架构概览

Angular 是一个用 HTML 和 TypeScript 构建客户端应用的平台与框架。 Angular 本身使用 TypeScript 写成的。它将核心功能和可选功能作为一组 TypeScript 库进行实现，你可以把它们导入你的应用中。

Angular 的基本构造块是 **NgModule**，它为**组件**提供了编译的上下文环境。 NgModule 会把相关的代码收集到一些功能集中。Angular 应用就是由一组 NgModule 定义出的。 应用至少会有一个用于引导应用的**根模块**，通常还会有很多**特性模块**。

- 组件定义**视图**。视图是一组可见的屏幕元素，Angular 可以根据你的程序逻辑和数据来选择和修改它们。 每个应用都至少有一个根组件。
- 组件使用**服务**。服务会提供那些与视图不直接相关的功能。服务提供商可以作为**依赖**被**注入**到组件中，这能让你的代码更加模块化、可复用，而且高效。

组件和服务都是简单的类，这些类使用**装饰器**来标出它们的类型，并提供元数据以告知 Angular 该如何使用它们。

- 组件类的元数据将组件类和一个用来定义视图的**模板**关联起来。 模板把普通的 HTML 和**指令**与**绑定标记（markup）**组合起来，这样 Angular 就可以在呈现 HTML 之前先修改这些 HTML。
- 服务的元数据提供了一些信息，Angular 要用这些信息来让组件可以通过**依赖注入（DI）**使用该服务。

应用的组件通常会定义很多视图，并进行分级组织。 Angular 提供了 `Router` 服务来帮助你定义视图之间的导航路径。 路由器提供了先进的浏览器内导航功能。

模块

Angular 定义了 `NgModule`，它和 JavaScript（ES2015） 的模块不同而且有一定的互补性。 NgModule 为一个组件集声明了编译的上下文环境，它专注于某个应用领域、某个工作流或一组紧密相关的能力。 NgModule 可以将其组件和一组相关代码（如服务）关联起来，形成功能单元。

每个 Angular 应用都有一个**根模块**，通常命名为 `AppModule`。根模块提供了用来启动应用的引导机制。 一个应用通常会包含很多功能模块。

像 JavaScript 模块一样，NgModule 也可以从其它 NgModule 中导入功能，并允许导出它们自己的功能供其它 NgModule 使用。 比如，要在你的应用中使用路由器（Router）服务，就要导入 `Router` 这个 NgModule。

把你的代码组织成一些清晰的功能模块，可以帮助管理复杂应用的开发工作并实现可复用性设计。 另外，这项技术还能让你获得**惰性加载**（也就是按需加载模块）的优点，以尽可能减小启动时需要加载的代码体积。

> 更深入的套路，参见模块简介。

组件

每个 Angular 应用都至少有一个组件，也就是**根组件**，它会把组件树和页面中的 DOM 连接起来。 每个组件都会定义一个类，其中包含应用的数据和逻辑，并与一个 HTML **模板**相关联，该模板定义了一个供目标环境下显示的视图。

`@Component` 装饰器表明紧随它的那个类是一个组件，并提供模板和该组件专属的元数据。

> 装饰器是一些用于修饰 JavaScript 类的函数。Angular 定义了许多装饰器，这些装饰器会把一些特定种类的元数据附加到类上，以便 Angular 了解这些这些类的含义以及该如何使用它们。
>
> 到网上学习关于装饰器的更多知识。

模板、指令和数据绑定

模板会把 HTML 和 Angular 的标记（markup）组合起来，这些标记可以在 HTML 元素显示出来之前修改它们。 模板中的**指令**会提供程序逻辑，而**绑定标记**会把你应用中的数据和 DOM 连接在一起。

- **事件绑定**让你的应用可以通过更新应用的数据来响应目标环境下的用户输入。
- **属性绑定**让你将从应用数据中计算出来的值插入到 HTML 中。

在视图显示出来之前，Angular 会先根据你的应用数据和逻辑来运行模板中的指令并解析绑定表达式，以修改 HTML 元素和 DOM。 Angular 支持**双向数据绑定**，这意味着 DOM 中发生的变化（比如用户的选择）同样可以反映回你的程序数据中。

你的模板也可以用**管道**转换要显示的值以增强用户体验。比如，可以使用管道来显示适合用户所在地区的日期和货币格式。 Angular 为一些通用的转换提供了预定义管道，你还可以定义自己的管道。

> 要了解对这些概念的深入讨论，参见组件介绍。

服务于依赖注入

对于与特定视图无关并希望跨组件共享的数据或逻辑，可以创建**服务**类。 服务类的定义通常紧跟在 “@Injectable” 装饰器之后。该装饰器提供的元数据可以让你的服务作为依赖**被注入到**客户组件中。

依赖注入（或 DI）让你可以保持组件类的精简和高效。有了 DI，组件就不用从服务器获取数据、验证用户输入或直接把日志写到控制台，而是会把这些任务委托给服务。

> 更深入的讨论，参见服务和 DI 简介。

路由

Angular 的 `Router` 模块提供了一个服务，它可以让你定义在应用的各个不同状态和视图层次结构之间导航时要使用的路径。 它的工作模型基于人们熟知的浏览器导航约定：

- 在地址栏输入 URL，浏览器就会导航到相应的页面。
- 在页面中点击链接，浏览器就会导航到一个新页面。
- 点击浏览器的前进和后退按钮，浏览器就会在你的浏览历史中向前或向后导航。

不过路由器会把类似 URL 的路径映射到视图而不是页面。 当用户执行一个动作时（比如点击链接），本应该在浏览器中加载一个新页面，但是路由器拦截了浏览器的这个行为，并显示或隐藏一个视图层次结构。

如果路由器认为当前的应用状态需要某些特定的功能，而定义此功能的模块尚未加载，路由器就会按需**惰性加载**此模块。

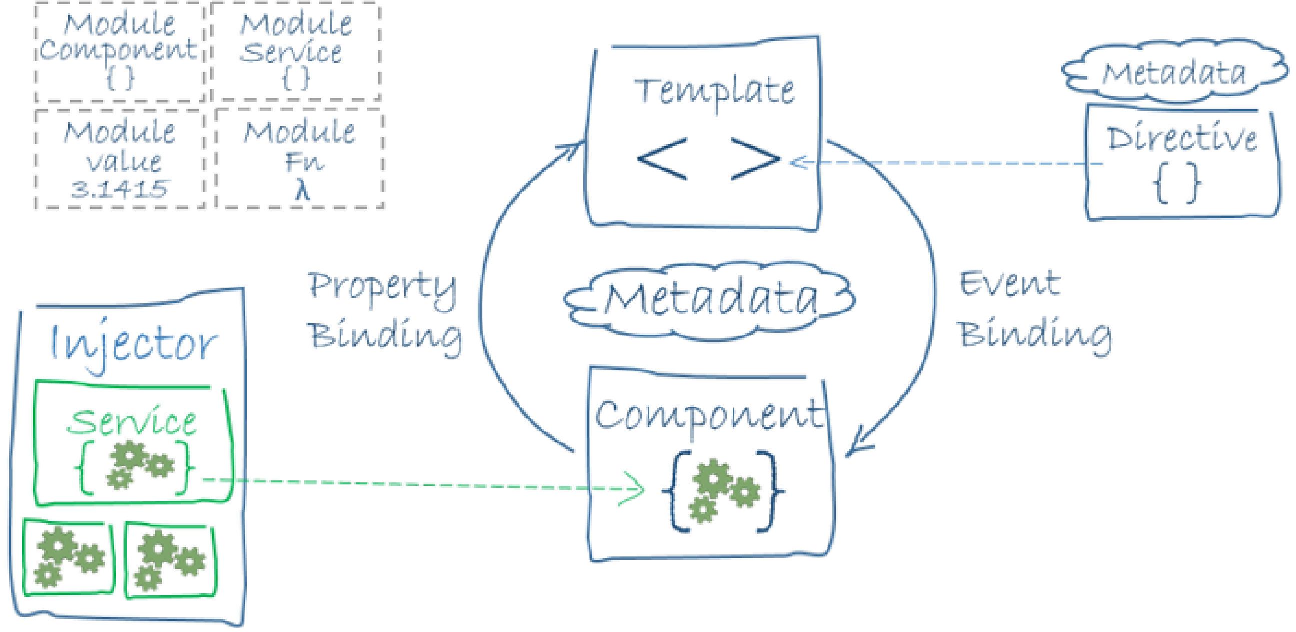
路由器会根据你应用中的导航规则和数据状态来拦截 URL。 当用户点击按钮、选择下拉框或收到其它任何来源的输入时，你可以导航到一个新视图。 路由器会在浏览器的历史日志中记录这个动作，所以前进和后退按钮也能正常工作。

要定义导航规则，你就要把**导航路径**和你的组件关联起来。 路径（path）使用类似 URL 的语法来和程序数据整合在一起，就像模板语法会把你的视图和程序数据整合起来一样。 然后你就可以用程序逻辑来决定要显示或隐藏哪些视图，以根据你制定的访问规则对用户的输入做出响应。

> 更深入的讨论，参见路由与导航。

接下来呢?

你已经学完了 Angular 应用的主要构造块的基础知识。 下面这张图展示了这些基础部分之间是如何关联起来的。



- 组件和模板共同定义了 Angular 的视图。
    - 组件类上的装饰器为其添加了元数据，其中包括指向相关模板的指针。
    - 组件模板中的指令和绑定标记会根据程序数据和程序逻辑修改这些视图。
- 依赖注入器会为组件提供一些服务，比如路由器服务就能让你定义如何在视图之间导航。

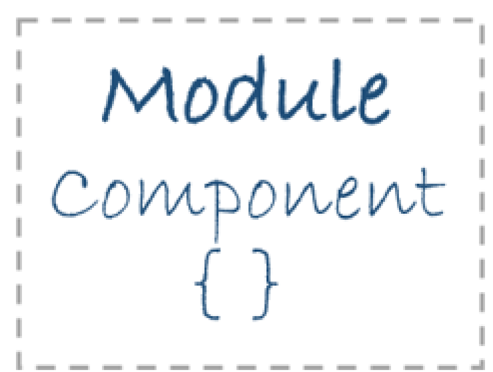
这些主题的详情在下列页面中有介绍：

- 模块
- 组件
    - 模板
    - 元数据
    - 绑定
    - 指令
    - 管道
- 服务于依赖注入

> 注意，这些页面中的代码都在在线例子 / 下载范例中。

当你熟悉了这些基础构造块之后，就可以在本文档中进一步查看它们的详情了。 要学习能帮你构建和发布应用的更多工具和技巧，参见后续步骤。

模块简介



Angular 应用是模块化的，它拥有自己的模块化系统，称作 **NgModule**。 一个 NgModule 就是一个容器，用于存放一些内聚的代码块，这些代码块专注于某个应用领域、某个工作流或一组紧密相关的功能。 它可以包含一些组件、服务提供商或其它代码文件，其作用域由包含它们的 NgModule 定义。 它还可以导入一些由其它模块中导出的功能，并导出一些指定的功能供其它 NgModule 使用。

每个 Angular 应用都至少有一个 NgModule 类，也就是根模块，它习惯上命名为 `AppModule`，并位于一个名叫 `app.module.ts` 的文件中。**引导**这个根模块就可以启动你的应用。

虽然小型的应用可能只有一个 NgModule，不过大多数应用都会有很多**特性模块**。应用的**根模块**之所以叫根模块，是因为它可以包含任意深度的层次化子模块。

`@NgModule` 元数据

NgModule 是一个带有 `@NgModule` 装饰器的类。`@NgModule` 装饰器是一个函数，它接受一个元数据对象，该对象的属性用来描述这个模块。其中最重要的属性如下。

- `declarations`（可声明对象表） —— 那些属于本 NgModule 的组件、**指令**、**管道**。
- `exports`（导出表） —— 那些能在其它模块的**组件模板**中使用的可声明对象的子集。
- `imports`（导入表） —— 那些导出了**本**模块中的组件模板所需的类的其它模块。
- `providers` —— 本模块向全局服务中贡献的那些服务的创建器。 这些服务能被本应用中的任何部分使用。 （你也可以在组件级别指定服务提供商，这通常是首选方式。）
- `bootstrap` —— 应用的主视图，称为**根组件**。它是应用中所有其它视图的宿主。只有**根模块**才应该设置这个 `bootstrap` 属性。

下面是一个简单的根 NgModule 定义：

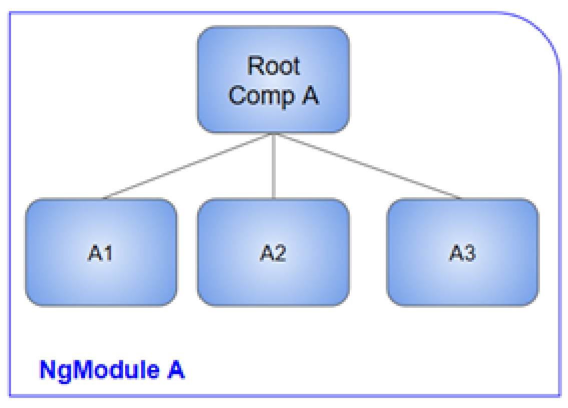
src/app/app.module.ts

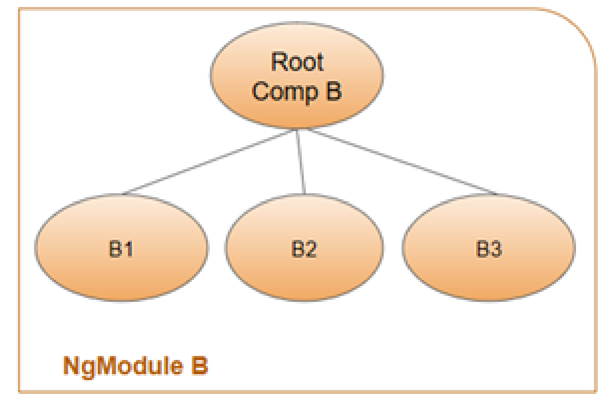
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

> 把 `AppComponent` 放到 `exports` 中只是为了演示导出的语法，这在本例子中实际上是没必要的。 根模块没有任何理由**导出**任何东西，因为其它模块永远不需要**导入**根模块。

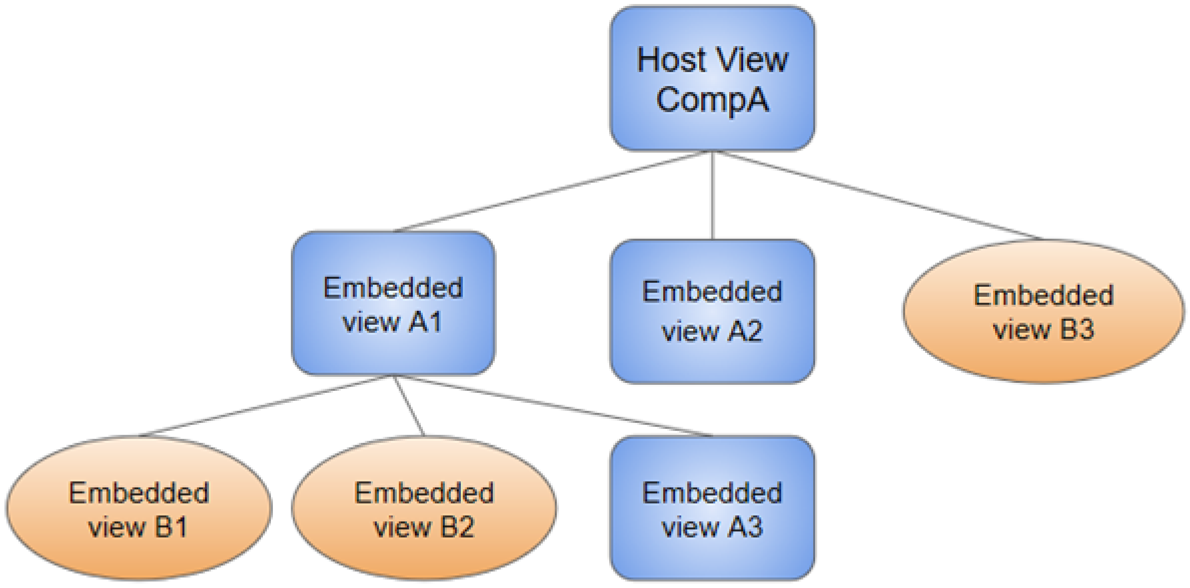
NgModule 和组件

NgModule 为其中的组件提供了一个**编译上下文环境**。根模块总会有一个根组件，并在引导期间创建它。 但是，任何模块都能包含任意数量的其它组件，这些组件可以通过路由器加载，也可以通过模板创建。那些属于这个 NgModule 的组件会共享同一个编译上下文环境。





组件及其模板共同定义**视图**。组件还可以包含**视图层次结构**，它能让你定义任意复杂的屏幕区域，可以将其作为一个整体进行创建、修改和销毁。 一个视图层次结构中可以混合使用由不同 NgModule 中的组件定义的视图。 这种情况很常见，特别是对一些 UI 库来说。



当你创建一个组件时，它直接与一个叫做**宿主视图**的视图关联起来。 宿主视图可以是视图层次结构的根，该视图层次结构可以包含一些**内嵌视图**，这些内嵌视图又是其它组件的宿主视图。 这些组件可以位于相同的 NgModule 中，也可以从其它 NgModule 中导入。 树中的视图可以嵌套到任意深度。

> 视图的这种层次结构是 Angular 在 DOM 和应用数据中检测与响应变更时的关键因素。

NgModule 和 JavaScript 的模块

NgModule 系统与 JavaScript（ES2015）用来管理 JavaScript 对象的模块系统不同，而且也没有直接关联。 这两种模块系统不同但**互补**。你可以使用它们来共同编写你的应用。

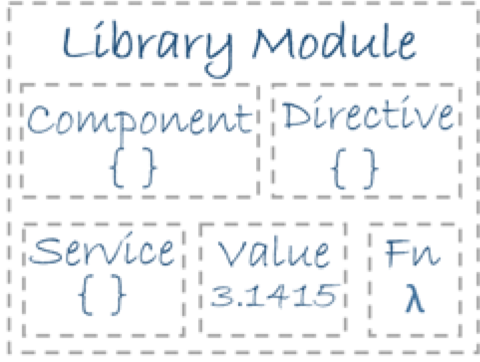
JavaScript 中，每个**文件**是一个模块，文件中定义的所有对象都从属于那个模块。 通过 `export` 关键字，模块可以把它的某些对象声明为公共的。 其它 JavaScript 模块可以使用**import 语句**来访问这些公共对象。

```
import { NgModule }     from '@angular/core';
import { AppComponent } from './app.component';
```

```
export class AppModule { }
```

> 学习更多关于 JavaScript 模块的知识。

Angular 自带的库



Angular 自带了一组 JavaScript 模块，你可以把它们看成库模块。每个 Angular 库的名称都带有 `@angular` 前缀。 使用 `npm` 包管理器安装它们，并使用 JavaScript 的 `import` 语句导入其中的各个部分。

例如，象下面这样，从 `@angular/core` 库中导入 `Component` 装饰器：

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

还可以使用 JavaScript 的导入语句从 Angular **库**中导入 Angular **模块**：

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

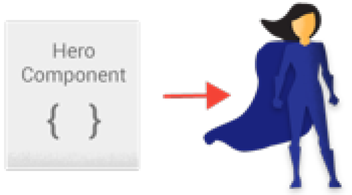
在上面这个简单的根模块范例中，应用的根模块需要来自 `BrowserModule` 中的素材。要访问这些素材，就要把它加入 `@NgModule` 元数据的 `imports` 中，代码如下：

```
imports:      [ BrowserModule ],
```

通过这种方式，你可以**同时**使用 Angular 和 JavaScript 的这两种模块系统。 虽然这两种模块系统容易混淆（它们共享了同样的词汇 `import` 和`export`），不过只要多用用你就会熟悉它们各自的语境了。

> 更多信息，参见 NgModules。

组件简介



组件控制屏幕上被称为**视图**的一小片区域。比如，教程中的下列视图都是由一个个组件所定义和控制的：

- 带有导航链接的应用根组件。
- 英雄列表。
- 英雄编辑器。

你在类中定义组件的应用逻辑，为视图提供支持。 组件通过一些由属性和方法组成的 API 与视图交互。

比如， `HeroListComponent` 有一个 `heroes` 属性，它会返回一个从服务中取到的英雄数组。 `HeroListComponent` 还有一个 `selectHero()` 方法，当用户从列表中选择一个英雄时，它会设置 `selectedHero` 属性的值。

src/app/hero-list.component.ts (class)

```
export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

当用户在应用中穿行时，Angular 就会创建、更新、销毁一些组件。 你的应用可以通过一些可选的生命周期钩子（比如`ngOnInit()`）来在每个特定的时机采取行动。

组件的元数据

`@Component` 装饰器会指出紧随其后的那个类是个组件类，并为其指定元数据。 在下面的范例代码中，你可以看到 `HeroListComponent` 只是一个普通类，完全没有 Angular 特有的标记或语法。 直到给它加上了 `@Component` 装饰器，它才变成了组件。

组件的元数据告诉 Angular 到哪里获取它需要的主要构造块，以创建和展示这个组件及其视图。 具体来说，它把一个**模板**（无论是直接内联在代码中还是引用的外部文件）和该组件关联起来。 该组件及其模板，共同描述了一个**视图**。

除了包含或指向模板之外，`@Component` 的元数据还会配置要如何在 HTML 中引用该组件，以及该组件需要哪些服务等等。

下面的例子中就是 `HeroListComponent` 的基础元数据：

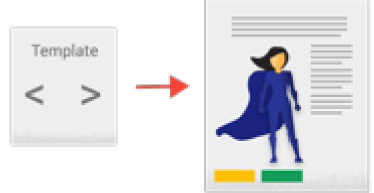
src/app/hero-list.component.ts (metadata)

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
/* . . . */
}
```

这个例子展示了一些最常用的 `@Component` 配置选项：

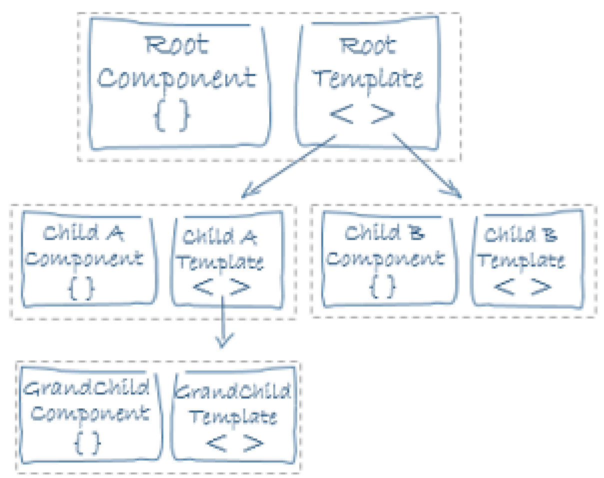
- `selector`：是一个 CSS 选择器，它会告诉 Angular，一旦在模板 HTML 中找到了这个选择器对应的标签，就创建并插入该组件的一个实例。 比如，如果应用的 HTML 中包含 `<app-hero-list></app-hero-list>`，Angular 就会在这些标签中插入一个 `HeroListComponent` 实例的视图。
- `templateUrl`：该组件的 HTML 模板文件相对于这个组件文件的地址。 另外，你还可以用 `template` 属性的值来提供内联的 HTML 模板。 这个模板定义了该组件的**宿主视图**。
- `providers` 是当前组件所需的依赖注入提供商的一个数组。在这个例子中，它告诉 Angular，该组件的构造函数需要一个 `HeroService` 实例，以获取要显示的英雄列表。

模板与视图



你要通过组件的配套模板来定义其视图。模板就是一种 HTML，它会告诉 Angular 如何渲染该组件。

视图通常会分层次进行组织，让你能以 UI 分区或页面为单位进行修改、显示或隐藏。 与组件直接关联的模板会定义该组件的**宿主视图**。该组件还可以定义一个**带层次结构的视图**，它包含一些**内嵌的视图**作为其它组件的宿主。



带层次结构的视图可以包含同一模块（NgModule）中组件的视图，也可以（而且经常会）包含其它模块中定义的组件的视图。

模板语法

模板很像标准的 HTML，但是它还包含 Angular 的模板语法，这些模板语法可以根据你的应用逻辑、应用状态和 DOM 数据来修改这些 HTML。 你的模板可以使用**数据绑定**来协调应用和 DOM 中的数据，使用**管道**在显示出来之前对其进行转换，使用**指令**来把程序逻辑应用到要显示的内容上。

比如，下面是本教程中 `HeroListComponent` 的模板：

src/app/hero-list.component.html

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero"></app-hero-detail>
```

这个模板使用了典型的 HTML 元素，比如 `<h2>` 和 `<p>`，还包括一些 Angular 的模板语法元素，如 `*ngFor`，`{{hero.name}}`，`click`、`[hero]` 和 `<app-hero-detail>`。这些模板语法元素告诉 Angular 该如何根据程序逻辑和数据在屏幕上渲染 HTML。

- `*ngFor` 指令告诉 Angular 在一个列表上进行迭代。
- `{{hero.name}}`、`(click)` 和 `[hero]` 把程序数据绑定到及绑定回 DOM，以响应用户的输入。更多内容参见稍后的数据绑定部分。
- 模板中的 `<app-hero-detail>` 标签是一个代表新组件 `HeroDetailComponent` 的元素。`HeroDetailComponent`（代码略）是 `HeroListComponent` 的一个子组件，它定义了英雄详情视图。 注意观察像这样的自定义组件是如何与原生 HTML 元素无缝的混合在一起的。

数据绑定

如果没有框架，你就要自己负责把数据值推送到 HTML 控件中，并把来自用户的响应转换成动作和对值的更新。 手动写这种数据推拉逻辑会很枯燥、容易出错，难以阅读 —— 用过 jQuery 的程序员一定深有体会。

Angular 支持**双向数据绑定**，这是一种对模板中的各个部件与组件中的各个部件进行协调的机制。 往模板 HTML 中添加绑定标记可以告诉 Angular 该如何连接它们。

下图显示了数据绑定标记的四种形式。每种形式都有一个方向 —— 从组件到 DOM、从 DOM 到组件或双向。

{{value}}

[property] = "value"

DOM

COMPONENT

(event) = "handler"

[(ng-model)] = "property"

这个来自 `HeroListComponent` 模板中的例子使用了其中的三种形式：

src/app/hero-list.component.html (binding)

```
<li>{{hero.name}}</li>
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
<li (click)="selectHero(hero)"></li>
```

- `{{hero.name}}`**插值表达式**在 `<li>` 标签中显示组件的 `hero.name` 属性的值。
- `[hero]`**属性绑定**把父组件 `HeroListComponent` 的 `selectedHero` 的值传到子组件 `HeroDetailComponent` 的 `hero` 属性中。
- 当用户点击某个英雄的名字时，`(click)` **事件绑定**会调用组件的 `selectHero` 方法。

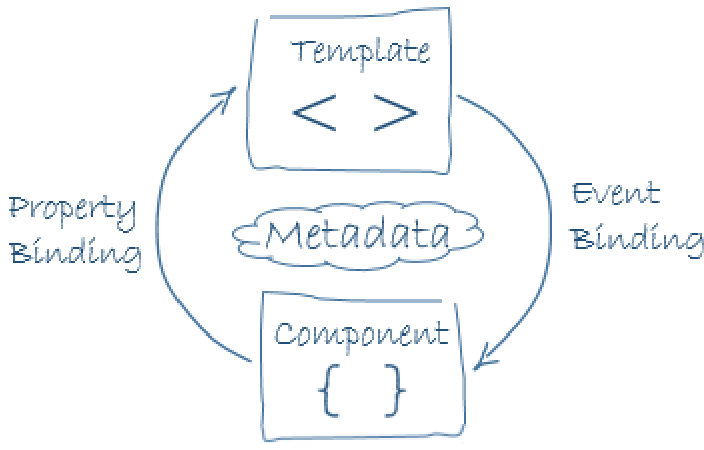
第四种重要的绑定形式是双向数据绑定，它把属性绑定和事件绑定组合成一种单独的写法。下面这个来自 `HeroDetailComponent` 模板中的例子通过 `ngModel` 指令使用了双向数据绑定：

src/app/hero-detail.component.html (ngModel)

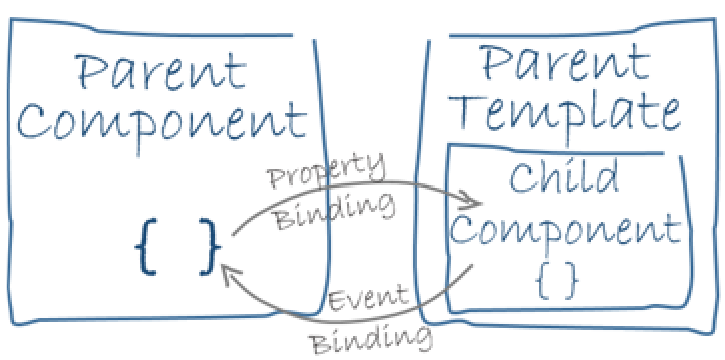
```
<input [(ngModel)]="hero.name">
```

在双向绑定中，数据属性值通过属性绑定从组件流到输入框。用户的修改通过事件绑定流回组件，把属性值设置为最新的值。

Angular 在每个 JavaScript 事件循环中处理**所有的**数据绑定，它会从组件树的根部开始，递归处理全部子组件。



数据绑定在模板及其组件之间的通讯中扮演了非常重要的角色，它对于父组件和子组件之间的通讯也同样重要。



管道

Angular 的管道可以让你在模板中声明显示值的转换逻辑。 带有 `@Pipe` 装饰器的类中会定义一个转换函数，用来把输入值转换成供视图显示用的输出值。

Angular 自带了很多管道，比如 date 管道和 currency 管道，完整的列表参见 Pipes API 列表。你也可以自己定义一些新管道。

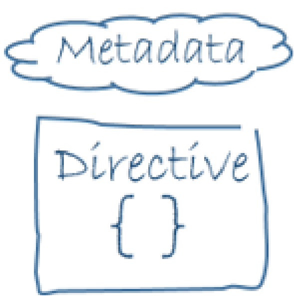
要在 HTML 模板中指定值的转换方式，请使用 管道操作符 (|)。

`{{interpolated_value | pipe_name}}`

你可以把管道串联起来，把一个管道函数的输出送给另一个管道函数进行转换。 管道还能接收一些参数，来控制它该如何进行转换。比如，你可以把要使用的日期格式传给 `date` 管道：

```html
<!-- Default format: output 'Jun 15, 2015'-->
 <p>Today is {{today | date}}</p>
<!-- fullDate format: output 'Monday, June 15, 2015'-->
<p>The date is {{today | date:'fullDate'}}</p>
 <!-- shortTime format: output '9:43 AM'-->
 <p>The time is {{today | date:'shortTime'}}</p>
```

指令



Angular 的模板是**动态的**。当 Angular 渲染它们的时候，会根据**指令**给出的指示对 DOM 进行转换。 指令就是一个带有 `@Directive` 装饰器的类。

组件从技术角度上说就是一个指令，但是由于组件对 Angular 应用来说非常独特、非常重要，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。

除组件外，还有两种指令：**结构型指令**和**属性型指令**。和组件一样，指令的元数据把指令类和一个 `selector` 关联起来，`selector` 用来把该指令插入到 HTML 中。 在模板中，指令通常作为属性出现在元素标签上，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。

结构型指令

结构型指令通过添加、移除或替换 DOM 元素来修改布局。 这个范例模板使用了两个内置的结构型指令来为要渲染的视图添加程序逻辑：

src/app/hero-list.component.html (structural)

```html
<li *ngFor="let hero of heroes"></li>
<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
```

- `*ngFor` 是一个迭代器，它要求 Angular 为 `heroes` 列表中的每个 `<li>` 渲染出一个 `<li>`。
- `*ngIf` 是个条件语句，只有当选中的英雄存在时，它才会包含 `HeroDetail` 组件。

属性型指令

属性型指令会修改现有元素的外观或行为。 在模板中，它们看起来就像普通的 HTML 属性一样，因此得名“属性型指令”。

`ngModel` 指令就是属性型指令的一个例子，它实现了双向数据绑定。 `ngModel` 修改现有元素（一般是 `<input>`）的行为：设置其显示属性值，并响应 change 事件。

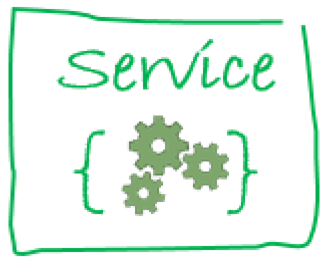
src/app/hero-detail.component.html (ngModel)

```
<input [(ngModel)]="hero.name">
```

Angular 还有很多预定义指令，它们或者修改布局结构（比如 ngSwitch），或者修改 DOM 元素和组件的某些方面（比如 ngStyle 和 ngClass）。

你还可以写自己的指令。像 `HeroListComponent` 这样的组件就是一种自定义指令，你还可以创建自定义的结构型指令和属性型指令。

服务与依赖注入简介



服务是一个广义的概念，它包括应用所需的任何值、函数或特性。狭义的服务是一个明确定义了用途的类。它应该做一些具体的事，并做好。

Angular 把组件和服务区分开，以提高模块性和复用性。

- 通过把组件中和视图有关的功能与其他类型的处理分离开，你可以让组件类更加精简、高效。 理想情况下，组件的工作只管用户体验，而不用顾及其它。 它应该提供用于数据绑定的属性和方法，以便作为视图（由模板渲染）和应用逻辑（通常包含一些模型的概念）的中介者。
- 组件不应该定义任何诸如从服务器获取数据、验证用户输入或直接往控制台中写日志等工作。 而要把这些任务委托给各种服务。通过把各种处理任务定义到可注入的服务类中，你可以让它可以被任何组件使用。 通过在不同的环境中注入同一种服务的不同提供商，你还可以让你的应用更具适应性。

Angular 不会**强制**遵循这些原则。它只会通过**依赖注入**让你能更容易地将应用逻辑分解为服务，并让这些服务可用于各个组件中。

服务范例

下面是一个服务类的范例，用于把日志记录到浏览器的控制台：

src/app/logger.service.ts (class)

```
export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}
```

服务也可以依赖其它服务。比如，这里的 `HeroService` 就依赖于 `Logger` 服务，它还用 `BackendService` 来获取英雄数据。`BackendService` 还可能再转而依赖 `HttpClient` 服务来从服务器异步获取英雄列表。

src/app/hero.service.ts (class)

```
export class HeroService {
  private heroes: Hero[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes;
  }
}
```

依赖注入（dependency injection）

组件是服务的消费者，也就是说，你可以把一个服务**注入**到组件中，让组件类得以访问该服务类。

在 Angular 中，要把一个类定义为服务，就要用 `@Injectable` 装饰器来提供元数据，以便让 Angular 可以把它作为**依赖**注入到组件中。

同样，也要使用 `@Injectable` 装饰器来表明一个组件或其它类（比如另一个服务、管道或 NgModule）**拥有**一个依赖。 依赖并不必然是服务，它也可能是函数或值等等。

依赖注入（通常简称 DI）被引入到 Angular 框架中，并且到处使用它，来为新建的组件提供所需的服务或其它东西。

- **注入器**是主要的机制。你不用自己创建 Angular 注入器。Angular 会在启动过程中为你创建全应用级注入器。
- 该注入器维护一个包含它已创建的依赖实例的**容器**，并尽可能复用它们。
- **提供商**是一个创建依赖的菜谱。对于服务来说，它通常就是这个服务类本身。你在应用中要用到的任何类都必须使用该应用的注入器注册一个提供商，以便注入器可以使用它来创建新实例。

当 Angular 创建组件类的新实例时，它会通过查看该组件类的构造函数，来决定该组件依赖哪些服务或其它依赖项。 比如 `HeroListComponent` 的构造函数中需要 `HeroService`：

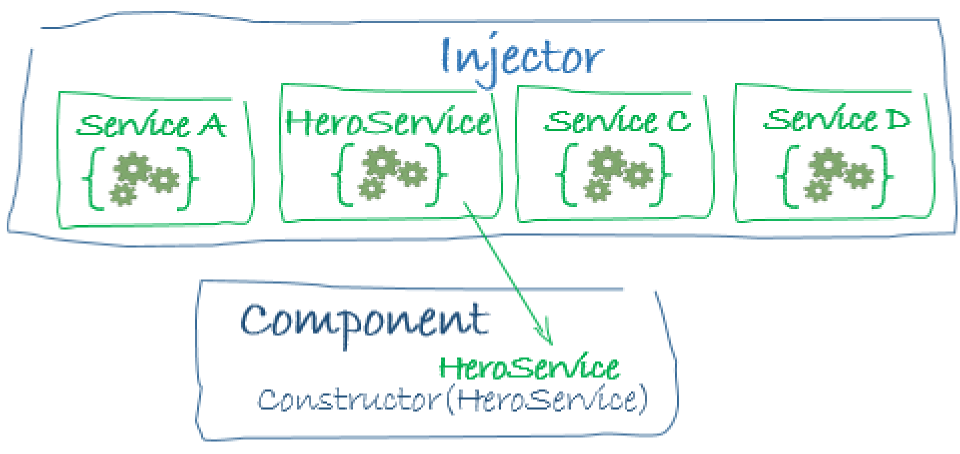
src/app/hero-list.component.ts (constructor)

```
constructor(private service: HeroService) { }
```

当 Angular 发现某个组件依赖某个服务时，它会首先检查是否该注入器中已经有了那个服务的任何现有实例。如果所请求的服务尚不存在，注入器就会使用以前注册的服务提供商来制作一个，并把它加入注入器中，然后把该服务返回给 Angular。

当所有请求的服务已解析并返回时，Angular 可以用这些服务实例为参数，调用该组件的构造函数。

`HeroService` 的注入过程如下所示：



提供服务

对于要用到的任何服务，你必须至少注册一个**提供商**。你可以在模块中或者组件中注册这些提供商。

- 当你往根模块中添加服务提供商时，服务的同一个实例会服务于你应用中的所有组件。

src/app/app.module.ts (module providers)

```
providers: [
  BackendService,
  HeroService,
  Logger
],
```

- 当你在组件级注册提供商时，你会为该组件的每一个新实例提供该服务的一个新实例。 要在组件级注册，就要在 `@Component` 元数据的 `providers` 属性中注册服务提供商。

src/app/hero-list.component.ts (component providers)

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
```

要了解更多细节，请参见依赖注入一节。

后续步骤：工具与技巧

一旦你理解了这些基本构造块，就可以开始学习更多能帮你开发和交付 Angular 应用的特性和工具了。Angular 提供了本文档中所包含的很多特性和服务。

响应式编程工具

- 生命周期钩子：通过实现生命周期钩子接口，可以窃听组件生命周期中的一些关键时刻 —— 从创建到销毁。
- 可观察对象（Observable）和事件处理：如何在组件和服务中使用可观察对象来发布和订阅任意类型的消息，比如用户交互事件和异步操作结果。

客户端与服务器的交互工具

- HTTP：用 HTTP 客户端与服务器通讯，以获取数据、保存数据或执行服务端动作。
- 服务端渲染：Angular Universal 会通过服务端渲染（SSR）技术在服务器上生成静态的应用页面。 这让你可以在服务器上运行 Angular 应用，以提升性能并在手机或低功耗设备上快速显示首屏，并为 Web 爬虫提供帮助（SEO）。
- Service Worker：Service Worker 是一个运行在浏览器中并为应用管理缓存的脚本。 Service Worker 的功能类似于网络代理。它们会拦截发出的 HTTP 请求，如果存在已缓存的响应，则直接返回它。通过使用 Service Worker 来减轻对网络的依赖，你可以显著提升用户体验。

特定领域的库

- 动画：使用 Angular 的动画库，你可以让组件支持动画行为，而不用深入了解动画技术或 CSS。
- Forms：通过基于 HTML 的验证和脏数据检查，来支持复杂的数据输入场景。

为开发周期提供支持

- 测试平台：对应用的各个部件运行单元测试，让它们好像在和 Angular 框架交互一样。
- 国际化：Angular 的国际化工具可以帮助你让应用可用于多种语言中。
- 编译：Angular 为开发环境提供了 JIT（即时）编译方式，为生产环境提供了 AOT（预先）编译方式。
- 安全指南：学习 Angular 对常见 Web 应用的弱点和工具（比如跨站脚本攻击）提供的内置防护措施。

环境搭建和发布工具

- 搭建本地开发环境：学习如何搭建用来开发《快速起步》的新项目。
- 安装：Angular CLI 应用和 Angular 本身依赖于一些由库提供的特性和功能，这些库都是以 npm 包的形式发布的。
- TypeScript 配置：TypeScript 是 Angular 应用开发的主要语言。
- 浏览器支持：学习如何让你的应用能和各种浏览器兼容。
- 发布：学习把你的应用发布到远端服务器的技巧。

显示数据

在 Angular 中最典型的数据显示方式，就是把 HTML 模板中的控件绑定到 Angular 组件的属性。

本章中，你将创建一个带英雄列表的组件。 你将显示英雄名字的列表，并根据条件在列表下方显示一条消息。

最终的用户界面是这样的：

Tour of Heroes

My favorite hero is: Windstorm

Heroes:

- Windstorm
- Bombasto
- Magneta
- Tornado

There are many heroes!

