Has anyone seen my {{item.name}}?'
})
export class LostItemComponent {
  @Input() item!: Item;
}
@Component({
  selector: 'app-unknown-item',
  template: '{{message}}'
})
export class UnknownItemComponent {
  @Input() item!: Item;
  get message() {
    return this.item && this.item.name ?
      `${this.item.name} is strange and mysterious.` :
      'A mystery wrapped in a fishbowl.';
  }
}
export const ItemSwitchComponents =
    [ StoutItemComponent, BestItemComponent, DeviceItemComponent, LostItemComponent,
UnknownItemComponent ];
```

3.6.4 模块

修改 src\examples 目录下的文件 examplesmodules1.module.ts,代码如例 3-7 所示。

【例 3-7】 修改文件 examplesmodules1.module.ts 的代码,设置启动组件。

```
import {NgModule} from '@angular/core';
import {AppDirectiveexampleModule} from './directiveexamples/app-directiveexample.module';
import {InsideDirectiveComponent} from './directiveexamples/inside-directive.component';
@NgModule({
  imports: [
    AppDirectiveexampleModule
  ],
  bootstrap: [InsideDirectiveComponent]
})
export class ExamplesmodulesModule1 {}
```

3.6.5 运行结果

保持其他文件不变并成功运行程序后,在浏览器地址栏中输入 localhost:4200,部分结果如图 3-1 所示。更多的结果请读者自己参考源代码进行验证。

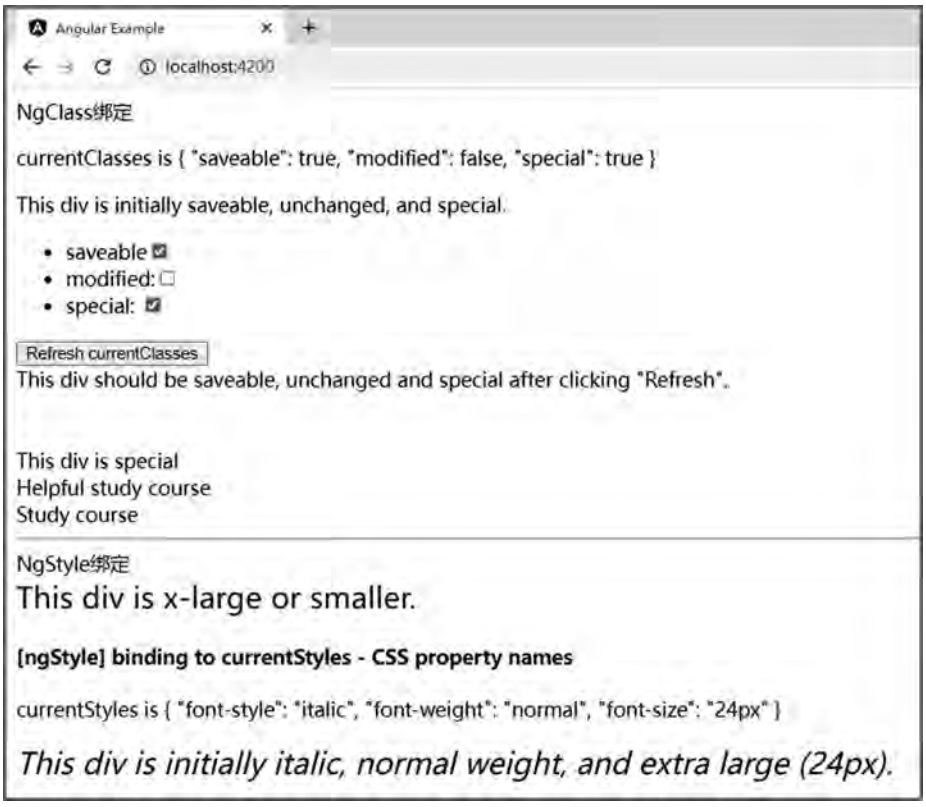

图 3-1　成功运行程序后在浏览器地址栏中输入 localhost:4200 的部分结果(从上往下)


微课视频

3.7　指令的综合应用开发

3.7.1　组件

在 src\examples\directiveexamples 目录下创建文件 directiveexamples.component.ts，代码如例 3-8 所示。

【例 3-8】　创建文件 directiveexamples.component.ts 的代码，定义组件。

