第5章

组件的组合、分解及其应用


微课视频

5.1 内容投影及其应用

5.1.1 常见的内容投影

Angular 内容投影是一种组件组合模式，可以在一个组件中插入或投影另一个组件，并使用另一个组件的内容。Angular 中内容投影的常见实现方法包括单插槽内容投影、多插槽内容投影和有条件的内容投影。在使用单插槽内容投影场景下，组件可以从单一来源（如组件）接收内容；而在多插槽内容投影场景下，组件可以从多个来源接收内容。换言之，一个组件可以插入一个或多个其他组件，使用条件内容投影的组件仅在满足特定条件时才渲染内容。

1. 单插槽内容投影

内容投影的最基本形式之一是单插槽内容投影。单插槽内容投影是指创建一个组件且可以在其中投影另一个组件。创建使用单插槽内容投影的组件的步骤包括创建一个组件；在创建的组件模板中添加< ng-content >元素并将希望投影的内容写入其中；有了< ng-content >元素，组件的用户（调用者）就可以将自己的消息另一个组件的内容投影到该组件中。< ng-content >元素是一个占位符，它不会创建真正的 DOM 元素，但会被替换成调用时的消息（内容），其自定义属性将被忽略。

2. 多插槽内容投影

一个组件可以具有多个插槽，每个插槽可以指定一个 CSS 选择器，该选择器会决定将哪些内容写入该插槽。此种模式被称为多插槽内容投影。使用此模式，可以用< ng-content >的 select 属性指定希望投影内容出现的位置。创建使用多插槽内容投影的组件的步骤包括创建一个组件；在创建的组件模板中添加< ng-content >元素并将希望投影的内容写入其中；将 select 属性添加到< ng-content >元素。Angular 使用的选择器支持标签名、属性、CSS 类和伪类：not 的任意组合。使用 question 属性的内容将投影到带有 select = [question]属性的< ng-content >元素。如果组件中包含不带 select 属性的< ng-content >元素，则该实例将接收所有与其他< ng-content >元素都不匹配的投影组件。

3. 条件内容投影

如果组件需要有条件地渲染内容或多次渲染内容，则应配置该组件以接收一个包含有条件渲染的内容的< ng-template >元素。在这种情况下，不建议使用< ng-content >元素。

因为只要组件的使用者(provider)提供了内容,即使该组件从未定义< ng-content >元素或该< ng-content >元素位于 ngIf 语句的内部,该内容也总会被初始化。使用< ng-template >元素可以让组件根据条件显式渲染内容,并可以进行多次渲染。在显式渲染< ng-template >元素之前,Angular 不会初始化该元素的内容。

< ng-template >进行条件内容投影的步骤包括创建一个组件;在接收< ng-template >元素创建的组件中使用< ng-container >元素渲染该模板;将< ng-container >元素包装在另一个元素(例如 div 元素)中后应用条件逻辑;在要投影内容的模板中将投影的内容写入< ng-template >元素中。组件可以使用@ContentChild()装饰器或@ContentChildren()装饰器获得对此模板内容的引用(即 TemplateRef)。借助于 TemplateRef,组件可以使用 ngTemplateOutlet 指令或 ViewContainerRef. createEmbeddedView()方法来渲染所引用的内容。如果是多插槽内容投影,则可以使用@ContentChildren()装饰器获取投影元素的查询列表。在某些情况下,可能希望将内容投影为其他元素,例如,如果要投影的内容可能是另一个元素的子元素,那么可以用 ngProjectAs 属性来完成此操作。< ng-container >元素是一个逻辑结构,可用于对其他 DOM 元素进行分组;但< ng-container > 本身不会在 DOM 树中渲染。

5.1.2 内容投影的应用

在项目 src\examples 目录下创建子目录 contentprojectexamples,在 src\examples\contentprojectexamples 目录下创建文件 zippybasic. component. ts,代码如例 5-1 所示。