> 这个在线例子 / 下载范例演示了本章中描述的所有语法和代码片段。

使用插值表达式显示组件属性

要显示组件的属性，最简单的方式就是通过插值表达式 (interpolation) 来绑定属性名。 要使用插值表达式，就把属性名包裹在双花括号里放进视图模板，如 `{{myHero}}`。

按照快速起步的说明，创建一个新项目，名为`displaying-data`。

删除 `app.component.html` 文件，这个范例中不再需要它了。

然后，到 `app.component.ts` 文件中修改组件的模板和代码。

修改完之后，它应该是这样的：

src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}</h1>
    <h2>My favorite hero is: {{myHero}}</h2>
    `
})
export class AppComponent {
  title = 'Tour of Heroes';
  myHero = 'Windstorm';
}
```

再把两个属性 `title` 和 `myHero` 添加到之前空白的组件中。

修改完的模板会使用双花括号形式的插值表达式来显示这两个模板属性：

src/app/app.component.ts (template)

```
template: `
  <h1>{{title}}</h1>
  <h2>My favorite hero is: {{myHero}}</h2>
  `
```

> 模板是包在 ECMAScript 2015 反引号 (`` ` ``) 中的一个多行字符串。 反引号 (`` ` ``) — 注意，不是单引号 (') — 允许把一个字符串写在多行上， 使 HTML 模板更容易阅读。

Angular 自动从组件中提取 `title` 和 `myHero` 属性的值，并且把这些值插入浏览器中。当这些属性发生变化时，Angular 就会自动刷新显示。

> 严格来说，“重新显示”是在某些与视图有关的异步事件之后发生的，例如，按键、定时器完成或对 HTTP 请求的响应。

注意，你没有调用 new 来创建 `AppComponent` 类的实例，是 Angular 替你创建了它。那么它是如何创建的呢？

注意 `@Component` 装饰器中指定的 CSS 选择器 `selector`，它指定了一个叫 `my-app` 的元素。 该元素是 `index.html` 的 `body` 里的占位符。

src/index.html (body)

```
<body>
  <app-root></app-root>
</body>
```

当你通过 `main.ts` 中的 `AppComponent` 类启动时，Angular 在 `index.html` 中查找一个 `<app-root>` 元素， 然后实例化一个 `AppComponent`，并将其渲染到 `<app-root>` 标签中。

运行应用。它应该显示出标题和英雄名：

Tour of Heroes

My favorite hero is: Windstorm

回顾一下前面所做的决定，看看还有哪些其它选择。

内联 (inline) 模板还是模板文件?

你可以在两种地方存放组件模板。 你可以使用 `template` 属性把它定义为**内联**的，或者把模板定义在一个独立的 HTML 文件中， 再通过 `@Component` 装饰器中的 `templateUrl` 属性， 在组件元数据中把它链接到组件。

到底选择内联 HTML 还是独立 HTML 取决于个人喜好、具体状况和组织级策略。 上面的应用选择内联 HTML ，是因为模板很小，而且没有额外的 HTML 文件显得这个演示简单些。

无论用哪种风格，模板数据绑定在访问组件属性方面都是完全一样的。

> 默认情况下，Angular CLI 生成组件时会带有模板文件，你可以通过参数覆盖它：
>
> ```
> ng generate component hero -it
> ```

使用构造函数还是变量初始化?

虽然这个例子使用了变量赋值的方式初始化组件，你还可以使用构造函数来声明和初始化属性。

```
export class AppCtorComponent {
  title: string;
  myHero: string;

  constructor() {
    this.title = 'Tour of Heroes';
    this.myHero = 'Windstorm';
  }
}
```

为了让本应用更加简短，它采用了更简单的“变量赋值”风格。

使用 **ngFor** 显示数组属性

要显示一个英雄列表，先向组件中添加一个英雄名字数组，然后把 `myHero` 重定义为数组中的第一个名字。

src/app/app.component.ts (class)

```
export class AppComponent {
  title = 'Tour of Heroes';
  heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
  myHero = this.heroes[0];
}
```

接着，在模板中使用 Angular 的 `ngFor` 指令来显示 `heroes` 列表中的每一项。

src/app/app.component.ts (template)

```
template: `
  <h1>{{title}}</h1>
  <h2>My favorite hero is: {{myHero}}</h2>
  <p>Heroes:</p>
  <ul>
    <li *ngFor="let hero of heroes">
      {{ hero }}
    </li>
  </ul>
`
```

这个界面使用了由 `<ul>` 和 `<li>` 标签组成的无序列表。`<li>` 元素里的 `*ngFor` 是 Angular 的“迭代”指令。 它将 `<li>` 元素及其子级标记为“迭代模板”:

src/app/app.component.ts (li)

```
<li *ngFor="let hero of heroes">
  {{ hero }}
</li>
```

> 不要忘记 `*ngFor` 中的前导星号 (*)。它是语法中不可或缺的一部分。 更多信息，见模板语法。

注意看 `ngFor` 双引号表达式中的 `hero`，它是一个模板输入变量。 更多模板输入变量的信息，见模板语法中的 微语法 (microsyntax)。

Angular 为列表中的每个条目复制一个 `<li>` 元素，在每个迭代中，把 `hero` 变量设置为当前条目（英雄）。 Angular 把 `hero` 变量作为双花括号插值表达式的上下文。

> 本例中，`ngFor` 用于显示一个“数组”， 但 `ngFor` 可以为任何可迭代的 (iterable) 对象重复渲染条目。

现在，英雄们出现在了一个无序列表中。

Tour of Heroes

My favorite hero is: Windstorm

Heroes:

- Windstorm
- Bombasto
- Magneta
- Tornado

为数据创建一个类

应用代码直接在组件内部直接定义了数据。 作为演示还可以，但它显然不是最佳实践。

现在使用的是到了一个字符串数组的绑定。在真实的应用中，大多是到一个对象数组的绑定。

要将此绑定转换成使用对象，需要把这个英雄名字数组变成 `Hero` 对象数组。但首先得有一个 `Hero` 类。

```
ng generate class hero
```

代码如下：

src/app/hero.ts

```
export class Hero {
  constructor(
    public id: number,
    public name: string) { }
}
```

你定义了一个类，具有一个构造函数和两个属性：`id` 和 `name`。

它可能看上去不像是有属性的类，但它确实有，利用的是 TypeScript 提供的简写形式 —— 用构造函数的参数直接定义属性。

来看第一个参数：

src/app/hero.ts (id)

```
public id: number,
```

这个简写语法做了很多:

- 声明了一个构造函数参数及其类型。
- 声明了一个同名的公共属性。
- 当创建该类的一个实例时，把该属性初始化为相应的参数值。

使用 Hero 类

导入了 `Hero` 类之后，组件的 `heroes` 属性就可以返回一个**类型化的**`Hero` 对象数组了。

src/app/app.component.ts (heroes)

```
heroes = [
  new Hero(1, 'Windstorm'),
  new Hero(13, 'Bombasto'),
  new Hero(15, 'Magneta'),
  new Hero(20, 'Tornado')
];
myHero = this.heroes[0];
```

接着，修改模板。 现在它显示的是英雄的 `id` 和 `name`。 要修复它，只显示英雄的 `name` 属性就行了。

src/app/app.component.ts (template)

```
template: `
  <h1>{{title}}</h1>
  <h2>My favorite hero is: {{myHero.name}}</h2>
  <p>Heroes:</p>
  <ul>
    <li *ngFor="let hero of heroes">
      {{ hero.name }}
    </li>
  </ul>
`
```

显示上还和以前一样，不过代码更清晰了。

通过 NgIf 进行条件显示

有时，应用需要只在特定情况下显示视图或视图的一部分。

来改一下这个例子，如果多于三位英雄，显示一条消息。

Angular 的 `ngIf` 指令会根据一个布尔条件来显示或移除一个元素。 来看看实际效果，把下列语句加到模板的底部：

src/app/app.component.ts (message)

```
<p *ngIf="heroes.length > 3">There are many heroes!</p>
```

> 不要忘了 `*ngIf` 中的前导星号 (*)。它是本语法中不可或缺的一部分。 更多 `ngIf` 和 `*` 的内容，见模板语法中的ngIf。

双引号中的模板表达式 `*ngIf="heros.length > 3"`，外观和行为很象 TypeScript 。 当组件中的英雄列表有三个以上的条目时，Angular 把这个段落添加到 DOM 中，于是消息显示了出来。 更多信息，见模板语法中的模板表达式。

> Angular 并不是在显示和隐藏这条消息，它是在从 DOM 中添加和移除这个段落元素。 这会提高性能，特别是在一些大的项目中有条件地包含或排除一大堆带着很多数据绑定的 HTML 时。

试一下。因为这个数组中有四个条目，所以消息应该显示出来。 回到 `app.component.ts`，从英雄数组中删除或注释掉一个元素。 浏览器应该自动刷新，消息应该会消失。

小结

现在你知道了如何使用：

- 带有双花括号的插值表达式 (interpolation) 来显示一个组件属性。
- 用 ngFor 显示数组。
- 用一个 TypeScript 类来为你的组件描述模型数据并显示模型的属性。
- 用 ngIf 根据一个布尔表达式有条件地显示一段 HTML。

下面是最终的代码：

src/app/app.component.ts *src/app/hero.ts* *src/app/app.module.ts* *main.ts*

```
1.  import { Component } from '@angular/core';
2.
3.  import { Hero } from './hero';
4.
5.  @Component({
6.    selector: 'app-root',
7.    template: `
8.    <h1>{{title}}</h1>
9.    <h2>My favorite hero is: {{myHero.name}}</h2>
10.   <p>Heroes:</p>
11.   <ul>
12.     <li *ngFor="let hero of heroes">
13.       {{ hero.name }}
14.       </li>
15.   </ul>
16.   <p *ngIf="heroes.length > 3">There are many heroes!</p>
17. `
18. })
19. export class AppComponent {
20.   title = 'Tour of Heroes';
21.   heroes = [
22.     new Hero(1, 'Windstorm'),
23.     new Hero(13, 'Bombasto'),
24.     new Hero(15, 'Magneta'),
25.     new Hero(20, 'Tornado')
26.   ];
27.   myHero = this.heroes[0];
28. }
```

模板语法

Angular 应用管理着用户之所见和所为，并通过 Component 类的实例（**组件**）和面向用户的模板来与用户交互。

从使用模型-视图-控制器 (MVC) 或模型-视图-视图模型 (MVVM) 的经验中，很多开发人员都熟悉了组件和模板这两个概念。 在 Angular 中，组件扮演着控制器或视图模型的角色，模板则扮演视图的角色。

这是一篇关于 Angular 模板语言的技术大全。 它解释了模板语言的基本原理，并描述了你将在文档中其它地方遇到的大部分语法。

这里还有很多代码片段用来解释技术点和概念，它们全都在模板语法的在线例子 / 下载范例中。

模板中的 HTML

HTML 是 Angular 模板的语言。几乎所有的 HTML 语法都是有效的模板语法。 但值得注意的例外是 `<script>` 元素，它被禁用了，以阻止脚本注入攻击的风险。（实际上，`<script>` 只是被忽略了。） 参见安全页了解详情。

有些合法的 HTML 被用在模板中是没有意义的。`<html>`、`<body>` 和 `<base>` 元素这个舞台上中并没有扮演有用的角色。剩下的所有元素基本上就都一样用了。

可以通过组件和指令来扩展模板中的 HTML 词汇。它们看上去就是新元素和属性。接下来将学习如何通过数据绑定来动态获取/设置 DOM（文档对象模型）的值。

首先看看数据绑定的第一种形式 —— 插值表达式，它展示了模板的 HTML 可以有多丰富。

插值表达式 ({{...}})

在以前的 Angular 教程中，你遇到过由双花括号括起来的插值表达式，`{{` 和 `}}`。

src/app/app.component.html

```
<p>My current hero is {{currentHero.name}}</p>
```

插值表达式可以把计算后的字符串插入到 HTML 元素标签内的文本或对标签的属性进行赋值。

src/app/app.component.html

```
<h3>
  {{title}}
  <img src="{{heroImageUrl}}" style="height:30px">
</h3>
```

在括号之间的“素材”，通常是组件属性的名字。Angular 会用组件中相应属性的字符串值，替换这个名字。 上例中，Angular 计算 `title` 和 `heroImageUrl` 属性的值，并把它们填在空白处。 首先显示粗体的应用标题，然后显示英雄的图片。

一般来说，括号间的素材是一个模板表达式，Angular 先对它求值，再把它转换成字符串。 下列插值表达式通过把括号中的两个数字相加说明了这一点：

src/app/app.component.html

```
<!-- "The sum of 1 + 1 is 2" -->
<p>The sum of 1 + 1 is {{1 + 1}}</p>
```

这个表达式可以调用宿主组件的方法，就像下面用的 `getVal()`：

src/app/app.component.html

```
<!-- "The sum of 1 + 1 is not 4" -->
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}</p>
```

Angular 对所有双花括号中的表达式求值，把求值的结果转换成字符串，并把它们跟相邻的字符串字面量连接起来。最后，把这个组合出来的插值结果赋给元素或指令的属性。

表面上看，你在元素标签之间插入了结果和对标签的属性进行了赋值。 这样思考起来很方便，并且这个误解很少给你带来麻烦。 但严格来讲，这是不对的。插值表达式是一个特殊的语法，Angular 把它转换成了属性绑定，后面将会解释这一点。

讲解属性绑定之前，先深入了解一下模板表达式和模板语句。

模板表达式

模板表达式产生一个值。 Angular 执行这个表达式，并把它赋值给绑定目标的属性，这个绑定目标可能是 HTML 元素、组件或指令。

`{{1 + 1}}` 中所包含的模板表达式是 `1 + 1`。 在属性绑定中会再次看到模板表达式，它出现在 `=` 右侧的引号中，就像这样：`[property]="expression"`。

编写模板表达式所用的语言看起来很像 JavaScript。 很多 JavaScript 表达式也是合法的模板表达式，但不是全部。

JavaScript 中那些具有或可能引发副作用的表达式是被禁止的，包括：

- 赋值 (`=`, `+=`, `-=`, ...)
- `new` 运算符
- 使用 `;` 或 `,` 的链式表达式
- 自增和自减运算符：`++` 和 `--`

和 JavaScript 语 法的其它显著不同包括：

- 不支持位运算 `|` 和 `&`
- 具有新的模板表达式运算符，比如 `|`、`?.` 和 `!`。

表达式上下文

典型的**表达式上下文**就是这个组件实例，它是各种绑定值的来源。 在下面的代码片段中，双花括号中的 `title` 和引号中的 `isUnchanged` 所引用的都是 `AppComponent` 中的属性。

src/app/app.component.html

```
{{title}}
<span [hidden]="isUnchanged">changed</span>
```

表达式的上下文可以包括组件之外的对象。 比如模板输入变量 (`let hero`)和模板引用变量(`#heroInput`)就是备选的上下文对象之一。

src/app/app.component.html

```
<div *ngFor="let hero of heroes">{{hero.name}}</div>
<input #heroInput> {{heroInput.value}}
```

表达式中的上下文变量是由**模板变量**、指令的**上下文变量**（如果有）和组件的**成员**叠加而成的。 如果你要引用的变量名存在于一个以上的命名空间中，那么，模板变量是最优先的，其次是指令的上下文变量，最后是组件的成员。

上一个例子中就体现了这种命名冲突。组件具有一个名叫 `hero` 的属性，而 `*ngFor` 声明了一个也叫 `hero` 的模板变量。 在 `{{hero.name}}` 表达式中的 `hero` 实际引用的是模板变量，而不是组件的属性。

模板表达式不能引用全局命名空间中的任何东西，比如 `window` 或 `document`。它们也不能调用 `console.log` 或 `Math.max`。 它们只能引用表达式上下文中的成员。

表达式指南

模板表达式能成就或毁掉一个应用。请遵循下列指南：

- 没有可见的副作用
- 执行迅速
- 非常简单
- 幂等性

超出上面指南外的情况应该只出现在那些你确信自己已经彻底理解的特定场景中。

没有可见的副作用

模板表达式除了目标属性的值以外，不应该改变应用的任何状态。

这条规则是 Angular “单向数据流”策略的基础。 永远不用担心读取组件值可能改变另外的显示值。 在一次单独的渲染过程中，视图应该总是稳定的。

执行迅速

Angular 会在每个变更检测周期后执行模板表达式。 它们可能在每一次按键或鼠标移动后被调用。

表达式应该快速结束，否则用户就会感到拖沓，特别是在较慢的设备上。 当计算代价较高时，应该考虑缓存那些从其它值计算得出的值。

非常简单

虽然也可以写出相当复杂的模板表达式，但不要那么写。

常规是属性名或方法调用。偶尔的逻辑取反 (`!`) 也还凑合。 其它情况下，应在组件中实现应用和业务逻辑，使开发和测试变得更容易。

幂等性

最好使用幂等的表达式，因为它没有副作用，并且能提升 Angular 变更检测的性能。

在 Angular 的术语中，幂等的表达式应该总是返回**完全相同的东西**，直到某个依赖值发生改变。

在单独的一次事件循环中，被依赖的值不应该改变。 如果幂等的表达式返回一个字符串或数字，连续调用它两次，也应该返回相同的字符串或数字。 如果幂等的表达式返回一个对象（包括 `Date` 或 `Array`），连续调用它两次，也应该返回同一个对象的**引用**。

模板语句

模板语句用来响应由绑定目标（如 HTML 元素、组件或指令）触发的事件。 模板语句将在事件绑定一节看到，它出现在 `=` 号右侧的引号中，就像这样：`(event)="statement"`。

src/app/app.component.html

```
<button (click)="deleteHero()">Delete hero</button>
```

模板语句**有副作用**。 这是事件处理的关键。因为你要根据用户的输入更新应用状态。

响应事件是 Angular 中“单向数据流”的另一面。 在一次事件循环中，可以随意改变任何地方的任何东西。

和模板表达式一样，模板**语句**使用的语言也像 JavaScript。 模板语句解析器和模板表达式解析器有所不同，特别之处在于它支持基本赋值 (`=`) 和表达式链 (`;` 和 `,`)。

然而，某些 JavaScript 语法仍然是不允许的：

- `new` 运算符
- 自增和自减运算符：`++` 和 `--`
- 操作并赋值，例如 `+=` 和 `-=`
- 位操作符 `|` 和 `&`
- 模板表达式运算符

语句上下文

和表达式中一样，语句只能引用语句上下文中 —— 通常是正在绑定事件的那个组件实例。

典型的**语句上下文**就是当前组件的实例。 `(click)="deleteHero()"` 中的 **deleteHero** 就是这个数据绑定组件上的一个方法。

src/app/app.component.html

```
<button (click)="deleteHero()">Delete hero</button>
```

语句上下文可以引用模板自身上下文中的属性。 在下面的例子中，就把模板的 `$event` 对象、模板输入变量 (`let hero`)和模板引用变量 (`#heroForm`)传给了组件中的一个事件处理器方法。

src/app/app.component.html

```
<button (click)="onSave($event)">Save</button>
<button *ngFor="let hero of heroes" (click)="deleteHero(hero)">{{hero.name}}
</button>
<form #heroForm (ngSubmit)="onSubmit(heroForm)"> ... </form>
```

模板上下文中的变量名的优先级高于组件上下文中的变量名。在上面的 `deleteHero(hero)` 中，`hero` 是一个模板输入变量，而不是组件中的 `hero` 属性。

模板语句不能引用全局命名空间的任何东西。比如不能引用 `window` 或 `document`，也不能调用 `console.log` 或 `Math.max`。

语句指南

和表达式一样，避免写复杂的模板语句。 常规是函数调用或者属性赋值。

现在，对模板表达式和语句有了一点感觉了吧。 除插值表达式外，还有各种各样的数据绑定语法，是学习它们是时候了。

绑定语法：概览

数据绑定是一种机制，用来协调用户所见和应用数据。 虽然你能往 HTML 推送值或者从 HTML 拉取值， 但如果把这些琐事交给数据绑定框架处理， 应用会更容易编写、阅读和维护。 只要简单地在绑定源和目标 HTML 元素之间声明绑定，框架就会完成这项工作。

Angular 提供了各种各样的数据绑定，本章将逐一讨论。 先从高层视角来看看 Angular 数据绑定及其语法。

绑定的类型可以根据数据流的方向分成三类： **从数据源到视图**、**从视图到数据源**以及双向的**从视图到数据源再到视图**。

数据方向	语法	绑定类型

单向
从数据源
到视图
	`{{expression}}`
	`[target]="expression"`
	`bind-target="expression"`
		插值表达式
		属性
		Attribute
		CSS 类
		样式

从视图到数据源的单向绑定	`(target)="statement"` `on-target="statement"`	事件
双向	`[(target)]="expression"` `bindon-target="expression"`	双向

> 译注：由于 HTML attribute 和 DOM property 在中文中都被翻译成了“属性”，无法区分， 而接下来的部分重点是对它们进行比较。
>
> 我们无法改变历史，因此，在本章的翻译中，保留了它们的英文形式，不加翻译，以免混淆。 本章中，如果提到“属性”的地方，一定是指 property，因为在 Angular 中，实际上很少涉及 attribute。
>
> 但在其它章节中，为简单起见，凡是能通过上下文明显区分开的，就仍统一译为“属性”， 区分不明显的，会加注英文。

除了插值表达式之外的绑定类型，在等号左边是目标名， 无论是包在括号中 (`[]`、`()`) 还是用前缀形式 (`bind-`、`on-`、`bindon-`) 。

这个目标名就是**属性（Property）**的名字。它可能看起来像是**元素属性（Attribute）**的名字，但它不是。 要理解它们的不同点，你必须尝试用另一种方式来审视模板中的 HTML。

新的思维模型

数据绑定的威力和允许用自定义标记扩展 HTML 词汇的能力，会让你把模板 HTML 当成 **HTML+**。

它其实**就是** HTML+。 但它也跟你曾使用的 HTML 有着显著的不同。 这里需要一种新的思维模型。

在正常的 HTML 开发过程中，你使用 HTML 元素来创建视觉结构， 通过把字符串常量设置到元素的 attribute 来修改那些元素。

src/app/app.component.html

```
<div class="special">Mental Model</div>
<img src="assets/images/hero.png">
<button disabled>Save</button>
```

在 Angular 模板中，你仍使用同样的方式创建结构和初始化 attribute 值。

然后，用封装了 HTML 的组件创建新元素，并把它们当作原生 HTML 元素在模板中使用。

src/app/app.component.html

```
<!-- Normal HTML -->
<div class="special">Mental Model</div>
<!-- Wow! A new element! -->
<app-hero-detail></app-hero-detail>
```

这就是 HTML+。

现在开始学习数据绑定。你碰到的第一种数据绑定是这样的：

src/app/app.component.html

```
<!-- Bind button disabled state to `isUnchanged` property -->
<button [disabled]="isUnchanged">Save</button>
```

过会儿再认识那个怪异的方括号记法。直觉告诉你，你正在绑定按钮的 `disabled` attribute。 并把它设置为组件的 `isUnchanged` 属性的当前值。

但你的直觉是错的！日常的 HTML 思维模式在误导着你。 实际上，一旦开始数据绑定，就不再跟 HTML attribute 打交道了。 这里不是设置 attribute，而是设置 DOM 元素、组件和指令的 property。

> ## HTML attribute 与 DOM property 的对比
>
> 要想理解 Angular 绑定如何工作，重点是搞清 HTML attribute 和 DOM property 之间的区别。
>
> attribute 是由 HTML 定义的。property 是由 DOM (Document Object Model) 定义的。

> - 少量 HTML attribute 和 property 之间有着 1:1 的映射，如 `id`。
> - 有些 HTML attribute 没有对应的 property，如 `colspan`。
> - 有些 DOM property 没有对应的 attribute，如 `textContent`。
> - 大量 HTML attribute 看起来映射到了 property…… 但却不像你想的那样！
>
> 最后一类尤其让人困惑…… 除非你能理解这个普遍原则：
>
> attribute **初始化** DOM property，然后它们的任务就完成了。property 的值可以改变；attribute 的值不能改变。
>
> 例如，当浏览器渲染 `<input type="text" value="Bob">` 时，它将创建相应 DOM 节点， 它的 `value` 这个 property 被**初始化为** “Bob”。
>
> 当用户在输入框中输入 “Sally” 时，DOM 元素的 `value` 这个 **property** 变成了 “Sally”。 但是该 HTML 的 `value` 这个 **attribute** 保持不变。如果你读取 input 元素的 attribute，就会发现确实没变： `input.getAttribute('value') // 返回 "Bob"`。
>
> HTML 的 `value` 这个 attribute 指定了**初始**值；DOM 的 `value` 这个 property 是**当前**值。
>
> `disabled` 这个 attribute 是另一种特例。按钮的 `disabled` 这个 **property** 是 `false`，因为默认情况下按钮是可用的。 当你添加 `disabled` 这个 **attribute** 时，只要它出现了按钮的 `disabled` 这个 **property** 就初始化为 `true`，于是按钮就被禁用了。
>
> 添加或删除 `disabled` 这个 **attribute** 会禁用或启用这个按钮。但 **attribute** 的值无关紧要，这就是你为什么没法通过 `<button disabled="false">仍被禁用</button>` 这种写法来启用按钮。
>
> 设置按钮的 `disabled` 这个 **property**（如，通过 Angular 绑定）可以禁用或启用这个按钮。 这就是 **property** 的价值。
>
> 就算名字相同，HTML attribute 和 DOM property 也不是同一样东西。

这句话值得再强调一次： 模板绑定是通过 **property** 和**事件**来工作的，而不是 **attribute**。

> 没有 ATTRIBUTE 的世界
>
> 在 Angular 的世界中，attribute 唯一的作用是用来初始化元素和指令的状态。 当进行数据绑定时，只是在与元素和指令的 property 和事件打交道，而 attribute 就完全靠边站了。

把这个思维模型牢牢的印在脑子里，接下来，学习什么是绑定目标。

绑定目标

数据绑定的目标是 DOM 中的某些东西。 这个目标可能是（元素 | 组件 | 指令的）property、 （元素 | 组件 | 指令的）事件，或(极少数情况下) attribute 名。 下面是的汇总表：

绑定类型	目标	范例

属性	元素的 property
	组件的 property
	指令的 property
		src/app/app.component.html
		`<img [src]="heroImageUrl">`
		`<app-hero-detail [hero]="currentHero"></app-hero-detail>`
		`<div [ngClass]="{'special': isSpecial}"></div>`

事件	元素的事件
	组件的事件
	指令的事件
		src/app/app.component.html
		`<button (click)="onSave()">Save</button>`
		`<app-hero-detail (deleteRequest)="deleteHero()">`
		`</app-hero-detail>`
		`<div (myClick)="clicked=$event" clickable>click me</div>`

双向	事件与 property	src/app/app.component.html `<input [(ngModel)]="name">`
Attribute	attribute（例外情况）	src/app/app.component.html `<button [attr.aria-label]="help">help</button>`

CSS 类	`class` property	src/app/app.component.html `<div [class.special]="isSpecial">Special</div>`
样式	`style` property	src/app/app.component.html `<button [style.color]="isSpecial ? 'red' : 'green'">`

放开眼界，来看看每种绑定类型的具体情况。

属性绑定 ([属性名])

当要把视图元素的属性 (property) 设置为模板表达式时，就要写模板的属性 (property) 绑定。

最常用的属性绑定是把元素属性设置为组件属性的值。 下面这个例子中，image 元素的 `src` 属性会被绑定到组件的 `heroImageUrl` 属性上：

src/app/app.component.html

```
<img [src]="heroImageUrl">
```

另一个例子是当组件说它 `isUnchanged`（未改变）时禁用按钮：

src/app/app.component.html

```
<button [disabled]="isUnchanged">Cancel is disabled</button>
```

另一个例子是设置指令的属性：

src/app/app.component.html

```
<div [ngClass]="classes">[ngClass] binding to the classes property</div>
```

还有另一个例子是设置自定义组件的模型属性（这是父子组件之间通讯的重要途径）：

src/app/app.component.html

```
<app-hero-detail [hero]="currentHero"></app-hero-detail>
```

单向输入

人们经常把属性绑定描述成**单向数据绑定**，因为值的流动是单向的，从组件的数据属性流动到目标元素的属性。

不能使用属性绑定来从目标元素拉取值，也不能绑定到目标元素的属性来读取它。只能设置它。

> 也不能使用属性 绑定 来**调用**目标元素上的方法。
>
> 如果这个元素触发了事件，可以通过事件绑定来监听它们。
>
> 如果必须读取目标元素上的属性或调用它的某个方法，得用另一种技术。 参见 API 参考手册中的 ViewChild 和 ContentChild。

绑定目标

包裹在方括号中的元素属性名标记着目标属性。下列代码中的目标属性是 image 元素的 `src` 属性。

src/app/app.component.html

```
<img [src]="heroImageUrl">
```

有些人喜欢用 `bind-` 前缀的可选形式，并称之为**规范形式**：

src/app/app.component.html

```
<img bind-src="heroImageUrl">
```

目标的名字总是 property 的名字。即使它看起来和别的名字一样。 看到 `src` 时，可能会把它当做 attribute。不！ 它不是！ 它是 image 元素的 property 名。

元素属性可能是最常见的绑定目标，但 Angular 会先去看这个名字是否是某个已知指令的属性名，就像下面的例子中一样：

src/app/app.component.html

```
<div [ngClass]="classes">[ngClass] binding to the classes property</div>
```

> 严格来说，Angular 正在匹配指令的输入属性的名字。 这个名字是指令的 `inputs` 数组中所列的名字，或者是带有 `@Input()` 装饰器的属性。 这些输入属性被映射为指令自己的属性。

如果名字没有匹配上已知指令或元素的属性，Angular 就会报告“未知指令”的错误。

消除副作用

正如以前讨论过的，模板表达式的计算不能有可见的副作用。表达式语言本身可以提供一部分安全保障。 不能在属性绑定表达式中对任何东西赋值，也不能使用自增、自减运算符。

当然，表达式可能会调用具有副作用的属性或方法。但 Angular 没法知道这一点，也没法阻止你。

表达式中可以调用像 `getFoo()` 这样的方法。只有你知道 `getFoo()` 干了什么。 如果 `getFoo()` 改变了某个东西，恰好又绑定到个这个东西，你就可能把自己坑了。 Angular 可能显示也可能不显示变化后的值。Angular 还可能检测到变化，并抛出警告型错误。 一般建议是，只绑定数据属性和那些只返回值而不做其它事情的方法。

返回恰当的类型

模板表达式应该返回目标属性所需类型的值。 如果目标属性想要个字符串，就返回字符串。 如果目标属性想要个数字，就返回数字。 如果目标属性想要个对象，就返回对象。

`HeroDetail` 组件的 `hero` 属性想要一个 `Hero` 对象，那就在属性绑定中精确地给它一个 `Hero` 对象：

src/app/app.component.html

```
<app-hero-detail [hero]="currentHero"></app-hero-detail>
```

别忘了方括号

方括号告诉 Angular 要计算模板表达式。 如果忘了加方括号，Angular 会把这个表达式当做字符串常量看待，并用该字符串来**初始化目标属性**。 它**不会**计算这个字符串。

不要出现这样的失误：

src/app/app.component.html

```
<!-- ERROR: HeroDetailComponent.hero expects a
     Hero object, not the string "currentHero" -->
  <app-hero-detail hero="currentHero"></app-hero-detail>
```

一次性字符串初始化

当满足下列条件时，**应该**省略括号：

- 目标属性接受字符串值。
- 字符串是个固定值，可以直接合并到模块中。
- 这个初始值永不改变。

你经常这样在标准 HTML 中用这种方式初始化 attribute，这种方式也可以用在初始化指令和组件的属性。 下面这个例子把 `HeroDetailComponent` 的 `prefix` 属性初始化为固定的字符串，而不是模板表达式。Angular 设置它，然后忘记它。

src/app/app.component.html

```
<app-hero-detail prefix="You are my" [hero]="currentHero"></app-hero-detail>
```

作为对比，`[hero]` 绑定是组件的 `currentHero` 属性的活绑定，它会一直随着更新。

属性绑定还是插值表达式?

你通常得在插值表达式和属性绑定之间做出选择。 下列这几对绑定做的事情完全相同：

src/app/app.component.html

```
<p><img src="{{heroImageUrl}}"> is the <i>interpolated</i> image.</p>
<p><img [src]="heroImageUrl"> is the <i>property bound</i> image.</p>

<p><span>"{{title}}" is the <i>interpolated</i> title.</span></p>
<p>"<span [innerHTML]="title"></span>" is the <i>property bound</i> title.</p>
```

在多数情况下，插值表达式是更方便的备选项。 实际上，在渲染视图之前，Angular 把这些插值表达式翻译成相应的属性绑定。

当要渲染的数据类型是字符串时，没有技术上的理由证明哪种形式更好。 你倾向于可读性，所以倾向于插值表达式。 建议建立代码风格规则，选择一种形式， 这样，既遵循了规则，又能让手头的任务做起来更自然。

但数据类型不是字符串时，就必须使用**属性绑定**了。

内容安全

假设下面的**恶意内容**

src/app/app.component.ts

```
evilTitle = 'Template <script>alert("evil never sleeps")</script>Syntax';
```

幸运的是，Angular 数据绑定对危险 HTML 有防备。 在显示它们之前，它对内容先进行**消毒**。 不管是插值表达式还是属性绑定，都不会允许带有 script 标签的 HTML 泄漏到浏览器中。

src/app/app.component.html

```
<!--
  Angular generates warnings for these two lines as it sanitizes them
  WARNING: sanitizing HTML stripped some content (see http://g.co/ng/security#xss).
 -->
<p><span>"{{evilTitle}}" is the <i>interpolated</i> evil title.</span></p>
<p>"<span [innerHTML]="evilTitle"></span>" is the <i>property bound</i> evil title.
</p>
```

插值表达式处理 script 标签与属性绑定有所不同，但是二者都只渲染没有危害的内容。

"Template <script>alert("evil never sleeps")</script>Syntax" is the *interpolated* evil title.

"Template Syntax" is the *property bound* evil title.

attribute、class 和 style 绑定

模板语法为那些不太适合使用属性绑定的场景提供了专门的单向数据绑定形式。

attribute 绑定

可以通过attribute 绑定来直接设置 attribute 的值。

> 这是“绑定到目标属性 (property)”这条规则中唯一的例外。这是唯一的能创建和设置 attribute 的绑定形式。

本章中，通篇都在说通过属性绑定来设置元素的属性总是好于用字符串设置 attribute。为什么 Angular 还提供了 attribute 绑定呢?

因为当元素没有属性可绑的时候，就必须使用 attribute 绑定。

考虑 ARIA， SVG 和 table 中的 colspan/rowspan 等 attribute。 它们是纯粹的 attribute，没有对应的属性可供绑定。

如果想写出类似下面这样的东西，就会暴露出痛点了：

```
<tr><td colspan="{{1 + 1}}">Three-Four</td></tr>
```

会得到这个错误：

```
Template parse errors:
Can't bind to 'colspan' since it isn't a known native property
```

正如提示中所说，`<td>` 元素没有 `colspan` 属性。 但是插值表达式和属性绑定只能设置**属性**，不能设置 attribute。

你需要 attribute 绑定来创建和绑定到这样的 attribute。

attribute 绑定的语法与属性绑定类似。 但方括号中的部分不是元素的属性名，而是由`attr`前缀，一个点 (`.`) 和 attribute 的名字组成。 可以通过值为字符串的表达式来设置 attribute 的值。

这里把 `[attr.colspan]` 绑定到一个计算值：

src/app/app.component.html

```
<table border=1>
  <!--  expression calculates colspan=2 -->
  <tr><td [attr.colspan]="1 + 1">One-Two</td></tr>

  <!-- ERROR: There is no `colspan` property to set!
    <tr><td colspan="{{1 + 1}}">Three-Four</td></tr>
  -->

  <tr><td>Five</td><td>Six</td></tr>
</table>
```

这里是表格渲染出来的样子：

One-Two	
Five	Six

attribute 绑定的主要用例之一是设置 ARIA attribute（译注：ARIA 指可访问性，用于给残障人士访问互联网提供便利）， 就像这个例子中一样：

src/app/app.component.html

```
<!-- create and set an aria attribute for assistive technology -->
<button [attr.aria-label]="actionName">{{actionName}} with Aria</button>
```

CSS 类绑定

借助 CSS 类绑定，可以从元素的 `class` attribute 上添加和移除 CSS 类名。

CSS 类绑定绑定的语法与属性绑定类似。 但方括号中的部分不是元素的属性名，而是由`class`前缀，一个点 (`.`)和 CSS 类的名字组成， 其中后两部分是可选的。形如：`[class.class-name]`。

下列例子示范了如何通过 CSS 类绑定来添加和移除应用的 "special" 类。不用绑定直接设置 attribute 时是这样的：

src/app/app.component.html

```
<!-- standard class attribute setting  -->
<div class="bad curly special">Bad curly special</div>
```

可以把它改写为绑定到所需 CSS 类名的绑定；这是一个或者全有或者全无的替换型绑定。 （译注：即当 badCurly 有值时 class 这个 attribute 设置的内容会被完全覆盖）

src/app/app.component.html

```
<!-- reset/override all class names with a binding  -->
<div class="bad curly special"
     [class]="badCurly">Bad curly</div>
```

最后，可以绑定到特定的类名。 当模板表达式的求值结果是真值时，Angular 会添加这个类，反之则移除它。

src/app/app.component.html

```
<!-- toggle the "special" class on/off with a property -->
<div [class.special]="isSpecial">The class binding is special</div>

<!-- binding to `class.special` trumps the class attribute -->
<div class="special"
     [class.special]="!isSpecial">This one is not so special</div>
```

> 虽然这是切换单一类名的好办法，但人们通常更喜欢使用 NgClass 指令 来同时管理多个类名。

样式绑定

通过样式绑定，可以设置内联样式。

样式绑定的语法与属性绑定类似。 但方括号中的部分不是元素的属性名，而由`style`前缀，一个点 (`.`)和 CSS 样式的属性名组成。 形如：`[style.style-property]`。

src/app/app.component.html

```
<button [style.color]="isSpecial ? 'red': 'green'">Red</button>
<button [style.background-color]="canSave ? 'cyan': 'grey'" >Save</button>
```

有些样式绑定中的样式带有单位。在这里，以根据条件用 “em” 和 “%” 来设置字体大小的单位。

src/app/app.component.html

```
<button [style.font-size.em]="isSpecial ? 3 : 1" >Big</button>
<button [style.font-size.%]="!isSpecial ? 150 : 50" >Small</button>
```

> 虽然这是设置单一样式的好办法，但人们通常更喜欢使用 NgStyle 指令 来同时设置多个内联样式。

> 注意，**样式属性**命名方法可以用中线命名法，像上面的一样 也可以用驼峰式命名法，如 `fontSize`。

事件绑定 ((事件名))

前面遇到的绑定的数据流都是单向的：从组件到元素。

但用户不会只盯着屏幕看。他们会在输入框中输入文本。他们会从列表中选取条目。 他们会点击按钮。这类用户动作可能导致反向的数据流：**从元素到组件。**

知道用户动作的唯一方式是监听某些事件，如按键、鼠标移动、点击和触摸屏幕。 可以通过 Angular 事件绑定来声明对哪些用户动作感兴趣。

事件绑定语法由等号左侧带圆括号的目标事件和右侧引号中的模板语句组成。 下面事件绑定监听按钮的点击事件。每当点击发生时，都会调用组件的 `onSave()` 方法。

src/app/app.component.html

```
<button (click)="onSave()">Save</button>
```

目标事件

圆括号中的名称 —— 比如 `(click)` —— 标记出目标事件。在下面例子中，目标是按钮的 click 事件。

src/app/app.component.html

```
<button (click)="onSave()">Save</button>
```

有些人更喜欢带 `on-` 前缀的备选形式，称之为规范形式：

src/app/app.component.html

```
<button on-click="onSave()">On Save</button>
```

元素事件可能是更常见的目标，但 Angular 会先看这个名字是否能匹配上已知指令的事件属性，就像下面这个例子：

src/app/app.component.html

```
<!-- `myClick` is an event on the custom `ClickDirective` -->
<div (myClick)="clickMessage=$event" clickable>click with myClick</div>
```

> 更多关于该 `myClick` 指令的解释，见给输入/输出属性起别名。

如果这个名字没能匹配到元素事件或已知指令的输出属性，Angular 就会报“未知指令”错误。

$event 和事件处理语句

在事件绑定中，Angular 会为目标事件设置事件处理器。

当事件发生时，这个处理器会执行模板语句。 典型的模板语句通常涉及到响应事件执行动作的接收器，例如从 HTML 控件中取得值，并存入模型。

绑定会通过名叫 `$event` 的事件对象传递关于此事件的信息（包括数据值）。

事件对象的形态取决于目标事件。如果目标事件是原生 DOM 元素事件， `$event` 就是 DOM 事件对象，它有像 `target` 和 `target.value` 这样的属性。

考虑这个范例：

src/app/app.component.html

```
<input [value]="currentHero.name"
       (input)="currentHero.name=$event.target.value" >
```

上面的代码在把输入框的 `value` 属性绑定到 `firstName` 属性。 要监听对值的修改，代码绑定到输入框的 `input` 事件。 当用户造成更改时， `input` 事件被触发，并在包含了 DOM 事件对象 (`$event`) 的上下文中执行这条语句。

要更新 `firstName` 属性，就要通过路径 `$event.target.value` 来获取更改后的值。

如果事件属于指令（回想一下，组件是指令的一种），那么 `$event` 具体是什么由指令决定。

使用 `EventEmitter` 实现自定义事件

通常，指令使用 Angular EventEmitter 来触发自定义事件。 指令创建一个 `EventEmitter` 实例，并且把它作为属性暴露出来。 指令调用 `EventEmitter.emit(payload)` 来触发事件，可以传入任何东西作为消息载荷。 父指令通过绑定到这个属性来监听事件，并通过 `$event` 对象来访问载荷。

假设 `HeroDetailComponent` 用于显示英雄的信息，并响应用户的动作。 虽然 `HeroDetailComponent` 包含删除按钮，但它自己并不知道该如何删除这个英雄。 最好的做法是触发事件来报告“删除用户”的请求。

下面的代码节选自 `HeroDetailComponent`：

src/app/hero-detail.component.ts (template)

```
template: `
<div>
  <img src="{{heroImageUrl}}">
  <span [style.text-decoration]="lineThrough">
    {{prefix}} {{hero?.name}}
  </span>
  <button (click)="delete()">Delete</button>
</div>`
```

src/app/hero-detail.component.ts (deleteRequest)

```
// This component makes a request but it can't actually delete a hero.
deleteRequest = new EventEmitter<Hero>();

delete() {
  this.deleteRequest.emit(this.hero);
}
```

组件定义了 `deleteRequest` 属性，它是 `EventEmitter` 实例。 当用户点击**删除**时，组件会调用 `delete()` 方法，让 `EventEmitter` 发出一个 `Hero` 对象。

现在，假设有个宿主的父组件，它绑定了 `HeroDetailComponent` 的 `deleteRequest` 事件。

src/app/app.component.html (event-binding-to-component)

```
<app-hero-detail (deleteRequest)="deleteHero($event)" [hero]="currentHero"></app-hero-detail>
```

当 `deleteRequest` 事件触发时，Angular 调用父组件的 `deleteHero` 方法， 在 `$event` 变量中传入**要删除的英雄**（来自 `HeroDetail`）。

模板语句有副作用

`deleteHero` 方法有副作用：它删除了一个英雄。 模板语句的副作用不仅没问题，反而正是所期望的。

删除这个英雄会更新模型，还可能触发其它修改，包括向远端服务器的查询和保存。 这些变更通过系统进行扩散，并最终显示到当前以及其它视图中。

双向数据绑定 (`[(...)]`)

你经常需要显示数据属性，并在用户作出更改时更新该属性。

在元素层面上，既要设置元素属性，又要监听元素事件变化。

Angular 为此提供一种特殊的**双向数据绑定**语法：`[(x)]`。 `[(x)]` 语法结合了**属性绑定**的方括号 `[x]` 和**事件绑定**的圆括号 `(x)`。

> [( )] = 盒子里的香蕉
>
> 想象**盒子里的香蕉**来记住方括号套圆括号。

当一个元素拥有可以设置的属性 `x` 和对应的事件 `xChange` 时，解释 `[(x)]` 语法就容易多了。 下面的 `SizerComponent` 符合这个模式。它有 `size` 属性和配套的 `sizeChange` 事件：

src/app/sizer.component.ts

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

@Component({
  selector: 'app-sizer',
  template: `
  <div>
    <button (click)="dec()" title="smaller">-</button>
    <button (click)="inc()" title="bigger">+</button>
    <label [style.font-size.px]="size">FontSize: {{size}}px</label>
  </div>`
})
export class SizerComponent {
  @Input()  size: number | string;
  @Output() sizeChange = new EventEmitter<number>();

  dec() { this.resize(-1); }
  inc() { this.resize(+1); }

  resize(delta: number) {
    this.size = Math.min(40, Math.max(8, +this.size + delta));
    this.sizeChange.emit(this.size);
  }
}
```

`size` 的初始值是一个输入值，来自属性绑定。（译注：注意 `size` 前面的 `@Input`） 点击按钮，在最小/最大值范围限制内增加或者减少 `size`。 然后用调整后的 `size` 触发 `sizeChange` 事件。

下面的例子中，`AppComponent.fontSize` 被双向绑定到 `SizerComponent`：

src/app/app.component.html (two-way-1)

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
<div [style.font-size.px]="fontSizePx">Resizable Text</div>
```

`SizerComponent.size` 初始值是 `AppComponent.fontSizePx`。 点击按钮时，通过双向绑定更新 `AppComponent.fontSizePx`。 被修改的 `AppComponent.fontSizePx` 通过**样式**绑定，改变文本的显示大小。

双向绑定语法实际上是**属性**绑定和**事件绑定**的语法糖。 Angular 将 `SizerComponent` 的绑定分解成这样：

src/app/app.component.html (two-way-2)

```
<app-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></app-sizer>
```

`$event` 变量包含了 `SizerComponent.sizeChange` 事件的荷载。 当用户点击按钮时，Angular 将 `$event` 赋值给 `AppComponent.fontSizePx`。

显然，比起单独绑定属性和事件，双向数据绑定语法显得非常方便。

如果能在像 `<input>` 和 `<select>` 这样的 HTML 元素上使用双向数据绑定就更好了。 可惜，原生 HTML 元素不遵循 `x` 值和 `xChange` 事件的模式。

幸运的是，Angular 以 **NgModel** 指令为桥梁，允许在表单元素上使用双向数据绑定。

内置指令

上一版本的 Angular 中包含了超过 70 个内置指令。 社区贡献了更多，这还没算为内部应用而创建的无数私有指令。

Earlier versions of Angular included over seventy built-in directives. The community contributed many more, and countless private directives have been created for internal applications.

在新版的 Angular 中不需要那么多指令。 使用更强大、更富有表现力的 Angular 绑定系统，其实可以达到同样的效果。 如果能用简单的绑定达到目的，为什么还要创建指令来处理点击事件呢？

src/app/app.component.html

```
<button (click)="onSave()">Save</button>
```

你仍然可以从简化复杂任务的指令中获益。 Angular 发布时仍然带有内置指令，只是没那么多了。 你仍会写自己的指令，只是没那么多了。

下面来看一下那些最常用的内置指令。它们可分为**属性型**指令 或 **结构型**指令。

内置**属性型**指令

属性型指令会监听和修改其它 HTML 元素或组件的行为、元素属性（Attribute）、DOM 属性（Property）。它们通常会作为 HTML 属性的名称而应用在元素上。

更多的细节参见**属性型指令**一章。 很多 Angular 模块，比如`RouterModule`和`FormsModule`都定义了自己的属性型指令。 本节将会介绍几个最常用的属性型指令：

- `NgClass` - 添加或移除一组 CSS 类
- `NgStyle` - 添加或移除一组 CSS 样式
- `NgModel` - 双向绑定到 HTML 表单元素

NgClass

你经常用动态添加或删除 CSS 类的方式来控制元素如何显示。 通过绑定到 `NgClass`，可以同时添加或移除多个类。

CSS 类绑定 是添加或删除**单个**类的最佳途径。

src/app/app.component.html

```
<!-- toggle the "special" class on/off with a property -->
<div [class.special]="isSpecial">The class binding is special</div>
```

当想要同时添加或移除**多个** CSS 类时，`NgClass` 指令可能是更好的选择。

试试把 `ngClass` 绑定到一个 key:value 形式的控制对象。这个对象中的每个 key 都是一个 CSS 类名，如果它的 value 是 `true`，这个类就会被加上，否则就会被移除。

组件方法 `setCurrentClasses` 可以把组件的属性 `currentClasses` 设置为一个对象，它将会根据三个其它组件的状态为 `true` 或 `false` 而添加或移除三个类。

src/app/app.component.ts

```
currentClasses: {};
setCurrentClasses() {
  // CSS classes: added/removed per current state of component properties
  this.currentClasses =  {
    'saveable': this.canSave,
    'modified': !this.isUnchanged,
    'special':  this.isSpecial
  };
}
```

把 `NgClass` 属性绑定到 `currentClasses`，根据它来设置此元素的 CSS 类：

src/app/app.component.html

```
<div [ngClass]="currentClasses">This div is initially saveable, unchanged, and special</div>
```

> 你既可以在初始化时调用 `setCurrentClassess()`，也可以在所依赖的属性变化时调用。

NgStyle

你可以根据组件的状态动态设置内联样式。 `NgStyle` 绑定可以同时设置多个内联样式。

样式绑定是设置**单一**样式值的简单方式。

src/app/app.component.html

```
<div [style.font-size]="isSpecial ? 'x-large' : 'smaller'" >
  This div is x-large or smaller.
</div>
```

如果要同时设置**多个**内联样式，`NgStyle` 指令可能是更好的选择。

`NgStyle` 需要绑定到一个 key:value 控制对象。 对象的每个 key 是样式名，它的 value 是能用于这个样式的任何值。

来看看组件的 `setCurrentStyles` 方法，它会根据另外三个属性的状态把组件的 `currentStyles` 属性设置为一个定义了三个样式的对象：

src/app/app.component.ts

```
currentStyles: {};
setCurrentStyles() {
  // CSS styles: set per current state of component properties
  this.currentStyles = {
    'font-style':  this.canSave      ? 'italic' : 'normal',
    'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
    'font-size':   this.isSpecial    ? '24px'   : '12px'
  };
}
```

把 `NgStyle` 属性绑定到 `currentStyles`，以据此设置此元素的样式：

src/app/app.component.html

```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large (24px).
</div>
```

> 你既可以在初始化时调用 `setCurrentStyles()`，也可以在所依赖的属性变化时调用。

NgModel - 使用`[(ngModel)]`双向绑定到表单元素

当开发数据输入表单时，你通常都要既显示数据属性又根据用户的更改去修改那个属性。

使用 `NgModel` 指令进行双向数据绑定可以简化这种工作。例子如下：

src/app/app.component.html (NgModel-1)

```
<input [(ngModel)]="currentHero.name">
```

使用 `ngModel` 时需要 `FormsModule`

在使用 `ngModel` 指令进行双向数据绑定之前，你必须导入 `FormsModule` 并把它添加到 Angular 模块的 `imports` 列表中。 要了解 `FormsModule` 和 `ngModel` 的更多知识，参见表单一章。

导入 `FormsModule` 并让 `[(ngModel)]` 可用的代码如下：

src/app/app.module.ts (FormsModule import)

```
import { NgModule } from '@angular/core';
import { BrowserModule }  from '@angular/platform-browser';
import { FormsModule } from '@angular/forms'; // <--- JavaScript import from
Angular

/* Other imports */

@NgModule({
  imports: [
    BrowserModule,
    FormsModule  // <--- import into the NgModule
  ],
  /* Other module metadata */
})
export class AppModule { }
```

[(ngModel)]内幕

Inside [(ngModel)]

回头看看 `name` 绑定，注意，你可以通过分别绑定到 `<input>` 元素的 `value` 属性和 `input` 事件来达到同样的效果。

src/app/app.component.html

```
<input [value]="currentHero.name"
       (input)="currentHero.name=$event.target.value" >
```

那样显得很笨重，谁会记得该设置哪个元素属性以及当用户修改时触发哪个事件？ 你该如何提取输入框中的文本并且更新数据属性？谁会希望每次都去查资料来确定这些？

`ngModel` 指令通过自己的输入属性 `ngModel` 和输出属性 `ngModelChange` 隐藏了那些细节。

src/app/app.component.html

```
<input
  [ngModel]="currentHero.name"
  (ngModelChange)="currentHero.name=$event">
```

> `ngModel` 输入属性会设置该元素的值，并通过 `ngModelChange` 的输出属性来监听元素值的变化。

> 各种元素都有很多特有的处理细节，因此 `NgModel` 指令只支持实现了ControlValueAccessor的元素，它们能让元素适配本协议。 `<input>` 输入框正是其中之一。 Angular 为所有的基础 HTML 表单都提供了**值访问器（Value accessor）**，**表单**一章展示了如何绑定它们。
>
> 你不能把 `[(ngModel)]` 用到非表单类的原生元素或第三方自定义组件上，除非写一个合适的**值访问器**，这种技巧超出了本章的范围。
>
> 你自己写的 Angular 组件不需要**值访问器**，因为你可以让值和事件的属性名适应 Angular 基本的双向绑定语法，而不使用 `NgModel`。 前面看过的 `sizer`就是使用这种技巧的例子。

使用独立的 `ngModel` 绑定优于绑定到该元素的原生属性，你可以做得更好。

你不用被迫两次引用这个数据属性，Angular 可以捕获该元素的数据属性，并且通过一个简单的声明来设置它，这样它就可以使用 `[(ngModel)]` 语法了。

src/app/app.component.html

```
<input [(ngModel)]="currentHero.name">
```

`[(ngModel)]` 就是你需要的一切吗？有没有什么理由回退到它的展开形式？

`[(ngModel)]` 语法只能**设置**数据绑定属性。 如果要做更多或者做点不一样的事，也可以写它的展开形式。

下面这个生造的例子强制输入框的内容变成大写：

src/app/app.component.html

```
<input
  [ngModel]="currentHero.name"
  (ngModelChange)="setUppercaseName($event)">
```

这里是所有这些变体的动画，包括这个大写转换的版本：

NgModel Binding

Result: Hercules

Hercules	without NgModel
Hercules	[(ngModel)]
Hercules	bindon-ngModel
Hercules	(ngModelChange) = "...firstName=$event"
Hercules	(ngModelChange) = "setUpperCaseFirstName($event)"

内置**结构型**指令

结构型指令的职责是 HTML 布局。 它们塑造或重塑 DOM 的**结构**，这通常是通过添加、移除和操纵它们所附加到的宿主元素来实现的。

关于结构型指令的详情参见**结构型指令**一章，在那里你将学到：

- 为什么要给结构型指令的名字加上(*)前缀?
- 当没有合适的宿主元素放置指令时，可用 `<ng-container>` 对元素进行分组。
- 如何写自己的结构型指令。
- 你只能往一个元素上应用一个结构型指令。

本节是对常见结构型指令的简介：

- `NgIf` - 根据条件把一个元素添加到 DOM 中或从 DOM 移除
- `NgSwitch` 一组指令，用来在多个可选视图之间切换。
- NgForOf - 对列表中的每个条目重复套用同一个模板

NgIf

通过把 `NgIf` 指令应用到元素上（称为**宿主元素**），你可以往 DOM 中添加或从 DOM 中移除这个元素。 在下面的例子中，该指令绑定到了类似于 `isActive` 这样的条件表达式。

src/app/app.component.html