```
import {Component} from '@angular/core';
@Component({
  selector: 'root',
  template: `
    NgIf 指令示例(布尔变量 showName 的值为{{showName}} )
    <div *ngIf = "showName"> showName 为真时显示</div>
    <hr>
    NgFor 指令示例
    <div *ngFor = "let author of authors; let i = index">{{i}} : {{author.name}} {{author.
book}}</div>
    <hr>
    NgSwitch 指令示例
    <div [ngSwitch] = "day">
      <b *ngSwitchCase = "days.MONDAY">星期一</b>
      <b *ngSwitchCase = "days.TUESDAY">星期二</b>
```

```
      <b *ngSwitchCase="days.WEDNESDAY">星期三</b>
      <b *ngSwitchCase="days.THURSDAY">星期四</b>
      <b *ngSwitchCase="days.FRIDAY">星期五</b>
      <b *ngSwitchDefault>休息日</b>
    </div>
    <hr>
NgClass 指令示例(先定义 CSS 样式,再在 ngClass 后面用对象、数组、字符串的方式引入所定义的样式)
    <div [ngClass]="objectStyleFormat">用对象的方式描述要定义的样式(大字)</div>
<div [ngClass]="{'redClassStyle': flag, 'blueClassStyle': !flag}">用对象的方式描述要定义的
样式(不是大字)</div>
    <div [ngClass]="arrayStyleFormat">用数组的方式描述要定义的样式(加粗大字)</div>
    <div [ngClass]="stringStyleFormat">用字符串的方式描述要定义的样式(加粗不是大字)</div>
    <hr>
    NgStyle 指令示例
    <div [ngStyle]="{'font-size':'18px'}"> 18px 字体</div>
    <div [ngStyle]="{'font-size':attr}"> 22px 字体</div>
    <div [ngStyle]="currentStyles"> 24px 加粗黑色斜体字</div>
  `,
  styles: ['.blackClassStyle {color: black} ' +
  '.blueClassStyle {color: blue} ' +
  '.largerFont {font-size: larger} ' +
  '.boldFont {font-weight:bold} ' +
  '.currentStyle  {font-style: italic; font-weight :  bold;font-size : 24px}'
  ]
})
export class DirectiveexamplesComponent {
  showName: boolean = true;
authors: any = [{name: '左丘明', book: '左传'}, {value: '陈寿', book: '三国志'}, {value: '范晔',
book: '后汉书'}];
  days = Days;
  day: Days = this.days.TUESDAY;
  flag = true ;
    objectStyleFormat = {'blackClassStyle': true, 'blueClassStyle': false, 'largerFont': true};
    arrayStyleFormat = ['blackClassStyle', 'largerFont', 'boldFont'];
    stringStyleFormat = "blackClassStyle boldFont ";
    currentStyles = {"font-style": "italic", "font-weight": "bold", "font-size":
"24px ", "color": "black"};
    attr = '22px'
}
export enum Days {
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY
}
```

3.7.2 模块

修改 src\examples 目录下的文件 examplesmodules1.module.ts,代码如例 3-9 所示。

【例 3-9】 修改文件 examplesmodules1.module.ts 的代码。

```
import {NgModule} from '@angular/core';
import{AppDirectiveexampleModule}from'./directiveexamples/app-directiveexample.module';
import {DirectiveexamplesComponent} from './directiveexamples/directiveexamples.component';
@NgModule({
```

```
  imports: [
    AppDirectiveexampleModule
  ],
  bootstrap: [DirectiveexamplesComponent]
})
export class ExamplesmodulesModule1 {}
```

3.7.3 运行结果

保持其他文件不变并成功运行程序后，在浏览器地址栏中输入 localhost:4200，结果如图 3-2 所示。

图 3-2 成功运行程序后在浏览器地址栏中输入 localhost:4200 的结果

习题 3

一、简答题

1. 简述对指令的理解。
2. 简述对各类指令的理解。

二、实验题

实现指令的应用开发。