【例 5-1】 创建文件 zippybasic. component. ts 的代码,定义带单插槽内容投影功能的组件。

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-zippy-basic',
  template: `
    <div>单插槽内容投影(Single-slot content projection)</div>
    <ng-content></ng-content>
  `
})
export class ZippybasicComponent {}
```

在 src\examples\contentprojectexamples 目录下创建文件 zippymultislot. component. ts,代码如例 5-2 所示。

【例 5-2】 创建文件 zippymultislot. component. ts 的代码,定义带多插槽内容投影功能的组件。

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-zippy-multislot',
  template: `
    <div>多插槽内容投影(Multi-slot content projection)</div>
    默认:
    <ng-content></ng-content>
    问题:
    <ng-content select="[question]"></ng-content>
  `
})
export class ZippymultislotComponent {}
```

在 src\examples\contentprojectexamples 目录下创建文件 zippyngprojectas. component. ts,代码如例 5-3 所示。

【例 5-3】 创建文件 zippyngprojectas. component. ts 的代码,定义带 select 属性的组件。

```
import {Component} from '@angular/core';
@Component({
  selector: 'app - zippy - ngprojectas',
  template: `
    < div >使用 ngProjectAs 属性的内容投影(Content projection with ngProjectAs)</div >
    默认:
    < ng - content ></ng - content >
    问题:
    < ng - content select = "[question]"></ng - content >
  `
})
export class ZippyngprojectasComponent {}
```

在 src\examples\contentprojectexamples 目录下创建文件 zippy. component. ts,代码如例 5-4 所示。

【例 5-4】 创建文件 zippy. component. ts 的代码,定义带有条件内容投影功能的组件。

```
import {Component, ContentChild, Input} from "@angular/core";
import {ZippycontentDirective} from "./zippycontent.directive";
let nextId = 0;
@Component({
  selector: 'app - example - zippy',
  template: `
    < ng - content ></ng - content >
    < div * ngIf = "expanded" [id] = "contentId">
      < ng - container [ngTemplateOutlet] = "content.templateRef"></ng - container >
    </div >
  `,
})
export class ZippyComponent {
  contentId = `zippy - ${nextId++}`;
  @Input() expanded = false;
  @ContentChild(ZippycontentDirective) content!: ZippycontentDirective;
}
```

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

【例 5-5】 创建文件 zippytoggle. directive. ts 的代码,定义指令。

```
import {Directive, HostBinding, HostListener} from "@angular/core";
import {ZippyComponent} from "./zippy.component";
@Directive({
  selector: 'button[appExampleZippyToggle]',
})
export class ZippytoggleDirective {
  @HostBinding('attr.aria - expanded') ariaExpanded = this.zippy.expanded;
  @HostBinding('attr.aria - controls') ariaControls = this.zippy.contentId;
  @HostListener('click') toggleZippy() {
```

```
    this.zippy.expanded = !this.zippy.expanded;
  }
  constructor(public zippy: ZippyComponent) {}
}
```

在 src\examples\contentprojectexamples 目录下创建文件 zippycontent.directive.ts，代码如例 5-6 所示。

【例 5-6】 创建文件 zippycontent.directive.ts 的代码，定义指令。

```
import {Directive, TemplateRef} from "@angular/core";
@Directive({
  selector: '[appExampleZippyContent]'
})
export class ZippycontentDirective {
  constructor(public templateRef: TemplateRef<unknown>) {}
}
```

在 src\examples\contentprojectexamples 目录下创建文件 contentprojectexample.component.ts，代码如例 5-7 所示。