```
<app-hero-detail *ngIf="isActive"></app-hero-detail>
```

> 别忘了 `ngIf` 前面的星号(`*`)。

当 `isActive` 表达式返回真值时，`NgIf` 把 `HeroDetailComponent` 添加到 DOM 中；为假时，`NgIf` 会从 DOM 中移除 `HeroDetailComponent`，并销毁该组件及其所有子组件。

这和显示/隐藏不是一回事

你也可以通过类绑定或样式绑定来显示或隐藏一个元素。

src/app/app.component.html

```
<!-- isSpecial is true -->
<div [class.hidden]="!isSpecial">Show with class</div>
<div [class.hidden]="isSpecial">Hide with class</div>

<!-- HeroDetail is in the DOM but hidden -->
<app-hero-detail [class.hidden]="isSpecial"></app-hero-detail>

<div [style.display]="isSpecial ? 'block' : 'none'">Show with style</div>
<div [style.display]="isSpecial ? 'none'  : 'block'">Hide with style</div>
```

但隐藏子树和用 `NgIf` 排除子树是截然不同的。

当隐藏子树时，它仍然留在 DOM 中。 子树中的组件及其状态仍然保留着。 即使对于不可见属性，Angular 也会继续检查变更。 子树可能占用相当可观的内存和运算资源。

当 `NgIf` 为 `false` 时，Angular 从 DOM 中物理地移除了这个元素子树。 它销毁了子树中的组件及其状态，也潜在释放了可观的资源，最终让用户体验到更好的性能。

显示/隐藏的技术对于只有少量子元素的元素是很好用的，但要当心别试图隐藏大型组件树。相比之下，`NgIf` 则是个更安全的选择。

防范空指针错误

`ngIf` 指令通常会用来防范空指针错误。 而显示/隐藏的方式是无法防范的，当一个表达式尝试访问空值的属性时，Angular 就会抛出一个异常。

这里我们用 `NgIf` 来保护了两个 `<div>` 防范空指针错误。 `currentHero` 的名字只有当存在 `currentHero` 时才会显示出来。 而 `nullHero` 永远不会显示。

src/app/app.component.html

```
<div *ngIf="currentHero">Hello, {{currentHero.name}}</div>
<div *ngIf="nullHero">Hello, {{nullHero.name}}</div>
```

> 参见稍后的**安全导航操作符**部分。

NgForOf

`NgFor` 是一个**重复器**指令 —— 自定义数据显示的一种方式。 你的目标是展示一个由多个条目组成的列表。首先定义了一个 HTML 块，它规定了单个条目应该如何显示。 再告诉 Angular 把这个块当做模板，渲染列表中的每个条目。

下例中，`NgFor` 应用在一个简单的 `<div>` 上：

src/app/app.component.html

```
<div *ngFor="let hero of heroes">{{hero.name}}</div>
```

也可以把 `NgFor` 应用在一个组件元素上，就下例这样：

src/app/app.component.html

```
<app-hero-detail *ngFor="let hero of heroes" [hero]="hero"></app-hero-detail>
```

> 不要忘了 `ngFor` 前面的星号 (`*`)。

赋值给 `*ngFor` 的文本是用于指导重复器如何工作的指令。

NgFor 微语法

赋值给 `*ngFor` 的字符串不是模板表达式。 它是一个**微语法** —— 由 Angular 自己解释的小型语言。在这个例子中，字符串 `"let hero of heroes"` 的含义是：

> **取出 `heroes` 数组中的每个英雄，把它存入局部变量 `hero` 中，并在每次迭代时对模板 HTML 可用**

Angular 把这个指令翻译成了一个 `<ng-template>` 包裹的宿主元素，然后使用这个模板重复创建出一组新元素，并且绑定到列表中的每一个 `hero`。

要了解**微语法**的更多知识，参见**结构型指令**一章。

模板输入变量

`hero` 前的 `let` 关键字创建了一个名叫 `hero` 的**模板输入变量**。 `ngFor` 指令在由父组件的 `heroes` 属性返回的 `heroes` 数组上迭代，每次迭代都从数组中把当前元素赋值给 `hero` 变量。

你可以在 `ngFor` 的宿主元素（及其子元素）中引用模板输入变量 `hero`，从而访问该英雄的属性。 这里它首先在一个插值表达式中被引用到，然后通过一个绑定把它传给了 `<hero-detail>` 组件的 `hero` 属性。

src/app/app.component.html

```
<div *ngFor="let hero of heroes">{{hero.name}}</div>
<app-hero-detail *ngFor="let hero of heroes" [hero]="hero"></app-hero-detail>
```

要了解更多**模板输入变量**的知识，参见**结构型指令**一章。

带索引的 `*ngFor`

`NgFor` 指令上下文中的 `index` 属性返回一个从零开始的索引，表示当前条目在迭代中的顺序。 你可以通过模板输入变量捕获这个 `index` 值，并把它用在模板中。

下面这个例子把 `index` 捕获到了 `i` 变量中，并且把它显示在英雄名字的前面。

src/app/app.component.html

```
<div *ngFor="let hero of heroes; let i=index">{{i + 1}} - {{hero.name}}</div>
```

> 要学习更多的**类似 index** 的值，例如 `last`、`even` 和 `odd`，请参阅 NgFor API 参考。

带 `trackBy` 的 `*ngFor`

`ngFor` 指令有时候会性能较差，特别是在大型列表中。 对一个条目的一丁点改动、移除或添加，都会导致级联的 DOM 操作。

例如，重新从服务器查询可以刷新包括所有新英雄在内的英雄列表。

他们中的绝大多数（如果不是所有的话）都是以前显示过的英雄。**你**知道这一点，是因为每个英雄的 `id` 没有变化。 但在 Angular 看来，它只是一个由新的对象引用构成的新列表， 它没有选择，只能清理旧列表、舍弃那些 DOM 元素，并且用新的 DOM 元素来重建一个新列表。

如果给它指定一个 `trackBy`，Angular 就可以避免这种折腾。 往组件中添加一个方法，它会返回 `NgFor`**应该**追踪的值。 在这里，这个值就是英雄的 `id`。

src/app/app.component.ts

```
trackByHeroes(index: number, hero: Hero): number { return hero.id; }
```

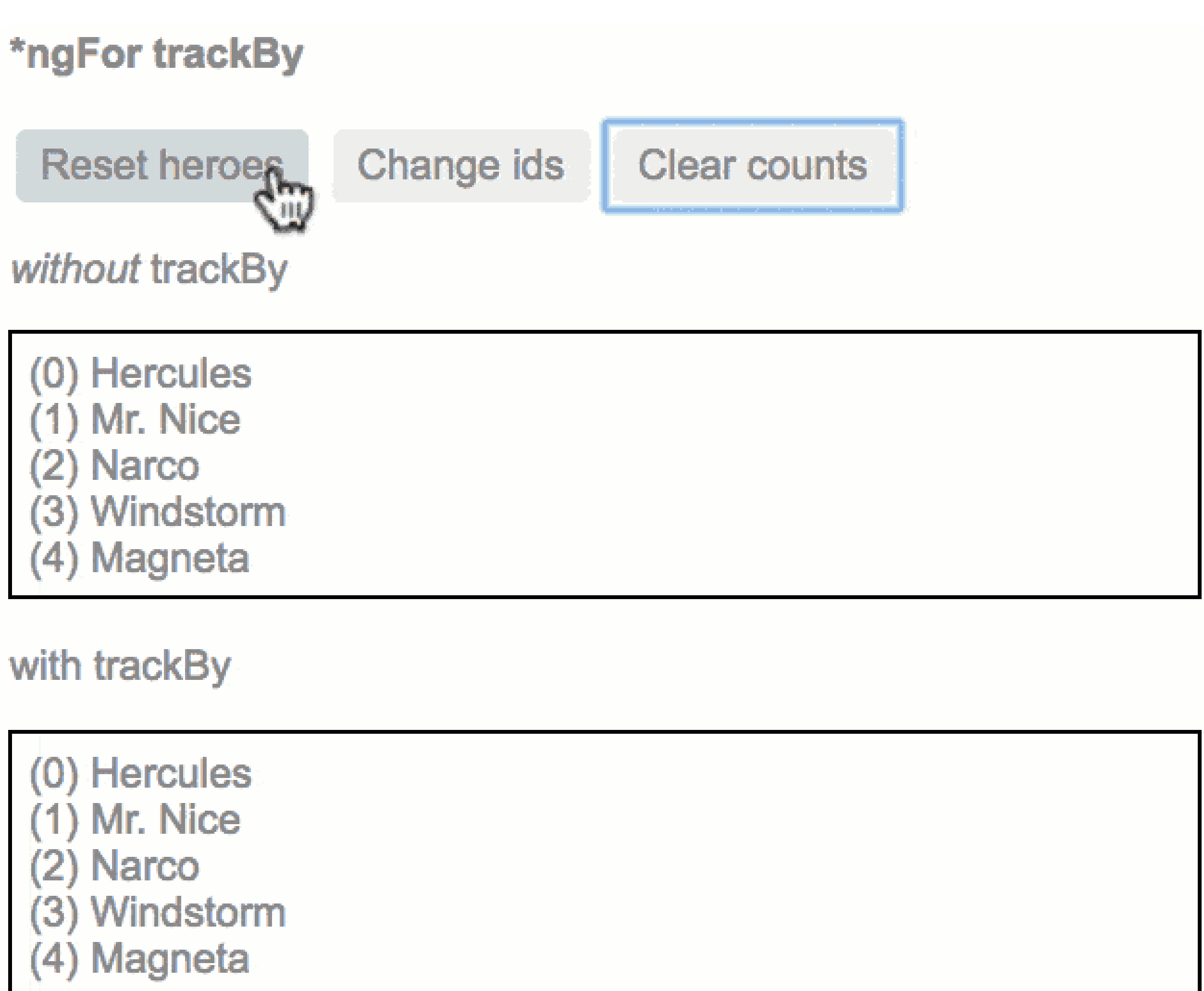
在微语法中，把 `trackBy` 设置为该方法。

src/app/app.component.html

```
<div *ngFor="let hero of heroes; trackBy: trackByHeroes">
  ({{hero.id}}) {{hero.name}}
</div>
```

这里展示了 `trackBy` 的效果。 "Reset heroes"会创建一个具有相同 `hero.id` 的新英雄。 "Change ids"则会创建一个具有新 `hero.id` 的新英雄。

- 如果没有 `trackBy`，这些按钮都会触发完全的 DOM 元素替换。
- 有了 `trackBy`，则只有修改了 `id` 的按钮才会触发元素替换。



`NgSwitch` 指令

`NgSwitch` 指令类似于 JavaScript 的 `switch` 语句。 它可以从多个可能的元素中根据**switch 条件**来显示某一个。 Angular 只会把**选中的**元素放进 DOM 中。

`NgSwitch` 实际上包括三个相互协作的指令：`NgSwitch`、`NgSwitchCase` 和 `NgSwitchDefault`，例子如下：

src/app/app.component.html

```
<div [ngSwitch]="currentHero.emotion">
  <app-happy-hero    *ngSwitchCase="'happy'"    [hero]="currentHero"></app-happy-hero>
  <app-sad-hero      *ngSwitchCase="'sad'"      [hero]="currentHero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'confused'" [hero]="currentHero"></app-confused-hero>
  <app-unknown-hero  *ngSwitchDefault           [hero]="currentHero"></app-unknown-hero>
</div>
```

Pick your favorite hero

◉ Hercules ○ Mr. Nice ○ Narco ○ Windstorm ○ Magneta

Wow. You like Hercules. What a happy hero ... just like you.

`NgSwitch` 是主控指令，要把它绑定到一个返回**候选值**的表达式。 本例子中的 `emotion` 是个字符串，但实际上这个候选值可以是任意类型。

绑定到 `[ngSwitch]`。如果试图用 `*ngSwitch` 的形式使用它就会报错，这是因为 `NgSwitch` 是一个**属性型**指令，而不是**结构型指令**。 它要修改的是所在元素的行为，而不会直接接触 DOM 结构。

绑定到 `*ngSwitchCase` 和 `*ngSwitchDefault` `NgSwitchCase` 和 `NgSwitchDefault` 指令都是**结构型指令**，因为它们会从 DOM 中添加或移除元素。

- `NgSwitchCase` 会在它绑定到的值等于候选值时，把它所在的元素加入到 DOM 中。
- `NgSwitchDefault` 会在没有任何一个 `NgSwitchCase` 被选中时把它所在的元素加入 DOM 中。

这组指令在要添加或移除**组件元素**时会非常有用。 这个例子会在 `hero-switch.components.ts` 中定义的四个“感人英雄”组件之间选择。 每个组件都有一个输入属性`hero`，它绑定到父组件的 `currentHero` 上。

这组指令在原生元素和Web Component上都可以正常工作。 比如，你可以把 `<confused-hero>` 分支改成这样：

src/app/app.component.html

```
<div *ngSwitchCase="'confused'">Are you as confused as {{currentHero.name}}?</div>
```

模板引用变量 (#var)

模板引用变量通常用来引用模板中的某个 DOM 元素，它还可以引用 Angular 组件或指令或Web Component。

使用井号 (#) 来声明引用变量。 `#phone` 的意思就是声明一个名叫 `phone` 的变量来引用 `<input>` 元素。

src/app/app.component.html

```
<input #phone placeholder="phone number">
```

你可以在模板中的任何地方引用模板引用变量。 比如声明在 `<input>` 上的 `phone` 变量就是在模板另一侧的 `<button>` 上使用的。

src/app/app.component.html

```
<input #phone placeholder="phone number">

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

模板引用变量怎么得到它的值?

大多数情况下，Angular 会把模板引用变量的值设置为声明它的那个元素。 在上一个例子中，`phone` 引用的是表示**电话号码**的 `<input>` 框。 "拨号"按钮的点击事件处理器把这个 **input** 值传给了组件的 `callPhone` 方法。 不过，指令也可以修改这种行为，让这个值引用到别处，比如它自身。 `NgForm` 指令就是这么做的。

下面是表单一章中表单范例的**简化版**。

src/app/hero-form.component.html

```
<form (ngSubmit)="onSubmit(heroForm)" #heroForm="ngForm">
  <div class="form-group">
    <label for="name">Name
      <input class="form-control" name="name" required [(ngModel)]="hero.name">
    </label>
  </div>
  <button type="submit" [disabled]="!heroForm.form.valid">Submit</button>
</form>
<div [hidden]="!heroForm.form.valid">
  {{submitMessage}}
</div>
```

模板引用变量 `heroForm` 在这个例子中出现了三次，中间隔着一大堆 HTML。 `heroForm` 的值是什么？

如果你没有导入过 `FormsModule`，Angular 就不会控制这个表单，那么它就是一个HTMLFormElement实例。这里的 `heroForm` 实际上是一个 Angular NgForm 指令的引用， 因此具备了跟踪表单中的每个控件的值和有效性的能力。

原生的 `<form>` 元素没有 `form` 属性，但 `NgForm` 指令有。这就解释了为何当 `heroForm.form.valid` 是无效时你可以禁用提交按钮， 并能把整个表单控件树传给父组件的 `onSubmit` 方法。

关于模板引用变量的注意事项

模板**引用**变量 (`#phone`) 和`*ngFor`部分看到过的模板**输入**变量 (`let phone`) 是不同的。 要了解详情，参见结构型指令一章。

模板引用变量的作用范围是**整个模板**。 不要在同一个模板中多次定义同一个变量名，否则它在运行期间的值是无法确定的。

你也可以用 `ref-` 前缀代替 `#`。 下面的例子中就用把 `fax` 变量声明成了 `ref-fax` 而不是 `#fax`。

src/app/app.component.html

```
<input ref-fax placeholder="fax number">
<button (click)="callFax(fax.value)">Fax</button>
```

输入和输出属性

输入属性是一个带有 `@Input` 装饰器的可设置属性。当它通过属性绑定的形式被绑定时，值会“流入”这个属性。

输出属性是一个带有 `@Output` 装饰器的可观察对象型的属性。 这个属性几乎总是返回 Angular 的 `EventEmitter`。 当它通过事件绑定的形式被绑定时，值会“流出”这个属性。

你只能通过它的输入和输出属性将其绑定到其它组件。

> 记住，所有的组件都是指令。
>
> 为简洁起见，以下讨论会涉及到组件，因为这个主题主要是组件作者所关心的问题。

讨论

在下面的例子中，`iconUrl` 和 `onSave` 是组件的成员，它们在 `=` 右侧引号语法中被引用了。

src/app/app.component.html

```
<img [src]="iconUrl"/>
<button (click)="onSave()">Save</button>
```

`iconUrl` 和 `onSave` 是 `AppComponent` 类的成员。但它们并没有带 `@Input()` 或 `@Output()` 装饰器。 Angular 不在乎。

你总是可以在组件自己的模板中绑定到组件的公共属性，而不用管它们是否输入（Input）属性或输出（Output）属性。

这是因为组件类和模板是紧耦合的，它们是同一个东西的两个部分，合起来构成组件。 组件类及其模板之间的交互属于实现细节。

绑定到其它组件

你也可以绑定到**其它**组件的属性。 这种绑定形式下，**其它**组件的属性位于等号（`=`）的**左侧**。

下面的例子中，`AppComponent` 的模板把 `AppComponent` 类的成员绑定到了 `HeroDetailComponent`（选择器为 `'app-hero-detail'`） 的属性上。

src/app/app.component.html

```
<app-hero-detail [hero]="currentHero" (deleteRequest)="deleteHero($event)">
</app-hero-detail>
```

Angular 的编译器**可能**会对这些绑定报错，就像这样：

```
Uncaught Error: Template parse errors:
Can't bind to 'hero' since it isn't a known property of 'app-hero-detail'
```

你自己知道 `HeroDetailComponent` 有两个属性 `hero` 和 `detectRequest`，但 Angular 编译器并不知道。

Angular 编译器不会绑定到其它组件的属性上 —— 除非这些属性是输入或输出属性。

这条规则是有充分理由的。

组件绑定到它**自己**的属性当然没问题。 该组件的作者对这些绑定有完全的控制权。

但是，其它组件不应该进行这种毫无限制的访问。 如果任何人都可以绑定到你的组件的任何属性上，那么这个组件就很难维护。 所以，外部组件应该只能绑定到组件的公共（允许绑定） API 上。

Angular 要求你**显式声明**那些 API。 它让**你**可以自己决定哪些属性是可以被外部组件绑定的。

TypeScript 的 `public` 是没用的

你不能用 TypeScript 的 `public` 和 `private` 访问控制符来标明组件的公共 API。

> 所有数据绑定属性必须是 TypeScript 的公共属性，Angular 永远不会绑定到 TypeScript 中的私有属性。

因此，Angular 需要一些其它方式来标记出那些允许被**外部**组件绑定到的属性。 这种**其它方式**，就是 `@Input()` 和 `@Output()` 装饰器。

声明输入与输出属性

在本章的例子中，绑定到 `HeroDetailComponent` 不会失败，这是因为这些要进行数据绑定的属性都带有 `@Input()` 和 `@Output()` 装饰器。

src/app/hero-detail.component.ts

```
@Input()  hero: Hero;
@Output() deleteRequest = new EventEmitter<Hero>();
```

> 另外，还可以在指令元数据的 `inputs` 或 `outputs` 数组中标记出这些成员。比如这个例子：

```
src/app/hero-detail.component.ts
```

```
@Component({
  inputs: ['hero'],
  outputs: ['deleteRequest'],
})
```

输入还是输出?

输入属性通常接收数据值。 **输出**属性暴露事件生产者，如 `EventEmitter` 对象。

输入和**输出**这两个词是从目标指令的角度来说的。

Input　　　　　　　　Output

```
<hero-detail [hero]="currentHero" (deleteRequest)="deleteHero($event)">
```

从 `HeroDetailComponent` 角度来看， `HeroDetailComponent.hero` 是个输入属性， 因为数据流从模板绑定表达式流**入**那个属性。

从 `HeroDetailComponent` 角度来看， `HeroDetailComponent.deleteRequest` 是个输出属性， 因为事件从那个属性流**出**，流向模板绑定语句中的处理器。

给输入/输出属性起别名

有时需要让输入/输出属性的公共名字不同于内部名字。

这是使用 attribute 指令时的常见情况。 指令的使用者期望绑定到指令名。例如，在 `<div>` 上用 `myClick` 选择器应用指令时， 希望绑定的事件属性也叫 `myClick`。

```
src/app/app.component.html
```

```
<div (myClick)="clickMessage=$event" clickable>click with myClick</div>
```

然而，在指令类中，直接用指令名作为自己的属性名通常都不是好的选择。 指令名很少能描述这个属性是干嘛的。 `myClick` 这个指令名对于用来发出 click 消息的属性就算不上一个好名字。

幸运的是，可以使用约定俗成的公共名字，同时在内部使用不同的名字。 在上面例子中，实际上是把 `myClick` 这个别名指向了指令自己的 `clicks` 属性。

把别名传进@Input/@Output 装饰器，就可以为属性指定别名，就像这样：

src/app/click.directive.ts

```
@Output('myClick') clicks = new EventEmitter<string>(); //  @Output(alias)
propertyName = ...
```

> 也可在 `inputs` 和 `outputs` 数组中为属性指定别名。 可以写一个冒号 (`:`) 分隔的字符串，**左侧**是指令中的属性名，**右侧**则是公共别名。
>
> src/app/click.directive.ts
>
> ```
> @Directive({
>   outputs: ['clicks:myClick']  // propertyName:alias
> })
> ```

模板表达式操作符

模板表达式语言使用了 JavaScript 语法的子集，并补充了几个用于特定场景的特殊操作符。 下面介绍其中的两个：**管道**和**安全导航操作符**。

管道操作符 (|)

在绑定之前，表达式的结果可能需要一些转换。例如，可能希望把数字显示成金额、强制文本变成大写，或者过滤列表以及进行排序。

Angular 管道对像这样的小型转换来说是个明智的选择。 管道是一个简单的函数，它接受一个输入值，并返回转换结果。 它们很容易用于模板表达式中，只要使用管道操作符 (`|`) 就行了。

src/app/app.component.html

```
<div>Title through uppercase pipe: {{title | uppercase}}</div>
```

管道操作符会把它左侧的表达式结果传给它右侧的管道函数。

还可以通过多个管道串联表达式：

src/app/app.component.html

```
<!-- Pipe chaining: convert title to uppercase, then to lowercase -->
<div>
  Title through a pipe chain:
  {{title | uppercase | lowercase}}
</div>
```

还能对它们使用参数:

src/app/app.component.html

```
<!-- pipe with configuration argument => "February 25, 1970" -->
<div>Birthdate: {{currentHero?.birthdate | date:'longDate'}}</div>
```

`json` 管道对调试绑定特别有用:

src/app/app.component.html (pipes-json)

```
<div>{{currentHero | json}}</div>
```

它生成的输出是这样的:

```
{ "id": 0, "name": "Hercules", "emotion": "happy",
  "birthdate": "1970-02-25T08:00:00.000Z",
  "url": "http://www.imdb.com/title/tt0065832/",
  "rate": 325 }
```

安全导航操作符 (?.) 和空属性路径

Angular 的安全导航操作符 (`?.`) 是一种流畅而便利的方式,用来保护出现在属性路径中 null 和 undefined 值。 下例中,当 `currentHero` 为空时,保护视图渲染器,让它免于失败。

src/app/app.component.html

```
The current hero's name is {{currentHero?.name}}
```

如果下列数据绑定中 `title` 属性为空,会发生什么?

src/app/app.component.html

```
The title is {{title}}
```

这个视图仍然被渲染出来，但是显示的值是空；只能看到 “The title is”，它后面却没有任何东西。 这是合理的行为。至少应用没有崩溃。

假设模板表达式涉及属性路径，在下例中，显示一个空 (null) 英雄的 `firstName`。

```
The null hero's name is {{nullHero.name}}
```

JavaScript 抛出了空引用错误，Angular 也是如此：

```
TypeError: Cannot read property 'name' of null in [null].
```

晕，**整个视图都不见了。**

如果确信 `hero` 属性永远不可能为空，可以声称这是合理的行为。 如果它必须不能为空，但它仍然是空值，实际上是制造了一个编程错误，它应该被捕获和修复。 这种情况应该抛出异常。

另一方面，属性路径中的空值可能会时常发生，特别是数据目前为空但最终会出现。

当等待数据的时候，视图渲染器不应该抱怨，而应该把这个空属性路径显示为空白，就像上面 `title` 属性那样。

不幸的是，当 `currentHero` 为空的时候，应用崩溃了。

可以通过用NgIf代码环绕它来解决这个问题。

src/app/app.component.html

```
<!--No hero, div not displayed, no error -->
<div *ngIf="nullHero">The null hero's name is {{nullHero.name}}</div>
```

或者可以尝试通过 `&&` 来把属性路径的各部分串起来，让它在遇到第一个空值的时候，就返回空。

src/app/app.component.html

```
The null hero's name is {{nullHero && nullHero.name}}
```

这些方法都有价值，但是会显得笨重，特别是当这个属性路径非常长的时候。 想象一下在一个很长的属性路径（如 `a.b.c.d`）中对空值提供保护。

Angular 安全导航操作符 (`?.`) 是在属性路径中保护空值的更加流畅、便利的方式。 表达式会在它遇到第一个空值的时候跳出。 显示是空的，但应用正常工作，而没有发生错误。

src/app/app.component.html

```
<!-- No hero, no problem! -->
The null hero's name is {{nullHero?.name}}
```

在像 `a?.b?.c?.d` 这样的长属性路径中，它工作得很完美。back to top

非空断言操作符（!）

在 TypeScript 2.0 中，你可以使用 `--strictNullChecks` 标志强制开启严格空值检查。TypeScript 就会确保不存在意料之外的 null 或 undefined。

在这种模式下，有类型的变量默认是不允许 null 或 undefined 值的，如果有未赋值的变量，或者试图把 null 或 undefined 赋值给不允许为空的变量，类型检查器就会抛出一个错误。

如果类型检查器在运行期间无法确定一个变量是 null 或 undefined，那么它也会抛出一个错误。 你自己可能知道它不会为空，但类型检查器不知道。 所以你要告诉类型检查器，它不会为空，这时就要用到**非空断言操作符**。

Angular 模板中的**非空断言操作符（`!`）也是同样的用途。

例如，在用*ngIf来检查过 `hero` 是已定义的之后，就可以断言 `hero` 属性一定是已定义的。

src/app/app.component.html

```
<!--No hero, no text -->
<div *ngIf="hero">
  The hero's name is {{hero!.name}}
</div>
```

在 Angular 编译器把你的模板转换成 TypeScript 代码时，这个操作符会防止 TypeScript 报告 "`hero.name` 可能为 null 或 undefined"的错误。

与**安全导航操作符**不同的是，非空断言操作符不会防止出现 null 或 undefined。 它只是告诉 TypeScript 的类型检查器对特定的属性表达式，不做 "严格空值检测"。

如果你打开了严格控制检测，那就要用到这个模板操作符，而其它情况下则是可选的。

回到顶部

类型转换函数 `$any` （$any(<表达式>)）

有时候，绑定表达式可能会报类型错误，并且它不能或很难指定类型。要消除这种报错，你可以使用 `$any` 转换函数来把表达式转换成 `any` 类型。

src/app/app.component.html

```
<!-- Accessing an undeclared member -->
<div>
  The hero's marker is {{$any(hero).marker}}
</div>
```

在这个例子中，当 Angular 编译器把模板转换成 TypeScript 代码时，`$any` 表达式可以防止 TypeScript 编译器报错说 `marker` 不是 `Hero` 接口的成员。

`$any` 转换函数可以和 `this` 联合使用，以便访问组件中未声明过的成员。

src/app/app.component.html

```
<!-- Accessing an undeclared member -->
<div>
  Undeclared members is {{$any(this).member}}
</div>
```

`$any` 转换函数可以在绑定表达式中任何可以进行方法调用的地方使用。

小结

你完成了模板语法的概述。现在，该把如何写组件和指令的知识投入到实际工作当中了。

生命周期钩子

每个组件都有一个被 Angular 管理的生命周期。

Angular 创建它，渲染它，创建并渲染它的子组件，在它被绑定的属性发生变化时检查它，并在它从 DOM 中被移除前销毁它。

Angular 提供了生命周期钩子，把这些关键生命时刻暴露出来，赋予你在它们发生时采取行动的能力。

除了那些组件内容和视图相关的钩子外,指令有相同生命周期钩子。

组件生命周期钩子概览

指令和组件的实例有一个生命周期：新建、更新和销毁。 通过实现一个或多个 Angular `core` 库里定义的**生命周期钩子**接口，开发者可以介入该生命周期中的这些关键时刻。

每个接口都有唯一的一个钩子方法，它们的名字是由接口名再加上 `ng` 前缀构成的。比如，`OnInit` 接口的钩子方法叫做 `ngOnInit`， Angular 在创建组件后立刻调用它，：

peek-a-boo.component.ts (excerpt)

```
export class PeekABoo implements OnInit {
  constructor(private logger: LoggerService) { }

  // implement OnInit's `ngOnInit` method
  ngOnInit() { this.logIt(`OnInit`); }

  logIt(msg: string) {
    this.logger.log(`#${nextId++} ${msg}`);
  }
}
```

没有指令或者组件会实现所有这些接口，并且有些钩子只对组件有意义。只有在指令/组件中**定义过的**那些钩子方法才会被 Angular 调用。

生命周期的顺序

当 Angular 使用构造函数新建一个组件或指令后，就会按下面的顺序在特定时刻调用这些生命周期钩子方法：

钩子	用途及时机
`ngOnChanges()`	当 Angular（重新）设置数据绑定输入属性时响应。 该方法接受当前和上一属性值的 `SimpleChanges` 对象 当被绑定的输入属性的值发生变化时调用，首次调用一定会发生在 `ngOnInit()` 之前。
`ngOnInit()`	在 Angular 第一次显示数据绑定和设置指令/组件的输入属性之后，初始化指令/组件。 在第一轮 `ngOnChanges()` 完成之后调用，只调用一次。
`ngDoCheck()`	检测，并在发生 Angular 无法或不愿意自己检测的变化时作出反应。 在每个 Angular 变更检测周期中调用，`ngOnChanges()` 和 `ngOnInit()` 之后。
`ngAfterContentInit()`	当把内容投影进组件之后调用。 第一次 `ngDoCheck()` 之后调用，只调用一次。
`ngAfterContentChecked()`	每次完成被投影组件内容的变更检测之后调用。 `ngAfterContentInit()` 和每次 `ngDoCheck()` 之后调用
`ngAfterViewInit()`	初始化完组件视图及其子视图之后调用。 第一次 `ngAfterContentChecked()` 之后调用，只调用一次。
`ngAfterViewChecked()`	每次做完组件视图和子视图的变更检测之后调用。 `ngAfterViewInit()` 和每次 `ngAfterContentChecked()` 之后调用。
`ngOnDestroy()`	当 Angular 每次销毁指令/组件之前调用并清扫。 在这儿反订阅可观察对象和分离事件处理器，以防内存泄漏。 在 Angular 销毁指令/组件之前调用。

接口是可选的（严格来说）

从纯技术的角度讲，接口对 JavaScript 和 TypeScript 的开发者都是可选的。JavaScript 语言本身没有接口。Angular 在运行时看不到 TypeScript 接口，因为它们在编译为 JavaScript 的时候已经消失了。

幸运的是，它们并不是必须的。 你并不需要在指令和组件上添加生命周期钩子接口就能获得钩子带来的好处。

Angular 会去检测这些指令和组件的类，一旦发现钩子方法被定义了，就调用它们。 Angular 会找到并调用像 `ngOnInit()` 这样的钩子方法，有没有接口无所谓。

虽然如此，在 TypeScript 指令类中添加接口是一项最佳实践，它可以获得强类型和 IDE 等编辑器带来的好处。

其它生命周期钩子

Angular 的其它子系统除了有这些组件钩子外，还可能有它们自己的生命周期钩子。

第三方库也可能会实现它们自己的钩子，以便让这些开发者在使用时能做更多的控制。

生命周期范例

在线例子 / 下载范例通过在受控于根组件 `AppComponent` 的一些组件上进行的一系列练习，演示了生命周期钩子的运作方式。

它们遵循了一个常用的模式：用**子组件**演示一个或多个生命周期钩子方法，而**父组件**被当作该**子组件**的测试台。

下面是每个练习简短的描述：

组件	说明
Peek-a-boo	展示每个生命周期钩子，每个钩子方法都会在屏幕上显示一条日志。
Spy	指令也同样有生命周期钩子。`SpyDirective` 可以利用 `ngOnInit` 和 `ngOnDestroy` 钩子在它所监视的每个元素被创建或销毁时输出日志。 本例把 `SpyDirective` 应用到父组件里的 `ngFor`**英雄**重复器(repeater)的 `<div>` 里面。
OnChanges	这里将会看到：每当组件的输入属性发生变化时，Angular 会如何以 `changes` 对象作为参数去调用 `ngOnChanges()` 钩子。 展示了该如何理解和使用 `changes` 对象。
DoCheck	实现了一个 `ngDoCheck()` 方法，通过它可以自定义变更检测逻辑。 这里将会看到：Angular 会用什么频度调用这个钩子，监视它的变化，并把这些变化输出成一条日志。
AfterView	显示 Angular 中的**视图**所指的是什么。 演示了 `ngAfterViewInit` 和 `ngAfterViewChecked` 钩子。
AfterContent	展示如何把外部内容投影进组件中，以及如何区分“投影进来的内容”和“组件的子视图”。 演示了 `ngAfterContentInit` 和 `ngAfterContentChecked` 钩子。
计数器	演示了组件和指令的组合，它们各自有自己的钩子。 在这个例子中，每当父组件递增它的输入属性 `counter` 时，`CounterComponent` 就会通过 `ngOnChanges` 记录一条变更。 同时，前一个例子中的 `SpyDirective` 被用于在 `CounterComponent` 上提供日志，它可以同时观察到日志的创建和销毁过程。

本文剩下的部分将详细讨论这些练习。

Peek-a-boo：全部钩子

`PeekABooComponent` 组件演示了组件中所有可能存在的钩子。

你可能很少、或者永远不会像这里一样实现所有这些接口。 之所以在 peek-a-boo 中这么做，是为了演示 Angular 是如何按照期望的顺序调用这些钩子的。

用户点击Create...按钮，然后点击Destroy...按钮后，日志的状态如下图所示：

Peek-A-Boo

Create PeekABooComponent

-- Lifecycle Hook Log --

#1 name is not known at construction
#2 OnChanges: name initialized to "Windstorm"
#3 OnInit
#4 DoCheck
#5 AfterContentInit
#6 AfterContentChecked
#7 AfterViewInit
#8 AfterViewChecked
#9 DoCheck
#10 AfterContentChecked
#11 AfterViewChecked
#12 DoCheck
#13 AfterContentChecked
#14 AfterViewChecked
#15 OnDestroy

日志信息的日志和所规定的钩子调用顺序是一致的： `OnChanges`、`OnInit`、`DoCheck` (3x)、`AfterContentInit`、`AfterContentChecked` (3x)、 `AfterViewInit`、`AfterViewChecked` (3x)和 `OnDestroy`

> 构造函数本质上不应该算作 Angular 的钩子。 记录确认了在创建期间那些输入属性(这里是 `name` 属性)没有被赋值。

如果用户点击**Update Hero**按钮，就会看到另一个 `OnChanges` 和至少两组 `DoCheck`、`AfterContentChecked` 和 `AfterViewChecked` 钩子。 显然，这三种钩子被触发了**很多次**，必须让这三种钩子里的逻辑尽可能的精简！

下一个例子就聚焦于这些钩子的细节上。

窥探 OnInit 和 OnDestroy

潜入这两个 spy 钩子来发现一个元素是什么时候被初始化或者销毁的。

指令是一种完美的渗透方式，这些英雄们永远不会知道该指令的存在。

> 不开玩笑了，注意下面两个关键点：

> 1. 就像对组件一样，Angular 也会对**指令**调用这些钩子方法。
>
> 2. 一个侦探(spy)指令可以让你在无法直接修改 DOM 对象实现代码的情况下，透视其内部细节。 显然，你不能修改一个原生 `<div>` 元素的实现代码。 你同样不能修改第三方组件。 但你用一个指令就能监视它们了。

这个鬼鬼祟祟的侦探指令很简单，几乎完全由 `ngOnInit()` 和 `ngOnDestroy()` 钩子组成，它通过一个注入进来的 `LoggerService` 来把消息记录到父组件中去。

src/app/spy.directive.ts

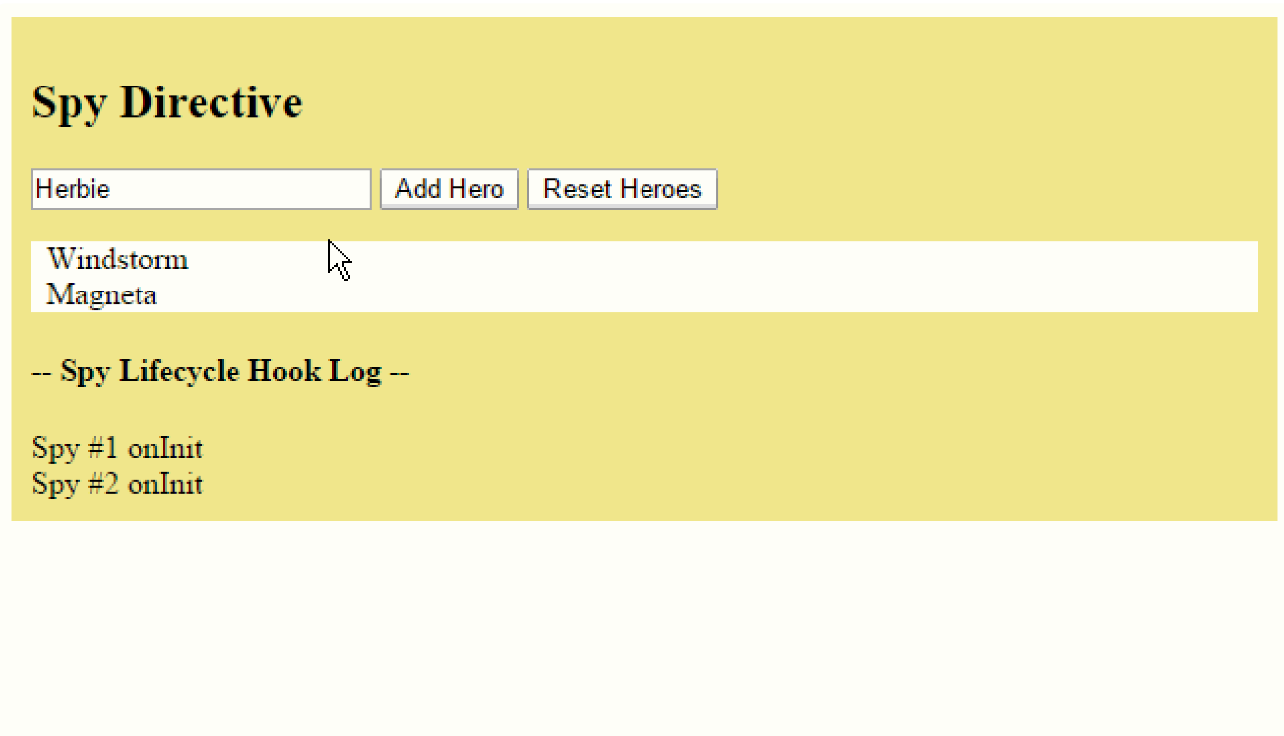
```
// Spy on any element to which it is applied.
// Usage: <div mySpy>...</div>
@Directive({selector: '[mySpy]'})
export class SpyDirective implements OnInit, OnDestroy {

  constructor(private logger: LoggerService) { }

  ngOnInit()     { this.logIt(`onInit`); }

  ngOnDestroy() { this.logIt(`onDestroy`); }

  private logIt(msg: string) {
    this.logger.log(`Spy #${nextId++} ${msg}`);
  }
}
```

你可以把这个侦探指令写到任何原生元素或组件元素上，它将与所在的组件同时初始化和销毁。 下面是把它附加到用来重复显示英雄数据的这个 `<div>` 上。

src/app/spy.component.html

```
<div *ngFor="let hero of heroes" mySpy class="heroes">
  {{hero}}
</div>
```

每个“侦探”的出生和死亡也同时标记出了存放英雄的那个 `<div>` 的出生和死亡。**钩子记录**中的结构是这样的：



添加一个英雄就会产生一个新的英雄 `<div>`。侦探的 `ngOnInit()` 记录下了这个事件。

Reset 按钮清除了这个 `heroes` 列表。 Angular 从 DOM 中移除了所有英雄的 div，并且同时销毁了附加在这些 div 上的侦探指令。 侦探的 `ngOnDestroy()` 方法汇报了它自己的临终时刻。

在真实的应用程序中，`ngOnInit()` 和 `ngOnDestroy()` 方法扮演着更重要的角色。

OnInit()钩子

使用 `ngOnInit()` 有两个原因：

1. 在构造函数之后马上执行复杂的初始化逻辑
2. 在 Angular 设置完输入属性之后，对该组件进行准备。

有经验的开发者会认同组件的构建应该很便宜和安全。

> Misko Hevery，Angular 项目的组长，在这里解释了你为什么应该避免复杂的构造函数逻辑。

不要在组件的构造函数中获取数据？ 在测试环境下新建组件时或在你决定要显示它之前，不应该担心它会尝试联系远程服务器。 构造函数中除了使用简单的值对局部变量进行初始化之外，什么都不应该做。

`ngOnInit()` 是组件获取初始数据的好地方。指南中讲解了如何这样做。

另外还要记住，在指令的**构造函数完成之前**，那些被绑定的输入属性还都没有值。 如果你需要基于这些属性的值来初始化这个指令，这种情况就会出问题。 而当 `ngOnInit()` 执行的时候，这些属性都已经被正确的赋

值过了。

> `ngOnChanges()` 方法是你访问这些属性的第一次机会，Angular 会在 `ngOnInit()` 之前调用它。 但是在那之后，Angular 还会调用 `ngOnChanges()` 很多次。而 `ngOnInit()` 只会被调用一次。

你可以信任 Angular 会在创建组件后立刻调用 `ngOnInit()` 方法。 这里是放置复杂初始化逻辑的好地方。

OnDestroy()钩子

一些清理逻辑**必须**在 Angular 销毁指令之前运行，把它们放在 `ngOnDestroy()` 中。

这是在该组件消失之前，可用来通知应用程序中其它部分的最后一个时间点。

这里是用来释放那些不会被垃圾收集器自动回收的各类资源的地方。 取消那些对可观察对象和 DOM 事件的订阅。停止定时器。注销该指令曾注册到全局服务或应用级服务中的各种回调函数。 如果不这么做，就会有导致内存泄露的风险。

OnChanges() 钩子

一旦检测到该组件(或指令)的**输入属性**发生了变化，Angular 就会调用它的 `ngOnChanges()` 方法。 本例监控 `OnChanges` 钩子。

on-changes.component.ts (excerpt)

```
ngOnChanges(changes: SimpleChanges) {
  for (let propName in changes) {
    let chng = changes[propName];
    let cur  = JSON.stringify(chng.currentValue);
    let prev = JSON.stringify(chng.previousValue);
    this.changeLog.push(`${propName}: currentValue = ${cur}, previousValue =
${prev}`);
  }
}
```

`ngOnChanges()` 方法获取了一个对象，它把每个发生变化的属性名都映射到了一个SimpleChange对象， 该对象中有属性的当前值和前一个值。这个钩子会在这些发生了变化的属性上进行迭代，并记录它们。

这个例子中的 `OnChangesComponent` 组件有两个输入属性：`hero` 和 `power`。

src/app/on-changes.component.ts

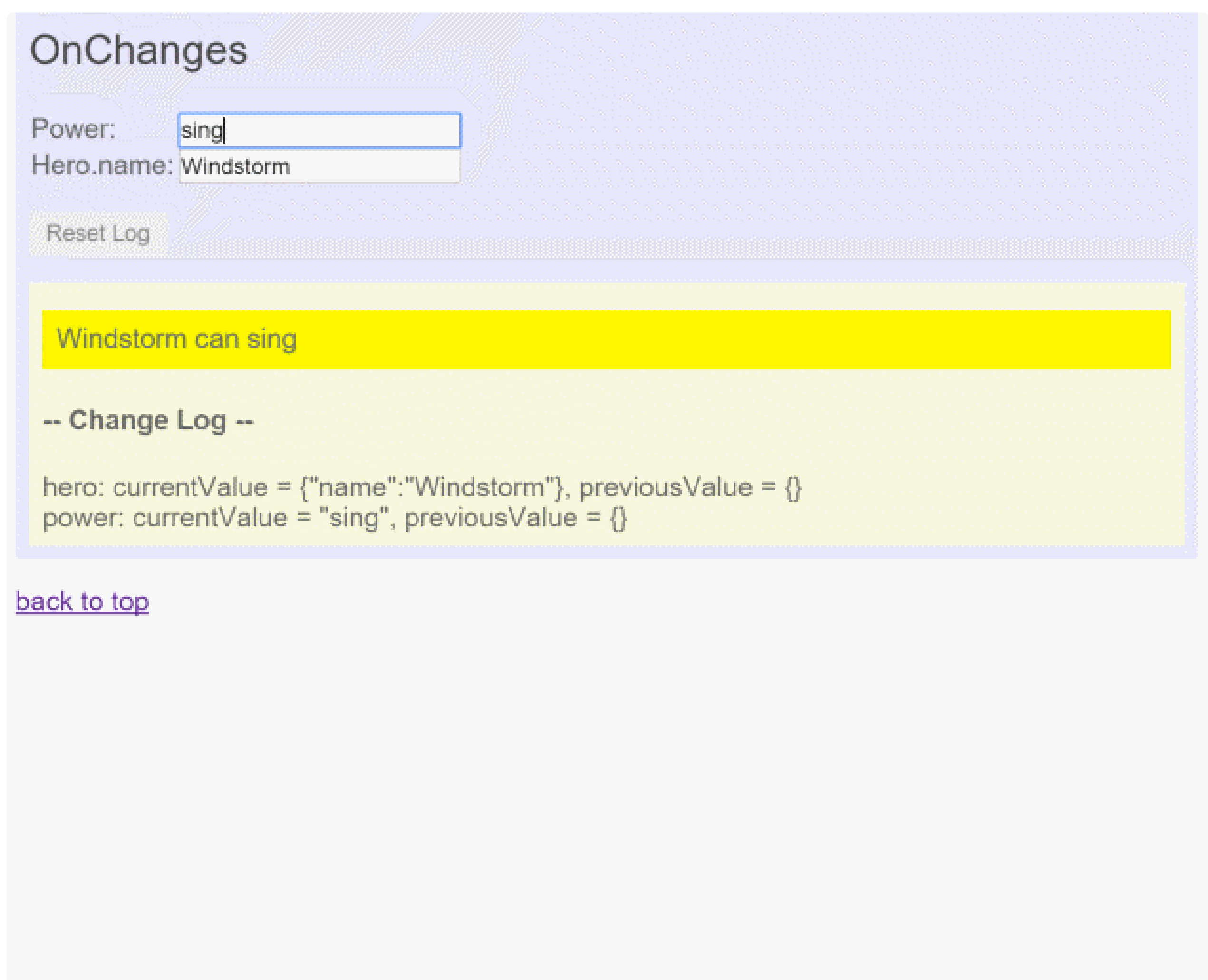
```
@Input() hero: Hero;
@Input() power: string;
```

宿主 `OnChangesParentComponent` 绑定了它们，就像这样：

src/app/on-changes-parent.component.html

```
<on-changes [hero]="hero" [power]="power"></on-changes>
```

下面是此例子中的当用户做出更改时的操作演示：



当 **power** 属性的字符串值变化时，相应的日志就出现了。 但是 `ngOnChanges` 并没有捕捉到 `hero.name` 的变化。 这是第一个意外。

Angular 只会在输入属性的值变化时调用这个钩子。 而 `hero` 属性的值是一个**到英雄对象的引用**。 Angular 不会关注这个英雄对象的 `name` 属性的变化。 这个英雄对象的**引用**没有发生变化，于是从 Angular 的视角看来，也就没有什么需要报告的变化了。

DoCheck() 钩子

使用 `DoCheck` 钩子来检测那些 Angular 自身无法捕获的变更并采取行动。

> 用这个方法来检测那些被 Angular 忽略的更改。

DoCheck 范例通过下面的 `ngDoCheck()` 实现扩展了 **OnChanges** 范例：

DoCheckComponent (ngDoCheck)

```
ngDoCheck() {

  if (this.hero.name !== this.oldHeroName) {
    this.changeDetected = true;
    this.changeLog.push(`DoCheck: Hero name changed to "${this.hero.name}" from
"${this.oldHeroName}"`);
    this.oldHeroName = this.hero.name;
  }

  if (this.power !== this.oldPower) {
    this.changeDetected = true;
    this.changeLog.push(`DoCheck: Power changed to "${this.power}" from
"${this.oldPower}"`);
    this.oldPower = this.power;
  }

  if (this.changeDetected) {
      this.noChangeCount = 0;
  } else {
      // log that hook was called when there was no relevant change.
      let count = this.noChangeCount += 1;
      let noChangeMsg = `DoCheck called ${count}x when no change to hero or power`;
      if (count === 1) {
        // add new "no change" message
        this.changeLog.push(noChangeMsg);
      } else {
        // update last "no change" message
        this.changeLog[this.changeLog.length - 1] = noChangeMsg;
      }
  }

  this.changeDetected = false;
}
```

该代码检测一些相关的值，捕获当前值并与以前的值进行比较。 当英雄或它的超能力发生了非实质性改变时，就会往日志中写一条特殊的消息。 这样你可以看到 `DoCheck` 被调用的频率。结果非常显眼：

DoCheck

Power: sing
Hero.name: Windstorm

Reset Log

Windstorm can sing

-- Change Log --

OnChanges: hero: currentValue = {"name":"Windstorm"}, previousValue = {}
OnChanges: power: currentValue = "sing", previousValue = {}
DoCheck: Hero name changed to "Windstorm" from ""
DoCheck: Power changed to "sing" from ""
DoCheck called 26x when no change to hero or power

虽然 `ngDoCheck()` 钩子可以可以监测到英雄的 `name` 什么时候发生了变化。但其开销很恐怖。 这个 `ngDoCheck` 钩子被非常频繁的调用 —— 在**每次**变更检测周期之后，发生了变化的每个地方都会调它。 在这个例子中，用户还没有做任何操作之前，它就被调用了超过二十次。

大部分检查的第一次调用都是在 Angular 首次渲染该页面中**其它不相关数据**时触发的。 仅仅把鼠标移到其它 `<input>` 中就会触发一次调用。 只有相对较少的调用才是由于对相关数据的修改而触发的。 显然，我们的实现必须非常轻量级，否则将损害用户体验。

AfterView 钩子

AfterView 例子展示了 `AfterViewInit()` 和 `AfterViewChecked()` 钩子，Angular 会在每次创建了组件的子视图后调用它们。

下面是一个子视图，它用来把英雄的名字显示在一个 `<input>` 中：

ChildComponent

```
@Component({
  selector: 'app-child-view',
  template: '<input [(ngModel)]="hero">'
})
export class ChildViewComponent {
  hero = 'Magneta';
}
```

`AfterViewComponent` 把这个子视图显示**在它的模板中**：

AfterViewComponent (template)

```
template: `
  <div>-- child view begins --</div>
    <app-child-view></app-child-view>
  <div>-- child view ends --</div>`
```

下列钩子基于**子视图中**的每一次数据变更采取行动，它只能通过带@ViewChild装饰器的属性来访问子视图。

AfterViewComponent (class excerpts)

```
export class AfterViewComponent implements  AfterViewChecked, AfterViewInit {
  private prevHero = '';

  // Query for a VIEW child of type `ChildViewComponent`
  @ViewChild(ChildViewComponent) viewChild: ChildViewComponent;

  ngAfterViewInit() {
    // viewChild is set after the view has been initialized
    this.logIt('AfterViewInit');
    this.doSomething();
  }

  ngAfterViewChecked() {
    // viewChild is updated after the view has been checked
    if (this.prevHero === this.viewChild.hero) {
      this.logIt('AfterViewChecked (no change)');
    } else {
      this.prevHero = this.viewChild.hero;
      this.logIt('AfterViewChecked');
      this.doSomething();
    }
  }
  // ...
}
```

遵循单向数据流规则

当英雄的名字超过 10 个字符时，`doSomething()` 方法就会更新屏幕。

AfterViewComponent (doSomething)

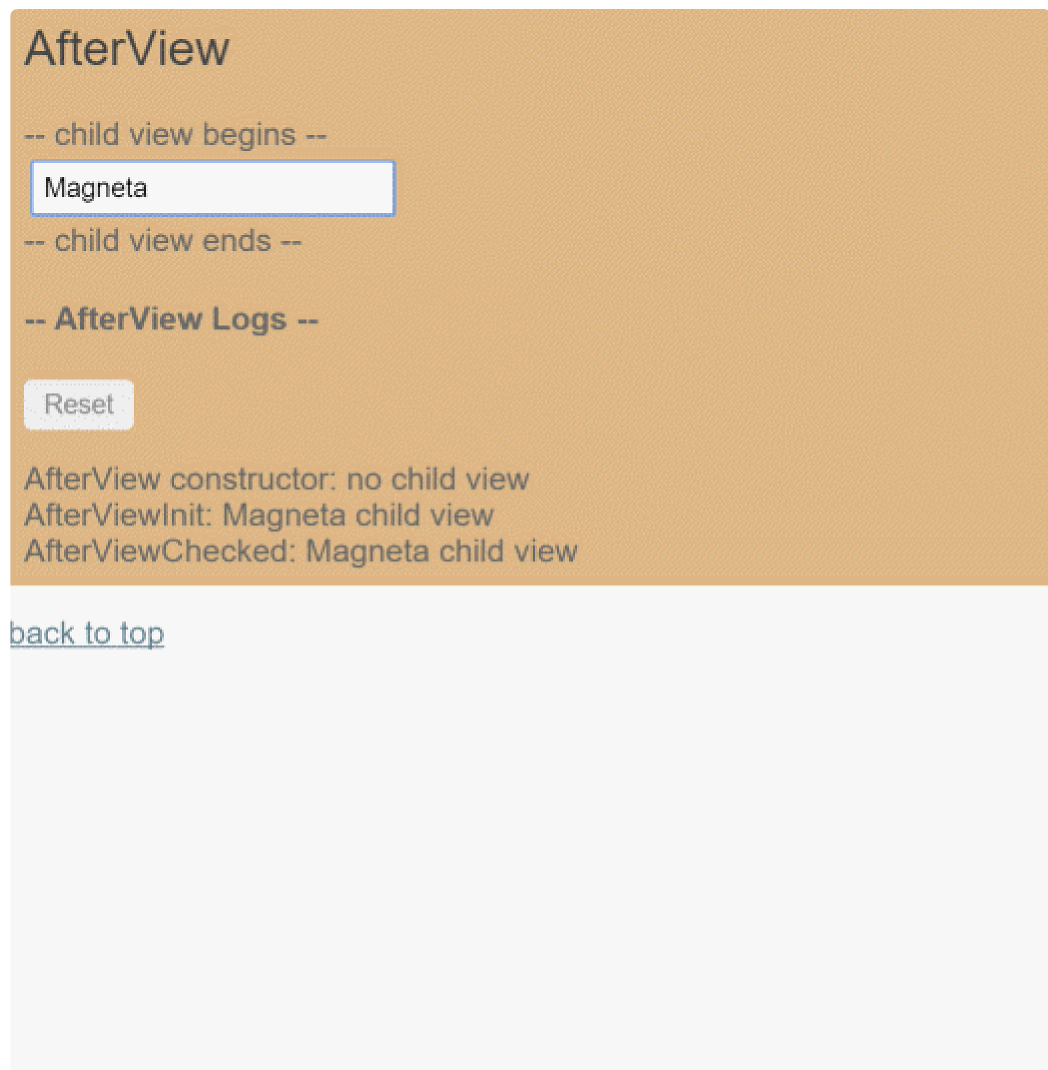
```
// This surrogate for real business logic sets the `comment`
private doSomething() {
  let c = this.viewChild.hero.length > 10 ? `That's a long name` : '';
  if (c !== this.comment) {
    // Wait a tick because the component's view has already been checked
    this.logger.tick_then(() => this.comment = c);
  }
}
```

为什么在更新 `comment` 属性之前，`doSomething()` 方法要等上一拍(tick)?

Angular 的“单向数据流”规则禁止在一个视图已经被组合好**之后**再更新视图。 而这两个钩子都是在组件的视图已经被组合好之后触发的。

如果立即更新组件中被绑定的 `comment` 属性，Angular 就会抛出一个错误(试试!)。 `LoggerService.tick_then()` 方法延迟更新日志一个回合（浏览器 JavaScript 周期回合），这样就够了。

这里是 **AfterView** 的操作演示：



注意，Angular 会频繁的调用 `AfterViewChecked()`，甚至在并没有需要关注的更改时也会触发。 所以务必把这个钩子方法写得尽可能精简，以免出现性能问题。

AfterContent 钩子

AfterContent 例子展示了 `AfterContentInit()` 和 `AfterContentChecked()` 钩子，Angular 会在外来内容被投影到组件中**之后**调用它们。

内容投影

内容投影是从组件外部导入 HTML 内容，并把它插入在组件模板中指定位置上的一种途径。

> AngularJS 的开发者大概知道一项叫做 **transclusion** 的技术，对，这就是它的马甲。

对比前一个例子考虑这个变化。 这次不再通过模板来把子视图包含进来，而是改为从 AfterContentComponent 的父组件中导入它。下面是父组件的模板：

AfterContentParentComponent (template excerpt)

```
`<after-content>
   <app-child></app-child>
 </after-content>`
```

注意，<app-child> 标签被包含在 <after-content> 标签中。 永远不要在组件标签的内部放任何内容 —— **除非你想把这些内容投影进这个组件中。**

现在来看下 <after-content> 组件的模板：

AfterContentComponent (template)

```
template: `
  <div>-- projected content begins --</div>
    <ng-content></ng-content>
  <div>-- projected content ends --</div>`
```

<ng-content> 标签是外来内容的**占位符**。 它告诉 Angular 在哪里插入这些外来内容。 在这里，被投影进去的内容就是来自父组件的 <app-child> 标签。

-- projected content begins --
Magneta
-- projected content ends --

> 下列迹象表明存在着**内容投影**：
>
> - 在组件的元素标签中有 HTML
> - 组件的模板中出现了 <ng-content> 标签

AfterContent 钩子

AfterContent 钩子和 **AfterView** 相似。关键的不同点是子组件的类型不同。

- **AfterView** 钩子所关心的是 `ViewChildren`，这些子组件的元素标签会出现在该组件的模板**里面**。
- **AfterContent** 钩子所关心的是 `ContentChildren`，这些子组件被 Angular 投影进该组件中。

下列 **AfterContent** 钩子基于**子级内容**中值的变化而采取相应的行动，它只能通过带有@ContentChild装饰器的属性来查询到“子级内容”。

AfterContentComponent (class excerpts)

```
export class AfterContentComponent implements AfterContentChecked,
AfterContentInit {
  private prevHero = '';
  comment = '';

  // Query for a CONTENT child of type `ChildComponent`
  @ContentChild(ChildComponent) contentChild: ChildComponent;

  ngAfterContentInit() {
    // contentChild is set after the content has been initialized
    this.logIt('AfterContentInit');
    this.doSomething();
  }

  ngAfterContentChecked() {
    // contentChild is updated after the content has been checked
    if (this.prevHero === this.contentChild.hero) {
      this.logIt('AfterContentChecked (no change)');
    } else {
      this.prevHero = this.contentChild.hero;
      this.logIt('AfterContentChecked');
      this.doSomething();
    }
  }
  // ...
}
```

使用 AfterContent 时，无需担心单向数据流规则

该组件的 `doSomething()` 方法立即更新了组件被绑定的 `comment` 属性。 它不用等下一回合。

回忆一下，Angular 在每次调用 **AfterView** 钩子之前也会同时调用 **AfterContent**。 Angular 在完成当前组件的视图合成之前，就已经完成了被投影内容的合成。 所以你仍然有机会去修改那个视图。

组件之间的交互

本烹饪宝典包含了常见的组件通讯场景，也就是让两个或多个组件之间共享信息的方法。

参见在线例子 / 下载范例。

通过输入型绑定把数据从父组件传到子组件。

`HeroChildComponent` 有两个**输入型属性**，它们通常带@Input 装饰器。

component-interaction/src/app/hero-child.component.ts

```
1. import { Component, Input } from '@angular/core';
2.
3. import { Hero } from './hero';
4.
5. @Component({
6.   selector: 'app-hero-child',
7.   template: `
8.     <h3>{{hero.name}} says:</h3>
9.     <p>I, {{hero.name}}, am at your service, {{masterName}}.</p>
10.   `
11. })
12. export class HeroChildComponent {
13.   @Input() hero: Hero;
14.   @Input('master') masterName: string;
15. }
```

第二个 `@Input` 为子组件的属性名 `masterName` 指定一个别名 `master`(译者注：不推荐为起别名，请参见风格指南).

父组件 `HeroParentComponent` 把子组件的 `HeroChildComponent` 放到 `*ngFor` 循环器中，把自己的 `master` 字符串属性绑定到子组件的 `master` 别名上，并把每个循环的 `hero` 实例绑定到子组件的 `hero` 属性。

component-interaction/src/app/hero-parent.component.ts

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

import { HEROES } from './hero';

@Component({
  selector: 'app-hero-parent',
  template: `
    <h2>{{master}} controls {{heroes.length}} heroes</h2>
    <app-hero-child *ngFor="let hero of heroes"
      [hero]="hero"
      [master]="master">
    </app-hero-child>
  `
})
export class HeroParentComponent {
  heroes = HEROES;
  master = 'Master';
}
```

运行应用程序会显示三个英雄：

Master controls 3 heroes

Mr. IQ says:

I, Mr. IQ, am at your service, Master.

Magneta says:

I, Magneta, am at your service, Master.

Bombasto says:

I, Bombasto, am at your service, Master.

测试一下！

端到端测试，用于确保所有的子组件都像所期待的那样被初始化并显示出来。

component-interaction/e2e/src/app.e2e-spec.ts

```
// ...
let _heroNames = ['Mr. IQ', 'Magneta', 'Bombasto'];
let _masterName = 'Master';

it('should pass properties to children properly', function () {
  let parent = element.all(by.tagName('app-hero-parent')).get(0);
  let heroes = parent.all(by.tagName('app-hero-child'));

  for (let i = 0; i < _heroNames.length; i++) {
    let childTitle = heroes.get(i).element(by.tagName('h3')).getText();
    let childDetail = heroes.get(i).element(by.tagName('p')).getText();
    expect(childTitle).toEqual(_heroNames[i] + ' says:');
    expect(childDetail).toContain(_masterName);
  }
});
// ...
```

回到顶部

通过 setter 截听输入属性值的变化

使用一个输入属性的 setter，以拦截父组件中值的变化，并采取行动。

子组件 `NameChildComponent` 的输入属性 `name` 上的这个 setter，会 trim 掉名字里的空格，并把空值替换成默认字符串。

component-interaction/src/app/name-child.component.ts

```
1. import { Component, Input } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-name-child',
5.   template: '<h3>"{{name}}"</h3>'
6. })
7. export class NameChildComponent {
8.   private _name = '';
9.
10.   @Input()
11.   set name(name: string) {
12.     this._name = (name && name.trim()) || '<no name set>';
13.   }
14.
15.   get name(): string { return this._name; }
16. }
```

下面的 `NameParentComponent` 展示了各种名字的处理方式，包括一个全是空格的名字。

component-interaction/src/app/name-parent.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-name-parent',
5.   template: `
6.   <h2>Master controls {{names.length}} names</h2>
7.   <app-name-child *ngFor="let name of names" [name]="name"></app-name-
   child>
8.   `
9. })
10. export class NameParentComponent {
11.   // Displays 'Mr. IQ', '<no name set>', 'Bombasto'
12.   names = ['Mr. IQ', '   ', '  Bombasto  '];
13. }
```

Master controls 3 names

"Mr. IQ"

"<no name set>"

"Bombasto"

测试一下!

端到端测试：输入属性的 setter，分别使用空名字和非空名字。

component-interaction/e2e/src/app.e2e-spec.ts

```
// ...
it('should display trimmed, non-empty names', function () {
  let _nonEmptyNameIndex = 0;
  let _nonEmptyName = '"Mr. IQ"';
  let parent = element.all(by.tagName('app-name-parent')).get(0);
  let hero = parent.all(by.tagName('app-name-child')).get(_nonEmptyNameIndex);

  let displayName = hero.element(by.tagName('h3')).getText();
  expect(displayName).toEqual(_nonEmptyName);
});

it('should replace empty name with default name', function () {
  let _emptyNameIndex = 1;
  let _defaultName = '"<no name set>"';
  let parent = element.all(by.tagName('app-name-parent')).get(0);
  let hero = parent.all(by.tagName('app-name-child')).get(_emptyNameIndex);

  let displayName = hero.element(by.tagName('h3')).getText();
  expect(displayName).toEqual(_defaultName);
});
// ...
```

回到顶部

通过ngOnChanges()来截听输入属性值的变化

使用 `OnChanges` 生命周期钩子接口的 `ngOnChanges()` 方法来监测输入属性值的变化并做出回应。

> 当需要监视多个、交互式输入属性的时候，本方法比用属性的 setter 更合适。
>
> 学习关于 `ngOnChanges()` 的更多知识，参见生命周期钩子一章。

这个 `VersionChildComponent` 会监测输入属性 `major` 和 `minor` 的变化，并把这些变化编写成日志以报告这些变化。

component-interaction/src/app/version-child.component.ts

```
import { Component, Input, OnChanges, SimpleChange } from '@angular/core';

@Component({
  selector: 'app-version-child',
  template: `
    <h3>Version {{major}}.{{minor}}</h3>
    <h4>Change log:</h4>
    <ul>
      <li *ngFor="let change of changeLog">{{change}}</li>
    </ul>
  `
})
export class VersionChildComponent implements OnChanges {
  @Input() major: number;
  @Input() minor: number;
  changeLog: string[] = [];

  ngOnChanges(changes: {[propKey: string]: SimpleChange}) {
    let log: string[] = [];
    for (let propName in changes) {
      let changedProp = changes[propName];
      let to = JSON.stringify(changedProp.currentValue);
      if (changedProp.isFirstChange()) {
        log.push(`Initial value of ${propName} set to ${to}`);
      } else {
        let from = JSON.stringify(changedProp.previousValue);
        log.push(`${propName} changed from ${from} to ${to}`);
      }
    }
    this.changeLog.push(log.join(', '));
  }
}
```

`VersionParentComponent` 提供 `minor` 和 `major` 值，把修改它们值的方法绑定到按钮上。

component-interaction/src/app/version-parent.component.ts

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

@Component({
  selector: 'app-version-parent',
  template: `
    <h2>Source code version</h2>
    <button (click)="newMinor()">New minor version</button>
    <button (click)="newMajor()">New major version</button>
    <app-version-child [major]="major" [minor]="minor"></app-version-child>
  `
})
export class VersionParentComponent {
  major = 1;
  minor = 23;

  newMinor() {
    this.minor++;
  }

  newMajor() {
    this.major++;
    this.minor = 0;
  }
}
```

下面是点击按钮的结果。

Source code version

New minor version New major version

Version 1.23

Change log:

- Initial value of major set to 1, Initial value of minor set to 23

测试一下！

测试确保**这两个**输入属性值都被初始化了，当点击按钮后，`ngOnChanges` 应该被调用，属性的值也符合预期。

component-interaction/e2e/src/app.e2e-spec.ts

```
1. // ...
2. // Test must all execute in this exact order
3. it('should set expected initial values', function () {
4.   let actual = getActual();
5.
6.   let initialLabel = 'Version 1.23';
7.   let initialLog = 'Initial value of major set to 1, Initial value of minor
   set to 23';
8.
9.   expect(actual.label).toBe(initialLabel);
10.  expect(actual.count).toBe(1);
11.  expect(actual.logs.get(0).getText()).toBe(initialLog);
12. });
13.
14. it('should set expected values after clicking \'Minor\' twice', function ()
   {
15.   let repoTag = element(by.tagName('app-version-parent'));
16.   let newMinorButton = repoTag.all(by.tagName('button')).get(0);
17.
18.   newMinorButton.click().then(function() {
19.     newMinorButton.click().then(function() {
20.       let actual = getActual();
21.
22.       let labelAfter2Minor = 'Version 1.25';
23.       let logAfter2Minor = 'minor changed from 24 to 25';
24.
25.       expect(actual.label).toBe(labelAfter2Minor);
26.       expect(actual.count).toBe(3);
27.       expect(actual.logs.get(2).getText()).toBe(logAfter2Minor);
28.     });
29.   });
30. });
31.
32. it('should set expected values after clicking \'Major\' once', function ()
   {
33.   let repoTag = element(by.tagName('app-version-parent'));
34.   let newMajorButton = repoTag.all(by.tagName('button')).get(1);
35.
36.   newMajorButton.click().then(function() {
37.     let actual = getActual();
38.
```

```
39.     let labelAfterMajor = 'Version 2.0';
40.     let logAfterMajor = 'major changed from 1 to 2, minor changed from 25
   to 0';
41. 
42.     expect(actual.label).toBe(labelAfterMajor);
43.     expect(actual.count).toBe(4);
44.     expect(actual.logs.get(3).getText()).toBe(logAfterMajor);
45.   });
46. });
47. 
48. function getActual() {
49.   let versionTag = element(by.tagName('app-version-child'));
50.   let label = versionTag.element(by.tagName('h3')).getText();
51.   let ul = versionTag.element((by.tagName('ul')));
52.   let logs = ul.all(by.tagName('li'));
53. 
54.   return {
55.     label: label,
56.     logs: logs,
57.     count: logs.count()
58.   };
59. }
60. // ...
```

回到顶部

父组件监听子组件的事件

子组件暴露一个 `EventEmitter` 属性，当事件发生时，子组件利用该属性 `emits`(向上弹射)事件。父组件绑定到这个事件属性，并在事件发生时作出回应。

子组件的 `EventEmitter` 属性是一个输出属性，通常带有@Output 装饰器，就像在 `VoterComponent` 中看到的。

component-interaction/src/app/voter.component.ts

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

@Component({
  selector: 'app-voter',
  template: `
    <h4>{{name}}</h4>
    <button (click)="vote(true)"  [disabled]="voted">Agree</button>
    <button (click)="vote(false)" [disabled]="voted">Disagree</button>
  `
})
export class VoterComponent {
  @Input()  name: string;
  @Output() onVoted = new EventEmitter<boolean>();
  voted = false;

  vote(agreed: boolean) {
    this.onVoted.emit(agreed);
    this.voted = true;
  }
}
```

点击按钮会触发 `true` 或 `false`(布尔型**有效载荷**)的事件。

父组件 `VoteTakerComponent` 绑定了一个事件处理器(`onVoted()`)，用来响应子组件的事件(`$event`)并更新一个计数器。

component-interaction/src/app/votetaker.component.ts

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

@Component({
  selector: 'app-vote-taker',
  template: `
    <h2>Should mankind colonize the Universe?</h2>
    <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>
    <app-voter *ngFor="let voter of voters"
      [name]="voter"
      (onVoted)="onVoted($event)">
    </app-voter>
  `
})
export class VoteTakerComponent {
  agreed = 0;
  disagreed = 0;
  voters = ['Mr. IQ', 'Ms. Universe', 'Bombasto'];

  onVoted(agreed: boolean) {
    agreed ? this.agreed++ : this.disagreed++;
  }
}
```

框架(Angular)把事件参数(用 `$event` 表示)传给事件处理方法，这个方法会处理：

Should mankind colonize the Universe?

Agree: 0, Disagree: 0

Mr. IQ

Agree Disagree

Ms. Universe

Agree Disagree

Bombasto

Agree Disagree

测试一下！

测试确保点击 **Agree** 和 **Disagree** 按钮时，计数器被正确更新。

component-interaction/e2e/src/app.e2e-spec.ts

```
// ...
it('should not emit the event initially', function () {
  let voteLabel = element(by.tagName('app-vote-taker'))
    .element(by.tagName('h3')).getText();
  expect(voteLabel).toBe('Agree: 0, Disagree: 0');
});

it('should process Agree vote', function () {
  let agreeButton1 = element.all(by.tagName('app-voter')).get(0)
    .all(by.tagName('button')).get(0);
  agreeButton1.click().then(function() {
    let voteLabel = element(by.tagName('app-vote-taker'))
      .element(by.tagName('h3')).getText();
    expect(voteLabel).toBe('Agree: 1, Disagree: 0');
  });
});

it('should process Disagree vote', function () {
  let agreeButton1 = element.all(by.tagName('app-voter')).get(1)
    .all(by.tagName('button')).get(1);
  agreeButton1.click().then(function() {
    let voteLabel = element(by.tagName('app-vote-taker'))
      .element(by.tagName('h3')).getText();
    expect(voteLabel).toBe('Agree: 1, Disagree: 1');
  });
});
// ...
```

回到顶部

父组件与子组件通过**本地变量**互动

父组件不能使用数据绑定来读取子组件的属性或调用子组件的方法。但可以在父组件模板里，新建一个本地变量来代表子组件，然后利用这个变量来读取子组件的属性和调用子组件的方法，如下例所示。

子组件 `CountdownTimerComponent` 进行倒计时，归零时发射一个导弹。`start` 和 `stop` 方法负责控制时钟并在模板里显示倒计时的状态信息。

component-interaction/src/app/countdown-timer.component.ts

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

@Component({
  selector: 'app-countdown-timer',
  template: '<p>{{message}}</p>'
})
export class CountdownTimerComponent implements OnInit, OnDestroy {

  intervalId = 0;
  message = '';
  seconds = 11;

  clearTimer() { clearInterval(this.intervalId); }

  ngOnInit()    { this.start(); }
  ngOnDestroy() { this.clearTimer(); }

  start() { this.countDown(); }
  stop()  {
    this.clearTimer();
    this.message = `Holding at T-${this.seconds} seconds`;
  }

  private countDown() {
    this.clearTimer();
    this.intervalId = window.setInterval(() => {
      this.seconds -= 1;
      if (this.seconds === 0) {
        this.message = 'Blast off!';
      } else {
        if (this.seconds < 0) { this.seconds = 10; } // reset
        this.message = `T-${this.seconds} seconds and counting`;
      }
    }, 1000);
  }
}
```

计时器组件的宿主组件 `CountdownLocalVarParentComponent` 如下：

component-interaction/src/app/countdown-parent.component.ts

```
import { Component }                from '@angular/core';
import { CountdownTimerComponent }  from './countdown-timer.component';

@Component({
  selector: 'app-countdown-parent-lv',
  template: `
  <h3>Countdown to Liftoff (via local variable)</h3>
  <button (click)="timer.start()">Start</button>
  <button (click)="timer.stop()">Stop</button>
  <div class="seconds">{{timer.seconds}}</div>
  <app-countdown-timer #timer></app-countdown-timer>
  `,
  styleUrls: ['../assets/demo.css']
})
export class CountdownLocalVarParentComponent { }
```

父组件不能通过数据绑定使用子组件的 `start` 和 `stop` 方法，也不能访问子组件的 `seconds` 属性。

把本地变量(`#timer`)放到(`<countdown-timer>`)标签中，用来代表子组件。这样父组件的模板就得到了子组件的引用，于是可以在父组件的模板中访问子组件的所有属性和方法。

这个例子把父组件的按钮绑定到子组件的 `start` 和 `stop` 方法，并用插值表达式来显示子组件的 `seconds` 属性。

下面是父组件和子组件一起工作时的效果。

Countdown to Liftoff

Start Stop

10

T-10 seconds and counting

测试一下!

测试确保在父组件模板中显示的秒数和子组件状态信息里的秒数同步。它还会点击 **Stop** 按钮来停止倒计时：

component-interaction/e2e/src/app.e2e-spec.ts

```
// ...
it('timer and parent seconds should match', function () {
  let parent = element(by.tagName(parentTag));
  let message = parent.element(by.tagName('app-countdown-timer')).getText();
  browser.sleep(10); // give `seconds` a chance to catchup with `message`
  let seconds = parent.element(by.className('seconds')).getText();
  expect(message).toContain(seconds);
});

it('should stop the countdown', function () {
  let parent = element(by.tagName(parentTag));
  let stopButton = parent.all(by.tagName('button')).get(1);

  stopButton.click().then(function() {
    let message = parent.element(by.tagName('app-countdown-timer')).getText();
    expect(message).toContain('Holding');
  });
});
// ...
```

回到顶部

父组件调用@ViewChild()

这个**本地变量**方法是个简单便利的方法。但是它也有局限性，因为父组件-子组件的连接必须全部在父组件的模板中进行。父组件本身的代码对子组件没有访问权。

如果父组件的**类**需要读取子组件的属性值或调用子组件的方法，就不能使用**本地变量**方法。

当父组件**类**需要这种访问时，可以把子组件作为 **ViewChild**，**注入**到父组件里面。

下面的例子用与倒计时相同的范例来解释这种技术。 它的外观或行为没有变化。子组件CountdownTimerComponent也和原来一样。

> 由**本地变量**切换到 **ViewChild** 技术的唯一目的就是做示范。

下面是父组件 `CountdownViewChildParentComponent`:

component-interaction/src/app/countdown-parent.component.ts

```
import { AfterViewInit, ViewChild } from '@angular/core';
import { Component }                from '@angular/core';
import { CountdownTimerComponent }  from './countdown-timer.component';

@Component({
  selector: 'app-countdown-parent-vc',
  template: `
  <h3>Countdown to Liftoff (via ViewChild)</h3>
  <button (click)="start()">Start</button>
  <button (click)="stop()">Stop</button>
  <div class="seconds">{{ seconds() }}</div>
  <app-countdown-timer></app-countdown-timer>
  `,
  styleUrls: ['../assets/demo.css']
})
export class CountdownViewChildParentComponent implements AfterViewInit {

  @ViewChild(CountdownTimerComponent)
  private timerComponent: CountdownTimerComponent;

  seconds() { return 0; }

  ngAfterViewInit() {
    // Redefine `seconds()` to get from the `CountdownTimerComponent.seconds` ...
    // but wait a tick first to avoid one-time devMode
    // unidirectional-data-flow-violation error
    setTimeout(() => this.seconds = () => this.timerComponent.seconds, 0);
  }

  start() { this.timerComponent.start(); }
  stop() { this.timerComponent.stop(); }
}
```

把子组件的视图插入到父组件类需要做一点额外的工作。

首先，你要使用 `ViewChild` 装饰器导入这个引用，并挂上 `AfterViewInit` 生命周期钩子。

接着，通过 `@ViewChild` 属性装饰器，将子组件 `CountdownTimerComponent` 注入到私有属性 `timerComponent` 里面。

组件元数据里就不再需要 `#timer` 本地变量了。而是把按钮绑定到父组件自己的 `start` 和 `stop` 方法，使用父组件的 `seconds` 方法的插值表达式来展示秒数变化。

这些方法可以直接访问被注入的计时器组件。

`ngAfterViewInit()` 生命周期钩子是非常重要的一步。被注入的计时器组件只有在 Angular 显示了父组件视图之后才能访问，所以它先把秒数显示为 0.

然后 Angular 会调用 `ngAfterViewInit` 生命周期钩子，但这时候再更新父组件视图的倒计时就已经太晚了。Angular 的单向数据流规则会阻止在同一个周期内更新父组件视图。应用在显示秒数之前会被迫**再等一轮**。

使用 `setTimeout()` 来等下一轮，然后改写 `seconds()` 方法，这样它接下来就会从注入的这个计时器组件里获取秒数的值。

测试一下！

使用和之前一样的倒计时测试。

回到顶部

父组件和子组件通过服务来通讯

父组件和它的子组件共享同一个服务，利用该服务**在家庭内部**实现双向通讯。

该服务实例的作用域被限制在父组件和其子组件内。这个组件子树之外的组件将无法访问该服务或者与它们通讯。

这个 `MissionService` 把 `MissionControlComponent` 和多个 `AstronautComponent` 子组件连接起来。

component-interaction/src/app/mission.service.ts

```
import { Injectable } from '@angular/core';
import { Subject }    from 'rxjs';

@Injectable()
export class MissionService {

  // Observable string sources
  private missionAnnouncedSource = new Subject<string>();
  private missionConfirmedSource = new Subject<string>();

  // Observable string streams
  missionAnnounced$ = this.missionAnnouncedSource.asObservable();
  missionConfirmed$ = this.missionConfirmedSource.asObservable();

  // Service message commands
  announceMission(mission: string) {
    this.missionAnnouncedSource.next(mission);
  }

  confirmMission(astronaut: string) {
    this.missionConfirmedSource.next(astronaut);
  }
}
```

`MissionControlComponent` 提供服务的实例，并将其共享给它的子组件(通过 `providers` 元数据数组)，子组件可以通过构造函数将该实例注入到自身。

component-interaction/src/app/missioncontrol.component.ts

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

import { MissionService }     from './mission.service';

@Component({
  selector: 'app-mission-control',
  template: `
    <h2>Mission Control</h2>
    <button (click)="announce()">Announce mission</button>
    <app-astronaut *ngFor="let astronaut of astronauts"
      [astronaut]="astronaut">
    </app-astronaut>
    <h3>History</h3>
    <ul>
      <li *ngFor="let event of history">{{event}}</li>
    </ul>
  `,
  providers: [MissionService]
})
export class MissionControlComponent {
  astronauts = ['Lovell', 'Swigert', 'Haise'];
  history: string[] = [];
  missions = ['Fly to the moon!',
              'Fly to mars!',
              'Fly to Vegas!'];
  nextMission = 0;

  constructor(private missionService: MissionService) {
    missionService.missionConfirmed$.subscribe(
      astronaut => {
        this.history.push(`${astronaut} confirmed the mission`);
      });
  }

  announce() {
    let mission = this.missions[this.nextMission++];
    this.missionService.announceMission(mission);
    this.history.push(`Mission "${mission}" announced`);
    if (this.nextMission >= this.missions.length) { this.nextMission = 0; }
  }
}
```

`AstronautComponent` 也通过自己的构造函数注入该服务。由于每个 `AstronautComponent` 都是 `MissionControlComponent` 的子组件，所以它们获取到的也是父组件的这个服务实例。

component-interaction/src/app/astronaut.component.ts

```
import { Component, Input, OnDestroy } from '@angular/core';

import { MissionService } from './mission.service';
import { Subscription }   from 'rxjs';

@Component({
  selector: 'app-astronaut',
  template: `
    <p>
      {{astronaut}}: <strong>{{mission}}</strong>
      <button
        (click)="confirm()"
        [disabled]="!announced || confirmed">
        Confirm
      </button>
    </p>
  `
})
export class AstronautComponent implements OnDestroy {
  @Input() astronaut: string;
  mission = '<no mission announced>';
  confirmed = false;
  announced = false;
  subscription: Subscription;

  constructor(private missionService: MissionService) {
    this.subscription = missionService.missionAnnounced$.subscribe(
      mission => {
        this.mission = mission;
        this.announced = true;
        this.confirmed = false;
    });
  }

  confirm() {
    this.confirmed = true;
    this.missionService.confirmMission(this.astronaut);
  }

  ngOnDestroy() {
    // prevent memory leak when component destroyed
```

```
42.     this.subscription.unsubscribe();
43.   }
44. }
```

> 注意，这个例子保存了 `subscription` 变量，并在 `AstronautComponent` 被销毁时调用 `unsubscribe()` 退订。 这是一个用于防止内存泄漏的保护措施。实际上，在这个应用程序中并没有这个风险，因为 `AstronautComponent` 的生命期和应用程序的生命期一样长。但在更复杂的应用程序环境中就不一定了。
>
> 不需要在 `MissionControlComponent` 中添加这个保护措施，因为它作为父组件，控制着 `MissionService` 的生命期。

History 日志证明了：在父组件 `MissionControlComponent` 和子组件 `AstronautComponent` 之间，信息通过该服务实现了双向传递。

Mission Control

Announce mission

Lovell: **<no mission announced>** Confirm

Swigert: **<no mission announced>** Confirm

Haise: **<no mission announced>** Confirm

History

测试一下！

测试确保点击父组件 `MissionControlComponent` 和子组件 `AstronautComponent` 两个的组件的按钮时，**History** 日志和预期的一样。

component-interaction/e2e/src/app.e2e-spec.ts

```
1. // ...
2. it('should announce a mission', function () {
3.   let missionControl = element(by.tagName('app-mission-control'));
4.   let announceButton = missionControl.all(by.tagName('button')).get(0);
5.   announceButton.click().then(function () {
6.     let history = missionControl.all(by.tagName('li'));
7.     expect(history.count()).toBe(1);
8.     expect(history.get(0).getText()).toMatch(/Mission.* announced/);
9.   });
10. });
11.
12. it('should confirm the mission by Lovell', function () {
13.   testConfirmMission(1, 2, 'Lovell');
14. });
15.
16. it('should confirm the mission by Haise', function () {
17.   testConfirmMission(3, 3, 'Haise');
18. });
19.
20. it('should confirm the mission by Swigert', function () {
21.   testConfirmMission(2, 4, 'Swigert');
22. });
23.
24. function testConfirmMission(buttonIndex: number, expectedLogCount: number,
    astronaut: string) {
25.   let _confirmedLog = ' confirmed the mission';
26.   let missionControl = element(by.tagName('app-mission-control'));
27.   let confirmButton =
    missionControl.all(by.tagName('button')).get(buttonIndex);
28.   confirmButton.click().then(function () {
29.     let history = missionControl.all(by.tagName('li'));
30.     expect(history.count()).toBe(expectedLogCount);
31.     expect(history.get(expectedLogCount - 1).getText()).toBe(astronaut +
    _confirmedLog);
32.   });
33. }
34. // ...
```

回到顶部

组件样式

Angular 应用使用标准的 CSS 来设置样式。这意味着你可以把关于 CSS 的那些知识和技能直接用于 Angular 程序中，例如：样式表、选择器、规则以及媒体查询等。

另外，Angular 还能把**组件样式**捆绑在组件上，以实现比标准样式表更加模块化的设计。

本章将会讲解如何加载和使用这些**组件样式**。

你可以运行在线例子 / 下载范例，在 Stackblitz 中试用并下载本页的代码。

使用组件样式

对你编写的每个 Angular 组件来说，除了定义 HTML 模板之外，还要定义用于模板的 CSS 样式、 指定任意的选择器、规则和媒体查询。

实现方式之一，是在组件的元数据中设置 `styles` 属性。 `styles` 属性可以接受一个包含 CSS 代码的字符串数组。 通常你只给它一个字符串就行了，如同下例：

src/app/hero-app.component.ts

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Tour of Heroes</h1>
    <app-hero-main [hero]="hero"></app-hero-main>
  `,
  styles: ['h1 { font-weight: normal; }']
})
export class HeroAppComponent {
/* . . . */
}
```

范围化的样式

> 在 `@Component` 的元数据中指定的样式只会对该组件的模板生效。

它们既不会被模板中嵌入的组件继承，也不会被通过内容投影（如 ng-content）嵌进来的组件继承。

在这个例子中， `h1` 的样式只对 `HeroAppComponent` 生效，既不会作用于内嵌的 `HeroMainComponent` ，也不会作用于应用中其它任何地方的 `<h1>` 标签。

这种范围限制就是所谓的**样式模块化**特性

- 可以使用对每个组件最有意义的 CSS 类名和选择器。
- 类名和选择器是仅属于组件内部的，它不会和应用中其它地方的类名和选择器出现冲突。
- 组件的样式**不会**因为别的地方修改了样式而被意外改变。
- 你可以让每个组件的 CSS 代码和它的 TypeScript、HTML 代码放在一起，这将促成清爽整洁的项目结构。
- 将来你可以修改或移除组件的 CSS 代码，而不用遍历整个应用来看它有没有被别处用到，只要看看当前组件就可以了。

特殊的选择器

组件样式中有一些从影子(Shadow) DOM 样式范围领域（记录在W3C的CSS Scoping Module Level 1中）引入的特殊**选择器**：

:host 选择器

使用 `:host` 伪类选择器，用来选择组件**宿主**元素中的元素（相对于组件模板**内部**的元素）。

src/app/hero-details.component.css

```
:host {
  display: block;
  border: 1px solid black;
}
```

`:host` 选择是是把宿主元素作为目标的**唯一**方式。除此之外，你将没办法指定它， 因为宿主不是组件自身模板的一部分，而是父组件模板的一部分。

要把宿主样式作为条件，就要像**函数**一样把其它选择器放在 `:host` 后面的括号中。

下一个例子再次把宿主元素作为目标，但是只有当它同时带有 `active` CSS 类的时候才会生效。

src/app/hero-details.component.css

```
:host(.active) {
  border-width: 3px;
}
```

:host-context 选择器

有时候，基于某些来自组件视图**外部**的条件应用样式是很有用的。 例如，在文档的 `<body>` 元素上可能有一个用于表示样式主题 (theme) 的 CSS 类，你应当基于它来决定组件的样式。

这时可以使用 `:host-context()` 伪类选择器。它也以类似 `:host()` 形式使用。它在当前组件宿主元素的**祖先节点**中查找 CSS 类， 直到文档的根节点为止。在与其它选择器组合使用时，它非常有用。

在下面的例子中，只有当某个祖先元素有 CSS 类 `theme-light` 时，才会把 `background-color` 样式应用到组件**内部**的所有 `<h2>` 元素中。

src/app/hero-details.component.css

```
:host-context(.theme-light) h2 {
  background-color: #eef;
}
```

已废弃 `/deep/`、`>>>` 和 `::ng-deep`

组件样式通常只会作用于组件自身的 HTML 上。

可以使用 `/deep/` 选择器来强制一个样式对各级子组件的视图也生效，它**不但作用于组件的子视图，也会作用于组件的内容。**

这个例子以所有的 `<h3>` 元素为目标，从宿主元素到当前元素再到 DOM 中的所有子元素：

src/app/hero-details.component.css

```
:host /deep/ h3 {
  font-style: italic;
}
```

`/deep/` 组合器还有两个别名：`>>>` 和 `::ng-deep`。

> `/deep/` 和 `>>>` 选择器只能被用在仿真 (emulated) 模式下。 这种方式是默认值，也是用得最多的方式。 更多信息，见控制视图封装模式一节。

> CSS 标准中用于 "刺穿 Shadow DOM" 的组合器已经被废弃，并将这个特性从主流浏览器和工具中移除。 因此，我们也将在 Angular 中移除对它们的支持（包括 `/deep/`、`>>>` 和 `::ng-deep`）。目前，建议先统一使用 `::ng-deep`，以便兼容将来的工具。

把样式加载进组件中

有几种方式把样式加入组件：

- 设置 `styles` 或 `styleUrls` 元数据
- 内联在模板的 HTML 中
- 通过 CSS 文件导入

上述作用域规则对所有这些加载模式都适用。

元数据中的样式

你可以给 `@Component` 装饰器添加一个 `styles` 数组型属性。

这个数组中的每一个字符串（通常也只有一个）定义一份 CSS。

src/app/hero-app.component.ts (CSS inline)

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Tour of Heroes</h1>
    <app-hero-main [hero]="hero"></app-hero-main>
  `,
  styles: ['h1 { font-weight: normal; }']
})
export class HeroAppComponent {
/* . . . */
}
```

> 注意：这些样式只对当前组件生效。 它们既不会作用于模板中嵌入的任何组件，也不会作用于投影进来的组件（如 `ng-content` ）。

当使用 `--inline-styles` 标识创建组件时，CLI 就会定义一个空的 `styles` 数组

```
ng generate component hero-app --inline-style
```

组件元数据中的样式文件

你可以通过把外部 CSS 文件添加到 `@Component` 的 `styleUrls` 属性中来加载外部样式。

src/app/hero-app.component.ts (CSS in file)   *src/app/hero-app.component.css*

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Tour of Heroes</h1>
    <app-hero-main [hero]="hero"></app-hero-main>
  `,
  styleUrls: ['./hero-app.component.css']
})
export class HeroAppComponent {
/* . . . */
}
```

注意：这些样式只对当前组件生效。 它们既不会作用于模板中嵌入的任何组件，也不会作用于投影进来的组件（如 `ng-content` ）。

你可以指定多个样式文件，甚至可以组合使用 `style` 和 `styleUrls` 方式。

CLI 会默认为你创建一个空白的样式表文件，并且在所生成组件的 `styleUrls` 中引用该文件。

```
ng generate component hero-app
```

模板内联样式

你也可以在组件的 HTML 模板中嵌入 `<style>` 标签。

src/app/hero-controls.component.ts

```
@Component({
  selector: 'app-hero-controls',
  template: `
    <style>
      button {
        background-color: white;
        border: 1px solid #777;
      }
    </style>
    <h3>Controls</h3>
    <button (click)="activate()">Activate</button>
  `
})
```

模板中的 link 标签

你也可以在组件的 HTML 模板中写 `<link>` 标签。

src/app/hero-team.component.ts

```
@Component({
  selector: 'app-hero-team',
  template: `
    <!-- We must use a relative URL so that the AOT compiler can find the stylesheet -->
    <link rel="stylesheet" href="../assets/hero-team.component.css">
    <h3>Team</h3>
    <ul>
      <li *ngFor="let member of hero.team">
        {{member}}
      </li>
    </ul>`
})
```

> link 标签的 `href` URL 必须是相对于**本应用的根路径**的，而不是相对于这个组件文件的。
>
> 当使用 CLI 进行构建时，要确保这个链接到的样式表文件被复制到了服务器上。参见 CLI 官方文档。

CSS @imports 语法

你还可以利用标准的 CSS `@import` 规则来把其它 CSS 文件导入到 CSS 文件中。

在**这种**情况下，URL 是相对于你正在导入的 CSS 文件的。

src/app/hero-details.component.css (excerpt)

```
/* The AOT compiler needs the `./` to show that this is local */
@import './hero-details-box.css';
```

外部以及全局样式文件

当使用 CLI 进行构建时，你必须配置 `angular.json` 文件，使其包含**所有外部资源**（包括外部的样式表文件）。

在它的 `styles` 区注册这些全局样式文件，默认情况下，它会有一个预先配置的全局 `styles.css` 文件。

要了解更多，参见 CLI 官方文档。

非 CSS 样式文件

如果使用 CLI 进行构建，那么你可以用 sass、less 或 stylus 来编写样式，并使用相应的扩展名（`.scss`、`.less`、`.styl`）把它们指定到 `@Component.styleUrls` 元数据中。例子如下：

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
...
```

CLI 的构建过程会运行相关的预处理器。

当使用 `ng generate component` 命令生成组件文件时，CLI 会默认生成一个空白的 CSS 样式文件（`.css`）。你可以配置 CLI，让它默认使用你喜欢的 CSS 预处理器，参见 CLI 官方文档 中的解释。

> 添加到 `@Component.styles` 数组中的字符串**必须写成 CSS**，因为 CLI 没法对这些内联的样式使用任何 CSS 预处理器。

控制视图的封装模式：原生 (Native)、仿真 (Emulated) 和无 (None)

像上面讨论过的一样，组件的 CSS 样式被封装进了自己的视图中，而不会影响到应用程序的其它部分。

通过在组件的元数据上设置**视图封装模式**，你可以分别控制**每个组件**的封装模式。 可选的封装模式一共有如下几种：

- `Native` 模式使用浏览器原生的 Shadow DOM 实现来为组件的宿主元素附加一个 Shadow DOM。组件的样式被包裹在这个 Shadow DOM 中。(译注：不进不出，没有样式能进来，组件样式出不去。)
- `Emulated` 模式（默认值）通过预处理（并改名）CSS 代码来模拟 Shadow DOM 的行为，以达到把 CSS 样式局限在组件视图中的目的。 更多信息，见附录 1 。(译注：只进不出，全局样式能进来，组件样式出不去)
- `None` 意味着 Angular 不使用视图封装。 Angular 会把 CSS 添加到全局样式中。而不会应用上前面讨论过的那些作用域规则、隔离和保护等。 从本质上来说，这跟把组件的样式直接放进 HTML 是一样的。(译注：能进能出。)

通过组件元数据中的 `encapsulation` 属性来设置组件封装模式：

src/app/quest-summary.component.ts