【例 5-7】 创建文件 contentprojectexample.component.ts 的代码，内容投影的综合应用。

```
import {Component} from "@angular/core";
@Component({
  selector: 'root',
  template: `
    <div>内容投影(Content Projection)</div>
    <app-zippy-basic>
      <p>内容投影很酷吧?</p>
    </app-zippy-basic>
    <hr />
    <app-zippy-multislot>
      <p question>
        内容投影很酷吧?
      </p>
      <p>开始学习内容投影吧!</p>
    </app-zippy-multislot>
    <hr />
    <div>zippy 示例</div>
    <app-example-zippy>
      <button appExampleZippyToggle>内容投影很酷吧?</button>
      <ng-template appExampleZippyContent>
        这取决于用它做什么
      </ng-template>
    </app-example-zippy>
    <hr />
    <app-zippy-ngprojectas>
      <p>开始学习内容投影吧!</p>
      <ng-container ngProjectAs="[question]">
        <p>内容投影很酷吧?</p>
      </ng-container>
    </app-zippy-ngprojectas>
  `,
  styles: ['p {font-family: Lato}']
```

```
})
export class ContentprojectexampleComponent {
}
```

在 src\examples\contentprojectexamples 目录下创建文件 app-content. module. ts,代码如例 5-8 所示。

【例 5-8】 创建文件 app-content. module. ts 的代码,定义路由并声明组件。

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform - browser';
import {RouterModule} from "@angular/router";
import {ZippybasicComponent} from "./zippybasic.component";
import {ZippymultislotComponent} from "./zippymultislot.component";
import {ZippyngprojectasComponent} from "./zippyngprojectas.component";
import {ZippytoggleDirective} from "./zippytoggle.directive";
import {ZippycontentDirective} from "./zippycontent.directive";
import {ZippyComponent} from "./zippy.component";
import {ContentprojectexampleComponent} from "./contentprojectexample.component";
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {path: 'content', component: ContentprojectexampleComponent},
    ]),
  ],
  declarations: [
    ZippybasicComponent,
    ZippymultislotComponent,
    ZippyngprojectasComponent,
    ZippytoggleDirective,
    ZippycontentDirective,
    ZippyComponent,
    ContentprojectexampleComponent,
  ],
})
export class AppEcontentModule { }
```

修改 src\examples 目录下的文件 examplesmodules1. module. ts,代码如例 5-9 所示。

【例 5-9】 修改文件 examplesmodules1. module. ts 的代码,设置启动组件。

```
import {NgModule} from '@angular/core';
import {AppEcontentModule} from "./contentprojectexamples/app - econtent.module";
import { ContentprojectexampleComponent } from ' ./contentprojectexamples/contentprojectexample.
component';
@NgModule({
  imports: [
    AppEcontentModule,
  ],
  bootstrap: [ContentprojectexampleComponent]
})
export class ExamplesmodulesModule1 {}
```

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 5-1 所示。

图 5-1　成功运行程序后在浏览器地址栏中输入 localhost:4200 的结果

5.2　视图封装及其应用


微课视频

5.2.1　视图封装模式

Angular 应用中组件的样式可以封装在组件的宿主元素中，这样它们就不会影响应用的其余部分的样式。组件的装饰器提供了 encapsulation 选项用来控制如何基于模式对每个组件的应用视图进行封装。encapsulation 选项包括 Shadow Dom、Emulated、None 等模式。

1. Shadow Dom 模式

Shadow Dom 模式下 Angular 使用浏览器内置的 Shadow DOM API 将组件的视图包含在 Shadow Root(用作组件的宿主元素)中，并以隔离的方式应用所提供的样式。Shadow Dom 模式仅适用于内置支持阴影 DOM 的浏览器。并非所有浏览器都支持它。

2. Emulated 模式

Emulated 模式下 Angular 会修改组件 CSS 选择器，使它们只应用于组件的视图而不影响应用中的其他元素(模拟 Shadow DOM 行为)。此模式也是默认模式和推荐模式。使用 Emulated 模式时，Angular 会预处理所有组件的样式，以便它们仅应用于组件的视图。在运行的 Angular 应用的 DOM 中，使用 Emulated 的组件所在的元素附加了一些额外的属

性(如_nghost 属性、_ngcontent 属性)。_nghost 属性被添加到包裹组件视图的元素(即宿主元素)中,这个元素是 Shadow DOM 封装中的 ShadowRoot;组件的宿主元素通常就是这种情况。_ngcontent 属性被添加到组件视图中的子元素上,这些属性用于将元素与其各自模拟的 ShadowRoot(具有匹配_nghost 属性的宿主元素)相匹配。Angular 独自实现这些属性的具体取值且隐藏实现细节。这些取值是自动生成的,不应在应用代码中引用;它们常用于生成组件的样式,这些组件样式会被注入 DOM 的< head >部分;并经过后期处理以便每个 CSS 选择器都能使用适当的_nghost 或_ngcontent 属性对样式进行扩充。这些修改后的选择器可以确保样式以相互独立且有针对性的方式应用于组件的视图。

3. None 模式

None 模式表示 Angular 不使用任何形式的视图封装,这意味着为组件指定的任何样式都是全局的,可以影响应用中存在的任何 HTML 元素。这种模式在本质上与将样式包含在 HTML 中是一样的。

可以在组件的装饰器中针对每个组件指定封装模式,这意味着应用程序中不同的组件可以使用不同的封装策略,但不建议这样做。Shadow Dom 组件的样式仅添加到 Shadow DOM 宿主中,确保它们仅影响各自组件视图中的元素。Emulated 组件的样式会添加到文档的< head >中,以使它们在整个应用中可用,但它们的选择器只会影响它们各自组件模板中的元素。None 组件的样式会添加到文档的< head >中,使它们在整个应用中可用,会影响文档中的任何匹配元素(因为是全局的)。Emulated 和 None 组件的样式会添加到每个 Shadow Dom 组件的 Shadow DOM 宿主中。None 组件的样式将影响 Shadow DOM 中的匹配元素。

5.2.2 视图封装的应用

在项目 src\examples 目录下创建子目录 viewencapsulationexamples,在 src\examples\viewencapsulationexamples 目录下创建文件 noencapsulation.component.ts,代码如例 5-10 所示。

【例 5-10】 创建文件 noencapsulation.component.ts 的代码,定义使用 None 模式的组件。