```
// warning: few browsers support shadow DOM encapsulation at this time
encapsulation: ViewEncapsulation.Native
```

原生(`Native`)模式只适用于有原生 Shadow DOM 支持的浏览器。 因此仍然受到很多限制，这就是为什么仿真 (`Emulated`) 模式是默认选项，并建议将其用于大多数情况。

查看仿真 (Emulated) 模式下生成的 CSS

当使用默认的仿真模式时，Angular 会对组件的所有样式进行预处理，让它们模仿出标准的 Shadow CSS 作用域规则。

在启用了仿真模式的 Angular 应用的 DOM 树中，每个 DOM 元素都被加上了一些额外的属性。

```
<hero-details _nghost-pmm-5>
  <h2 _ngcontent-pmm-5>Mister Fantastic</h2>
  <hero-team _ngcontent-pmm-5 _nghost-pmm-6>
    <h3 _ngcontent-pmm-6>Team</h3>
  </hero-team>
</hero-detail>
```

生成出的属性分为两种：

- 一个元素在原生封装方式下可能是 Shadow DOM 的宿主，在这里被自动添加上一个 `_nghost` 属性。这是组件宿主元素的典型情况。
- 组件视图中的每一个元素，都有一个 `_ngcontent` 属性，它会标记出该元素是哪个宿主的模拟 Shadow DOM。

这些属性的具体值并不重要。它们是自动生成的，并且你永远不会在程序代码中直接引用到它们。 但它们会作为生成的组件样式的目标，就像 DOM 的 `<head>` 中一样：

```
[_nghost-pmm-5] {
  display: block;
  border: 1px solid black;
}

h3[_ngcontent-pmm-6] {
  background-color: white;
  border: 1px solid #777;
}
```

这些就是那些样式被处理后的结果，每个选择器都被增加了 `_nghost` 或 `_ngcontent` 属性选择器。 这些额外的选择器实现了本文所描述的这些作用域规则。

Angular 元素（Elements）概览

Angular 元素就是打包成自定义元素的 Angular 组件。所谓自定义元素就是一套与具体框架无关的用于定义新 HTML 元素的 Web 标准。

自定义元素这项特性目前受到了 Chrome、Opera 和 Safari 的支持，在其它浏览器中也能通过腻子脚本（参见浏览器支持）加以支持。 自定义元素扩展了 HTML，它允许你定义一个由 JavaScript 代码创建和控制的标签。 浏览器会维护一个自定义元素（也叫 Web Components）的注册表 `CustomElementRegistry`，它把一个可实例化的 JavaScript 类映射到 HTML 标签上。

`@angular/elements` 包导出了一个 `createCustomElement()` API，它在 Angular 组件接口与变更检测功能和内置 DOM API 之间建立了一个桥梁。

把组件转换成自定义元素会让所需的 Angular 基础设施也可用在浏览器中。创建自定义元素非常简单直接，它会自动把你的组件视图对接到变更检测和数据绑定机制，会把 Angular 的功能映射到原生 HTML 中的等价物。

使用自定义元素

自定义元素会自举启动 —— 它们在添加到 DOM 中时就会自行启动自己，并在从 DOM 中移除时自行销毁自己。一旦自定义元素添加到了任何页面的 DOM 中，它的外观和行为就和其它的 HTML 元素一样了，不需要对 Angular 的术语或使用约定有任何特殊的了解。

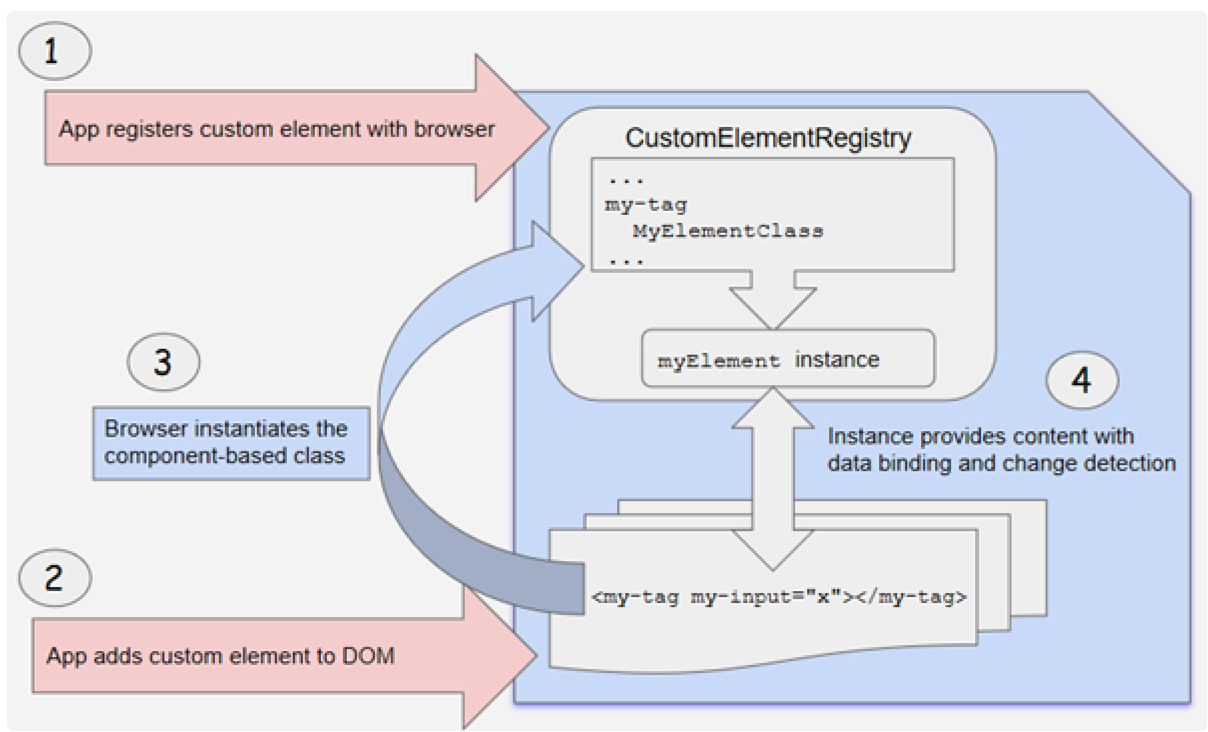
- **Angular 应用中的简易动态内容**
  把组件转换成自定义元素为你在 Angular 应用中创建动态 HTML 内容提供了一种简单的方式。 在 Angular 应用中，你直接添加到 DOM 中的 HTML 内容是不会经过 Angular 处理的，除非你使用**动态组件**来借助自己的代码把 HTML 标签与你的应用数据关联起来并参与变更检测。而使用自定义组件，所有这些装配工作都是自动的。
- **富内容应用**
  如果你有一个富内容应用（比如正在展示本文档的这个），自定义元素能让你的内容提供者使用复杂的 Angular 功能，而不要求他了解 Angular 的知识。比如，像本文档这样的 Angular 指南是使用 Angular 导航工具直接添加到 DOM 中的，但是其中可以包含特殊的元素，比如 `<code-snippet>`，它可以执行复杂的操作。 你所要告诉你的内容提供者的一切，就是这个自定义元素的语法。他们不需要了解关于 Angular 的任何知识，也不需要了解你的组件的数据结构或实现。

工作原理

使用 `createCustomElement()` 函数来把组件转换成一个可注册成浏览器中自定义元素的类。 注册完这个配置好的类之后，你就可以在内容中像内置 HTML 元素一样使用这个新元素了，比如直接把它加到 DOM 中：

```
<my-popup message="Use Angular!"></my-popup>
```

当你的自定义元素放进页面中时，浏览器会创建一个已注册类的实例。其内容是由组件模板提供的，它使用 Angular 模板语法，并且使用组件和 DOM 数据进行渲染。组件的输入属性（Property）对应于该元素的输入属性（Attribute）。



> 我们正在使用的这些自定义元素也可以被用在使用其它框架构建的 Web 应用中。 Angular 框架的一个最小化的、自包含的版本，会注入成一个服务，以支持变更检测和数据绑定功能。 要了解更多，参见这个视频演讲。

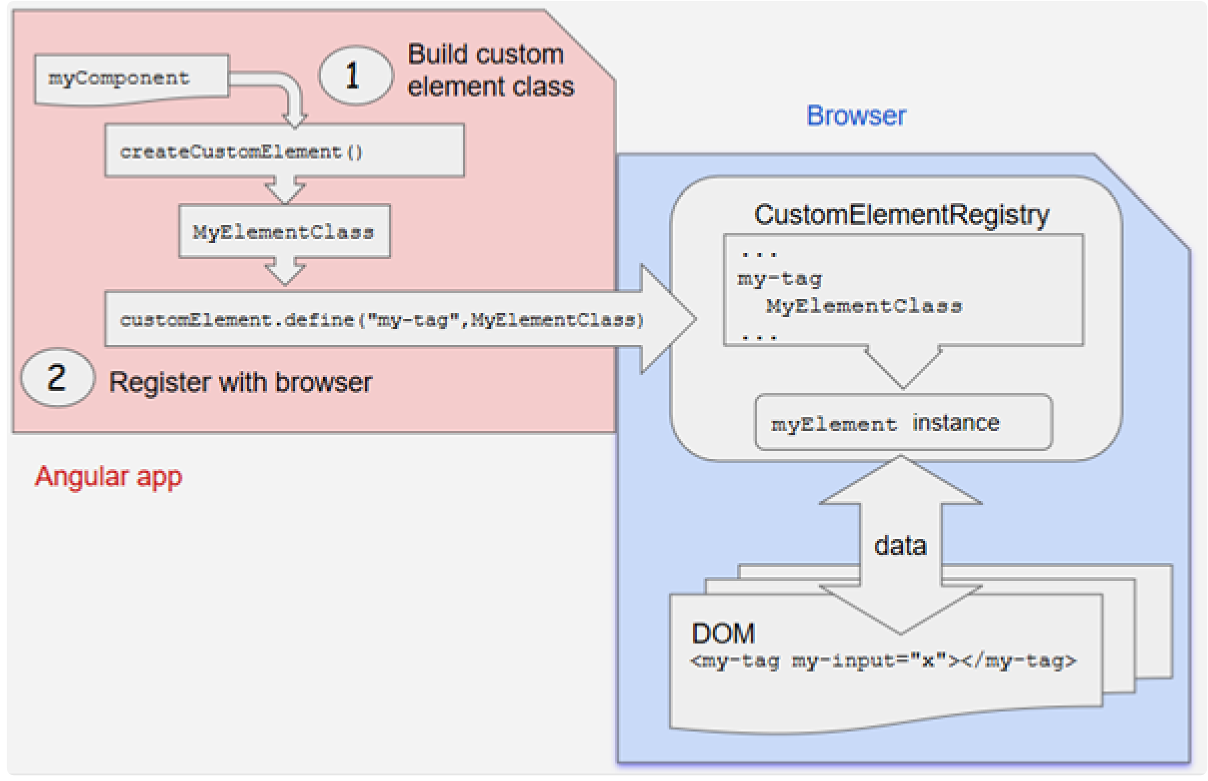
把组件转换成自定义元素

Angular 提供了 `createCustomElement()` 函数，以支持把 Angular 组件及其依赖转换成自定义元素。该函数会收集该组件的 `Observable` 型属性，提供浏览器创建和销毁实例时所需的 Angular 功能，还会对变更进行检测并做出响应。

这个转换过程实现了 `NgElementConstructor` 接口，并创建了一个构造器类，用于生成该组件的一个自举型实例。

然后用 JavaScript 的 `customElements.define()` 函数把这个配置好的构造器和相关的自定义元素标签注册到浏览器的 `CustomElementRegistry` 中。 当浏览器遇到这个已注册元素的标签时，就会使用该构造器来创建一

个自定义元素的实例。



映射

寄宿着 Angular 组件的自定义元素在组件中定义的"数据及逻辑"和标准的 DOM API 之间建立了一座桥梁。组件的属性和逻辑会直接映射到 HTML 属性和浏览器的事件系统中。

- 用于创建的 API 会解析该组件，以查找输入属性（Property），并在这个自定义元素上定义相应的属性（Attribute）。 它把属性名转换成与自定义元素兼容的形式（自定义元素不区分大小写），生成的属性名会使用中线分隔的小写形式。 比如，对于带有 `@Input('myInputProp') inputProp` 的组件，其对应的自定义元素会带有一个 `my-input-prop` 属性。
- 组件的输出属性会用 HTML 自定义事件的形式进行分发，自定义事件的名字就是这个输出属性的名字。 比如，对于带有 `@Output() valueChanged = new EventEmitter()` 属性的组件，其相应的自定义元素将会分发名叫 "valueChanged" 的事件，事件中所携带的数据存储在该事件对象的 `detail` 属性中。 如果你提供了别名，就改用这个别名。比如，`@Output('myClick') clicks = new EventEmitter<string>();` 会导致分发名为 "myClick" 事件。

要了解更多，请参见 Web Components 的文档：Creating custom events。

自定义元素的浏览器支持

最近开发的 Web 平台特性：自定义元素目前在一些浏览器中实现了原生支持，而其它浏览器或者尚未决定，或者已经制订了计划。

浏览器	自定义元素支持
Chrome	原生支持。
Opera	原生支持。
Safari	原生支持。
Firefox	把 `dom.webcomponents.enabled` 和 `dom.webcomponents.customelements.enabled` 首选项设置为 true。计划在版本 60/61 中提供原生支持。
Edge	正在实现。

对于原生支持了自定义元素的浏览器，该规范要求开发人员使用 ES2016 的类来定义自定义元素 —— 开发人员可以在项目的 `tsconfig.json` 中设置 `target: "es2015"` 属性来满足这一要求。并不是所有浏览器都支持自定义元素和 ES2015，开发人员也可以选择使用腻子脚本来让它支持老式浏览器和 ES5 的代码。

使用 Angular CLI 可以自动为你的项目添加正确的腻子脚本：`ng add @angular/elements --name=*your_project_name*`。

- 要了解关于腻子脚本的更多信息，参见腻子脚本的相关文档。
- 要了解 Angular 浏览器支持的更多信息，参见浏览器支持。

范例：弹窗服务

以前，如果你要在运行期间把一个组件添加到应用中，就不得不定义**动态组件**。你还要把动态组件添加到模块的 `entryComponents` 列表中，以便应用在启动时能找到它，然后还要加载它、把它附加到 DOM 中的元素上，并且装配所有的依赖、变更检测和事件处理，详见动态组件加载器。

用 Angular 自定义组件会让这个过程更简单、更透明。它会自动提供所有基础设施和框架，而你要做的就是定义所需的各种事件处理逻辑。（如果你不准备在应用中直接用它，还要把该组件在编译时排除出去。）

这个弹窗服务的范例应用定义了一个组件，你可以动态加载它也可以把它转换成自定义组件。

- `popup.component.ts` 定义了一个简单的弹窗元素，用于显示一条输入消息，附带一些动画和样式。
- `popup.service.ts` 创建了一个可注入的服务，它提供了两种方式来执行 PopupComponent：作为动态组件或作为自定义元素。注意动态组件的方式需要更多的代码来做搭建工作。
- `app.module.ts` 把 PopupComponent 添加到模块的 `entryComponents` 列表中，而从编译过程中排除它，以消除启动时的警告和错误。
- `app.component.ts` 定义了该应用的根组件，它借助 PopupService 在运行时把这个弹窗添加到 DOM 中。在应用运行期间，根组件的构造函数会把 PopupComponent 转换成自定义元素。

为了对比，这个范例中同时演示了这两种方式。一个按钮使用动态加载的方式添加弹窗，另一个按钮使用自定义元素的方式。可以看到，两者的结果是一样的，其差别只是准备过程不同。

popup.component.ts *popup.service.ts* *app.module.ts* *app.component.ts*

```
1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2. import { AnimationEvent } from '@angular/animations';
3. import { animate, state, style, transition, trigger } from
   '@angular/animations';
4.
5. @Component({
6.   selector: 'my-popup',
7.   template: 'Popup: {{message}}',
8.   host: {
9.     '[@state]': 'state',
10.    '(@state.done)': 'onAnimationDone($event)',
11.  },
12.  animations: [
13.    trigger('state', [
14.      state('opened', style({transform: 'translateY(0%)'})),
15.      state('void, closed', style({transform: 'translateY(100%)', opacity:
   0})),
16.      transition('* => *', animate('100ms ease-in')),
17.    ])
18.  ],
19.  styles: [`
20.    :host {
21.      position: absolute;
22.      bottom: 0;
23.      left: 0;
24.      right: 0;
25.      background: #009cff;
26.      height: 48px;
27.      padding: 16px;
28.      display: flex;
```

```
29.        align-items: center;
30.        border-top: 1px solid black;
31.        font-size: 24px;
32.      }
33.    `]
34.  })
35.
36.  export class PopupComponent {
37.    private state: 'opened' | 'closed' = 'closed';
38.
39.    @Input()
40.    set message(message: string) {
41.      this._message = message;
42.      this.state = 'opened';
43.
44.      setTimeout(() => this.state = 'closed', 2000);
45.    }
46.    get message(): string { return this._message; }
47.    _message: string;
48.
49.    @Output()
50.    closed = new EventEmitter();
51.
52.    onAnimationDone(e: AnimationEvent) {
53.      if (e.toState === 'closed') {
54.        this.closed.next();
55.      }
56.    }
57.  }
```

动态组件加载器

组件的模板不会永远是固定的。应用可能会需要在运行期间加载一些新的组件。

这本烹饪书为你展示如何使用 `ComponentFactoryResolver` 来动态添加组件。

到在线例子 / 下载范例查看本烹饪书的源码。

动态组件加载

下面的例子展示了如何构建动态广告条。

英雄管理局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。 再用只支持静态组件结构的模板显然是不现实的。

你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。

src/app/ad.directive.ts

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

@Directive({
  selector: '[ad-host]',
})
export class AdDirective {
  constructor(public viewContainerRef: ViewContainerRef) { }
}
```

`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：`ad-host`，它就是你将应用到元素上的指令。下一节会展示该如何做。

加载组件

广告条的大部分实现代码都在 `ad-banner.component.ts` 中。 为了让这个例子简单点，HTML 被直接放在了 `@Component` 装饰器的 `template` 属性中。

`<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 `ad-host`。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。

src/app/ad-banner.component.ts (template)

```
template: `
            <div class="ad-banner">
              <h3>Advertisements</h3>
              <ng-template ad-host></ng-template>
            </div>
          `
```

`<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。

解析组件

深入看看 `ad-banner.component.ts` 中的方法。

`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。

src/app/ad-banner.component.ts (excerpt)

```
export class AdBannerComponent implements OnInit, OnDestroy {
  @Input() ads: AdItem[];
  currentAdIndex = -1;
  @ViewChild(AdDirective) adHost: AdDirective;
  interval: any;

  constructor(private componentFactoryResolver: ComponentFactoryResolver) { }

  ngOnInit() {
    this.loadComponent();
    this.getAds();
  }

  ngOnDestroy() {
    clearInterval(this.interval);
  }

  loadComponent() {
    this.currentAdIndex = (this.currentAdIndex + 1) % this.ads.length;
    let adItem = this.ads[this.currentAdIndex];

    let componentFactory =
this.componentFactoryResolver.resolveComponentFactory(adItem.component);

    let viewContainerRef = this.adHost.viewContainerRef;
    viewContainerRef.clear();

    let componentRef = viewContainerRef.createComponent(componentFactory);
    (<AdComponent>componentRef.instance).data = adItem.data;
  }

  getAds() {
    this.interval = setInterval(() => {
      this.loadComponent();
    }, 3000);
  }
}
```

这里的 `loadComponent()` 方法很重要。 来一步步看看。首先，它选取了一个广告。

> `loadComponent()` 如何选择广告
>
> `loadComponent()` 方法使用某种算法选择了一个广告。
>
> （译注：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`。

在 `loadComponent()` 选取了一个广告之后，它使用 `ComponentFactoryResolver` 来为每个具体的组件解析出一个 `ComponentFactory`。 然后 `ComponentFactory` 会为每一个组件创建一个实例。

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。

回忆一下，`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。

对选择器的引用

通常，Angular 编译器会为模板中所引用的每个组件都生成一个 `ComponentFactory` 类。 但是，对于动态加载的组件，模板中不会出现对它们的选择器的引用。

要想确保编译器照常生成工厂类，就要把这些动态加载的组件添加到 `NgModule` 的 `entryComponents` 数组中：

src/app/app.module.ts (entry components)

```
entryComponents: [ HeroJobAdComponent, HeroProfileComponent ],
```

公共的 `AdComponent` 接口

在广告条中，所有组件都实现了一个公共接口 `AdComponent`，它定义了一个标准化的 API，来把数据传给组件。

下面就是两个范例组件及其 `AdComponent` 接口：

hero-job-ad.component.ts *hero-profile.component.ts* *ad.component.ts*

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

import { AdComponent }      from './ad.component';

@Component({
  template: `
    <div class="job-ad">
      <h4>{{data.headline}}</h4>

      {{data.body}}
    </div>
  `
})
export class HeroJobAdComponent implements AdComponent {
  @Input() data: any;

}
```

最终的广告栏

最终的广告栏是这样的：

参见在线例子 / 下载范例。

属性型指令

属性型指令用于改变一个 DOM 元素的外观或行为。

你可以到这里试试：属性型指令范例 / 下载范例。

指令概览

在 Angular 中有三种类型的指令：

1. 组件 — 拥有模板的指令
2. 结构型指令 — 通过添加和移除 DOM 元素改变 DOM 布局的指令
3. 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。

创建一个简单的属性型指令

属性型指令至少需要一个带有 `@Directive` 装饰器的控制器类。该装饰器指定了一个用于标识属性的选择器。控制器类实现了指令需要的指令行为。

本章展示了如何创建一个简单的属性型指令 **myHighlight** ，当用户把鼠标悬停在一个元素上时，改变它的背景色。你可以这样用它：

src/app/app.component.html (applied)

```
<p appHighlight>Highlight me!</p>
```

编写指令代码

在命令行窗口下用 CLI 命令创建指令类文件。

```
ng generate directive highlight
```

CLI 会创建 `src/app/highlight.directive.ts` 及相应的测试文件（`.../spec.ts`），并且在根模块 `AppModule` 中声明这个指令类。

> 和组件一样，这些指令也必须在Angular 模块中进行声明。

生成的 `src/app/highlight.directive.ts` 文件如下：

src/app/highlight.directive.ts

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

这里导入的 `Directive` 符号提供了 Angular 的 `@Directive` 装饰器。

`@Directive` 装饰器的配置属性中指定了该指令的 CSS 属性型选择器 `[appHighlight]`

这里的方括号(`[]`)表示它的属性型选择器。 Angular 会在模板中定位每个名叫 `appHighlight` 的元素，并且为这些元素加上本指令的逻辑。

正因如此，这类指令被称为 属性选择器 。

> ## 为什么不直接叫做 "highlight"?
>
> 尽管 **highlight** 是一个比 **myHighlight** 更简洁的名字，而且它确实也能工作。 但是最佳实践是在选择器名字前面添加前缀，以确保它们不会与标准 HTML 属性冲突。 它同时减少了与第三方指令名字发生冲突的危险。
>
> 确认你没有给 `highlight` 指令添加`ng`前缀。 那个前缀属于 Angular，使用它可能导致难以诊断的 bug。例如，这个简短的前缀 `my` 可以帮助你区分自定义指令。

紧跟在 `@Directive` 元数据之后的就是该指令的控制器类，名叫 `HighlightDirective`，它包含了该指令的逻辑（目前为空逻辑）。然后导出 `HighlightDirective`，以便它能在别处访问到。

现在，把刚才生成的 `src/app/highlight.directive.ts` 编辑成这样：

src/app/highlight.directive.ts

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

`import` 语句还从 Angular 的 `core` 库中导入了一个 `ElementRef` 符号。

你可以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。

这里的第一个实现把宿主元素的背景色设置为了黄色。

使用属性型指令

要想使用这个新的 `HighlightDirective`，就往根组件 `AppComponent` 的模板中添加一个 `<p>` 元素，并把该指令作为一个属性使用。

src/app/app.component.html

```
<p appHighlight>Highlight me!</p>
```

运行这个应用以查看 `HighlightDirective` 的实际效果。

```
ng serve
```

总结：Angular 在宿主元素 `<p>` 上发现了一个 `appHighlight` 属性。 然后它创建了一个 `HighlightDirective` 类的实例，并把所在元素的引用注入到了指令的构造函数中。 在构造函数中，该指令把 `<p>` 元素的背景设置为了黄色。

响应用户引发的事件

当前，`appHighlight` 只是简单的设置元素的颜色。 这个指令应该在用户鼠标悬浮一个元素时，设置它的颜色。

先把 `HostListener` 加进导入列表中。

src/app/highlight.directive.ts (imports)

```
import { Directive, ElementRef, HostListener } from '@angular/core';
```

然后使用 `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。

src/app/highlight.directive.ts (mouse-methods)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
}

@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}

private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
```

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

> 当然，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。 但这种方法至少有三个问题：
>
> 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。
> 3. 必须直接和 DOM API 打交道，应该避免这样做。

这些处理器委托了一个辅助方法来为 DOM 元素（`el`）设置颜色。

这个辅助方法（`highlight`）被从构造函数中提取了出来。 修改后的构造函数只负责声明要注入的元素 `el: ElementRef`。

src/app/highlight.directive.ts (constructor)

```
constructor(private el: ElementRef) { }
```

下面是修改后的指令代码：

src/app/highlight.directive.ts

```
import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef) { }

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
```

运行本应用并确认：当把鼠标移到 `p` 上的时候，背景色就出现了，而移开的时候，它消失了。

Highlight me!

使用 `@Input` 数据绑定向指令传递值

高亮的颜色目前是硬编码在指令中的，这不够灵活。 在这一节中，你应该让指令的使用者可以指定要用哪种颜色进行高亮。

先从 `@angular/core` 中导入 `Input`。

src/app/highlight.directive.ts (imports)

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

然后把 `highlightColor` 属性添加到指令类中，就像这样：

src/app/highlight.directive.ts (highlightColor)

```
@Input() highlightColor: string;
```

绑定到 @Input 属性

注意看 `@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，参见稍后了解更多。

试试把下列指令绑定变量添加到 `AppComponent` 的模板中：

src/app/app.component.html (excerpt)

```
<p appHighlight highlightColor="yellow">Highlighted in yellow</p>
<p appHighlight [highlightColor]="'orange'">Highlighted in orange</p>
```

把 `color` 属性添加到 `AppComponent` 中：

src/app/app.component.ts (class)

```
export class AppComponent {
  color = 'yellow';
}
```

让它通过属性绑定来控制高亮颜色。

src/app/app.component.html (excerpt)

```
<p appHighlight [highlightColor]="color">Highlighted with parent component's
color</p>
```

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：

src/app/app.component.html (color)

```
<p [appHighlight]="color">Highlight me!</p>
```

`[appHighlight]` 属性同时做了两件事：把这个高亮指令应用到了 `<p>` 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。

你还要把该指令的 `highlightColor` 改名为 `myHighlight`，因为它是颜色属性目前的绑定名。

src/app/highlight.directive.ts (renamed to match directive selector)

```
@Input() appHighlight: string;
```

这可不好。因为 `appHighlight` 是一个糟糕的属性名，而且不能反映该属性的意图。

绑定到 @Input 别名

幸运的是，你可以随意命名该指令的属性，并且给它指定一个用于绑定的别名。

恢复原始属性名，并在 `@Input` 的参数中把选择器 `myHighlight` 指定为别名。

src/app/highlight.directive.ts (color property with alias)

```
@Input('appHighlight') highlightColor: string;
```

在指令内部，该属性叫 `highlightColor`，在外部，你绑定到它地方，它叫 `appHighlight`。

这是最好的结果：理想的内部属性名，理想的绑定语法：

src/app/app.component.html (color)

```
<p [appHighlight]="color">Highlight me!</p>
```

现在，你通过别名绑定到了 `highlightColor` 属性，并修改 `onMouseEnter()` 方法来使用它。 如果有人忘了绑定到 `appHighlightColor`，那就用红色进行高亮。

src/app/highlight.directive.ts (mouse enter)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || 'red');
}
```

这是最终版本的指令类。

src/app/highlight.directive.ts (excerpt)

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) { }

  @Input('appHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || 'red');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

写个测试程序试验下

凭空想象该指令如何工作可不容易。 在本节，你将把 `AppComponent` 改成一个测试程序，它让你可以通过单选按钮来选取高亮颜色，并且把你选取的颜色绑定到指令中。

把 `app.component.html` 修改成这样：

src/app/app.component.html (v2)

```
<h1>My First Attribute Directive</h1>

<h4>Pick a highlight color</h4>
<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>
```

修改 `AppComponent.color`，让它不再有初始值。

src/app/app.component.ts (class)

```
export class AppComponent {
  color: string;
}
```

下面是测试程序和指令的动图。

绑定到第二个属性

本例的指令只有一个可定制属性，真实的应用通常需要更多。

目前，默认颜色（它在用户选取了高亮颜色之前一直有效）被硬编码为红色。应该允许模板的开发者设置默认颜色。

把第二个名叫 `defaultColor` 的输入属性添加到 `HighlightDirective` 中：

src/app/highlight.directive.ts (defaultColor)

```
@Input() defaultColor: string;
```

修改该指令的 `onMouseEnter`，让它首先尝试使用 `highlightColor` 进行高亮，然后用 `defaultColor`，如果它们都没有指定，那就用红色作为后备。

src/app/highlight.directive.ts (mouse-enter)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || this.defaultColor || 'red');
}
```

当已经绑定过 `appHighlight` 属性时，要如何绑定到第二个属性呢？

像组件一样，你也可以绑定到指令的很多属性，只要把它们依次写在模板中就行了。 开发者可以绑定到 `AppComponent.color`，并且用紫罗兰色作为默认颜色，代码如下：

src/app/app.component.html (defaultColor)

```
<p [appHighlight]="color" defaultColor="violet">
  Highlight me too!
</p>
```

Angular 之所以知道 `defaultColor` 绑定属于 `HighlightDirective`，是因为你已经通过 `@Input` 装饰器把它设置成了**公共**属性。

当这些代码完成时，测试程序工作时的动图如下：

小结

本章介绍了如何：

- 构建一个属性型指令，它用于修改一个元素的行为。
- 把一个指令应用到模板中的某个元素上。
- 响应事件以改变指令的行为。
- 把值绑定到指令中。

最终的源码如下：

app/app.component.ts　app/app.component.html　app/highlight.directive.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  color: string;
}
```

你还可以体验和下载属性型指令范例 / 下载范例.

附录：为什么要加@Input？

在这个例子中 `hightlightColor` 是 `HighlightDirective` 的一个**输入型**属性。你见过它没有用别名时的代码：

src/app/highlight.directive.ts (color)

```
@Input() highlightColor: string;
```

也见过用别名时的代码：

src/app/highlight.directive.ts (color)

```
@Input('appHighlight') highlightColor: string;
```

无论哪种方式，`@Input` 装饰器都告诉 Angular，该属性是**公共的**，并且能被父组件绑定。 如果没有 `@Input`，Angular 就会拒绝绑定到该属性。

但你以前也曾经把模板 HTML 绑定到组件的属性，而且从来没有用过 `@Input`。 差异何在?

差异在于信任度不同。 Angular 把组件的模板看做**从属于**该组件的。 组件和它的模板默认会相互信任。 这也就是意味着，组件自己的模板可以绑定到组件的**任意**属性，无论是否使用了 `@Input` 装饰器。

但组件或指令不应该盲目的信任其它组件或指令。 因此组件或指令的属性默认是不能被绑定的。 从 Angular 绑定机制的角度来看，它们是**私有**的，而当添加了 `@Input` 时，它们变成了**公共**的 只有这样，它们才能被其它组件或属性绑定。

你可以根据属性名在绑定中出现的位置来判定是否要加 `@Input`。

- 当它出现在等号**右侧**的模板表达式中时，它属于模板所在的组件，不需要 `@Input` 装饰器。
- 当它出现在等号左边的方括号（[ ]）中时，该属性属于**其它**组件或指令，它必须带有 `@Input` 装饰器。

试用此原理分析下列范例:

src/app/app.component.html (color)

```
<p [appHighlight]="color">Highlight me!</p>
```

- `color` 属性位于右侧的绑定表达式中，它属于模板所在的组件。 该模板和组件相互信任。因此 `color` 不需要 `@Input` 装饰器。
- `appHighlight` 属性位于左侧，它引用了 `HighlightDirective` 中一个**带别名的**属性，它不是模板所属组件的一部分，因此存在信任问题。 所以，该属性必须带 `@Input` 装饰器。

结构型指令

本章将看看 Angular 如何用**结构型指令**操纵 DOM 树，以及你该如何写自己的结构型指令来完成同样的任务。

试试在线例子 / 下载范例。

什么是结构型指令?

结构型指令的职责是 HTML 布局。 它们塑造或重塑 DOM 的结构，比如添加、移除或维护这些元素。

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。

结构型指令非常容易识别。 在这个例子中，星号（*）被放在指令的属性名之前。

src/app/app.component.html (ngif)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

没有方括号，没有圆括号，只是把 `*ngIf` 设置为一个字符串。

在这个例子中，你将学到星号(*)这个简写方法，而这个字符串是一个**微语法**，而不是通常的模板表达式。 Angular 会解开这个语法糖，变成一个 `<ng-template>` 标记，包裹着宿主元素及其子元素。 每个结构型指令都可以用这个模板做点不同的事情。

三个常用的内置结构型指令 —— NgIf、NgFor和NgSwitch...。 你在**模板语法**一章中学过它，并且在 Angular 文档的例子中到处都在用它。下面是模板中的例子：

src/app/app.component.html (built-in)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>

<ul>
  <li *ngFor="let hero of heroes">{{hero.name}}</li>
</ul>

<div [ngSwitch]="hero?.emotion">
  <app-happy-hero    *ngSwitchCase="'happy'"    [hero]="hero"></app-happy-hero>
  <app-sad-hero      *ngSwitchCase="'sad'"      [hero]="hero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'app-confused'" [hero]="hero"></app-confused-
hero>
  <app-unknown-hero  *ngSwitchDefault           [hero]="hero"></app-unknown-hero>
</div>
```

本章不会重复讲如何**使用**它们，而是解释它们的**工作原理**以及如何写自己的结构型指令。

> 指令的拼写形式
>
> 在本章中，你将看到指令同时具有两种拼写形式**大驼峰** `UpperCamelCase` **和小驼峰** `lowerCamelCase`，**比如你已经看过的** `NgIf` **和** `ngIf`。 **这里的原因在于，**`NgIf` **引用的是指令的**类名，**而** `ngIf` **引用的是指令的**属性名*。
>
> 指令的**类名**拼写成**大驼峰形式**（`NgIf`），而它的**属性名**则拼写成**小驼峰形式**（`ngIf`）。 本章会在谈论指令的属性和工作原理时引用指令的**类名**，在描述如何在 HTML 模板中把该指令应用到元素时，引用指令的**属性名**。

> 还有另外两种 Angular 指令，在本开发指南的其它地方有讲解：(1) 组件 (2) 属性型指令。
>
> **组件**可以在原生 HTML 元素中管理一小片区域的 HTML。从技术角度说，它就是一个带模板的指令。
>
> **属性型**指令会改变某个元素、组件或其它指令的外观或行为。 比如，内置的`NgStyle`指令可以同时修改元素的多个样式。
>
> 你可以在一个宿主元素上应用多个**属性型**指令，但只能应用一个**结构型**指令。

NgIf 案例分析

`NgIf` 是一个很好的结构型指令案例：它接受一个布尔值，并据此让一整块 DOM 树出现或消失。

src/app/app.component.html (ngif-true)

```
<p *ngIf="true">
  Expression is true and ngIf is true.
  This paragraph is in the DOM.
</p>
<p *ngIf="false">
  Expression is false and ngIf is false.
  This paragraph is not in the DOM.
</p>
```

`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。 使用浏览器的开发者工具就可以确认这一点。

```
<p _ngcontent-c0>
  Expression is true and ngIf is true.
  This paragraph is in the DOM.
</p>
<!--bindings={
  "ng-reflect-ng-if": "false"
}-->
```

可以看到第一段文字出现在了 DOM 中，而第二段则没有，在第二段的位置上是一个关于“绑定”的注释（稍后有更多讲解）。

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。

为什么**移除**而不是**隐藏**？

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。

src/app/app.component.html (display-none)

```
<p [style.display]="'block'">
  Expression sets display to "block".
  This paragraph is visible.
</p>
<p [style.display]="'none'">
  Expression sets display to "none".
  This paragraph is hidden but still in the DOM.
</p>
```

当不可见时，这个元素仍然留在 DOM 中。

```
<p _ngcontent-fwv-0 style="display: block;">
  Expression sets display to "block"" .
  This paragraph is visible.
</p>
<p _ngcontent-fwv-0 style="display: none;">
  "
    Expression sets display to "none" .
    This paragraph is hidden but still in the DOM.
  "
</p>
```

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。

同样的考量也适用于每一个结构型指令，无论是内置的还是自定义的。 你应该提醒自己慎重考虑添加元素、移除元素以及创建和销毁组件的后果。

星号（*）前缀

你可能注意到了指令名的星号（*）前缀，并且困惑于为什么需要它以及它是做什么的。

这里的 `*ngIf` 会在 `hero` 存在时显示英雄的名字。

src/app/app.component.html (asterisk)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

星号是一个用来简化更复杂语法的“语法糖”。 从内部实现来说，Angular 把 `*ngIf` **属性** 翻译成一个 `<ng-template>` **元素** 并用它来包裹宿主元素，代码如下：

src/app/app.component.html (ngif-template)

```
<ng-template [ngIf]="hero">
  <div class="name">{{hero.name}}</div>
</ng-template>
```

- `*ngIf` 指令被移到了 `<ng-template>` 元素上。在那里它变成了一个属性绑定 `[ngIf]`。
- `<div>` 上的其余部分，包括它的 `class` 属性在内，移到了内部的 `<ng-template>` 元素上。

第一种形态永远不会真的渲染出来。 只有最终产出的结果才会出现在 DOM 中。

```
<!--bindings={
  "ng-reflect-ng-if": "[object Object]"
}-->
<div _ngcontent-c0>Mr. Nice</div>
```

Angular 会在真正渲染的时候填充 `<ng-template>` 的内容，并且把 `<ng-template>` 替换为一个供诊断用的注释。

`NgFor`和`NgSwitch...`指令也都遵循同样的模式。

`*ngFor` 内幕

Angular 会把 `*ngFor` 用同样的方式把星号（）**语法的** `template`属性**转换成** `<ng-template>`元素*。

这里有一个 `NgFor` 的全特性应用，同时用了这三种写法：

src/app/app.component.html (inside-ngfor)

```
<div *ngFor="let hero of heroes; let i=index; let odd=odd; trackBy: trackById"
[class.odd]="odd">
  ({{i}}) {{hero.name}}
</div>

<ng-template ngFor let-hero [ngForOf]="heroes" let-i="index" let-odd="odd"
[ngForTrackBy]="trackById">
  <div [class.odd]="odd">({{i}}) {{hero.name}}</div>
</ng-template>
```

它明显比 `ngIf` 复杂得多，确实如此。 `NgFor` 指令比本章展示过的 `NgIf` 具有更多的必选特性和可选特性。至少 `NgFor` 会需要一个循环变量（`let hero`）和一个列表（`heroes`）。

你可以通过把一个字符串赋值给 `ngFor` 来启用这些特性，这个字符串使用 Angular 的微语法。

> `ngFor` 字符串**之外**的每一样东西都会留在宿主元素（`<div>`）上，也就是说它移到了 `<ng-template>` 内部。 在这个例子中， `[ngClass]="odd"` 留在了 `<div>` 上。

微语法

Angular 微语法能让你通过简短的、友好的字符串来配置一个指令。 微语法解析器把这个字符串翻译成 `<ng-template>` 上的属性：

- `let` 关键字声明一个模板输入变量，你会在模板中引用它。本例子中，这个输入变量就是 `hero`、 `i` 和 `odd`。 解析器会把 `let hero`、 `let i` 和 `let odd` 翻译成命名变量 `let-hero`、 `let-i` 和 `let-odd`。
- 微语法解析器接收 `of` 和 `trackby`，把它们首字母大写（`of` -> `Of`, `trackBy` -> `TrackBy`）， 并且给它们加上指令的属性名（`ngFor`）前缀，最终生成的名字是 `ngForOf` 和 `ngForTrackBy`。 还有两个 `NgFor` 的**输入属性**，指令据此了解到列表是 `heroes`，而 track-by 函数是 `trackById`。
- `NgFor` 指令在列表上循环，每个循环中都会设置和重置它自己的**上下文**对象上的属性。 这些属性包括 `index` 和 `odd` 以及一个特殊的属性名 `$implicit`（隐式变量）。
- `let-i` 和 `let-odd` 变量是通过 `let i=index` 和 `let odd=odd` 来定义的。 Angular 把它们设置为**上下文**对象中的 `index` 和 `odd` 属性的当前值。
- 这里并没有指定 `let-hero` 的上下文属性。它的来源是隐式的。 Angular 将 `let-hero` 设置为此上下文中 `$implicit` 属性的值， 它是由 `NgFor` 用当前迭代中的英雄初始化的。
- API 参考手册中描述了 `NgFor` 指令的其它属性和上下文属性。
- `NgFor` 是由 `NgForOf` 指令来实现的。请参阅NgForOf API reference来了解 `NgForOf` 指令的更多属性及其上下文属性。

这些微语法机制在你写自己的结构型指令时也同样有效，参考`NgIf` 的源码 和`NgFor` 的源码 可以学到更多。

模板输入变量

模板输入变量是这样一种变量，你可以**在单个实例的模板中**引用它的值。 这个例子中有好几个模板输入变量：`hero`、`i` 和 `odd`。 它们都是用 `let` 作为前导关键字。

模板输入变量和模板引用变量是不同的，无论是在**语义**上还是**语法**上。

你使用 `let` 关键字（如 `let hero`）在模板中声明一个模板**输入**变量。 这个变量的范围被限制在所重复模板的**单一实例**上。 事实上，你可以在其它结构型指令中使用同样的变量名。

而声明模板**引用**变量使用的是给变量名加 `#` 前缀的方式（`#var`）。 一个**引用**变量引用的是它所附着到的元素、组件或指令。它可以在**整个模板**的**任意位置**访问。

模板**输入**变量和**引用**变量具有各自独立的命名空间。`let hero` 中的 `hero` 和 `#hero` 中的 `hero` 并不是同一个变量。

每个宿主元素上只能有一个结构型指令

有时你会希望只有当特定的条件为真时才重复渲染一个 HTML 块。 你可能试过把 `*ngFor` 和 `*ngIf` 放在同一个宿主元素上，但 Angular 不允许。这是因为你在一个元素上只能放一个**结构型**指令。

原因很简单。结构型指令可能会对宿主元素及其子元素做很复杂的事。当两个指令放在同一个元素上时，谁先谁后？`NgIf` 优先还是 `NgFor` 优先？`NgIf` 可以取消 `NgFor` 的效果吗？ 如果要这样做，Angular 应该如何把这种能力泛化，以取消其它结构型指令的效果呢？

对这些问题，没有办法简单回答。而禁止多个结构型指令则可以简单地解决这个问题。 这种情况下有一个简单的解决方案：把 `*ngIf` 放在一个"容器"元素上，再包装进 `*ngFor` 元素。 这个元素可以使用`ng-container`，以免引入一个新的 HTML 层级。

`NgSwitch` 内幕

Angular 的 `NgSwitch` 实际上是一组相互合作的指令：`NgSwitch`、`NgSwitchCase` 和 `NgSwitchDefault`。

例子如下：

src/app/app.component.html (ngswitch)

```
<div [ngSwitch]="hero?.emotion">
  <app-happy-hero    *ngSwitchCase="'happy'"    [hero]="hero"></app-happy-hero>
  <app-sad-hero      *ngSwitchCase="'sad'"      [hero]="hero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'app-confused'" [hero]="hero"></app-confused-
hero>
  <app-unknown-hero  *ngSwitchDefault           [hero]="hero"></app-unknown-hero>
</div>
```

一个值(`hero.emotion`)被被赋值给了 `NgSwitch`，以决定要显示哪一个分支。

`NgSwitch` 本身不是结构型指令，而是一个**属性型**指令，它控制其它两个 switch 指令的行为。 这也就是为什么你要写成 `[ngSwitch]` 而不是 `*ngSwitch` 的原因。

`NgSwitchCase` 和 `NgSwitchDefault` **都是**结构型指令。 因此你要使用星号（`*`）前缀来把它们附着到元素上。 `NgSwitchCase` 会在它的值匹配上选项值的时候显示它的宿主元素。 `NgSwitchDefault` 则会当没有兄弟 `NgSwitchCase` 匹配上时显示它的宿主元素。

> 指令所在的元素就是它的宿主元素。 `<happy-hero>` 是 `*ngSwitchCase` 的宿主元素。 `<unknown-hero>` 是 `*ngSwitchDefault` 的宿主元素。

像其它的结构型指令一样，`NgSwitchCase` 和 `NgSwitchDefault` 也可以解开语法糖，变成 `<ng-template>` 的形式。

src/app/app.component.html (ngswitch-template)

```
<div [ngSwitch]="hero?.emotion">
  <ng-template [ngSwitchCase]="'happy'">
    <app-happy-hero [hero]="hero"></app-happy-hero>
  </ng-template>
  <ng-template [ngSwitchCase]="'sad'">
    <app-sad-hero [hero]="hero"></app-sad-hero>
  </ng-template>
  <ng-template [ngSwitchCase]="'confused'">
    <app-confused-hero [hero]="hero"></app-confused-hero>
  </ng-template >
  <ng-template ngSwitchDefault>
    <app-unknown-hero [hero]="hero"></app-unknown-hero>
  </ng-template>
</div>
```

优先使用星号（`*`）语法

星号（`*`）语法比不带语法糖的形式更加清晰。 如果找不到单一的元素来应用该指令，可以使用<ng-container>作为该指令的容器。

虽然很少有理由在模板中使用结构型指令的**属性**形式和**元素**形式，但这些幕后知识仍然是很重要的，即：Angular 会创建 `<ng-template>`，还要了解它的工作原理。 当需要写自己的结构型指令时，你就要使用 `<ng-template>`。

<ng-template>指令

<ng-template>是一个 Angular 元素，用来渲染 HTML。 它永远不会直接显示出来。 事实上，在渲染视图之前，Angular 会把 `<ng-template>` 及其内容**替换为**一个注释。

如果没有使用结构型指令，而仅仅把一些别的元素包装进 `<ng-template>` 中，那些元素就是不可见的。 在下面的这个短语"Hip! Hip! Hooray!"中，中间的这个 "Hip!"（欢呼声） 就是如此。

src/app/app.component.html (template-tag)

```
<p>Hip!</p>
<ng-template>
  <p>Hip!</p>
</ng-template>
<p>Hooray!</p>
```

Angular 抹掉了中间的那个 "Hip!" ，让欢呼声显得不再那么热烈了。

```
<p _ngcontent-c0>Hip!</p>
<!---->
<p _ngcontent-c0>Hooray!</p>
```

Hip!

Hooray!

结构型指令会让 `<ng-template>` 正常工作，在你写自己的结构型指令时就会看到这一点。

使用<ng-container>把一些兄弟元素归为一组

通常都要有一个**根**元素作为结构型指令的数组。 列表元素（`<li>`） 就是一个典型的供 `NgFor` 使用的宿主元素。

src/app/app.component.html (ngfor-li)

```
<li *ngFor="let hero of heroes">{{hero.name}}</li>
```

当没有这样一个单一的宿主元素时，你就可以把这些内容包裹在一个原生的 HTML 容器元素中，比如 `<div>`，并且把结构型指令附加到这个"包裹"上。

src/app/app.component.html (ngif)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

但引入另一个容器元素（通常是 `<span>` 或 `<div>`）来把一些元素归到一个单一的**根元素**下，通常也会带来问题。注意，是"通常"而不是"总会"。

这种用于分组的元素可能会破坏模板的外观表现，因为 CSS 的样式既不曾期待也不会接受这种新的元素布局。 比如，假设你有下列分段布局。

src/app/app.component.html (ngif-span)

```
<p>
  I turned the corner
  <span *ngIf="hero">
    and saw {{hero.name}}. I waved
  </span>
  and continued on my way.
</p>
```

而你的 CSS 样式规则是应用于 `<p>` 元素下的 `<span>` 的。

src/app/app.component.css (p-span)

```
p span { color: red; font-size: 70%; }
```

这样渲染出来的段落就会非常奇怪。

I turned the corner and saw Mr. Nice. I waved and continued on my way.

本来为其它地方准备的 `p span` 样式，被意外的应用到了这里。

另一个问题是：有些 HTML 元素需要所有的直属下级都具有特定的类型。 比如，`<select>` 元素要求直属下级必须为 `<option>`，那就没办法把这些选项包装进 `<div>` 或 `<span>` 中。

如果这样做：

src/app/app.component.html (select-span)

```
<div>
  Pick your favorite hero
  (<label><input type="checkbox" checked (change)="showSad = !showSad">show sad</label>)
</div>
<select [(ngModel)]="hero">
  <span *ngFor="let h of heroes">
    <span *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </span>
  </span>
</select>
```

下拉列表就是空的。

Pick your favorite hero, who is ☑ not sad

▼

浏览器不会显示 <span> 中的 <option>。

<ng-container> 的救赎

Angular 的 <ng-container> 是一个分组元素，但它不会污染样式或元素布局，因为 Angular **压根不会把它放进 DOM** 中。

下面是重新实现的条件化段落，这次使用 <ng-container>。

src/app/app.component.html (ngif-ngcontainer)

```
<p>
  I turned the corner
  <ng-container *ngIf="hero">
    and saw {{hero.name}}. I waved
  </ng-container>
  and continued on my way.
</p>
```

这次就渲染对了。

I turned the corner and saw Mr. Nice. I waved and continued on my way.

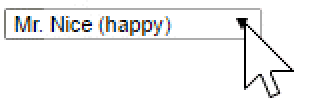
现在用 `<ng-container>` 来根据条件排除选择框中的某个 `<option>`。

src/app/app.component.html (select-ngcontainer)

```
<div>
  Pick your favorite hero
  (<label><input type="checkbox" checked (change)="showSad = !showSad">show
sad</label>)
</div>
<select [(ngModel)]="hero">
  <ng-container *ngFor="let h of heroes">
    <ng-container *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </ng-container>
  </ng-container>
</select>
```

下拉框也工作正常。

Pick your favorite hero, who is ☑ not sad



`<ng-container>` 是一个由 Angular 解析器负责识别处理的语法元素。 它不是一个指令、组件、类或接口，更像是 JavaScript 中 `if` 块中的花括号。

```
if (someCondition) {
  statement1;
  statement2;
  statement3;
}
```

没有这些花括号，JavaScript 只会执行第一句，而你原本的意图是把其中的所有语句都视为一体来根据条件执行。 而 `<ng-container>` 满足了 Angular 模板中类似的需求。

写一个结构型指令

在本节中，你会写一个名叫 `UnlessDirective` 的结构型指令，它是 `NgIf` 的反义词。 `NgIf` 在条件为 `true` 的时候显示模板内容，而 `UnlessDirective` 则会在条件为 `false` 时显示模板内容。

src/app/app.component.html (appUnless-1)

```
<p *appUnless="condition">Show this sentence unless the condition is true.</p>
```

创建指令很像创建组件。

- 导入 `Directive` 装饰器（而不再是 `Component`）。
- 导入符号 `Input`、`TemplateRef` 和 `ViewContainerRef`，你在**任何**结构型指令中都会需要它们。
- 给指令类添加装饰器。
- 设置 CSS **属性选择器**，以便在模板中标识出这个指令该应用于哪个元素。

这里是起点：

src/app/unless.directive.ts (skeleton)

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

@Directive({ selector: '[appUnless]'})
export class UnlessDirective {
}
```

指令的**选择器**通常是把指令的属性名括在方括号中，如 `[appUnless]`。 这个方括号定义出了一个 CSS 属性选择器。

该指令的**属性名**应该拼写成**小驼峰**形式，并且带有一个前缀。 但是，这个前缀不能用 `ng`，因为它只属于 Angular 本身。 请选择一些简短的，适合你自己或公司的前缀。 在这个例子中，前缀是 `my`。

指令的**类名**用 `Directive` 结尾，参见风格指南。 但 Angular 自己的指令例外。

TemplateRef 和 ViewContainerRef

像这个例子一样的简单结构型指令会从 Angular 生成的 `<ng-template>` 元素中创建一个**内嵌的视图**，并把这个视图插入到一个**视图容器**中，紧挨着本指令原来的宿主元素 `<p>`（译注：注意不是子节点，而是兄弟节点）。

你可以使用`TemplateRef`取得 `<ng-template>` 的内容，并通过`ViewContainerRef`来访问这个**视图容器**。

你可以把它们都注入到指令的构造函数中，作为该类的私有属性。

src/app/unless.directive.ts (ctor)

```
constructor(
  private templateRef: TemplateRef<any>,
  private viewContainer: ViewContainerRef) { }
```

appUnless 属性

该指令的使用者会把一个 true/false 条件绑定到 `[appUnless]` 属性上。 也就是说，该指令需要一个带有 `@Input` 的 `appUnless` 属性。

> 要了解关于 `@Input` 的更多知识，参见**模板语法**一章。

src/app/unless.directive.ts (set)

```
@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;
  }
}
```

一旦该值的条件发生了变化，Angular 就会去设置 `appUnless` 属性。因为不能用 `appUnless` 属性，所以你要为它定义一个设置器（setter）。

- 如果条件为假，并且以前尚未创建过该视图，就告诉**视图容器（ViewContainer）**根据模板创建一个**内嵌视图**。
- 如果条件为真，并且视图已经显示出来了，就会清除该容器，并销毁该视图。

没有人会读取 `appUnless` 属性，因此它不需要定义 getter。

完整的指令代码如下：

src/app/unless.directive.ts (excerpt)

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

/**
 * Add the template content to the DOM unless the condition is true.
 */
@Directive({ selector: '[appUnless]'})
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @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;
    }
  }
}
```

把这个指令添加到 AppModule 的 `declarations` 数组中。

然后创建一些 HTML 来试用一下。

src/app/app.component.html (appUnless)

```
<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>
```

当 `condition` 为 `false` 时，顶部的段落就会显示出来，而底部的段落消失了。 当 `condition` 为 `true` 时，顶部的段落被移除了，而底部的段落显示了出来。

The condition is currently false . Toggle condition to true

(A) This paragraph is displayed because the condition is false.

小结

你可以去在线例子 / 下载范例中下载本章的源码。

本章相关的代码如下：

app.component.ts   app.component.html   app.component.css   app.mo

```
1. import { Component } from '@angular/core';
2.
3. import { Hero, heroes } from './hero';
4.
5. @Component({
6.   selector: 'app-root',
7.   templateUrl: './app.component.html',
8.   styleUrls: [ './app.component.css' ]
9. })
10. export class AppComponent {
11.   heroes = heroes;
12.   hero = this.heroes[0];
13.
14.   condition = false;
15.   logs: string[] = [];
16.   showSad = true;
17.   status = 'ready';
18.
19.   trackById(index: number, hero: Hero): number { return hero.id; }
20. }
```

你学到了

- 结构型指令可以操纵 HTML 的元素布局。
- 当没有合适的容器元素时，可以使用`<ng-container>`对元素进行分组。
- Angular 会把星号（*）语法解开成 `<ng-template>`。
- 内置指令 `NgIf`、`NgFor` 和 `NgSwitch` 的工作原理。
- **微语法**如何展开成`<ng-template>`。
- 写了一个自定义结构型指令 —— `UnlessDirective`。

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。 比如，几乎每个人都更喜欢简单的日期格式，例如1988-04-15，而不是服务端传过来的原始字符串格式 —— Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time)。

显然，有些值最好显示成用户友好的格式。你很快就会发现，在很多不同的应用中，都在重复做出某些相同的变换。 你几乎会把它们看做某种 CSS 样式，事实上，你也确实更喜欢在 HTML 模板中应用它们 —— 就像 CSS 样式一样。

通过引入 Angular 管道，你可以把这种简单的“显示-值”转换器声明在 HTML 中。

你可以运行在线例子 / 下载范例，在 Stackblitz 中试用并下载本页的代码。

使用管道

管道把数据作为输入，然后转换它，给出期望的输出。 你要把组件的 `birthday` 属性转换成对人类更友好的日期格式。

src/app/hero-birthday1.component.ts

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```

重点看下组件的模板。

src/app/app.component.html

```
<p>The hero's birthday is {{ birthday | date }}</p>
```

在这个插值表达式中，你让组件的 `birthday` 值通过管道操作符(|)流动到 右侧的Date 管道函数中。所有管道都会用这种方式工作。

内置的管道

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。

> 要学习更多内置管道的知识，参见API 参考手册，并用“pipe”为关键词对结果进行过滤。
>
> Angular 没有 `FilterPipe` 或 `OrderByPipe` 管道，原因在后面的附录中有解释。

对管道进行参数化

管道可能接受任何数量的可选参数来对它的输出进行微调。 可以在管道名后面添加一个冒号(:)再跟一个参数值，来为管道添加参数(比如 `currency:'EUR'`)。 如果这个管道可以接受多个参数，那么就用冒号来分隔这些参数值(比如 `slice:1:5`)。

修改生日模板，来为这个日期管道提供一个格式化参数。 当格式化完该英雄的 4 月 15 日生日之后，它应该被渲染成04/15/88。

src/app/app.component.html

```
<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>
```

参数值可以是任何有效的模板表达式（参见模板语法中的模板表达式部分），比如字符串字面量或组件的属性。 换句话说，借助属性绑定，你也可以像用绑定来控制生日的值一样，控制生日的显示格式。

来写第二个组件，它把管道的格式参数**绑定**到该组件的 `format` 属性。这里是新组件的模板：

src/app/hero-birthday2.component.ts (template)

```
template: `
  <p>The hero's birthday is {{ birthday | date:format }}</p>
  <button (click)="toggleFormat()">Toggle Format</button>
`
```

你还能在模板中添加一个按钮，并把它的点击事件绑定到组件的 `toggleFormat()` 方法。 此方法会在短日期格式(`'shortDate'`)和长日期格式(`'fullDate'`)之间切换组件的 `format` 属性。

src/app/hero-birthday2.component.ts (class)

```
export class HeroBirthday2Component {
  birthday = new Date(1988, 3, 15); // April 15, 1988
  toggle = true; // start with true == shortDate

  get format()   { return this.toggle ? 'shortDate' : 'fullDate'; }
  toggleFormat() { this.toggle = !this.toggle; }
}
```

当你点击按钮的时候，显示的日志会在“04/15/1988”和“Friday, April 15, 1988”之间切换。

The hero's birthday is 4/15/1988

Toggle Format

> 要了解更多 `DatePipes` 的格式选项，请参阅API 文档。

链式管道

你可以把管道串联在一起，以组合出一些潜在的有用功能。 下面这个例子中，要把 `birthday` 串联到 `DatePipe` 管道，然后又串联到 `UpperCasePipe`，这样就可以把生日显示成大写形式了。 生日被显示成了APR 15, 1988：

src/app/app.component.html

```
The chained hero's birthday is
{{ birthday | date | uppercase}}
```

下面这个显示FRIDAY, APRIL 15, 1988的例子用同样的方式链接了这两个管道，而且同时还给 `date` 管道传进去一个参数。

src/app/app.component.html

```
The chained hero's birthday is
{{  birthday | date:'fullDate' | uppercase}}
```

自定义管道

你还可以写自己的自定义管道。 下面就是一个名叫 `ExponentialStrengthPipe` 的管道，它可以放大英雄的能力：

src/app/exponential-strength.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
*/
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

在这个管道的定义中体现了几个关键点：

- 管道是一个带有“管道元数据(pipe metadata)”装饰器的类。
- 这个管道类实现了 `PipeTransform` 接口的 `transform` 方法，该方法接受一个输入值和一些可选参数，并返回转换后的值。
- 当每个输入值被传给 `transform` 方法时，还会带上另一个参数，比如你这个管道就有一个 `exponent`(放大指数) 参数。
- 可以通过 `@Pipe` 装饰器来告诉 Angular：这是一个管道。该装饰器是从 Angular 的 `core` 库中引入的。
- 这个 `@Pipe` 装饰器允许你定义管道的名字，这个名字会被用在模板表达式中。它必须是一个有效的 JavaScript 标识符。 比如，你这个管道的名字是 `exponentialStrength`。

> ### PipeTransform 接口
>
> `transform` 方法是管道的基本要素。 `PipeTransform`**接口**中定义了它，并用它指导各种工具和编译器。理论上说，它是可选的。Angular 不会管它，而是直接查找并执行 `transform` 方法。

现在，你需要一个组件来演示这个管道。

src/app/power-booster.component.ts

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

@Component({
  selector: 'app-power-booster',
  template: `
    <h2>Power Booster</h2>
    <p>Super power boost: {{2 | exponentialStrength: 10}}</p>
  `
})
export class PowerBoosterComponent { }
```

Power Booster

Super power boost: 1024

请注意以下几点：

- 你使用自定义管道的方式和内置管道完全相同。
- 你必须在 `AppModule` 的 `declarations` 数组中包含这个管道。

别忘了 `DECLARATIONS` 数组

你必须手动注册自定义管道。如果忘了，Angular 就会报告一个错误。 在前一个例子中你没有把 `DatePipe` 列进去，这是因为 Angular 所有的内置管道都已经预注册过了。

试一下这个在线例子 / 下载范例，并通过修改值和模板中的可选部分来体会其行为。

能力倍增计算器

仅仅升级模板来测试这个自定义管道其实没多大意思。 干脆把这个例子升级为“能力提升计算器”，它可以把该管道和使用 `ngModel` 的双向数据绑定组合起来。

src/app/power-boost-calculator.component.ts

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

@Component({
  selector: 'app-power-boost-calculator',
  template: `
    <h2>Power Boost Calculator</h2>
    <div>Normal power: <input [(ngModel)]="power"></div>
    <div>Boost factor: <input [(ngModel)]="factor"></div>
    <p>
      Super Hero Power: {{power | exponentialStrength: factor}}
    </p>
  `
})
export class PowerBoostCalculatorComponent {
  power = 5;
  factor = 1;
}
```

Power Boost Calculator

Normal power: 5
Boost factor: 1

Super Hero Power: 5

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 JavaScript 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

在下一个例子中，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。下面是它的模板：

src/app/flying-heroes.component.html (v1)

```
New hero:
  <input type="text" #box
          (keyup.enter)="addHero(box.value); box.value=''"
          placeholder="hero name">
  <button (click)="reset()">Reset</button>
  <div *ngFor="let hero of heroes">
    {{hero.name}}
  </div>
```

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。

src/app/flying-heroes.component.ts (v1)

```
export class FlyingHeroesComponent {
  heroes: any[] = [];
  canFly = true;
  constructor() { this.reset(); }

  addHero(name: string) {
    name = name.trim();
    if (!name) { return; }
    let hero = {name, canFly: this.canFly};
    this.heroes.push(hero);
  }

  reset() { this.heroes = HEROES.slice(); }
}
```

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。

“会飞的英雄”管道（FlyingHeroesPipe）

往 `*ngFor` 重复器中添加一个 `FlyingHeroesPipe` 管道，这个管道能过滤出所有会飞的英雄。

src/app/flying-heroes.component.html (flyers)

```
<div *ngFor="let hero of (heroes | flyingHeroes)">
  {{hero.name}}
</div>
```

下面是 `FlyingHeroesPipe` 的实现，它遵循了以前讲过的那些写自定义管道的模式。

src/app/flying-heroes.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

import { Flyer } from './heroes';

@Pipe({ name: 'flyingHeroes' })
export class FlyingHeroesPipe implements PipeTransform {
  transform(allHeroes: Flyer[]) {
    return allHeroes.filter(hero => hero.canFly);
  }
}
```

当运行在线例子 / 下载范例时，你看到一种奇怪的行为。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。

注意这里是如何添加新英雄的：

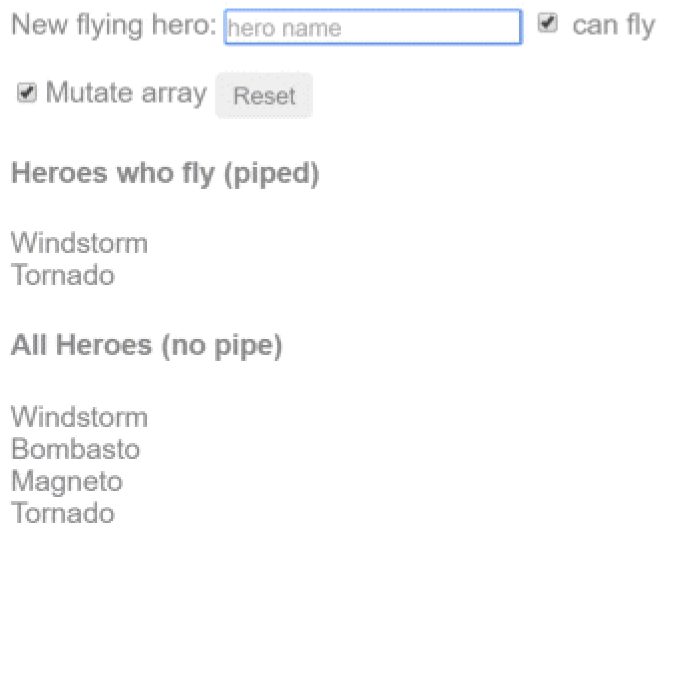
src/app/flying-heroes.component.ts

```
this.heroes.push(hero);
```

当你往 `heroes` 数组中添加一个新的英雄时，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。**

要修复它，就要创建一个新数组，把这个英雄追加进去，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，这次它就包括新的飞行英雄了。

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**飞行英雄**的例子用检查框和其它显示内容扩展了原有代码，来帮你体验这些效果。



直接替换这个数组是通知 Angular 更新显示的一种高效方式。 你该什么时候替换这个数组呢？当数据变化的时候。 在这个**玩具级**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英雄。

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。

为了过滤出会飞的英雄，考虑使用**非纯(impure)管道**。

纯(pure)管道与非纯(impure)管道

有两类管道：纯的与非纯的。 默认情况下，管道都是纯的。以前见到的每个管道都是纯的。 通过把它的 `pure` 标志设置为 `false`，你可以制作一个非纯管道。你可以像这样让 `FlyingHeroesPipe` 变成非纯的：

src/app/flying-heroes.pipe.ts

```
@Pipe({
  name: 'flyingHeroesImpure',
  pure: false
})
```

在继续往下走之前，先理解一下**纯**和**非纯**之间的区别，从**纯**管道开始。

纯管道

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。 如果不能，你就**可以**转回到非纯管道。

> 或者你也可以完全不用管道。 有时候，使用组件的属性能比用管道更好的达到目的，后面会再讨论这一点。

非纯管道

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。

非纯管道 *FlyingHeroesPipe*

把 `FlyingHeroesPipe` 换成了 `FlyingHeroesImpurePipe`。 下面是完整的实现：

FlyingHeroesImpurePipe　　*FlyingHeroesPipe*

```
@Pipe({
  name: 'flyingHeroesImpure',
  pure: false
})
export class FlyingHeroesImpurePipe extends FlyingHeroesPipe {}
```

你把它从 `FlyingHeroesPipe` 中继承下来，以证明无需改动内部代码。 唯一的区别是管道元数据中的 `pure` 标志。

这是一个很好地非纯管道候选者，因为它的 `transform` 函数又小又快。

src/app/flying-heroes.pipe.ts (filter)

```
return allHeroes.filter(hero => hero.canFly);
```

你可以从 `FlyingHeroesComponent` 派生出一个 `FlyingHeroesImpureComponent`。

src/app/flying-heroes-impure.component.html (excerpt)

```
<div *ngFor="let hero of (heroes | flyingHeroesImpure)">
  {{hero.name}}
</div>
```

唯一的重大改动就是管道。 你可以在在线例子 / 下载范例中确认，当你添加新的英雄甚至修改 `heroes` 数组时，这个**会飞的英雄**的显示也跟着更新了。

非纯 *AsyncPipe*

Angular 的 `AsyncPipe` 是一个有趣的非纯管道的例子。 `AsyncPipe` 接受一个 `Promise` 或 `Observable` 作为输入，并且自动订阅这个输入，最终返回它们给出的值。

`AsyncPipe` 管道是有状态的。 该管道维护着一个所输入的 `Observable` 的订阅，并且持续从那个 `Observable` 中发出新到的值。

下面例子使用该 `async` 管道把一个消息字符串(`message$`)的 `Observable` 绑定到视图中。

src/app/hero-async-message.component.ts

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

import { Observable, interval } from 'rxjs';
import { map, take } from 'rxjs/operators';

@Component({
  selector: 'app-hero-message',
  template: `
    <h2>Async Hero Message and AsyncPipe</h2>
    <p>Message: {{ message$ | async }}</p>
    <button (click)="resend()">Resend</button>`,
})
export class HeroAsyncMessageComponent {
  message$: Observable<string>;

  private messages = [
    'You are my hero!',
    'You are the best hero!',
    'Will you be my hero?'
  ];

  constructor() { this.resend(); }

  resend() {
    this.message$ = interval(500).pipe(
      map(i => this.messages[i]),
      take(this.messages.length)
    );
  }
}
```

这个 Async 管道节省了组件的样板代码。 组件不用订阅这个异步数据源，而且不用在被销毁时取消订阅(如果订阅了而忘了反订阅容易导致隐晦的内存泄露)。

一个非纯而且带缓存的管道

来写更多的非纯管道：一个向服务器发起 HTTP 请求的管道。

时刻记住，非纯管道可能每隔几微秒就会被调用一次。 如果你不小心点，这个管道就会发起一大堆请求“攻击”服务器。

下面这个管道只有当所请求的 URL 发生变化时才会向服务器发起请求。它会缓存服务器的响应。 代码如下，它使用Angular http客户端来接收数据

src/app/fetch-json.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
import { HttpClient }          from '@angular/common/http';
@Pipe({
  name: 'fetch',
  pure: false
})
export class FetchJsonPipe  implements PipeTransform {
  private cachedData: any = null;
  private cachedUrl = '';

  constructor(private http: HttpClient) { }

  transform(url: string): any {
    if (url !== this.cachedUrl) {
      this.cachedData = null;
      this.cachedUrl = url;
      this.http.get(url).subscribe( result => this.cachedData = result );
    }

    return this.cachedData;
  }
}
```

接下来在一个测试挽具组件中演示一下它，该组件的模板中定义了两个使用到此管道的绑定，它们都从 `heroes.json` 文件中取得英雄数据。

src/app/hero-list.component.ts

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

@Component({
  selector: 'app-hero-list',
  template: `
    <h2>Heroes from JSON File</h2>

    <div *ngFor="let hero of ('assets/heroes.json' | fetch) ">
      {{hero.name}}
    </div>

    <p>Heroes as JSON:
      {{'assets/heroes.json' | fetch | json}}
    </p>`
})
export class HeroListComponent { }
```

组件渲染起来是这样的：

> **Heroes from JSON File**
>
> Windstorm
> Bombasto
> Magneto
> Tornado
>
> Heroes as JSON: [{ "name": "Windstorm" }, { "name": "Bombasto" }, { "name": "Magneto" }, { "name": "Tornado" }]

这个管道上的断点请求数据的过程显示：

- 每个绑定都有它自己的管道实例。
- 每个管道实例都缓存了它自己的 URL 和数据。
- 每个管道实例都只调用一次服务器。

JsonPipe

第二个绑定除了用到 `FetchPipe` 之外还链接了更多管道。 它通过串联上内置管道 `JsonPipe` 来把英雄数据显示成了 JSON 格式。

借助 JSON 管道进行调试

JsonPipe为你诊断数据绑定的某些神秘错误或为做进一步绑定而探查数据时，提供了一个简单途径。

纯管道与纯函数

纯管道使用纯函数。 纯函数是指在处理输入并返回结果时，不会产生任何副作用的函数。 给定相同的输入，它们总是返回相同的输出。

在本章前面讨论的管道都是用纯函数实现的。 内置的 `DatePipe` 就是一个用纯函数实现的纯管道。 `ExponentialStrengthPipe` 是如此， `FlyingHeroesComponent` 也是如此。 不久前你刚看过的 `FlyingHeroesImpurePipe` 就是一个**用纯函数实现的非纯管道**。

但是一个**纯管道**必须总是用**纯函数**实现。忽略这个警告将导致失败并带来一大堆这样的控制台错误：表达式在被检查后被变更。

下一步

管道能很好的封装和共享的通用“值-显示”转换逻辑。可以像样式一样使用它们，把它们扔到模板表达式中，以提升视图的表现力和可用性。

要浏览 Angular 的所有内置管道，请到API 参考手册。 学着写写自定义管道，并贡献给开发社区。

附录：没有 **FilterPipe** 或者 **OrderByPipe**

Angular 没有随身发布过滤或列表排序的管道。 熟悉 AngularJS 的开发人员应该知道 `filter` 和 `orderBy` 过滤器，但在 Angular 中它们没有等价物。

这并不是疏忽。Angular 不想提供这些管道，因为 (a) 它们性能堪忧，以及 (b) 它们会阻止比较激进的代码最小化(minification)。 无论是 `filter` 还是 `orderBy` 都需要它的参数引用对象型属性。 你在前面学过，这样的管道必然是**非纯管道**，并且 Angular 会在几乎每一次变更检测周期中调用非纯管道。

过滤、 特别是排序是昂贵的操作。 当 Angular 每秒调用很多次这类管道函数时，即使是中等规模的列表都可能严重降低用户体验。 在 AngularJS 程序中， `filter` 和 `orderBy` 经常被误用，结果连累到 Angular 自身，人们抱怨说它太慢。 从某种意义上，这也不冤：谁叫 AngularJS 把 `filter` 和 `orderBy` 作为首发队员呢？是它自己准备了这个性能陷阱。

虽然不是很明显，但代码最小化方面也存在风险。想象一个用于英雄列表的排序管道。该列表可能根据英雄原始属性中的 `name` 和 `planet` 进行排序，就像这样：

```
<!-- NOT REAL CODE! -->
<div *ngFor="let hero of heroes | orderBy:'name,planet'"></div>
```

你使用文本字符串来标记出排序字段，期望管道通过索引形式(如 `hero['name']`)引用属性的值。 不幸的是，激进的代码最小化策略会**改变**`Hero` 类的属性名，所以 `Hero.name` 和 `Hero.planet` 可能会被变成 `Hero.a` 和 `Hero.b`。 显然，`hero['name']` 是无法正常工作的。

然而有些人可能不想做那么激进的最小化， Angular 作为一个产品不应该拒绝那些想做激进的最小化的人。 所以，Angular 开发组决定随 Angular 一起发布的每样东西，都应该能被安全的最小化。

Angular 开发组和一些有经验的 Angular 开发者强烈建议你：把你的过滤和排序逻辑挪进组件本身。 组件可以对外暴露一个 `filteredHeroes` 或 `sortedHeroes` 属性，这样它就获得控制权，以决定要用什么频度去执行其它辅助逻辑。 你原本准备实现为管道，并在整个应用中共享的那些功能，都能被改写为一个过滤/排序的服务，并注入到组件中。

如果你不需要顾虑这些性能和最小化问题，也可以创建自己的管道来实现这些功能(参考FlyingHeroesPipe中的写法)或到社区中去找找。

动画

动画是现代 Web 应用设计中一个很重要的方面。好的用户界面要能在不同的状态之间更平滑的转场。如果需要，还可以用适当的动画来吸引注意力。 设计良好的动画不但会让 UI 更有趣，还会让它更容易使用。

概览

Angular 的动画系统赋予了制作各种动画效果的能力，以构建出与原生 CSS 动画性能相同的动画。 你还获得了额外的让动画逻辑与其它应用代码紧紧集成在一起的能力，这让动画可以被更容易的触发与控制。

> Angular 动画是基于标准的Web 动画 API(Web Animations API)构建的，它们在支持此 API 的浏览器中会用原生方式工作。
>
> 对于 Angular 6，如果浏览器没有提供对 Web 动画 API 的原生支持，Angular 就会自动改用 CSS 的关键帧动画作为后备实现。这意味，除非要在代码中使用 AnimationBuilder ，否则不必使用相关的腻子脚本。 如果你要在代码中使用 AnimationBuilder ，就要从 Angular CLI 自动生成的 `polyfills.ts` 文件中反注释掉 `web-animations-js` 腻子脚本。

> 本章中引用的这个例子可以到在线例子 / 下载范例去体验。

准备工作

在往应用中添加动画之前，你要首先在应用的根模块中引入一些与动画有关的模块和函数。

app.module.ts (animation module import excerpt)

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

@NgModule({
  imports: [ BrowserModule, BrowserAnimationsModule ],
  // ... more stuff ...
})
export class AppModule { }
```

基本例子

这里的动画例子用来给英雄列表添加动画。

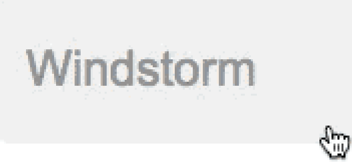
`Hero` 类有一个 `name` 属性、一个 `state` 属性（用于表明该英雄是否为激活状态）和一个 `toggleState()` 函数，用来在这两种状态之间切换。

hero.service.ts (Hero class)

```
export class Hero {
  constructor(public name: string, public state = 'inactive') { }

  toggleState() {
    this.state = this.state === 'active' ? 'inactive' : 'active';
  }
}
```

在屏幕的顶部（`app.hero-team-builder.component.ts`）是一系列按钮，用于从列表中添加和删除英雄（通过 `HeroService`）。 这些按钮会引起列表的变化，同时可以看到列表中的所有范例组件。

快速起步范例：在两个状态间转场



你可以构建一个简单的动画，它会让一个元素用模型驱动的方式在两个状态之间转场。

动画会被定义在 `@Component` 元数据中。

hero-list-basic.component.ts

```
import {
  Component,
  Input
} from '@angular/core';
import {
  trigger,
  state,
  style,
  animate,
  transition
} from '@angular/animations';
```

通过这些，可以在组件元数据中定义一个名叫 `heroState` 的**动画触发器**。它在两个状态 `active` 和 `inactive` 之间进行转场。 当英雄处于激活状态时，它会把该元素显示得稍微大一点、亮一点。

hero-list-basic.component.ts (@Component excerpt)

```
animations: [
  trigger('heroState', [
    state('inactive', style({
      backgroundColor: '#eee',
      transform: 'scale(1)'
    })),
    state('active',   style({
      backgroundColor: '#cfd8dc',
      transform: 'scale(1.1)'
    })),
    transition('inactive => active', animate('100ms ease-in')),
    transition('active => inactive', animate('100ms ease-out'))
  ])
]
```

> 在这个例子中，你在元数据中用内联的方式定义了动画样式(`color` 和 `transform`)。在即将到来的一个 Angular 版本中，还将支持从组件的 CSS 样式表中提取样式。

现在，使用 `[@triggerName]` 语法来把刚刚定义的动画附加到组件模板中一个或多个元素上。

hero-list-basic.component.ts (excerpt)

```
template: `
  <ul>
    <li *ngFor="let hero of heroes"
        [@heroState]="hero.state"
        (click)="hero.toggleState()">
      {{hero.name}}
    </li>
  </ul>
`,
```

这里，动画触发器被添加到了由 `ngFor` 重复出来的每一个元素上。每个重复出来的元素都有独立的动画效果。 然后把 `@triggerName` 属性(Attribute)的值设置成表达式 `hero.state`。这个值应该是 `inactive` 或 `active` 之一。

通过这些设置，一旦英雄对象的状态发生了变化，就会触发一个转场动画。下面是完整的组件实现：

hero-list-basic.component.ts

```
1. import {
2.   Component,
3.   Input
4. } from '@angular/core';
5. import {
6.   trigger,
7.   state,
8.   style,
9.   animate,
10.   transition
11. } from '@angular/animations';
12.
13. import { Hero } from './hero.service';
14.
15. @Component({
16.   selector: 'app-hero-list-basic',
17.   template: `
18.     <ul>
19.       <li *ngFor="let hero of heroes"
20.           [@heroState]="hero.state"
21.           (click)="hero.toggleState()">
22.         {{hero.name}}
23.       </li>
24.     </ul>
25.   `,
26.   styleUrls: ['./hero-list.component.css'],
27.   animations: [
28.     trigger('heroState', [
29.       state('inactive', style({
30.         backgroundColor: '#eee',
31.         transform: 'scale(1)'
32.       })),
33.       state('active',   style({
34.         backgroundColor: '#cfd8dc',
35.         transform: 'scale(1.1)'
36.       })),
37.       transition('inactive => active', animate('100ms ease-in')),
38.       transition('active => inactive', animate('100ms ease-out'))
39.     ])
40.   ]
41. })
```

```
42. export class HeroListBasicComponent {
43.    @Input() heroes: Hero[];
44. }
```

状态与转场

Angular 动画是由状态和状态之间的转场效果所定义的。

动画状态是一个由程序代码中定义的字符串值。在上面的例子中，`'active'` 和 `'inactive'` 是基于英雄对象的逻辑状态的。 状态的来源可以是像本例中这样简单的对象属性，也可以是由方法计算出来的值。重点是，你要能从组件模板中读取它。

你可以为每个动画状态定义了**一组样式**：

src/app/hero-list-basic.component.ts

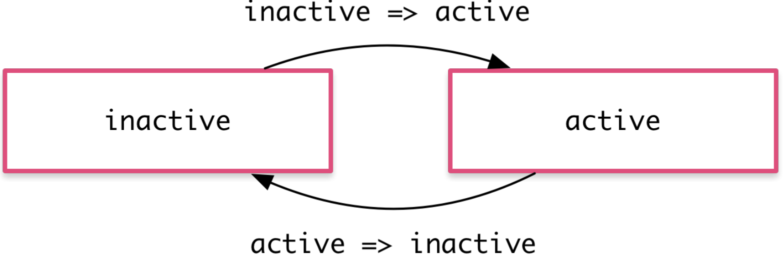
```
state('inactive', style({
  backgroundColor: '#eee',
  transform: 'scale(1)'
})),
state('active',   style({
  backgroundColor: '#cfd8dc',
  transform: 'scale(1.1)'
})),
```

这些 `state` 具体定义了每个状态的**最终样式**。一旦元素转场到那个状态，该样式就会被应用到此元素上，**当它留在此状态时**，这些样式也会一直保持着。 从这个意义上讲，这里其实并不只是在定义动画，而是在定义该元素在不同状态时应该具有的样式。

定义完状态，就能定义在状态之间的各种**转场**了。每个转场都会控制一条在一组样式和下一组样式之间切换的时间线：

src/app/hero-list-basic.component.ts

```
transition('inactive => active', animate('100ms ease-in')),
transition('active => inactive', animate('100ms ease-out'))
```



如果多个转场都有同样的时间线配置，就可以把它们合并进同一个 `transition` 定义中：

src/app/hero-list-combined-transitions.component.ts

```
transition('inactive => active, active => inactive',
 animate('100ms ease-out'))
```

如果要对同一个转场的两个方向都使用相同的时间线（就像前面的例子中那样），就可以使用 `<=>` 这种简写语法：

src/app/hero-list-twoway.component.ts

```
transition('inactive <=> active', animate('100ms ease-out'))
```

有时希望一些样式只在动画期间生效，但在结束后并不保留它们。这时可以把这些样式内联在 `transition` 中进行定义。 在这个例子中，该元素会立刻获得一组样式，然后动态转场到下一个状态。当转场结束时，这些样式并不会被保留，因为它们并没有被定义在 `state` 中。

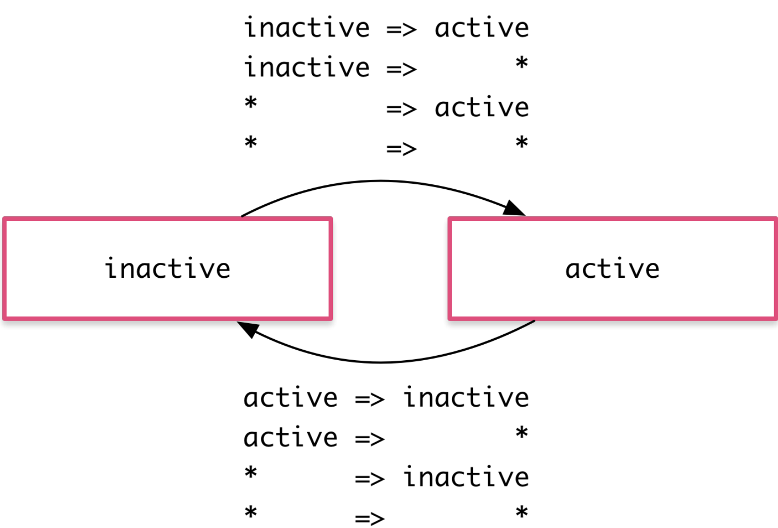
src/app/hero-list-inline-styles.component.ts

```
transition('inactive => active', [
  style({
    backgroundColor: '#cfd8dc',
    transform: 'scale(1.3)'
  }),
  animate('80ms ease-in', style({
    backgroundColor: '#eee',
    transform: 'scale(1)'
  }))
]),
```

`*`(通配符)状态

`*`(通配符)状态匹配**任何**动画状态。当定义那些不需要管当前处于什么状态的样式及转场时，这很有用。比如：

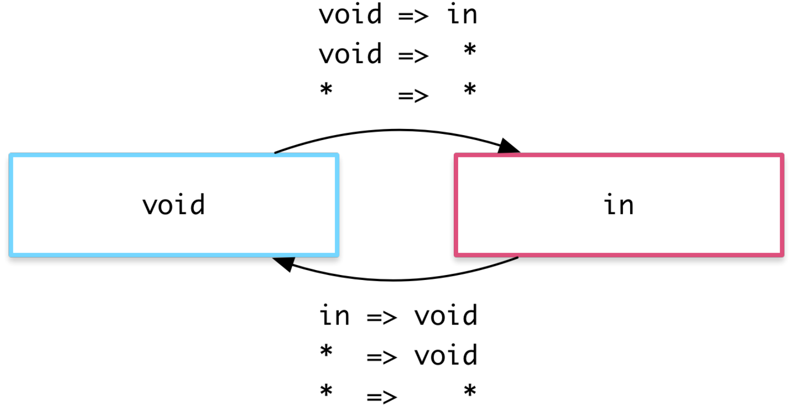
- 当该元素的状态从 `active` 变成任何其它状态时，`active => *` 转场都会生效。
- 当在**任意**两个状态之间切换时，`* => *` 转场都会生效。



`void` 状态

有一种叫做 `void` 的特殊状态，它可以应用在任何动画中。它表示元素**没有**被附加到视图。这种情况可能是由于它尚未被添加进来或者已经被移除了。 `void` 状态在定义“进场”和“离场”的动画时会非常有用。

比如当一个元素离开视图时，`* => void` 转场就会生效，而不管它在离场以前是什么状态。



`*` 通配符状态也能匹配 `void`。

例子：进场与离场

使用 `void` 和 `*` 状态，可以定义元素进场与离场时的转场动画：

- 进场：`void => *`
- 离场：`* => void`

例如，在下面的 `animations` 数组中，这两个转场语句使用 `void => *` 和 `* => void` 语法来让该元素以动画形式进入和离开当前视图。

hero-list-enter-leave.component.ts (excerpt)

```
animations: [
  trigger('flyInOut', [
    state('in', style({transform: 'translateX(0)'})),
    transition('void => *', [
      style({transform: 'translateX(-100%)'}),
      animate(100)
    ]),
    transition('* => void', [
      animate(100, style({transform: 'translateX(100%)'}))
    ])
  ])
]
```

注意，在这个例子中，这些样式在转场定义中被直接应用到了 `void` 状态，但并没有一个单独的 `state(void)` 定义。 这么做是因为希望在进场与离场时使用不一样的转换效果：元素从左侧进场，从右侧离开。

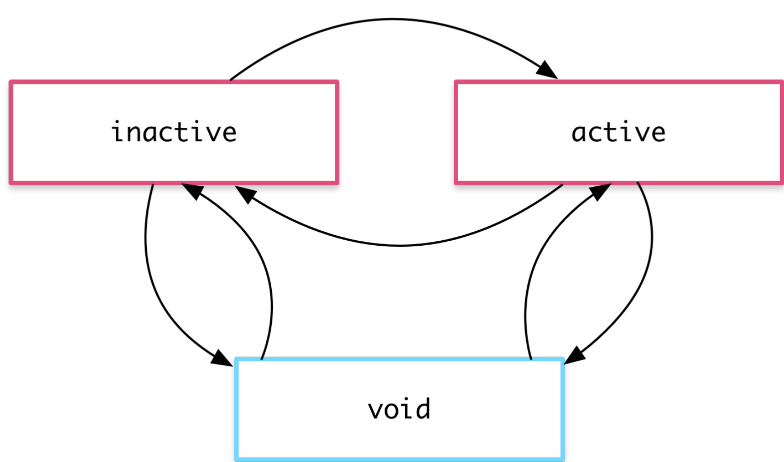
> 这两个常见的动画有自己的别名：
>
> ```
> transition(':enter', [ ... ]); // void => *
> transition(':leave', [ ... ]); // * => void
> ```

范例：从不同的状态下进场和离场

通过把英雄的状态用作动画的状态，还能把该动画跟以前的转场动画组合成一个复合动画。这让你能根据该英雄的当前状态为其配置不同的进场与离场动画：

- 非激活英雄进场：`void => inactive`
- 激活英雄进场：`void => active`
- 非激活英雄离场：`inactive => void`
- 激活英雄离场：`active => void`

现在就对每一种转场都有了细粒度的控制：



hero-list-enter-leave.component.ts (excerpt)

```
animations: [
  trigger('heroState', [
    state('inactive', style({transform: 'translateX(0) scale(1)'})),
    state('active',   style({transform: 'translateX(0) scale(1.1)'})),
    transition('inactive => active', animate('100ms ease-in')),
    transition('active => inactive', animate('100ms ease-out')),
    transition('void => inactive', [
      style({transform: 'translateX(-100%) scale(1)'}),
      animate(100)
    ]),
    transition('inactive => void', [
      animate(100, style({transform: 'translateX(100%) scale(1)'}))
    ]),
    transition('void => active', [
      style({transform: 'translateX(0) scale(0)'}),
      animate(200)
    ]),
    transition('active => void', [
      animate(200, style({transform: 'translateX(0) scale(0)'}))
    ])
  ])
]
```

可动的(Animatable)属性与单位

由于 Angular 的动画支持是基于 Web Animations 标准的，所以也能支持浏览器认为可以**参与动画**的任何属性。这些属性包括位置(position)、大小(size)、变换(transform)、颜色(color)、边框(border)等很多属性。W3C 维护着 一个“可动”属性列表。

尺寸类属性(如位置、大小、边框等)包括一个数字值和一个用来定义长度单位的后缀：

- `'50px'`
- `'3em'`
- `'100%'`

对大多数尺寸类属性而言，还能只定义一个数字，那就表示它使用的是像素(px)数：

- `50` 相当于 `'50px'`

自动属性值计算

有时候，你在开始运行之前都无法知道某个样式属性的值。比如，元素的宽度和高度往往依赖于它们的内容和屏幕的尺寸。处理这些属性对 CSS 动画而言通常是相当棘手的。

如果用 Angular 动画，就可以用一个特殊的 `*` 属性值来处理这种情况。该属性的值将会在运行期被计算出来，然后插入到这个动画中。

这个例子中的“离场”动画会取得该元素在离场前的高度，并且把它从这个高度用动画转场到 0 高度：

src/app/hero-list-auto.component.ts

```
animations: [
  trigger('shrinkOut', [
    state('in', style({height: '*'})),
    transition('* => void', [
      style({height: '*'}),
      animate(250, style({height: 0}))
    ])
  ])
]
```

动画时间线

对每一个动画转场效果，有三种时间线属性可以调整：持续时间(duration)、延迟(delay)和缓动(easing)函数。它们被合并到了一个单独的**转场时间线字符串。**

持续时间

持续时间控制动画从开始到结束要花多长时间。可以用三种方式定义持续时间：

- 作为一个普通数字，以毫秒为单位，如：`100`
- 作为一个字符串，以毫秒为单位，如：`'100ms'`
- 作为一个字符串，以秒为单位，如：`'0.1s'`

延迟

延迟控制的是在动画已经触发但尚未真正开始转场之前要等待多久。可以把它添加到字符串中的持续时间后面，它的选项格式也跟持续时间是一样的：

- 等待 100 毫秒，然后运行 200 毫秒：`'0.2s 100ms'`。

缓动函数

缓动函数用于控制动画在运行期间如何加速和减速。比如：使用 `ease-in` 函数意味着动画开始时相对缓慢，然后在进行中逐步加速。可以通过在这个字符串中的持续时间和延迟后面添加**第三个**值来控制使用哪个缓动函数(如果没有定义延迟就作为**第二个**值)。

- 等待 100 毫秒，然后运行 200 毫秒，并且带缓动：`'0.2s 100ms ease-out'`
- 运行 200 毫秒，并且带缓动：`'0.2s ease-in-out'`

例子

这里是两个自定义时间线的动态演示。“进场”和“离场”都持续 200 毫秒，也就是 `0.2s`，但它们有不同的缓动函数。“离场”动画会在 100 毫秒的延迟之后开始，也就是 `'0.2s 10 ease-out'`：

hero-list-timings.component.ts (excerpt)

```
animations: [
  trigger('flyInOut', [
    state('in', style({opacity: 1, transform: 'translateX(0)'})),
    transition('void => *', [
      style({
        opacity: 0,
        transform: 'translateX(-100%)'
      }),
      animate('0.2s ease-in')
    ]),
    transition('* => void', [
      animate('0.2s 0.1s ease-out', style({
        opacity: 0,
        transform: 'translateX(100%)'
      }))
    ])
  ])
]
```

基于关键帧(Keyframes)的多阶段动画

通过定义动画的**关键帧**，可以把两组样式之间的简单转场，升级成一种更复杂的动画，它会在转场期间经历一个或多个中间样式。

每个关键帧都可以被指定一个**偏移量**，用来定义该关键帧将被用在动画期间的哪个时间点。偏移量是一个介于 0(表示动画起点)和 1(表示动画终点)之间的数组。

这个例子使用关键帧来为进场和离场动画添加一些“反弹效果”：

hero-list-multistep.component.ts (excerpt)

```
animations: [
  trigger('flyInOut', [
    state('in', style({transform: 'translateX(0)'})),
    transition('void => *', [
      animate(300, keyframes([
        style({opacity: 0, transform: 'translateX(-100%)', offset: 0}),
        style({opacity: 1, transform: 'translateX(15px)',  offset: 0.3}),
        style({opacity: 1, transform: 'translateX(0)',     offset: 1.0})
      ]))
    ]),
    transition('* => void', [
      animate(300, keyframes([
        style({opacity: 1, transform: 'translateX(0)',     offset: 0}),
        style({opacity: 1, transform: 'translateX(-15px)', offset: 0.7}),
        style({opacity: 0, transform: 'translateX(100%)',  offset: 1.0})
      ]))
    ])
  ])
]
```

注意，这个偏移量并**不是**用绝对数字定义的时间段，而是在 0 到 1 之间的相对值（百分比）。动画的最终时间线会基于关键帧的偏移量、持续时间、延迟和缓动函数计算出来。

为关键帧定义偏移量是可选的。如果省略它们，偏移量会自动根据帧数平均分布出来。例如，三个未定义过偏移量的关键帧会分别获得偏移量：`0`、`0.5` 和 `1`。

并行动画组(Group)

你已经知道该如何在同一时间段进行多个样式的动画了：只要把它们都放进同一个 `style()` 定义中就行了！

但你也可能会希望为同时发生的几个动画配置不同的**时间线**。比如，同时对两个 CSS 属性做动画，但又得为它们定义不同的缓动函数。

这种情况下就可以用动画**组**来解决了。在这个例子中，同时在进场和离场时使用了组，以便能让它们使用两种不同的时间线配置。 它们被同时应用到同一个元素上，但又彼此独立运行：

hero-list-groups.component.ts (excerpt)

```
animations: [
  trigger('flyInOut', [
    state('in', style({width: 120, transform: 'translateX(0)', opacity: 1})),
    transition('void => *', [
      style({width: 10, transform: 'translateX(50px)', opacity: 0}),
      group([
        animate('0.3s 0.1s ease', style({
          transform: 'translateX(0)',
          width: 120
        })),
        animate('0.3s ease', style({
          opacity: 1
        }))
      ])
    ]),
    transition('* => void', [
      group([
        animate('0.3s ease', style({
          transform: 'translateX(50px)',
          width: 10
        })),
        animate('0.3s 0.2s ease', style({
          opacity: 0
        }))
      ])
    ])
  ])
]
```

其中一个动画组对元素的 `transform` 和 `width` 做动画，另一个组则对 `opacity` 做动画。

动画回调

当动画开始和结束时，会触发一个回调。

对于例子中的这个关键帧，你有一个叫做 `@flyInOut` 的 `trigger`。在那里你可以挂钩到那些回调，比如：

hero-list-multistep.component.ts (excerpt)

```
template: `
  <ul>
    <li *ngFor="let hero of heroes"
        (@flyInOut.start)="animationStarted($event)"
        (@flyInOut.done)="animationDone($event)"
        [@flyInOut]="'in'">
      {{hero.name}}
    </li>
  </ul>
`,
```

这些回调接收一个 `AnimationTransitionEvent` 参数，它包含一些有用的属性，例如 `fromState`，`toState` 和 `totalTime`。

无论动画是否实际执行过，那些回调都会触发。

用户输入

当用户点击链接、按下按钮或者输入文字时，这些用户动作都会产生 DOM 事件。 本章解释如何使用 Angular 事件绑定语法把这些事件绑定到事件处理器。

运行在线例子 / 下载范例

绑定到用户输入事件

你可以使用 Angular 事件绑定机制来响应任何 DOM 事件。 许多 DOM 事件是由用户输入触发的。绑定这些事件可以获取用户输入。

要绑定 DOM 事件，只要把 DOM 事件的名字包裹在圆括号中，然后用放在引号中的模板语句对它赋值就可以了。

下例展示了一个事件绑定，它实现了一个点击事件处理器：

src/app/click-me.component.ts

```
<button (click)="onClickMe()">Click me!</button>
```

等号左边的 `(click)` 表示把按钮的点击事件作为绑定目标。 等号右边引号中的文本是模板语句，通过调用组件的 `onClickMe` 方法来响应这个点击事件。

写绑定时，需要知道模板语句的执行上下文。 出现在模板语句中的每个标识符都属于特定的上下文对象。 这个对象通常都是控制此模板的 Angular 组件。 上例中只显示了一行 HTML，那段 HTML 片段属于下面这个组件：

src/app/click-me.component.ts

```
@Component({
  selector: 'app-click-me',
  template: `
    <button (click)="onClickMe()">Click me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = '';

  onClickMe() {
    this.clickMessage = 'You are my hero!';
  }
}
```

当用户点击按钮时，Angular 调用 `ClickMeComponent` 的 `onClickMe` 方法。

通过 $event 对象取得用户输入

DOM 事件可以携带可能对组件有用的信息。 本节将展示如何绑定输入框的 `keyup` 事件，在每个敲击键盘时获取用户输入。

下面的代码监听 `keyup` 事件，并将整个事件载荷 (`$event`) 传递给组件的事件处理器。

src/app/keyup.components.ts (template v.1)

```
template: `
  <input (keyup)="onKey($event)">
  <p>{{values}}</p>
`
```

当用户按下并释放一个按键时，触发 `keyup` 事件，Angular 在 `$event` 变量提供一个相应的 DOM 事件对象，上面的代码将它作为参数传递给 `onKey()` 方法。

src/app/keyup.components.ts (class v.1)

```
export class KeyUpComponent_v1 {
  values = '';

  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

`$event` 对象的属性取决于 DOM 事件的类型。例如，鼠标事件与输入框编辑事件包含了不同的信息。

所有标准 DOM 事件对象都有一个 `target` 属性， 引用触发该事件的元素。 在本例中， `target` 是<input> 元素， `event.target.value` 返回该元素的当前内容。

在组件的 `onKey()` 方法中，把输入框的值和分隔符 (|) 追加组件的 `values` 属性。 使用插值表达式来把存放累加结果的 `values` 属性回显到屏幕上。

假设用户输入字母“abc”，然后用退格键一个一个删除它们。 用户界面将显示：

```
a | ab | abc | ab | a | |
```

Give me some keys!