```
import {Component, ViewEncapsulation} from "@angular/core";
@Component({
  selector: 'app-no-encapsulation',
  template: `
    <div class="inside">
    <div>None</div>
    <div class="none-message">没有任何视图封装</div>
    </div>
  `,
  styles: ['div, .none-message { color: red}  ' +
  '.inside{border: 1px solid black;width: 200px}'],
  //使用全局样式,没有任何视图封装
  encapsulation: ViewEncapsulation.None,
})
export class NoencapsulationComponent { }
```

在 src\examples\viewencapsulationexamples 目录下创建文件 emulatedencapsulation.component.ts,代码如例 5-11 所示。

【例 5-11】 创建文件 emulatedencapsulation.component.ts 的代码,定义 Emulated 模式组件。

```
import {Component, ViewEncapsulation} from "@angular/core";
@Component({
  selector: 'app-emulated-encapsulation',
  template: `
    <div class="middle">
    <div>Emulated(默认编译器)</div>
    <div class="emulated-message">使用垫片(shimmed) CSS 来模拟原生行为</div>
    <app-no-encapsulation></app-no-encapsulation>
    </div>
  `,
  styles: ['h2, .emulated-message {color: black;} ' +
  '.middle{border: 2px solid blue; width: 400px}'],
  encapsulation: ViewEncapsulation.Emulated,
})
export class EmulatedencapsulationComponent { }
```

在 src\examples\viewencapsulationexamples 目录下创建文件 shadowdomencapsulation.component.ts,代码如例 5-12 所示。

【例 5-12】 创建文件 shadowdomencapsulation.component.ts 的代码,定义 ShadowDom 模式组件。

```
import {Component, ViewEncapsulation} from "@angular/core";
@Component({
  //用 ViewEncapsulation.ShadowDom 时,selector 的命名须用连接符,如 app-root
  selector: 'app-root',
  template: `
    <div class="outside">
      <div class="shadow-message">ShadowDOM 封装</div>
      <app-emulated-encapsulation></app-emulated-encapsulation>
      <app-no-encapsulation></app-no-encapsulation>
    </div>
  `,
  styles: ['h2, .shadow-message {color: blue;}' +
  '.outside{border: 1px solid red;width: 600px}'],
  encapsulation: ViewEncapsulation.ShadowDom,
})
export class ShadowdomencapsulationComponent { }
```

在 src\examples\viewencapsulationexamples 目录下创建文件 viewencapsulationexample.component.ts,代码如例 5-13 所示。

【例 5-13】 创建文件 viewencapsulationexample.component.ts 的代码,定义组件。

```
import {Component} from "@angular/core";
@Component({
  selector: 'root',
  template: `
```

```
      <app-root></app-root>
    `
})
export class ViewencapsulationexampleComponent { }
```

5.2.3 模块和运行结果

在 src\examples\viewencapsulationexamples 目录下创建文件 app-encapsulation.module.ts,代码如例 5-14 所示。

【例 5-14】 创建文件 app-encapsulation.module.ts 的代码,定义路由并声明组件。

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule} from "@angular/router";
import {EmulatedencapsulationComponent} from "./emulatedencapsulation.component";
import {NoencapsulationComponent} from "./noencapsulation.component";
import {ShadowdomencapsulationComponent} from "./shadowdomencapsulation.component";
import {ViewencapsulationexampleComponent} from "./viewencapsulationexample.component";
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {path: 'encapsulation', component: ViewencapsulationexampleComponent},
    ]),
  ],
  declarations: [
    EmulatedencapsulationComponent,
    NoencapsulationComponent,
    ShadowdomencapsulationComponent,
    ViewencapsulationexampleComponent,
  ],
})
export class AppEncapsulationModule { }
```

修改 src\examples 目录下的文件 examplesmodules1.module.ts,代码如例 5-15 所示。

【例 5-15】 修改文件 examplesmodules1.module.ts 的代码,设置启动组件。

```
import {NgModule} from '@angular/core';
import {AppEncapsulationModule} from"./viewencapsulationexamples/app-encapsulation.module";
import {ViewencapsulationexampleComponent} from './viewencapsulationexamples/viewencapsul-
ationexample.component';
@NgModule({
  imports: [
    AppEncapsulationModule,
  ],
  bootstrap: [ViewencapsulationexampleComponent]
})
export class ExamplesmodulesModule1 {}
```

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 5-2 所示。

图 5-2　成功运行程序后在浏览器地址栏中输入 localhost:4200 的结果

5.3　依赖注入及其应用


微课视频

5.3.1　依赖注入概述

依赖注入(DI)是一种设计模式,在这种设计模式中,类(如组件、模块、服务等)会从外部源请求依赖项而不是创建它们。依赖项是指某个类执行其功能所需的服务或对象。Angular 的 DI 框架会在实例化某个类时为其提供依赖;可以使用 DI 来提高应用的灵活性和模块化程度。@Injectable()装饰器会指定 Angular 在 DI 体系中使用所定义的类。

注入某些服务会使它们对组件可见。要将依赖项注入组件的 constructor()方法中,提供具有此依赖项类型的构造函数参数。当创建一个带有参数的 constructor()方法的类时,还需要指定参数类型和关于这些参数的元数据,以便 Angular 可以注入正确的服务。

通过配置提供者,可以把服务提供给那些需要它们的应用部件。依赖提供者会使用 DI 令牌来配置注入器,会将提供者与依赖项注入令牌(或叫 DI 令牌)并关联起来。注入器会用它来提供这个依赖值的具体的运行时版本,允许 Angular 创建任何内部依赖项的映射。DI 令牌会充当该映射的键名,如果把服务类指定为提供者令牌,那么注入器的默认行为是用 new 来实例化该类。尽管许多依赖项的值是通过类提供的,但扩展的 provide 对象可以将不同种类的提供者与 DI 令牌相关联;也可以用一个替代提供者来配置注入器,以指定另一些同样能提供日志功能的对象。因此,可以使用服务类来配置注入器,以提供一个替代类、一个对象或一个工厂函数。