> 或者，你可以用 `event.key` 替代 `event.target.value`，积累各个按键本身，这样同样的用户输入可以产生：
>
> ```
> a | b | c | backspace | backspace | backspace |
> ```

$event的类型

上例将 `$event` 转换为 `any` 类型。 这样简化了代码，但是有成本。 没有任何类型信息能够揭示事件对象的属性，防止简单的错误。

下面的例子，使用了带类型方法：

src/app/keyup.components.ts (class v.1 - typed)

```
export class KeyUpComponent_v1 {
  values = '';


  onKey(event: KeyboardEvent) { // with type info
    this.values += (<HTMLInputElement>event.target).value + ' | ';
  }
}
```

`$event` 的类型现在是 `KeyboardEvent`。 不是所有的元素都有 `value` 属性，所以它将 `target` 转换为输入元素。 `OnKey` 方法更加清晰的表达了它期望从模板得到什么，以及它是如何解析事件的。

传入 $event 是靠不住的做法

类型化事件对象揭露了重要的一点，即反对把整个 DOM 事件传到方法中，因为这样组件会知道太多模板的信息。 只有当它知道更多它本不应了解的 HTML 实现细节时，它才能提取信息。 这就违反了模板（**用户看到的**）和组件（**应用如何处理用户数据**）之间的分离关注原则。

下面将介绍如何用模板引用变量来解决这个问题。

从一个模板引用变量中获得用户输入

还有另一种获取用户数据的方式：使用 Angular 的模板引用变量。 这些变量提供了从模块中直接访问元素的能力。 在标识符前加上井号 (#) 就能声明一个模板引用变量。

下面的例子使用了局部模板变量，在一个超简单的模板中实现按键反馈功能。

src/app/loop-back.component.ts

```
@Component({
  selector: 'app-loop-back',
  template: `
    <input #box (keyup)="0">
    <p>{{box.value}}</p>
  `
})
export class LoopbackComponent { }
```

这个模板引用变量名叫 `box`，在 `<input>` 元素声明，它引用 `<input>` 元素本身。 代码使用 `box` 获得输入元素的 `value` 值，并通过插值表达式把它显示在 `<p>` 标签中。

这个模板完全是完全自包含的。它没有绑定到组件，组件也没做任何事情。

在输入框中输入，就会看到每次按键时，显示也随之更新了。

keyup loop-back component

> 除非你绑定一个事件，否则这将完全无法工作。
>
> 只有在应用做了些异步事件（如击键），Angular 才更新绑定（并最终影响到屏幕）。 本例代码将 `keyup` 事件绑定到了数字 0，这可能是最短的模板语句了。 虽然这个语句不做什么，但它满足 Angular 的要求，所以 Angular 将更新屏幕。

从模板变量获得输入框比通过 `$event` 对象更加简单。 下面的代码重写了之前 `keyup` 示例，它使用变量来获得用户输入。

src/app/keyup.components.ts (v2)

```
@Component({
  selector: 'app-key-up2',
  template: `
    <input #box (keyup)="onKey(box.value)">
    <p>{{values}}</p>
  `
})
export class KeyUpComponent_v2 {
  values = '';
  onKey(value: string) {
    this.values += value + ' | ';
  }
}
```

这个方法最漂亮的一点是：组件代码从视图中获得了干净的数据值。再也不用了解 `$event` 变量及其结构了。

按键事件过滤（通过 `key.enter`）

`(keyup)` 事件处理器监听**每一次按键**。 有时只在意**回车**键，因为它标志着用户结束输入。 解决这个问题的一种方法是检查每个 `$event.keyCode`，只有键值是**回车**键时才采取行动。

更简单的方法是：绑定到 Angular 的 `keyup.enter` 模拟事件。 然后，只有当用户敲**回车**键时，Angular 才会调用事件处理器。

src/app/keyup.components.ts (v3)

```
@Component({
  selector: 'app-key-up3',
  template: `
    <input #box (keyup.enter)="onEnter(box.value)">
    <p>{{value}}</p>
  `
})
export class KeyUpComponent_v3 {
  value = '';
  onEnter(value: string) { this.value = value; }
}
```

下面展示了它是如何工作的。

Type away! Press [enter] when done

失去焦点事件 (blur)

前上例中，如果用户没有先按回车键，而是移开了鼠标，点击了页面中其它地方，输入框的当前值就会丢失。 只有当用户按下了回车键候，组件的 `values` 属性才能更新。

下面通过同时监听输入框的回车键和失去焦点事件来修正这个问题。

src/app/keyup.components.ts (v4)

```
@Component({
  selector: 'app-key-up4',
  template: `
    <input #box
      (keyup.enter)="update(box.value)"
      (blur)="update(box.value)">

    <p>{{value}}</p>
  `
})
export class KeyUpComponent_v4 {
  value = '';
  update(value: string) { this.value = value; }
}
```

把它们放在一起

上一章介绍了如何显示数据。 本章展示了事件绑定技术。

现在，在一个微型应用中一起使用它们，应用能显示一个英雄列表，并把新的英雄加到列表中。 用户可以通过输入英雄名和点击“添加”按钮来添加英雄。

Little Tour of Heroes

Add

- Windstorm
- Bombasto
- Magneta
- Tornado

下面就是“简版英雄指南”组件。

src/app/little-tour.component.ts

```
@Component({
  selector: 'app-little-tour',
  template: `
    <input #newHero
      (keyup.enter)="addHero(newHero.value)"
      (blur)="addHero(newHero.value); newHero.value='' ">

    <button (click)="addHero(newHero.value)">Add</button>

    <ul><li *ngFor="let hero of heroes">{{hero}}</li></ul>
  `
})
export class LittleTourComponent {
  heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
  addHero(newHero: string) {
    if (newHero) {
      this.heroes.push(newHero);
    }
  }
}
```

小结

- 使用模板变量来引用元素 — `newHero` 模板变量引用了 `<input>` 元素。 你可以在 `<input>` 的任何兄弟或子级元素中引用 `newHero`。
- 传递数值，而非元素 — 获取输入框的值并将**它**传递给组件的 `addHero`，而不要传递 `newHero`。
- 保持模板语句简单 — `(blur)` 事件被绑定到两个 JavaScript 语句。 第一句调用 `addHero`。第二句 `newHero.value=''` 在添加新英雄到列表中后清除输入框。

源代码

下面是本章讨论过的所有源码。

click-me.component.ts *keyup.components.ts* *loop-back.component.ts* *little-to*

```
1. import { Component } from '@angular/core';
2.
3. @Component({
```

```
4.   selector: 'app-click-me',
5.   template: `
6.     <button (click)="onClickMe()">Click me!</button>
7.     {{clickMessage}}`
8. })
9. export class ClickMeComponent {
10.   clickMessage = '';
11.
12.   onClickMe() {
13.     this.clickMessage = 'You are my hero!';
14.   }
15. }
```

小结

你已经掌握了响应用户输入和操作的基础技术。

这些技术对小规模演示很实用，但是在处理大量用户输入时，很容易变得累赘和笨拙。 要在数据录入字段和模型属性之间传递数据，双向数据绑定是更加优雅和简洁的方式。 下一章 `表单` 解释了如何用 `NgModel` 来进行双向绑定。

表单

表单是商业应用的支柱，你用它来执行登录、求助、下单、预订机票、安排会议，以及不计其数的其它数据录入任务。

在开发表单时，创建数据方面的体验是非常重要的，它能指引用户明细、高效的完成工作流程。

开发表单需要设计能力（那超出了本章的范围），而框架支持**双向数据绑定、变更检测、验证和错误处理**，而本章你将会学到它们。

这个页面演示了如何从草稿构建一个简单的表单。这个过程中你将学会如何：

- 用组件和模板构建 Angular 表单
- 用 `ngModel` 创建双向数据绑定，以读取和写入输入控件的值
- 跟踪状态的变化，并验证表单控件
- 使用特殊的 CSS 类来跟踪控件的状态并给出视觉反馈
- 向用户显示验证错误提示，以及启用/禁用表单控件
- 使用模板引用变量在 HTML 元素之间共享信息

你可以运行在线例子 / 下载范例，在 Stackblitz 中试用并下载本页的代码。

模板驱动表单 (template-driven forms)

通常，使用 Angular 模板语法编写模板，结合本章所描述的表单专用指令和技术来构建表单。

> 你还可以使用响应式（也叫模型驱动）的方式来构建表单。不过本章中只介绍模板驱动表单。

利用 Angular 模板，可以构建几乎所有表单 — 登录表单、联系人表单…… 以及任何的商务表单。 可以创造性的摆放各种控件、把它们绑定到数据、指定校验规则、显示校验错误、有条件的禁用或 启用特定的控件、触发内置的视觉反馈等等，不胜枚举。

它用起来很简单，这是因为 Angular 处理了大多数重复、单调的任务，这让你可以不必亲自操刀、身陷其中。

你将学习构建如下的“模板驱动”表单：

Hero Form

Name

Dr IQ

Alter Ego

Chuck Overstreet

Hero Power

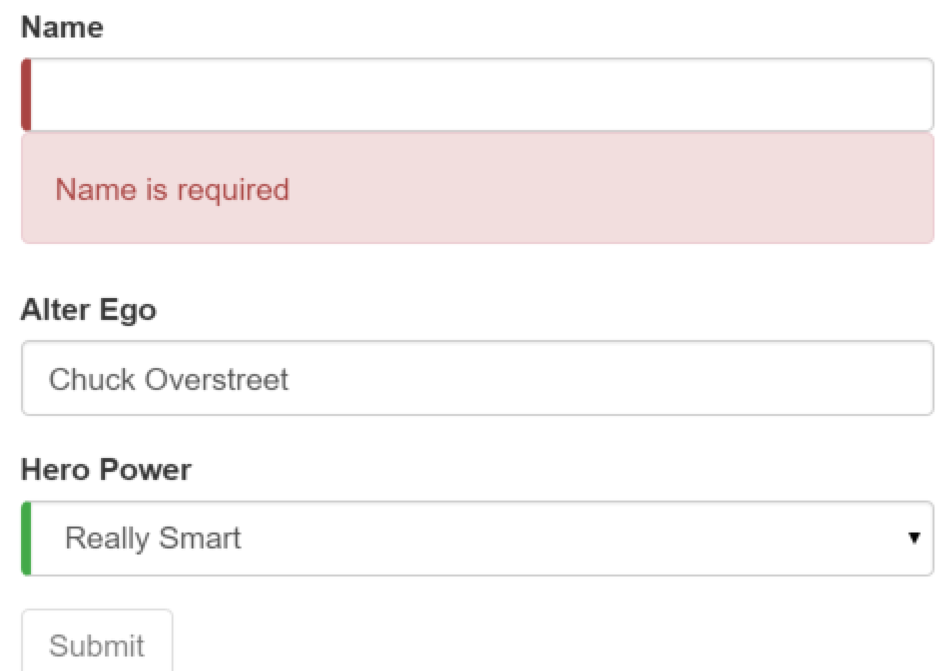
Really Smart

Submit

这里是**英雄职业介绍所**，使用这个表单来维护候选英雄们的个人信息。每个英雄都需要一份工作。 公司的使命就是让合适的英雄去应对恰当的危机！

表单中的三个字段，其中两个是必填的。必填的字段在左侧有个绿色的竖条，方便用户分辨哪些是必填项。

如果删除了英雄的名字，表单就会用醒目的样式把验证错误显示出来。



注意，提交按钮被禁用了，而且输入控件左侧的“必填”条从绿色变为了红色。

> 稍后，会使用标准 CSS 来定制“必填”条的颜色和位置。

你将一点点构建出此表单：

1. 创建 `Hero` 模型类
2. 创建控制此表单的组件。
3. 创建具有初始表单布局的模板。
4. 使用 `ngModel` 双向数据绑定语法把数据属性绑定到每个表单输入控件。
5. 往每个表单输入控件上添加 `name` 属性 (attribute)。
6. 添加自定义 CSS 来提供视觉反馈。
7. 显示和隐藏有效性验证的错误信息。
8. 使用 ngSubmit 处理表单提交。
9. 禁用此表单的提交按钮，直到表单变为有效。

准备工作

创建一个名为 `angular-forms` 的新项目：

```
ng new angular-forms
```

创建 Hero 模型类

当用户输入表单数据时，需要捕获它们的变化，并更新到模型的实例中。 除非知道模型里有什么，否则无法设计表单的布局。

最简单的模型是个“属性包”，用来存放应用中一件事物的事实。 这里使用三个必备字段 (`id`、`name`、`power`)，和一个可选字段 (`alterEgo`，译注：中文含义是第二人格，例如 X 战警中的 Jean / 黑凤凰)。

使用 Angular CLI 生成一个名叫 `Hero` 的新类：

```
ng generate class Hero
```

内容如下：

src/app/hero.ts

```
export class Hero {

  constructor(
    public id: number,
    public name: string,
    public power: string,
    public alterEgo?: string
  ) {  }

}
```

这是一个少量需求和零行为的贫血模型。对演示来说很完美。

TypeScript 编译器为每个 `public` 构造函数参数生成一个公共字段，在创建新的英雄实例时，自动把参数值赋给这些公共字段。

`alterEgo` 是可选的，调用构造函数时可省略，注意 `alterEgo?` 中的问号 (?)。

可以这样创建新英雄：

```
let myHero =  new Hero(42, 'SkyDog',
                       'Fetch any object at any distance',
                       'Leslie Rollover');
console.log('My hero is called ' + myHero.name); // "My hero is called SkyDog"
```

创建表单组件

Angular 表单分为两部分：基于 HTML 的**模板**和组件**类**，用来程序处理数据和用户交互。 先从组件类开始，是因为它可以简要说明英雄编辑器能做什么。

使用 Angular CLI 生成一个名叫 `HeroForm` 的新组件：

```
ng generate component HeroForm
```

内容如下：

src/app/hero-form/hero-form.component.ts (v1)

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

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

@Component({
  selector: 'app-hero-form',
  templateUrl: './hero-form.component.html',
  styleUrls: ['./hero-form.component.css']
})
export class HeroFormComponent {

  powers = ['Really Smart', 'Super Flexible',
            'Super Hot', 'Weather Changer'];

  model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');

  submitted = false;

  onSubmit() { this.submitted = true; }

  // TODO: Remove this when we're done
  get diagnostic() { return JSON.stringify(this.model); }
}
```

这个组件没有什么特别的地方，没有表单相关的东西，与之前写过的组件没什么不同。

只需要前面章节中学过的概念，就可以完全理解这个组件：

- 这段代码导入了 Angular 核心库以及你刚刚创建的 `Hero` 模型。
- `@Component` 选择器“hero-form”表示可以用 `<hero-form>` 标签把这个表单放进父模板。
- `moduleId: module.id` 属性设置了基地址，用于从相对模块路径加载 `templateUrl`。
- `templateUrl` 属性指向一个独立的 HTML 模板文件。

接下来，你可以注入一个数据服务，以获取或保存真实的数据，或者把这些属性暴露为输入属性和输出属性（参见Template Syntax中的输入和输出属性）来绑定到一个父组件。这不是现在需要关心的问题，未来的更改不会影响到这个表单。

- 你添加一个 `diagnostic` 属性，以返回这个模型的 JSON 形式。在开发过程中，它用于调试，最后清理时会丢弃它。

修改 app.module.ts

`app.module.ts` 定义了应用的根模块。其中标识即将用到的外部模块，以及声明属于本模块中的组件，例如 `HeroFormComponent`。

因为模板驱动的表单位于它们自己的模块，所以在使用表单之前，需要将 `FormsModule` 添加到应用模块的 `imports` 数组中。

对它做如下修改：

src/app/app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';

import { AppComponent }  from './app.component';
import { HeroFormComponent } from './hero-form/hero-form.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ],
  declarations: [
    AppComponent,
    HeroFormComponent
  ],
  providers: [],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

> 有两处更改
>
> 1. 导入 `FormsModule`。
> 2. 把 `FormsModule` 添加到 `ngModule` 装饰器的 `imports` 列表中，这样应用就能访问模板驱动表单的所有特性，包括 `ngModel`。

> 如果某个组件、指令或管道是属于 `imports` 中所导入的某个模块的，那就**不能再**把它再声明到本模块的 `declarations` 数组中。 如果它是你自己写的，并且确实属于当前模块，**才应该**把它声明在 `declarations` 数组中。

修改 app.component.ts

`AppComponent` 是应用的根组件，`HeroFormComponent` 将被放在其中。

把模板中的内容替换成如下代码：

src/app/app.component.html

```
<app-hero-form></app-hero-form>
```

> 这里只做了两处修改。 `template` 中只剩下这个新的元素标签，即组件的 `selector` 属性。这样当应用组件被加载时，就会显示这个英雄表单。 同样别忘了从类中移除了 `name` 字段。

创建初始 HTML 表单模板

修改模板文件，内容如下：

src/app/hero-form/hero-form.component.html

```
1. <div class="container">
2.     <h1>Hero Form</h1>
3.     <form>
4.       <div class="form-group">
5.         <label for="name">Name</label>
6.         <input type="text" class="form-control" id="name" required>
7.       </div>
8.
9.       <div class="form-group">
10.        <label for="alterEgo">Alter Ego</label>
11.        <input type="text" class="form-control" id="alterEgo">
12.      </div>
13.
14.      <button type="submit" class="btn btn-success">Submit</button>
15.
16.    </form>
17. </div>
```

这只是一段普通的旧式 HTML 5 代码。这里有两个 `Hero` 字段，`name` 和 `alterEgo`，供用户输入。

Name `<input>` 控件具有 HTML5 的 `required` 属性；但 **Alter Ego** `<input>` 控件没有，因为 `alterEgo` 字段是可选的。

在底部添加个 **Submit** 按钮，它还带一些 CSS 样式类。

你还没有真正用到 Angular。没有绑定，没有额外的指令，只有布局。

> 在模板驱动表单中，你只要导入了 `FormsModule` 就不用对 `<form>` 做任何改动来使用 `FormsModule`。接下来你会看到它的原理。

`container`、`form-group`、`form-control` 和 `btn` 类来自 Twitter Bootstrap。这些类纯粹是装饰品。Bootstrap 为这个表单提供了一些样式。

> ANGULAR 表单不需要任何样式库
>
> Angular 不需要 `container`、`form-group`、`form-control` 和 `btn` 类， 或者外部库的任何样式。Angular 应用可以使用任何 CSS 库…… ， 或者啥都不用。

要添加样式表，就打开 `styles.css`，并把下列代码添加到顶部：

src/styles.css

```
@import url('https://unpkg.com/bootstrap@3.3.7/dist/css/bootstrap.min.css');
```

用 ngFor 添加超能力

英雄必须从认证过的固定列表中选择一项超能力。 这个列表位于 `HeroFormComponent` 中。

在表单中添加 `select`，用 `ngFor` 把 `powers` 列表绑定到列表选项。 之前的显示数据一章中见过 `ngFor`。

在 **Alter Ego** 的紧下方添加如下 HTML：

src/app/hero-form/hero-form.component.html (powers)

```
<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control" id="power" required>
    <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
  </select>
</div>
```

列表中的每一项超能力都会渲染成 <option> 标签。 模板输入变量 `pow` 在每个迭代指向不同的超能力，使用双花括号插值表达式语法来显示它的名称。

使用 ngModel 进行双向数据绑定

如果立即运行此应用，你将会失望。

Hero Form

Name

Alter Ego

Hero Power

Really Smart

因为还没有绑定到某个英雄，所以看不到任何数据。 解决方案见前面的章节。 显示数据介绍了属性绑定。 用户输入介绍了如何通过事件绑定来监听 DOM 事件，以及如何用显示值更新组件的属性。

现在，需要同时进行显示、监听和提取。

虽然可以在表单中再次使用这些技术。 但是，这里将介绍个新东西，`[(ngModel)]` 语法，使表单绑定到模型的工作变得超级简单。

找到 **Name** 对应的 <input> 标签，并且像这样修改它：

src/app/hero-form/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name">
TODO: remove this: {{model.name}}
```

> 在 input 标签后添加用于诊断的插值表达式，以看清正在发生什么事。 给自己留个备注，提醒你完成后移除它。

聚焦到绑定语法 `[(ngModel)]="..."` 上。

你需要更多的工作来显示数据。在表单中声明一个模板变量。往 `<form>` 标签中加入 `#heroForm="ngForm"`，代码如下：

src/app/hero-form/hero-form.component.html (excerpt)

```
<form #heroForm="ngForm">
```

`heroForm` 变量是一个到 `NgForm` 指令的引用，它代表该表单的整体。

> ## NgForm 指令
>
> 什么是 `NgForm` 指令？ 但你明明没有添加过NgForm指令啊！
>
> Angular 替你做了。Angular 会在 `<form>` 标签上自动创建并附加一个 `NgForm` 指令。
>
> `NgForm` 指令为 `form` 增补了一些额外特性。 它会控制那些带有 `ngModel` 指令和 `name` 属性的元素，监听他们的属性（包括其有效性）。 它还有自己的 `valid` 属性，这个属性只有在**它包含的每个控件**都有效时才是真。

如果现在运行这个应用，开始在**姓名**输入框中键入，添加和删除字符，将看到它们从插值结果中显示和消失。 某一瞬间，它可能是这样的：

Dr IQ 3000

TODO: remove this: Dr IQ 3000

诊断信息可以证明，数据确实从输入框流动到模型，再反向流动回来。

> 这就是双向数据绑定！要了解更多信息，参见模板语法页的使用 NgModel 进行双向绑定。

注意，`<input>` 标签还添加了 `name` 属性 (attribute)，并设置为 "name"，表示英雄的名字。 使用任何唯一的值都可以，但使用具有描述性的名字会更有帮助。 当在表单中使用 `[(ngModel)]` 时，必须要定义 `name` 属性。

> 在内部，Angular 创建了一些 `FormControl`，并把它们注册到 `NgForm` 指令，再将该指令附加到 `<form>` 标签。 注册每个 `FormControl` 时，使用 `name` 属性值作为键值。本章后面会讨论 `NgForm`。

为**第二人格**和**超能力**属性添加类似的 `[(ngModel)]` 绑定和 `name` 属性。 抛弃输入框的绑定消息，在组件顶部添加到 `diagnostic` 属性的新绑定。 这样就能确认双向数据绑定**在整个 Hero 模型上**都能正常工作了。

修改之后，这个表单的核心是这样的：

src/app/hero-form/hero-form.component.html (excerpt)

```
{{diagnostic}}
<div class="form-group">
  <label for="name">Name</label>
  <input type="text" class="form-control" id="name"
         required
         [(ngModel)]="model.name" name="name">
</div>

<div class="form-group">
  <label for="alterEgo">Alter Ego</label>
  <input type="text"  class="form-control" id="alterEgo"
         [(ngModel)]="model.alterEgo" name="alterEgo">
</div>

<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control"  id="power"
          required
          [(ngModel)]="model.power" name="power">
    <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
  </select>
</div>
```

> - 每个 input 元素都有 `id` 属性， `label` 元素的 `for` 属性用它来匹配到对应的输入控件。
> - 每个 input 元素都有 `name` 属性，Angular 表单用它注册控件。

如果现在运行本应用，修改 Hero 模型的每个属性，表单是这样的：

Hero Form

{"id":18,"name":"Dr IQ 3000","power":"Super Flexible","alterEgo":"Chuck OverUnderStreet"}

Name

Dr IQ 3000

Alter Ego

Chuck OverUnderStreet

Hero Power

Super Flexible

表单顶部的诊断信息反映出所做的一切更改。

表单顶部的 `{{diagnostic}}` 绑定已经完成了它的使命，删除它。

通过 ngModel 跟踪修改状态与有效性验证

在表单中使用 `ngModel` 可以获得比仅使用双向数据绑定更多的控制权。它还会告诉你很多信息：用户碰过此控件吗？它的值变化了吗？数据变得无效了吗？

NgModel 指令不仅仅跟踪状态。它还使用特定的 Angular CSS 类来更新控件，以反映当前状态。 可以利用这些 CSS 类来修改控件的外观，显示或隐藏消息。

状态	为真时的 CSS 类	为假时的 CSS 类
控件被访问过。	`ng-touched`	`ng-untouched`
控件的值变化了。	`ng-dirty`	`ng-pristine`
控件的值有效。	`ng-valid`	`ng-invalid`

往姓名 `<input>` 标签上添加名叫 spy 的临时模板引用变量， 然后用这个 spy 来显示它上面的所有 CSS 类。

src/app/hero-form/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" id="name"
  required
  [(ngModel)]="model.name" name="name"
  #spy>
<br>TODO: remove this: {{spy.className}}
```

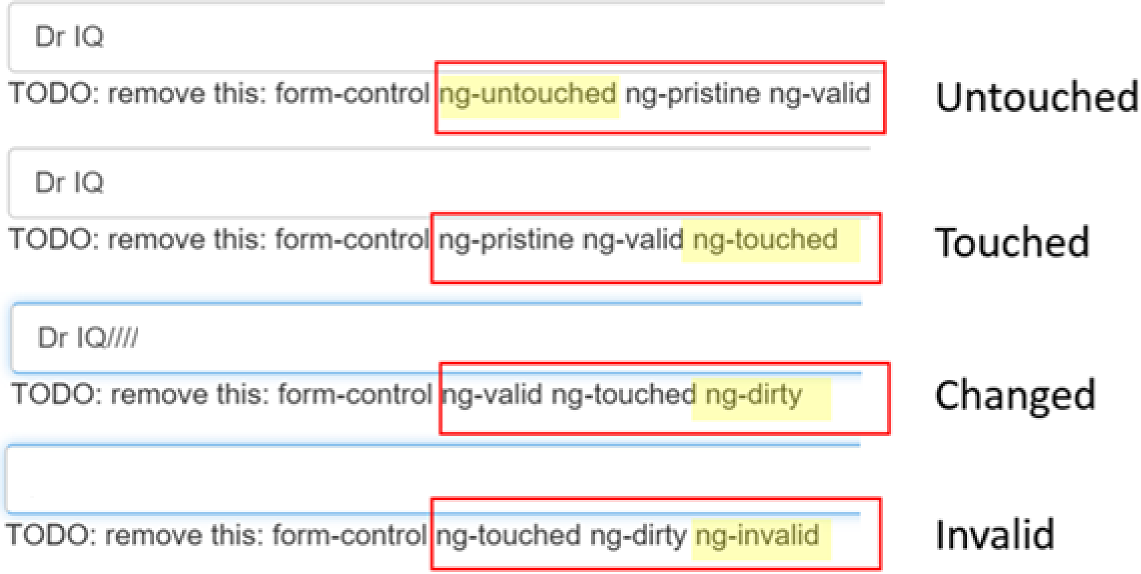
现在，运行本应用，并让**姓名**输入框获得焦点。 然后严格按照下面四个步骤来做：

1. 查看输入框，但别碰它。
2. 点击输入框，然后点击输入框外面。
3. 在名字的末尾添加些斜杠。
4. 删除名字。

动作和它对应的效果如下：

Dr IQ
TODO: remove this: form-control ng-untouched ng-pristine ng-valid

你会看到下列转换及其类名：

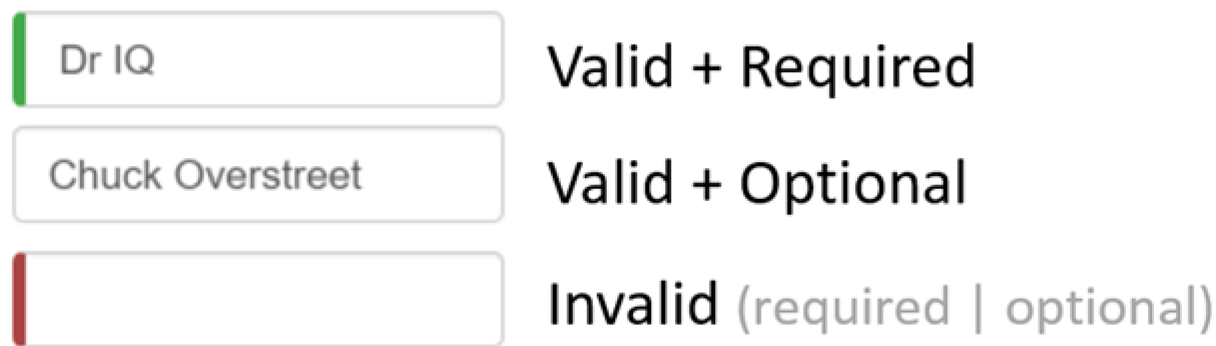


(`ng-valid` | `ng-invalid`)这一对是最有趣的部分，因为当数据变得无效时，你希望发出强力的视觉信号， 还想要标记出必填字段。可以通过加入自定义 CSS 来提供视觉反馈。

删除模板引用变量 `#spy` 和 `TODO`，因为它们已经完成了使命。

添加用于视觉反馈的自定义 CSS

可以在输入框的左侧添加带颜色的竖条，用于标记必填字段和无效输入：



在新建的 `forms.css` 文件中，添加两个样式来实现这一效果。把这个文件添加到项目中，与 `index.html` 相邻。

src/assets/forms.css

```
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}

.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}
```

修改 `index.html` 中的 `<head>`，以包含这个样式表：

src/index.html (styles)

```
<link rel="stylesheet" href="assets/forms.css">
```

显示和隐藏验证错误信息

你还能做的更好。“Name” 输入框是必填的，清空它会让左侧的条变红。这表示**某些东西**是错的，但用户不知道错在哪里，或者如何纠正。 可以借助 `ng-invalid` 类来给出有用的提示。

当用户删除姓名时，应该是这样的：

要达到这个效果，在 `<input>` 标签中添加：

- 模板引用变量
- “is required”消息，放在邻近的 `<div>` 元素中，只有当控件无效时，才显示它。

下面这个例子中把一条错误信息添加到了`name`输入框中：

src/app/hero-form/hero-form.component.html (excerpt)

```
<label for="name">Name</label>
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name"
       #name="ngModel">
<div [hidden]="name.valid || name.pristine"
     class="alert alert-danger">
  Name is required
</div>
```

模板引用变量可以访问模板中输入框的 Angular 控件。 这里，创建了名叫 `name` 的变量，并且赋值为 "ngModel"。

> 为什么是 “ngModel”？ 指令的 exportAs 属性告诉 Angular 如何链接模板引用变量到指令。 这里把 `name` 设置为 `ngModel` 是因为 `ngModel` 指令的 `exportAs` 属性设置成了 “ngModel”。

你把 `div` 元素的 `hidden` 属性绑定到 `name` 控件的属性，这样就可以控制“姓名”字段错误信息的可见性了。

src/app/hero-form/hero-form.component.html (hidden-error-msg)

```
<div [hidden]="name.valid || name.pristine"
     class="alert alert-danger">
```

上例中，当控件是有效的 (valid) 或全新的 (pristine) 时，隐藏消息。“全新的”意味着从它被显示在表单中开始，用户还从未修改过它的值。

这种用户体验取决于开发人员的选择。有些人会希望任何时候都显示这条消息。 如果忽略了 `pristine` 状态，就会只在值有效时隐藏此消息。 如果往这个组件中传入全新（空）的英雄，或者无效的英雄，将立刻看到错误信息 —— 虽然你还啥都没做。

有些人会为这种行为感到不安。它们希望只有在用户做出无效的更改时才显示这个消息。 如果当控件是“全新”状态时也隐藏消息，就能达到这个目的。 在往表单中添加新英雄时，将看到这种选择的重要性。

英雄的**第二人格**是可选项，所以不用改它。

英雄的**超能力**选项是必填的。 只要愿意，可以往 `<select>` 上添加相同的错误处理。 但没有必要，这个选择框已经限制了“超能力”只能选有效值。

现在，你要在这个表单中添加新的英雄。 在表单的底部放置“New Hero（新增英雄）”按钮，并把它的点击事件绑定到 `newHero` 组件。

src/app/hero-form/hero-form.component.html (New Hero button)

```
<button type="button" class="btn btn-default" (click)="newHero()">New Hero</button>
```

src/app/hero-form/hero-form.component.ts (New Hero method)

```
newHero() {
  this.model = new Hero(42, '', '');
}
```

再次运行应用，点击 **New Hero** 按钮，表单被清空了。 输入框左侧的**必填项**竖条是红色的，表示 `name` 和 `power` 属性是无效的。 这可以理解，因为有一些必填字段。 错误信息是隐藏的，因为表单还是全新的，还没有修改任何东西。

输入名字，再次点击 **New Hero** 按钮。 这次，出现了错误信息！为什么？你不希望显示新（空）的英雄时，出现错误信息。

使用浏览器工具审查这个元素就会发现，这个 **name** 输入框并不是全新的。 表单记得你在点击 **New Hero** 前输入的名字。 更换了英雄对象**并不会重置控件的“全新”状态**。

你必须清除所有标记，在调用 `newHero()` 方法后调用表单的 `reset()` 方法即可。

src/app/hero-form/hero-form.component.html (Reset the form)

```
<button type="button" class="btn btn-default" (click)="newHero();
heroForm.reset()">New Hero</button>
```

现在点击“New Hero”重设表单和它的控制标记。

使用 ngSubmit 提交该表单

在填表完成之后，用户还应该能提交这个表单。 “Submit（提交）”按钮位于表单的底部，它自己不做任何事，但因为有特殊的 type 值 (`type="submit"`)，所以会触发表单提交。

现在这样仅仅触发“表单提交”是没用的。 要让它有用，就要把该表单的 `ngSubmit` 事件属性绑定到英雄表单组件的 `onSubmit()` 方法上：

src/app/hero-form/hero-form.component.html (ngSubmit)

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">
```

你已经定义了一个模板引用变量 `#heroForm`，并且把赋值为“ngForm”。 现在，就可以在“Submit”按钮中访问这个表单了。

你要把表单的总体有效性通过 `heroForm` 变量绑定到此按钮的 `disabled` 属性上，代码如下：

src/app/hero-form/hero-form.component.html (submit-button)

```
<button type="submit" class="btn btn-success"
[disabled]="!heroForm.form.valid">Submit</button>
```

重新运行应用。表单打开时，状态是有效的，按钮是可用的。

现在，如果你删除**姓名**，就会违反“必填姓名”规则，就会像以前那样显示出错误信息。同时，Submit 按钮也被禁用了。

没感觉吗？再想一会儿。如果没有 Angular `NgForm` 的帮助，又该怎么让按钮的禁用/启用状态和表单的有效性关联起来呢？

有了 Angular，它就是这么简单：

1. 定义模板引用变量，放在（强化过的）form 元素上
2. 从很多行之外的按钮上引用这个变量。

切换两个表单区域（额外的奖励）

提交表单还是不够激动人心。

> 对演示来说，这个收场很平淡的。老实说，即使让它更出彩，也无法教给你任何关于表单的新知识。但这是练习新学到的绑定技能的好机会。 如果你不感兴趣，可以跳到本章的总结部分。

来实现一些更炫的视觉效果吧。 隐藏掉数据输入框，显示一些其它东西。

先把表单包裹进 `<div>` 中，再把它的 `hidden` 属性绑定到 `HeroFormComponent.submitted` 属性。

src/app/hero-form/hero-form.component.html (excerpt)

```
<div [hidden]="submitted">
  <h1>Hero Form</h1>
  <form (ngSubmit)="onSubmit()" #heroForm="ngForm">

     <!-- ... all of the form ... -->

  </form>
</div>
```

主表单从一开始就是可见的，因为 `submitted` 属性是 false，直到提交了这个表单。 来自 `HeroFormComponent` 的代码片段证实了这一点：

src/app/hero-form/hero-form.component.ts (submitted)

```
submitted = false;

onSubmit() { this.submitted = true; }
```

当点击 Submit 按钮时，`submitted` 标志会变成 true，并且表单像预想中一样消失了。

现在，当表单处于已提交状态时，需要显示一些别的东西。 在刚刚写的 `<div>` 包装下方，添加下列 HTML 语句：

src/app/hero-form/hero-form.component.html (excerpt)

```
<div [hidden]="!submitted">
  <h2>You submitted the following:</h2>
  <div class="row">
    <div class="col-xs-3">Name</div>
    <div class="col-xs-9  pull-left">{{ model.name }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Alter Ego</div>
    <div class="col-xs-9 pull-left">{{ model.alterEgo }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Power</div>
    <div class="col-xs-9 pull-left">{{ model.power }}</div>
  </div>
  <br>
  <button class="btn btn-primary" (click)="submitted=false">Edit</button>
</div>
```

英雄又出现了，它通过插值表达式绑定显示为只读内容。 这一小段 HTML 只在组件处于已提交状态时才会显示。

这段 HTML 包含一个 “Edit（编辑）”按钮，将 click 事件绑定到表达式，用于清除 `submitted` 标志。

当点 **Edit** 按钮时，这个只读块消失了，可编辑的表单重新出现了。

小结

本章讨论的 Angular 表单技术利用了下列框架特性来支持数据修改、验证和更多操作：

- Angular HTML 表单模板。
- 带有 `@Component` 装饰器的表单组件类。
- 通过绑定到 `NgForm.ngSubmit` 事件属性来处理表单提交。
- 模板引用变量，例如 `#heroForm` 和 `#name`。
- `[(ngModel)]` 语法用来实现双向数据绑定。
- `name` 属性的用途是有效性验证和对表单元素的变更进行追踪。
- 指向 input 控件的引用变量上的 `valid` 属性，可用于检查控件是否有效、是否显示/隐藏错误信息。
- 通过绑定到 `NgForm` 的有效性状态，控制 **Submit** 按钮的禁用状态。
- 定制 CSS 类来给用户提供无效控件的视觉反馈。

下面是该应用最终版本的代码：

< **hero-form/hero-form.component.ts** *hero-form/hero-form.component.html* >

```
1. import { Component } from '@angular/core';
2.
3. import { Hero }    from '../hero';
4.
5. @Component({
6.   selector: 'app-hero-form',
7.   templateUrl: './hero-form.component.html',
8.   styleUrls: ['./hero-form.component.css']
9. })
10. export class HeroFormComponent {
11.
12.   powers = ['Really Smart', 'Super Flexible',
13.             'Super Hot', 'Weather Changer'];
14.
15.   model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');
16.
17.   submitted = false;
18.
19.   onSubmit() { this.submitted = true; }
20.
21.   newHero() {
22.     this.model = new Hero(42, '', '');
23.   }
24. }
```

表单验证

通过验证用户输入的准确性和完整性，来增强整体数据质量。

本文展示了如何在界面中如何验证用户输入，并显示有用的验证信息，先使用模板驱动表单方式，再使用响应式表单方式。

> 参见表单和响应式表单了解关于这些选择的更多知识。

模板驱动验证

为了往模板驱动表单中添加验证机制，你要添加一些验证属性，就像原生的 HTML 表单验证器。 Angular 会用指令来匹配这些具有验证功能的指令。

每当表单控件中的值发生变化时，Angular 就会进行验证，并生成一个验证错误的列表（对应着 INVALID 状态）或者 null（对应着 VALID 状态）。

你可以通过把 `ngModel` 导出成局部模板变量来查看该控件的状态。 比如下面这个例子就把 `NgModel` 导出成了一个名叫 `name` 的变量：

template/hero-form-template.component.html (name)

```
<input id="name" name="name" class="form-control"
       required minlength="4" appForbiddenName="bob"
       [(ngModel)]="hero.name" #name="ngModel" >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
     class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>

</div>
```

请注意以下几点:

- `<input>` 元素带有一些 HTML 验证属性：`required` 和 `minlength`。它还带有一个自定义的验证器指令 `forbiddenName`。要了解更多信息，参见自定义验证器一节。
- `#name="ngModel"` 把 `NgModel` 导出成了一个名叫 `name` 的局部变量。`NgModel` 把自己控制的 `FormControl` 实例的属性映射出去，让你能在模板中检查控件的状态，比如 `valid` 和 `dirty`。要了解完整的控件属性，参见 API 参考手册中的AbstractControl。
- `<div>` 元素的 `*ngIf` 揭露了一套嵌套消息 `divs`，但是只在有“name”错误和控制器为 `dirty` 或者 `touched`。
- 每个嵌套的 `<div>` 为其中一个可能出现的验证错误显示一条自定义消息。比如 `required`、`minlength` 和 `forbiddenName`。

> ### 为何检查 dirty 和 touched?
>
> 你肯定不希望应用在用户还没有编辑过表单的时候就给他们显示错误提示。 对 `dirty` 和 `touched` 的检查可以避免这种问题。改变控件的值会改变控件的 `dirty`（脏）状态，而当控件失去焦点时，就会改变控件的 `touched`（碰过）状态。

响应式表单的验证

在响应式表单中，真正的源码都在组件类中。不应该通过模板上的属性来添加验证器，而应该在组件类中直接把验证器函数添加到表单控件模型上（`FormControl`）。然后，一旦控件发生了变化，Angular 就会调用这些函数。

验证器函数

有两种验证器函数：同步验证器和异步验证器。

- 同步验证器函数接受一个控件实例，然后返回一组验证错误或 `null`。你可以在实例化一个 `FormControl` 时把它作为构造函数的第二个参数传进去。
- 异步验证器函数接受一个控件实例，并返回一个承诺（Promise）或可观察对象（Observable），它们稍后会发出一组验证错误或者 `null`。你可以在实例化一个 `FormControl` 时把它作为构造函数的第三个参数传进去。

注意：出于性能方面的考虑，只有在所有同步验证器都通过之后，Angular 才会运行异步验证器。当每一个异步验证器都执行完之后，才会设置这些验证错误。

内置验证器

你可以写自己的验证器，也可以使用一些 Angular 内置的验证器。

模板驱动表单中可用的那些属性型验证器（如 `required`、`minlength` 等）对应于 `Validators` 类中的同名函数。要想查看内置验证器的全列表，参见 API 参考手册中的验证器部分。

要想把这个英雄表单改造成一个响应式表单，你还是用那些内置验证器，但这次改为用它们的函数形态。

reactive/hero-form-reactive.component.ts (validator functions)

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    'name': new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator.
    ]),
    'alterEgo': new FormControl(this.hero.alterEgo),
    'power': new FormControl(this.hero.power, Validators.required)
  });
}

get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }
```

注意

- `name` 控件设置了两个内置验证器：`Validators.required` 和 `Validators.minLength(4)`。要了解更多信息，参见本章的自定义验证器一节。
- 由于这些验证器都是同步验证器，因此你要把它们作为第二个参数传进去。
- 可以通过把这些函数放进一个数组后传进去，可以支持多重验证器。
- 这个例子添加了一些 getter 方法。在响应式表单中，你通常会通过它所属的控件组（FormGroup）的 `get` 方法来访问表单控件，但有时候为模板定义一些 getter 作为简短形式。

如果你到模板中找到 name 输入框，就会发现它和模板驱动的例子很相似。

reactive/hero-form-reactive.component.html (name with error msg)

```
<input id="name" class="form-control"
       formControlName="name" required >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
     class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>
</div>
```

关键改动是:

- 该表单不再导出任何指令，而是使用组件类中定义的 `name` 读取器。
- `required` 属性仍然存在，虽然验证不再需要它，但你仍然要在模板中保留它，以支持 CSS 样式或可访问性。

自定义验证器

由于内置验证器无法适用于所有应用场景，有时候你还是得创建自定义验证器。

考虑前面的例子中的 `forbiddenNameValidator` 函数。该函数的定义看起来是这样的:

shared/forbidden-name.directive.ts (forbiddenNameValidator)