5.3.2　依赖注入的实现方法

类提供者的语法实际上是一种简写形式,它会扩展成一个由 Provider 接口定义的提供者配置对象。不同的类可以提供相同的服务。如果替代类提供者有自己的依赖,就在父模块或组件的元数据属性 providers 中指定那些依赖。要为类提供者设置别名,在 providers 数组中使用 useExisting 属性指定别名和类提供者。通常,编写同一个父组件别名提供者的变体时会使用 forwardRef;若要为多个父类型指定别名(每个类型都有自己的类接口令牌),则要配置 provideParent()方法以接收更多的参数。

要注入一个对象,可以用 useValue 选项来配置注入器。常用的对象字面量是配置对

象。若要提供并注入配置对象,则要在@NgModule()装饰器的 providers 数组中指定该对象,可以定义和使用一个 InjectionToken 对象来为非类的依赖选择一个提供者令牌。借助@Inject()参数装饰器,可以把这个配置对象注入构造函数中。

虽然 TypeScript 的 AppConfig 接口可以在类中提供类型支持,但它在依赖注入时却没有任何作用。在 TypeScript 中,接口是一项设计时的部件,它没有可供 DI 框架使用的运行时表示形式或令牌。当转译器把 TypeScript 转换成 JavaScript 时,接口就会消失,因为 JavaScript 没有接口。由于 Angular 在运行时没有接口,所以接口不能作为令牌,也不能注入它。如果想在运行前,根据尚不可用的信息创建可变的依赖值,则可以使用工厂提供者(即采用工厂模式)。

5.3.3 服务类

在项目 src\examples 根目录下创建 injectexamples 子目录,在 src\examples\injectexamples 目录下创建文件 hero.ts,代码如例 5-16 所示。

【例 5-16】 创建文件 hero.ts 的代码,定义接口和数组。

```
export interface Hero {
  id: number;
  name: string;
  isSecret: boolean;
}
export const HEROES: Hero[] = [
  {id: 1, isSecret: false, name: '张三丰'},
  {id: 2, isSecret: false, name: '李斯'},
  {id: 3, isSecret: false, name: '王阳明'},
  {id: 4, isSecret: false, name: '朱熹'},
];
```

在 src\examples\injectexamples 目录下创建文件 hero.service.ts,代码如例 5-17 所示。

【例 5-17】 创建文件 hero.service.ts 的代码,定义类 HeroService。

```
import {Injectable} from '@angular/core';
import {Logger} from "./logger.service";
import {UserService} from "./user.service";
import {HEROES} from "./hero";
@Injectable({
  providedIn: 'root',
  useFactory: (logger: Logger, userService: UserService) =>
    new HeroService(logger, userService.user.isAuthorized),
  deps: [Logger, UserService],     //要用到类 Logger 和 UserService
})
export class HeroService {
  constructor(
    private logger: Logger,
    private isAuthorized: boolean) {
  }
  getHeroes() {
    const auth = this.isAuthorized ? 'authorized ' : 'unauthorized';
    this.logger.log(`Getting heroes for ${auth} user.`);
```

```
    return HEROES.filter(hero => this.isAuthorized || !hero.isSecret);
  }
}
```

在 src\examples\injectexamples 目录下创建文件 user.service.ts，代码如例 5-18 所示。

【例 5-18】 创建文件 user.service.ts 的代码，定义类 User 和 UserService。

```
import {Injectable} from '@angular/core';
export class User {
  constructor(
    public name: string,
    public isAuthorized = false) {
  }
}
const bob = new User('Bob', false);
@Injectable({
  providedIn: 'root'
})
export class UserService {
  user = bob;
}
```

在 src\examples\injectexamples 目录下创建文件 logger.service.ts，代码如例 5-19 所示。

【例 5-19】 创建文件 logger.service.ts 的代码，定义类 Logger。

```
import {Injectable} from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class Logger {
  logs: string[] = [];
  log(message: string) {
    this.logs.push(message);
    console.log(message);
  }
}
```

5.3.4 组件

在 src\examples\injectexamples 目录下创建文件 herolist.component.ts，代码如例 5-20 所示。

【例 5-20】 创建文件 herolist.component.ts 的代码，定义组件 app-hero-list。

```
import {Component} from '@angular/core';
import {Hero} from './hero';
import {HeroService} from './hero.service';
@Component({
  selector: 'app-hero-list',
  template: `
    <div *ngFor="let hero of heroes">
      {{hero.id}} - {{hero.name}}
      ({{hero.isSecret ? 'secret' : 'public'}})
```

```
      </div>
    `,
})
export class HeroListComponent {
  heroes: Hero[];
  constructor(heroService: HeroService) {
    this.heroes = heroService.getHeroes();
  }
}
```

在 src\examples\injectexamples 目录下创建文件 heroes.component.ts,代码如例 5-21 所示。

【例 5-21】 创建文件 heroes.component.ts 的代码,定义组件 app-heroes。

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-heroes',
  template: `
    <div>人名列表</div>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent {
}
```

在 src\examples\injectexamples 目录下创建文件 heroestsp.component.ts,代码如例 5-22 所示。

【例 5-22】 创建文件 heroestsp.component.ts 的代码,定义组件 app-heroes-tsp。

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-heroes-tsp',
  template: `
    <div>人名列表</div>
    <app-hero-list></app-hero-list>
  `
})
export class HeroestspComponent {
}
```

在 src\examples\injectexamples 目录下创建文件 injectexamples.component.ts,代码如例 5-23 所示。

【例 5-23】 创建文件 injectexamples.component.ts 的代码,调用三个组件以形成对比。

```
import {Component} from '@angular/core';
@Component({
  selector: 'root',
  template: `
    <app-heroes id="authorized" *ngIf="isAuthorized"></app-heroes>
    <app-heroes id="unauthorized" *ngIf="!isAuthorized"></app-heroes>
    <hr>
    <div>人名列表</div>
    <app-hero-list></app-hero-list>
```

```
    <hr>
    <app-heroes-tsp id="tspAuthorized" *ngIf="isAuthorized"></app-heroes-tsp>
  `
})
export class InjectexamplesComponent {
  isAuthorized = true;
}
```

5.3.5 模块和运行结果

在 src\examples\injectexamples 目录下创建文件 app-inject.module.ts，代码如例 5-24 所示。

【例 5-24】 创建文件 app-inject.module.ts 的代码，定义路由并声明组件。

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RouterModule} from "@angular/router";
import {InjectexamplesComponent} from "./injectexamples.component";
import {HeroesComponent} from "./heroes.component";
import {HeroListComponent} from "./herolist.component";
import {HeroestspComponent} from "./heroestsp.component";
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {path: 'inject', component: InjectexamplesComponent},
    ]),
  ],
  declarations: [
    HeroesComponent,
    HeroListComponent,
    HeroestspComponent,
    InjectexamplesComponent
  ],
})
export class AppInjectModule { }
```

修改 src\examples 目录下的文件 examplesmodules1.module.ts，代码如例 5-25 所示。

【例 5-25】 修改文件 examplesmodules1.module.ts 的代码，设置启动组件。

```
import {NgModule} from '@angular/core';
import {AppInjectModule} from "./injectexamples/app-inject.module";
import {InjectexamplesComponent} from './injectexamples/injectexamples.component';
@NgModule({
  imports: [
    AppInjectModule,
  ],
  bootstrap: [InjectexamplesComponent]
})
export class ExamplesmodulesModule1 {}
```

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 5-3 所示。

图 5-3　成功运行程序后在浏览器地址栏中输入 localhost:4200 的结果

习题 5

一、简答题

1. 简述对内容投影的理解。
2. 简述对视图封装的理解。
3. 简述对依赖注入的理解。

二、实验题

1. 实现内容投影的应用开发。
2. 实现视图封装的应用开发。
3. 实现依赖注入的应用开发。