```
/** A hero's name can't match the given regular expression */
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
  return (control: AbstractControl): {[key: string]: any} => {
    const forbidden = nameRe.test(control.value);
    return forbidden ? {'forbiddenName': {value: control.value}} : null;
  };
}
```

这个函数实际上是一个工厂，它接受一个用来检测指定名字是否已被禁用的正则表达式，并返回一个验证器函数。

在本例中，禁止的名字是“bob”； 验证器会拒绝任何带有“bob”的英雄名字。 在其他地方，只要配置的正则表达式可以匹配上，它可能拒绝“alice”或者任何其他名字。

`forbiddenNameValidator` 工厂函数返回配置好的验证器函数。 该函数接受一个 Angular 控制器对象，并在控制器值有效时返回 null，或无效时返回验证错误对象。 验证错误对象通常有一个名为验证秘钥（`forbiddenName`）的属性。其值为一个任意词典，你可以用来插入错误信息（`{name}`）。

自定义异步验证器和同步验证器很像，只是它们必须返回一个稍后会输出 null 或“验证错误对象”的承诺（Promise）或可观察对象，如果是可观察对象，那么它必须在某个时间点被完成（complete），那时候这个表单就会使用它输出的最后一个值作为验证结果。（译注：HTTP 服务是自动完成的，但是某些自定义的可观察对象可能需要手动调用 complete 方法）

添加响应式表单

在响应式表单组件中，添加自定义验证器相当简单。你所要做的一切就是直接把这个函数传给 `FormControl`
。

reactive/hero-form-reactive.component.ts (validator functions)

```
this.heroForm = new FormGroup({
  'name': new FormControl(this.hero.name, [
    Validators.required,
    Validators.minLength(4),
    forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom
validator.
  ]),
  'alterEgo': new FormControl(this.hero.alterEgo),
  'power': new FormControl(this.hero.power, Validators.required)
});
```

添加到模板驱动表单

在模板驱动表单中，你不用直接访问 `FormControl` 实例。所以不能像响应式表单中那样把验证器传进去，而应该在模板中添加一个指令。

`ForbiddenValidatorDirective` 指令相当于 `forbiddenNameValidator` 的包装器。

Angular 在验证流程中的识别出指令的作用，是因为指令把自己注册到了 `NG_VALIDATORS` 提供商中，该提供商拥有一组可扩展的验证器。

shared/forbidden-name.directive.ts (providers)

```
providers: [{provide: NG_VALIDATORS, useExisting: ForbiddenValidatorDirective,
multi: true}]
```

然后该指令类实现了 `Validator` 接口，以便它能简单的与 Angular 表单集成在一起。这个指令的其余部分有助于你理解它们是如何协作的：

shared/forbidden-name.directive.ts (directive)

```
1. @Directive({
2.   selector: '[appForbiddenName]',
3.   providers: [{provide: NG_VALIDATORS, useExisting:
   ForbiddenValidatorDirective, multi: true}]
4. })
5. export class ForbiddenValidatorDirective implements Validator {
6.   @Input('appForbiddenName') forbiddenName: string;
7. 
8.   validate(control: AbstractControl): {[key: string]: any} {
9.     return this.forbiddenName ? forbiddenNameValidator(new
   RegExp(this.forbiddenName, 'i'))(control)
10.                               : null;
11.   }
12. }
```

一旦 `ForbiddenValidatorDirective` 写好了，你只要把 `forbiddenName` 选择器添加到输入框上就可以激活这个验证器了。比如：

template/hero-form-template.component.html (forbidden-name-input)

```
<input id="name" name="name" class="form-control"
       required minlength="4" appForbiddenName="bob"
       [(ngModel)]="hero.name" #name="ngModel" >
```

> 你可能注意到了自定义验证器指令是用 `useExisting` 而不是 `useClass` 来实例化的。注册的验证器必须是这个 `ForbiddenValidatorDirective` 实例本身，也就是表单中 `forbiddenName` 属性被绑定到了"bob"的那个。如果用 `useClass` 来代替 `useExisting`，就会注册一个新的类实例，而它是没有 `forbiddenName` 的。

表示控件状态的 CSS 类

像 AngularJS 中一样，Angular 会自动把很多控件属性作为 CSS 类映射到控件所在的元素上。你可以使用这些类来根据表单状态给表单控件元素添加样式。目前支持下列类：

- `.ng-valid`
- `.ng-invalid`
- `.ng-pending`
- `.ng-pristine`
- `.ng-dirty`
- `.ng-untouched`
- `.ng-touched`

这个英雄表单使用 `.ng-valid` 和 `.ng-invalid` 来设置每个表单控件的边框颜色。

forms.css (status classes)

```
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}

.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}
```

你可以运行在线例子 / 下载范例来查看完整的响应式和模板驱动表单的代码。

响应式表单

响应式表单是 Angular 中用**响应式**风格创建表单的技术。 本章会在构建“英雄详情编辑器”的过程中，逐步讲解响应式表单的概念。

试试Reactive Forms (final) in Stackblitz / 下载范例。

你还可以运行Reactive Forms Demo in Stackblitz / 下载范例，并从顶部选取一个中间步骤。

响应式表单简介

Angular 提供了两种构建表单的技术：**响应式**表单和**模板驱动**表单。 这两项技术都属于 `@angular/forms` 库，并且共享一组公共的表单控件类。

但是它们在设计哲学、编程风格和具体技术上有显著区别。 所以，它们都有自己的模块：`ReactiveFormsModule` 和 `FormsModule`。

响应式表单

Angular 的**响应式**表单能让实现**响应式编程风格**更容易，这种编程风格更倾向于在非 UI 的**数据模型**（通常接收自服务器）之间显式的管理数据流， 并且用一个 UI 导向的**表单模型**来保存屏幕上 HTML 控件的状态和值。 响应式表单可以让使用响应式编程模式、测试和校验变得更容易。

使用**响应式**表单，你可以在组件中创建表单控件的对象树，并使用本章中传授的技巧把它们绑定到组件模板中的原生表单控件元素上。

你可以在组件类中直接创建和维护表单控件对象。由于组件类可以同时访问数据模型和表单控件结构， 因此你可以把表单模型值的变化推送到表单控件中，并把变化后的值拉取回来。 组件可以监听表单控件状态的变化，并对此做出响应。

直接使用表单控件对象的优点之一是值和有效性状态的更新总是同步的，并且在你的控制之下。 你不会遇到时序问题，这个问题有时在模板驱动表单中会成为灾难。而且响应式表单更容易进行单元测试。

在响应式编程范式中，组件会负责维护**数据模型**的不可变性，把模型当做纯粹的原始数据源。 组件不会直接更新数据模型，而是把用户的修改提取出来，把它们转发给外部的组件或服务，外部程序才会使用这些进行处理（比如保存它们）， 并且给组件返回一个新的**数据模型**，以反映模型状态的变化。

使用响应式表单的指令，并不要求你遵循所有的响应式编程原则，但它能让你更容易使用响应式编程方法，从而更愿意使用它。

模板驱动表单

在模板一章中介绍过的**模板驱动**表单，是一种完全不同的方式。

你把 HTML 表单控件（比如 `<input>` 和 `<select>`）放进组件模板中，并用 `ngModel` 等指令把它们绑定到组件中**数据模型**的属性上。

你不用自己创建 Angular 表单控件对象。Angular 指令会使用数据绑定中的信息创建它们。 你不用自己推送和拉取数据。Angular 使用 `ngModel` 来替你管理它们。 当用户做出修改时，Angular 会据此更新可变的**数据模型**。

因此，`ngModel` 并不是 `ReactiveFormsModule` 模块的一部分。

虽然这意味着组件中的代码更少，但是模板驱动表单是异步工作的，这可能在更高级的场景中让开发复杂化。

异步 vs. 同步

响应式表单是同步的而模板驱动表单是异步的。

使用响应式表单，你会在代码中创建整个表单控件树。 你可以立即更新一个值或者深入到表单中的任意节点，因为所有的控件都始终是可用的。

模板驱动表单会委托指令来创建它们的表单控件。 为了消除“检查完后又变化了”的错误，这些指令需要消耗一个以上的变更检测周期来构建整个控件树。 这意味着在从组件类中操纵任何控件之前，你都必须先等待一个节拍。

比如，如果你用 `@ViewChild(NgForm)` 查询来注入表单控件，并在生命周期钩子 `ngAfterViewInit`中检查它，就会发现它没有子控件。 你必须使用 `setTimeout` 等待一个节拍才能从控件中提取值、测试有效性，或把它设置为新值。

模板驱动表单的异步性让单元测试也变得复杂化了。 你必须把测试代码包裹在 `async()` 或 `fakeAsync()` 中来解决要查阅的值尚不存在的情况。 使用响应式表单，在所期望的时机一切都是可用的。

选择响应式表单还是模板驱动表单?

响应式表单和模板驱动表单是两种架构范式，各有优缺点。 请自行选择更合适的方法，甚至可以在同一个应用中同时使用它们。

本章其余的部分只专注于**响应式**范式以及响应式表单技术的详情。 要了解关于**模板驱动表单**的更多信息，参见表单一章。

在下一节，你要先准备一个响应式表单范例的项目，然后就可以开始学习Angular 表单类，并在响应式表单中使用它们了。

准备工作

创建一个名叫`angular-reactive-forms`的新项目：

```
ng new angular-reactive-forms
```

创建数据模型

本章的焦点是响应式表单组件以及编辑一个英雄。 你需要一个 `Hero` 类和一些英雄数据。

使用 CLI 创建一个名叫 `data-model` 的新类：

```
ng generate class data-model
```

并把下列内容复制到 `data-model.ts` 中：

src/app/data-model.ts

```
export class Hero {
  id = 0;
  name = '';
  addresses: Address[];
}

export class Address {
  street = '';
  city   = '';
  state  = '';
  zip    = '';
}

export const heroes: Hero[] = [
  {
    id: 1,
    name: 'Whirlwind',
    addresses: [
      {street: '123 Main',  city: 'Anywhere', state: 'CA',  zip: '94801'},
      {street: '456 Maple', city: 'Somewhere', state: 'VA', zip: '23226'},
    ]
  },
  {
    id: 2,
    name: 'Bombastic',
    addresses: [
      {street: '789 Elm',  city: 'Smallville', state: 'OH',  zip: '04501'},
    ]
  },
  {
    id: 3,
    name: 'Magneta',
    addresses: [ ]
  },
];

export const states = ['CA', 'MD', 'OH', 'VA'];
```

这个文件导出两个类和两个常量。`Address` 和 `Hero` 类定义应用的**数据模型。** `heroes` 和 `states` 常量提供测试数据。

创建**响应式表单**组件

生成一个名叫 `HeroDetail` 的新组件：

```
ng generate component HeroDetail
```

并导入：

src/app/hero-detail/hero-detail.component.ts

```
import { FormControl } from '@angular/forms';
```

接下来，创建并导出一个带 `FormControl` 的 `HeroDetailComponent` 类。 `FormControl` 是一个指令，它允许你直接创建并管理一个 `FormControl` 实例。

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
export class HeroDetailComponent1 {
  name = new FormControl();
}
```

这里创建了一个名叫 `name` 的 `FormControl`。 它将会绑定到模板中的一个 `<input>` 元素，表示英雄的名字。

`FormControl` 构造函数接收三个可选参数： 初始值、验证器数组和异步验证器数组。

> 最简单的控件并不需要数据或验证器，但是在实际应用中，大部分表单控件都会同时具备它们。 要想深入了解 `Validators`，参见表单验证一章。

创建模板

现在，把组件的模板文件 `src/app/hero-detail.component.html` 修改为如下内容：

src/app/hero-detail/hero-detail.component.html

```
<h2>Hero Detail</h2>
<h3><i>Just a FormControl</i></h3>
<label class="center-block">Name:
  <input class="form-control" [formControl]="name">
</label>
```

要让 Angular 知道你希望把这个输入框关联到类中的 `FormControl` 型属性 `name`，就要在模板中的 `<input>` 上加一句 `[formControl]="name"`。

> 请忽略 CSS 类 `form-control`，它属于Bootstrap CSS library而不是 Angular。 它会为表单添加样式，但是对表单的逻辑毫无影响。

导入 `ReactiveFormsModule`

`HeroDetailComponent` 的模板中使用了来自 `ReactiveFormsModule` 的 `formControlName`。

在 `app.module.ts` 中做了下面两件事：

1. 使用 JavaScript 的 `import` 语句访问 `ReactiveFormsModule` 和 `HeroDetailComponent`。
2. 把 `ReactiveFormsModule` 添加到 `AppModule` 的 `imports` 列表中。

src/app/app.module.ts (excerpt)

```
import { NgModule }              from '@angular/core';
import { BrowserModule }         from '@angular/platform-browser';
import { ReactiveFormsModule } from '@angular/forms';  // <-- #1 import module

import { AppComponent }          from './app.component';
import { HeroDetailComponent } from './hero-detail/hero-detail.component';

@NgModule({
  declarations: [
    AppComponent,
    HeroDetailComponent,
  ],
  imports: [
    BrowserModule,
    ReactiveFormsModule // <-- #2 add to @NgModule imports
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

显示 `HeroDetailComponent`

修改 `AppComponent` 的模板，以便显示 `HeroDetailComponent`。

src/app/app.component.html

```
<div class="container">
  <h1>Reactive Forms</h1>
  <app-hero-detail></app-hero-detail>
</div>
```

基础的表单类

本文使用四个基础类来构建响应式表单：

类	说明
`AbstractControl`	`AbstractControl`是这三个具体表单类的抽象基类。 并为它们提供了一些共同的行为和属性。
`FormControl`	**FormControl** 用于跟踪一个**单独的**表单控件的值和有效性状态。它对应于一个 HTML 表单控件，比如 `<input>` 或 `<select>`。
`FormGroup`	`FormGroup`用于 跟踪**一组**`AbstractControl` 的实例的值和有效性状态。 该组的属性中包含了它的子控件。 组件中的顶级表单就是一个 `FormGroup`。
`FormArray`	`FormArray`用于跟踪 `AbstractControl` 实例组成的有序数组的值和有效性状态。

为应用添加样式

要在 `AppComponent` 和 `HeroDetailComponent` 的模板中使用 Bootstrap 中的 CSS 类。请把 `bootstrap` 的**CSS样式表文件**添加到 `style.css` 的头部：

styles.css

```
@import url('https://unpkg.com/bootstrap@3.3.7/dist/css/bootstrap.min.css');
```

这些做好之后，启动应用服务器：

```
ng serve
```

浏览器应该显示成这样：

Hero Detail

Just a FormControl

添加 FormGroup

通常，如果有多个 `FormControl`，你要把它们都注册进一个父 `FormGroup` 中。 只要把它添加到 `hero-detail.component.ts` 的 `imports` 区就可以了。

src/app/hero-detail/hero-detail.component.ts

```
import { Component }              from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
```

在这个类中，把 `FormControl` 包裹进了一个名叫 `heroForm` 的 `FormGroup` 中，代码如下：

src/app/hero-detail/hero-detail.component.ts

```
export class HeroDetailComponent2 {
  heroForm = new FormGroup ({
    name: new FormControl()
  });
}
```

现在你改完了这个类，该把它映射到模板中了。把 `hero-detail.component.html` 改成这样：

src/app/hero-detail/hero-detail.component.html

```
<h2>Hero Detail</h2>
<h3><i>FormControl in a FormGroup</i></h3>
<form [formGroup]="heroForm">
  <div class="form-group">
    <label class="center-block">Name:
      <input class="form-control" formControlName="name">
    </label>
  </div>
</form>
```

注意，现在单行输入框位于一个 `form` 元素中。

`formGroup` 是一个响应式表单的指令，它拿到一个现有 `FormGroup` 实例，并把它关联到一个 HTML 元素上。这种情况下，它关联到的是 `<form>` 元素上的 `FormGroup` 实例 `heroForm`。

由于现在有了一个 `FormGroup`，因此你必须修改模板语法来把这个 `<input>` 关联到组件类中对应的 `FormControl` 上。 以前没有父 `FormGroup` 的时候，`[formControl]="name"` 也能正常工作，因为该指令可以

独立工作，也就是说，不在 `FormGroup` 中时它也能用。 有了 `FormGroup`，`name` 这个 `<input>` 就需要再添加一个语法 `formControlName=name`，以便让它关联到类中正确的 `FormControl` 上。 这个语法告诉 Angular，查阅父 `FormGroup`（这里是 `heroForm`），然后在这个 `FormGroup` 中查阅一个名叫 `name` 的 `FormControl`。

表单模型概览

当用户在 `<input>` 中输入数据时，它的值就会进入这个表单模型。 要想知道表单模型是什么样的，请在 `hero-detail.component.html` 的 `<form>` 标签紧后面添加如下代码：

src/app/hero-detail/hero-detail.component.html

```
<p>Form value: {{ heroForm.value | json }}</p>
```

`heroForm.value` 会返回表单模型。 用 `JsonPipe` 管道把这个模型以 JSON 格式渲染到浏览器中。

Hero Detail

FormControl in a FormGroup

Name:

Form value: { "name": null }

最初的 `name` 属性是个空字符串，在 **name** `<input>` 中输入之后，可以看到这些按键出现在了 JSON 中。

在真实的应用中，表单很快就会变大。 `FormBuilder` 能让表单开发和维护变得更简单。

`FormBuilder` 简介

`FormBuilder` 类能通过处理控件创建的细节问题来帮你减少重复劳动。

要使用 `FormBuilder`，就要先把它导入到 `hero-detail.component.ts` 中。你可以删除 `FormControl`：

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
import { Component }              from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';
```

遵循下列步骤来用 `FormBuilder` 把 `HeroDetailComponent` 重构得更容易读写：

- 明确把 `heroForm` 属性的类型声明为 `FormGroup`，稍后你会初始化它。
- 把 `FormBuilder` 注入到构造函数中。
- 添加一个名叫 `createForm()` 的新方法，它会用 `FormBuilder` 来定义 `heroForm`。
- 在构造函数中调用 `createForm()`。

修改过的 `HeroDetailComponent` 代码如下：

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
export class HeroDetailComponent3 {
  heroForm: FormGroup; // <--- heroForm is of type FormGroup

  constructor(private fb: FormBuilder) { // <--- inject FormBuilder
    this.createForm();
  }

  createForm() {
    this.heroForm = this.fb.group({
      name: '', // <--- the FormControl called "name"
    });
  }
}
```

`FormBuilder.group` 是一个用来创建 `FormGroup` 的工厂方法，它接受一个对象，对象的键和值分别是 `FormControl` 的名字和它的定义。 在这个例子中，`name` 控件的初始值是空字符串。

把一组控件定义在一个单一对象中，可以让你的代码更加紧凑、易读。 因为你不必写一系列重复的 `new FormControl(...)` 语句。

`Validators.required`

虽然本章不会深入讲解验证机制，但还是有一个例子来示范如何简单的在响应式表单中使用 `Validators.required`。

首先，导入 `Validators` 符号。

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
import { Component }                          from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
```

要想让 `name` 这个 `FormControl` 是必须的，请把 `FormGroup` 中的 `name` 属性改为一个数组。第一个条目是 `name` 的初始值，第二个是 `required` 验证器：`Validators.required`。

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
this.heroForm = this.fb.group({
  name: ['', Validators.required ],
});
```

> 响应式验证器是一些简单、可组合的函数。 在模板驱动表单中配置验证器有些困难，因为你必须把验证器包装进指令中。

修改模板底部的诊断信息，以显示表单的有效性状态。

src/app/hero-detail/hero-detail.component.html (excerpt)

```
<p>Form value: {{ heroForm.value | json }}</p>
<p>Form status: {{ heroForm.status | json }}</p>
```

浏览器会显示下列内容：

Hero Detail

A FormGroup with a single FormControl using FormBuilder

Name:

Form value: { "name": "" }

Form status: "INVALID"

`Validators.required` 生效了，但状态还是 `INVALID`，因为输入框中还没有值。 在输入框中输入，就会看到这个状态从 `INVALID` 变成了 `VALID`。

在真实的应用中，你要把这些诊断信息替换成用户友好的信息。

在本章的其余部分，`Validators.required` 是可有可无的，但在每个与此范例配置相同的范例中都会保留它。

要了解 Angular 表单验证器的更多知识，参见表单验证器一章。

更多的 `FormControl`

本节要添加一些 `FormControl`，用来表示住址、一项超能力，和一个副手。

另外，住址中有一个所在州属性，用户将会从 `<select>` 框中选择一个州，你会用 `<option>` 元素渲染各个州。从 `data-model.ts` 中导入 `states`（州列表）。

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
import { Component }                          from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

import { states } from '../data-model';
```

声明 `states` 属性并往 `heroForm` 中添加一些表示住址的 `FormControl`，代码如下：

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
export class HeroDetailComponent4 {
  heroForm: FormGroup;
  states = states;

  constructor(private fb: FormBuilder) {
    this.createForm();
  }

  createForm() {
    this.heroForm = this.fb.group({
      name: ['', Validators.required ],
      street: '',
      city: '',
      state: '',
      zip: '',
      power: '',
      sidekick: ''
    });
  }
}
```

然后把下列代码添加到 `hero-detail.component.html` 文件中。

src/app/hero-detail/hero-detail.component.html

```
<h2>Hero Detail</h2>
<h3><i>A FormGroup with multiple FormControls</i></h3>
<form [formGroup]="heroForm">
  <div class="form-group">
    <label class="center-block">Name:
      <input class="form-control" formControlName="name">
    </label>
  </div>
  <div class="form-group">
    <label class="center-block">Street:
      <input class="form-control" formControlName="street">
    </label>
  </div>
  <div class="form-group">
    <label class="center-block">City:
      <input class="form-control" formControlName="city">
    </label>
  </div>
  <div class="form-group">
    <label class="center-block">State:
      <select class="form-control" formControlName="state">
          <option *ngFor="let state of states" [value]="state">{{state}}</option>
      </select>
    </label>
  </div>
  <div class="form-group">
    <label class="center-block">Zip Code:
      <input class="form-control" formControlName="zip">
    </label>
  </div>
  <div class="form-group radio">
    <h4>Super power:</h4>
    <label class="center-block"><input type="radio" formControlName="power"
value="flight">Flight</label>
    <label class="center-block"><input type="radio" formControlName="power"
value="x-ray vision">X-ray vision</label>
    <label class="center-block"><input type="radio" formControlName="power"
value="strength">Strength</label>
  </div>
  <div class="checkbox">
    <label class="center-block">
```

```
      <input type="checkbox" formControlName="sidekick">I have a sidekick.
    </label>
  </div>
</form>


<p>Form value: {{ heroForm.value | json }}</p>
```

> **注意**：不用管这些脚本中提到的 `form-group`、`form-control`、`center-block` 和 `checkbox` 等。它们是来自 **Bootstrap** 的 CSS 类，Angular 本身不会管它们。 注意 `formGroupName` 和 `formControlName` 属性。 他们是 Angular 指令，用于把相应的 HTML 控件绑定到组件中的 `FormGroup` 和 `FormControl` 类型的属性上。

修改过的模板包含更多文本输入框，一个 `state` 选择框，`power` （超能力）的单选按钮和一个 `sidekick` 检查框。

你要用 `[value]="state"` 来绑定 `<option>` 的 `value` 属性。 如果不绑定这个值，这个选择框就会显示来自数据模型中的第一个选项。

组件**类**定义了控件属性而不用管它们在模板中的表现形式。 你可以像定义 `name` 控件一样定义 `state`、`power` 和 `sidekick` 控件，并用 `formControlName` 指令来指定 `FormControl` 的名字。

参见 API 参考手册中的radio buttons、 selects和 checkboxes

多级 `FormGroup`

要想更有效的管理这个表单的大小，你可以把一些相关的 `FormControl` 组织到多级 `FormGroup` 中。 比如，`street`、`city`、`state` 和 `zip` 就可以作为一个名叫 `address` 的 `FormGroup` 中的理想属性。 用这种方式，多级表单组和控件可以让你轻松地映射多层结构的数据模型，以帮你跟踪这组相关控件的有效性和状态。

你用 `FormBuilder` 在这个名叫 `heroForm` 的组件中创建一个 `FormGroup`，并把它用作父 `FormGroup`。 再次使用 `FormBuilder` 创建一个子级 `FormGroup`，其中包括这些住址控件。把结果赋值给父 `FormGroup` 中新的 `address` 属性。

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
export class HeroDetailComponent5 {
  heroForm: FormGroup;
  states = states;

  constructor(private fb: FormBuilder) {
    this.createForm();
  }

  createForm() {
    this.heroForm = this.fb.group({ // <-- the parent FormGroup
      name: ['', Validators.required ],
      address: this.fb.group({ // <-- the child FormGroup
        street: '',
        city: '',
        state: '',
        zip: ''
      }),
      power: '',
      sidekick: ''
    });
  }
}
```

当你修改组件类中表单控件的结构时，还必须对组件模板进行相应的调整。

在 `hero-detail.component.html` 中，把与住址有关的 `FormControl` 包裹进一个 `div` 中。 往这个 `<div>` 上添加一个 `formGroupName` 指令，并且把它绑定到 `"address"` 上。 这个 `address` 属性是一个 `FormGroup`，它的父 `FormGroup` 就是 `heroForm`。 把这个 `name` `<input>` 留在此 `<div>` 中。

要让这个变化更加明显，在文本的顶部加入一个 `<h4>` 头：**Secret Lair**。 新的住址组的 HTML 如下：

src/app/hero-detail/hero-detail.component.html (excerpt)

```
<div formGroupName="address" class="well well-lg">
  <h4>Secret Lair</h4>
  <div class="form-group">
    <label class="center-block">Street:
      <input class="form-control" formControlName="street">
    </label>
  </div>
  <div class="form-group">
    <label class="center-block">City:
      <input class="form-control" formControlName="city">
    </label>
  </div>
  <div class="form-group">
    <label class="center-block">State:
      <select class="form-control" formControlName="state">
        <option *ngFor="let state of states" [value]="state">{{state}}</option>
      </select>
    </label>
  </div>
  <div class="form-group">
    <label class="center-block">Zip Code:
      <input class="form-control" formControlName="zip">
    </label>
  </div>
</div>
```

做完这些之后，浏览器中的 JSON 输出就变成了带有多级 `FormGroup` 的表单模型。

heroForm value: { "name": "", "address": { "street": "", "city": "", "state": "", "zip": "" } }

这时模板和表单模型在彼此通讯了。

查看 `FormControl` 的属性

你可以使用 `.get()` 方法来提取表单中一个单独 `FormControl` 的状态。 你可以在组件类中这么做，或者通过往模板中添加下列代码来把它显示在页面中，就添加在 `{{form.value | json}}` 插值表达式的紧后面：

src/app/hero-detail/hero-detail.component.html

```
<p>Name value: {{ heroForm.get('name').value }}</p>
```

要点取得 `FormGroup` 中的 `FormControl` 的状态，使用点语法来指定到控件的路径。

src/app/hero-detail/hero-detail.component.html

```
<p>Street value: {{ heroForm.get('address.street').value}}</p>
```

> **注意**：如果你正在边看边跟着写代码，当你到达 `FormArray` 那节时，别忘了移除到 `address.street` 的引用。那一节中，你要在组件类中修改这个地址的名字，如果你把它留在模板中，就会抛出一个错误。

你可以使用此技术来显示 `FromControl` 的任意属性，代码如下：

属性	说明
`myControl.value`	`FormControl` 的值。
`myControl.status`	`FormControl` 的有效性。可能的值有 `VALID`、`INVALID`、`PENDING` 或 `DISABLED`。
`myControl.pristine`	如果用户**尚未**改变过这个控件的值，则为 `true`。它总是与 `myControl.dirty` 相反。
`myControl.untouched`	如果用户尚未进入这个 HTML 控件，也没有触发过它的 `blur`（失去焦点）事件，则为 `true`。 它是 `myControl.touched` 的反义词。

要了解 `FormControl` 的更多属性，参见 API 参考手册的**AbstractControl**部分。

检查 `FormControl` 属性的另一个原因是确保用户输入了有效的值。 要了解更多关于 Angular 表单验证的知识，参见表单验证一章。

数据模型与表单模型

此刻，表单显示的是空值。 `HeroDetailComponent` 应该显示一个英雄的值，这个值可能接收自远端服务器。

在这个应用中， `HeroDetailComponent` 从它的父组件 `HeroListComponent` 中取得一个英雄。

来自服务器的 `hero` 就是数据模型，而 `FormControl` 的结构就是表单模型。

组件必须把数据模型中的英雄值复制到表单模型中。这里隐含着两个非常重要的点。

1. 开发人员必须理解数据模型是如何映射到表单模型中的属性的。
2. 用户修改时的数据流是从 DOM 元素流向表单模型的，而不是数据模型。

表单控件永远不会修改**数据模型**。

表单模型和数据模型的结构并不需要精确匹配。在一个特定的屏幕上，你通常只会展现数据模型的一个子集。 但是表单模型的形态越接近数据模型，事情就会越简单。

在 `HeroDetailComponent` 中，这两个模型是非常接近的。

回忆一下 `data-model.ts` 中 `Hero` 和 `Address` 的 定义:

src/app/data-model.ts (classes)

```
export class Hero {
  id = 0;
  name = '';
  addresses: Address[];
}

export class Address {
  street = '';
  city   = '';
  state  = '';
  zip    = '';
}
```

这里又是组件的 `FormGroup` 定义。

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
this.heroForm = this.fb.group({
  name: ['', Validators.required ],
  address: this.fb.group({
    street: '',
    city: '',
    state: '',
    zip: ''
  }),
  power: '',
  sidekick: ''
});
```

在这些模型中有两点显著的差异：

1. `Hero` 有一个 `id`。表单模型中则没有，因为你通常不会把主键展示给用户。
2. `Hero` 有一个住址数组。这个表单模型只表示了一个住址，稍后的 `FormArray`则可以表示多个。

保持这两个模型的形态尽可能接近，可以在下一节中轻松使用 `patchValue()` 和 `setValue()` 方法把数据模型拷贝到表单模型中。

首先把 `address` 这个 `FormGroup` 的定义重构成这样：

src/app/hero-detail/hero-detail.component.ts

```
this.heroForm = this.fb.group({
  name: ['', Validators.required ],
  address: this.fb.group(new Address()), // <-- a FormGroup with a new address
  power: '',
  sidekick: ''
});
```

为了确保从 `data-model` 中导入，你可以引用 `Hero` 和 `Address` 类：

src/app/hero-detail/hero-detail.component.ts

```
import { Address, Hero, states } from '../data-model';
```

使用 `setValue()` 和 `patchValue()` 来操纵表单模型

> **注意**：如果你正在跟着写代码，那么本节是可选的，因为剩下的步骤并不依赖它。

以前，你创建了控件，并同时初始化它的值。 你也可以稍后用 `setValue()` 和 `patchValue()` 来初始化或重置这些值。

`setValue()`

借助`setValue()`，你可以设置每个表单控件的值，只要把与表单模型的属性精确匹配的数据模型传进去就可以了。

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
this.heroForm.setValue({
  name:    this.hero.name,
  address: this.hero.addresses[0] || new Address()
});
```

`setValue()` 方法会在赋值给任何表单控件之前先检查数据对象的值。

它不会接受一个与 `FormGroup` 结构不同或缺少表单组中任何一个控件的数据对象。 这种方式下，如果你有什么拼写错误或控件嵌套的不正确，它就能返回一些有用的错误信息。 反之，`patchValue()` 会默默地失败。

注意，你几乎可以直接把这个 `hero` 用作 `setValue()` 的参数，因为它的形态与组件的 `FormGroup` 结构是非常像的。

你现在只能显示英雄的第一个住址，不过你还必须考虑 `hero` 完全没有住址的可能性。 就像这个在数据对象参数中对 `address` 属性进行有条件的设置：

src/app/hero-detail/hero-detail.component.ts

```
address: this.hero.addresses[0] || new Address()
```

`patchValue()`

借助`patchValue()`，你可以通过提供一个只包含要更新的控件的键值对象来把值赋给 `FormGroup` 中的指定控件。

这个例子只会设置表单的 `name` 控件。

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
this.heroForm.patchValue({
  name: this.hero.name
});
```

借助`patchValue()`，你可以更灵活地解决数据模型和表单模型之间的差异。 但是和 `setValue()` 不同，`patchValue()` 不会检查缺失的控件值，并且不会抛出有用的错误信息。

创建 `HeroListComponent` 和 `HeroService`

要更好地演示后面的响应式表单技巧，可以通过加入 `HeroListComponent` 和 `HeroService` 来为这个范例添加更多功能。

`HeroDetalComponent` 是一个嵌套在 `HeroListComponent` 的**主从**视图中的子组件。如果把它们放在一起就是这样的：

Select a hero:

Refresh Whirlwind Bombastic Magneta

Hero Detail

Editing: Magneta

Name:

Magneta

首先使用下列命令添加一个 `HeroListComponent`：

```
ng generate component HeroList
```

把 `HeroListComponent` 修改为如下内容：

hero-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Observable }        from 'rxjs';
import { finalize } from 'rxjs/operators';

import { Hero }        from '../data-model';
import { HeroService } from '../hero.service';

@Component({
  selector: 'app-hero-list',
  templateUrl: './hero-list.component.html',
  styleUrls: ['./hero-list.component.css']
})
export class HeroListComponent implements OnInit {
  heroes: Observable<Hero[]>;
  isLoading = false;
  selectedHero: Hero;

  constructor(private heroService: HeroService) { }

  ngOnInit() { this.getHeroes(); }

  getHeroes() {
    this.isLoading = true;
    this.heroes = this.heroService.getHeroes()
                      // TODO: error handling
                      .pipe(finalize(() => this.isLoading = false));
    this.selectedHero = undefined;
  }

  select(hero: Hero) { this.selectedHero = hero; }
}
```

接着使用下列命令添加 `HeroService`:

```
ng generate service Hero
```

然后，把它的内容改为：

hero.service.ts

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

import { Observable, of } from 'rxjs';
import { delay } from 'rxjs/operators';

import { Hero, heroes } from './data-model';

@Injectable()
export class HeroService {

  delayMs = 500;

  // Fake server get; assume nothing can go wrong
  getHeroes(): Observable<Hero[]> {
    return of(heroes).pipe(delay(this.delayMs)); // simulate latency with delay
  }

  // Fake server update; assume nothing can go wrong
  updateHero(hero: Hero): Observable<Hero>  {
    const oldHero = heroes.find(h => h.id === hero.id);
    const newHero = Object.assign(oldHero, hero); // Demo: mutate cached hero
    return of(newHero).pipe(delay(this.delayMs)); // simulate latency with delay
  }
}
```

`HeroListComponent` 使用一个注入进来的 `HeroService` 来从服务器获取英雄列表，然后用一系列按钮把这些英雄展示给用户。 `HeroService` 模拟了 HTTP 服务。 它返回一个英雄组成的 `Observable` 对象，并会在短暂的延迟之后被解析出来，这是为了模拟网络延迟，并展示应用在自然延迟下的异步效果。

当用户点击一个英雄时，组件设置它的 `selectedHero` 属性，它绑定到 `HeroDetailComponent` 的 `@Input()` 属性 `hero` 上。 `HeroDetailComponent` 检测到英雄的变化，并使用当前英雄的值重置此表单。

"刷新"按钮会清除英雄列表和当前选中的英雄，然后重新获取英雄列表。

注意，`hero-list.component.ts` 从 `rxjs` 中导入了 `Observable` 和 `finalize`，而 `hero.service.ts` 导入了 `Observable`、`of` 和 `delay` 操作符。

`HeroListComponent` 和 `HeroService` 的其它实现细节超出了本教程的范围。 不过，它所涉及的技术包含在文档的其它部分，包括《英雄指南》的 这里 和 这里。

要使用 `HeroService`，就要把它导入到 `AppModule` 中，并添加到 `providers` 数组里。 要使用 `HeroListComponent`，就要导入它、声明它并导出它：

app.module.ts (excerpts)

```
// add JavaScript imports
import { HeroListComponent }   from './hero-list/hero-list.component';
import { HeroService }         from './hero.service';

@NgModule({
  declarations: [
    AppComponent,
    HeroDetailComponent,
    HeroListComponent // <--declare HeroListComponent
  ],
  // ...
  exports: [
    AppComponent,
    HeroDetailComponent,
    HeroListComponent // <-- export HeroListComponent
  ],
  providers: [ HeroService ], // <-- provide HeroService
```

接下来，把 `HeroListComponent` 的模板升级为：

hero-list.component.html

```
<h3 *ngIf="isLoading"><i>Loading heroes ... </i></h3>
<h3 *ngIf="!isLoading">Select a hero:</h3>

<nav>
  <button (click)="getHeroes()" class="btn btn-primary">Refresh</button>
  <a *ngFor="let hero of heroes | async" (click)="select(hero)">{{hero.name}}</a>
</nav>

<div *ngIf="selectedHero">
  <hr>
  <h2>Hero Detail</h2>
  <h3>Editing: {{selectedHero.name}}</h3>
  <app-hero-detail [hero]="selectedHero"></app-hero-detail>
</div>
```

这些修改需要反映到 `AppComponent` 模板中。把 `app.component.html` 替换为如下内容，以便把 `HeroDetailComponent` 替换为 `HeroListComponent`：

app.component.html

```
<div class="container">
  <h1>Reactive Forms</h1>
  <app-hero-list></app-hero-list>
</div>
```

最后，为 `HeroDetailComponent` 添加一个 `@Input()` 属性，让它能从 `HeroListComponent` 中接收数据。 别忘了也要把来自 `@angular/core` 的 `Input` 符号 `import` 进来。

hero-detail.component.ts (excerpt)

```
@Input() hero: Hero;
```

你先，你就可以点击一个按钮并渲染一个表单了。

何时设置表单模型的值（`ngOnChanges`）

何时设置表单模型的值取决于组件何时获得数据模型的值。

`HeroListComponent` 会给用户显示英雄的名字。 当用户点击某个英雄时，`HeroListComponent` 会通过绑定到 `hero` 这个输入属性，把选中的英雄传给 `HeroDetailComponent`。

hero-list.component.html (simplified)

```
<nav>
  <a *ngFor="let hero of heroes | async" (click)="select(hero)">{{hero.name}}</a>
</nav>

<div *ngIf="selectedHero">
  <app-hero-detail [hero]="selectedHero"></app-hero-detail>
</div>
```

这种方式下，每当用户选择一个新英雄时，`HeroDetailComponent` 的 `hero` 值就会发生变化。 你可以通过 ngOnChanges 生命周期钩子来调用 `setValue()`。只要 `hero` 这个输入属性发生了变化，Angular 就会调用这个钩子。

重置表单

首先，在 `hero-detail.component.ts` 中导入 `OnChanges` 符号。

src/app/hero-detail/hero-detail.component.ts (core imports)

```
import { Component, Input, OnChanges }             from '@angular/core';
```

接着，让 Angular 知道 `HeroDetailComponent` 实现了 `OnChanges`：

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
export class HeroDetailComponent implements OnChanges {
```

向该类中添加 `ngOnChanges` 方法，代码如下：

src/app/hero-detail/hero-detail.component.ts (ngOnchanges)

```
ngOnChanges() {
  this.rebuildForm();
}
```

注意，它调用了 `rebuildForm()`，该函数是一个方法，在这里你可以对值进行设置。 你可以把 `rebuildForm()` 命名为任何对你有意义的名字。 它不是 Angular 内置的，而是你自己创建的方法，用以更有效的利用 `ngOnChanges` 钩子。

src/app/hero-detail/hero-detail.component.ts

```
rebuildForm() {
  this.heroForm.reset({
    name: this.hero.name,
    address: this.hero.addresses[0] || new Address()
  });
}
```

`rebuildForm()` 方法会做两件事：重置英雄的名字和地址。

使用 `FormArray` 来表示 `FormGroup` 数组

`FormGroup` 是一个命名对象，它的属性值是 `FormControl` 和其它的 `FormGroup`。

有时你需要表示任意数量的控件或控件组。 比如，一个英雄可能拥有 0、1 或任意数量的住址。

`Hero.addresses` 属性就是一个 `Address` 实例的数组。 一个 `address` 的 `FormGroup` 可以显示一个 `Address` 对象。 而 `FormArray` 可以显示一个 `address` `FormGroup` 的数组。

要访问 `FormArray` 类，请先把它导入 `hero-detail.component.ts` 中：

src/app/hero-detail/hero-detail.component.ts (excerpt)

```
import { Component, Input, OnChanges }                 from '@angular/core';
import { FormArray, FormBuilder, FormGroup, Validators } from '@angular/forms';

import { Address, Hero, states } from '../data-model';
```

要使用 `FormArray`，就要这么做：

1. 在数组中定义条目 `FormControl` 或 `FormGroup`。
2. 把这个数组初始化微一组从**数据模型**中的数据创建的条目。
3. 根据用户的需求添加或移除这些条目。

为 `Hero.addresses` 定义了一个 `FormArray`，并且让用户添加或修改这些住址。

你需要在 `HeroDetailComponent` 的 `createForm()` 中重新定义表单模型，它现在只在 `address` `FormGroup` 中显示第一个英雄住址。

src/app/hero-detail/hero-detail.component.ts

```
this.heroForm = this.fb.group({
  name: ['', Validators.required ],
  address: this.fb.group(new Address()), // <-- a FormGroup with a new address
  power: '',
  sidekick: ''
});
```

从 `address`（住址）到 *`secretLairs`（秘密小屋）

在用户看来，英雄们没有**住址**。 只有我们凡人才有**住址**，英雄们拥有的是**秘密小屋**！ 把 `FormGroup` 型的住址替换为 `FormArray` 型的 `secretLairs` 定义：

src/app/hero-detail/hero-detail-8.component.ts

```
this.heroForm = this.fb.group({
  name: ['', Validators.required ],
  secretLairs: this.fb.array([]), // <-- secretLairs as an empty FormArray
  power: '',
  sidekick: ''
});
```

在 `hero-detail.component.html` 中 把 `formGroupName="address"` 改为 `formArrayName="secretLairs"`。

src/app/hero-detail/hero-detail.component.ts

```
<div formArrayName="secretLairs" class="well well-lg">
```

> 把表单的控件名从 `address` 改为 `secretLairs` 时导致了一个重要问题：**表单模型**与**数据模型**不再匹配了。
>
> 显然，必须在两者之间建立关联。但它在应用领域中的意义不限于此，它可以用于任何东西。
>
> **展现**的需求经常会与**数据**的需求不同。 响应式表单的方法既强调这种差异，也能为这种差异提供了便利。

初始化 `FormArray` 型的 `secretLairs`

默认的表单显示一个无地址的无名英雄。

你需要一个方法来用实际英雄的地址填充（或重新填充） `secretLairs`， 而不用管父组件 `HeroListComponent` 何时把 `@Input()` 属性 `HeroDetailComponent.hero` 设置为一个新的 `Hero`。

下面的 `setAddresses()` 方法把 `secretLairs` 这个 `FormArray` 替换为一个新的 `FormArray`，使用一组表示英雄地址的 `FormGroup` 来进行初始化。在 `HeroDetailComponent` 类中添加下列内容：

src/app/hero-detail/hero-detail.component.ts

```
setAddresses(addresses: Address[]) {
  const addressFGs = addresses.map(address => this.fb.group(address));
  const addressFormArray = this.fb.array(addressFGs);
  this.heroForm.setControl('secretLairs', addressFormArray);
}
```

注意，你使用 `FormGroup.setControl()` 方法，而不是 `setValue()` 方法来替换前一个 `FormArray`。 你所要替换的是**控件**，而不是控件的**值**。

还要注意， `secretLairs` 数组中包含的是**`FormGroup`，而不是 `Address`。

接着，在 `rebuildForm()` 中调用 `setAddresses()`：

src/app/hero-detail/hero-detail.component.ts

```
rebuildForm() {
  this.heroForm.reset({
    name: this.hero.name
  });
  this.setAddresses(this.hero.addresses);
}
```

获取 `FormArray`

`HeroDetailComponent` 应该能从 `secretLairs` `FormArray` 中显示、添加和删除条目。

使用 `FormGroup.get()` 方法来获取到 `FormArray` 的引用。 把这个表达式包装进一个名叫 `secretLairs` 的便捷属性中来让它更清晰，并供复用。 在 `HeroDetailComponent` 中添加下列内容。

src/app/hero-detail/hero-detail.component.ts (secretLairs property)

```
get secretLairs(): FormArray {
  return this.heroForm.get('secretLairs') as FormArray;
};
```

显示 `FormArray`

当前 HTML 模板显示单个的地址 `FormGroup`。 要把它修改成能显示 0、1 或更多个表示英雄地址的 `FormGroup`。

要改的部分主要是把以前表示地址的 HTML 模板包裹进一个 `<div>` 中，并且使用 `*ngFor` 来重复渲染这个 `<div>`。

写这个 `*ngFor` 有三个要点：

1. 在 `*ngFor` 的 `<div>` 之外套上另一个包装 `<div>`，并且把它的 `formArrayName` 指令设为 `"secretLairs"`。 这一步为内部的表单控件建立了一个 `FormArray` 型的 `secretLairs` 作为上下文，以便重复渲染 HTML 模板。
2. 这些重复条目的数据源是 `FormArray.controls` 而不是 `FormArray` 本身。 每个控件都是一个 `FormGroup` 型的地址对象，与以前的模板 HTML 所期望的格式完全一样。
3. 每个被重复渲染的 `FormGroup` 都需要一个独一无二的 `formGroupName`，它必须是 `FormGroup` 在这个 `FormArray` 中的索引。 你将复用这个索引，以便为每个地址组合出一个独一无二的标签。

下面是 HTML 模板中**秘密小屋**部分的代码骨架：

src/app/hero-detail/hero-detail.component.html (*ngFor)

```
<div formArrayName="secretLairs" class="well well-lg">
  <div *ngFor="let address of secretLairs.controls; let i=index"
[formGroupName]="i" >
    <!-- The repeated address template -->
  </div>
</div>
```

这里是**秘密小屋**部分的完整模板：

src/app/hero-detail/hero-detail.component.html (excerpt)

```
1. <div formArrayName="secretLairs" class="well well-lg">
2.   <div *ngFor="let address of secretLairs.controls; let i=index"
   [formGroupName]="i" >
3.     <!-- The repeated address template -->
4.     <h4>Address #{{i + 1}}</h4>
5.     <div style="margin-left: 1em;">
6.       <div class="form-group">
7.         <label class="center-block">Street:
8.           <input class="form-control" formControlName="street">
9.         </label>
10.       </div>
11.       <div class="form-group">
12.         <label class="center-block">City:
13.           <input class="form-control" formControlName="city">
14.         </label>
15.       </div>
16.       <div class="form-group">
17.         <label class="center-block">State:
18.           <select class="form-control" formControlName="state">
19.             <option *ngFor="let state of states" [value]="state">{{state}}
   </option>
20.           </select>
21.         </label>
22.       </div>
23.       <div class="form-group">
24.         <label class="center-block">Zip Code:
25.           <input class="form-control" formControlName="zip">
26.         </label>
27.       </div>
28.     </div>
29.     <br>
30.     <!-- End of the repeated address template -->
31.   </div>
32. </div>
```

把新的小屋添加到 `FormArray` 中

添加一个 `addLair()` 方法，它获取 `secretLairs` 数组，并把新的表示地址的 `FormGroup` 添加到其中。

src/app/hero-detail/hero-detail.component.ts (addLair method)

```
addLair() {
  this.secretLairs.push(this.fb.group(new Address()));
}
```

把一个按钮放在表单中，以便用户可以添加新的**秘密小屋**，并把它传给组件的 `addLair()` 方法。

src/app/hero-detail/hero-detail.component.html (addLair button)

```
<button (click)="addLair()" type="button">Add a Secret Lair</button>
```

> 务必确保添加了 `type="button"` 属性。 因为如果不明确指定类型，按钮的默认类型就是“submit”（提交）。 当你稍后添加了提交表单的动作时，每个“submit”按钮都是触发一次提交操作，而它将可能会做一些处理，比如保存当前的修改。 你显然不会希望每当用户点击“Add a Secret Lair”按钮时就保存一次。

试试看!

回到浏览器中，选择名叫“Magneta”的英雄。 "Magneta"没有地址，你会在表单底部的诊断用 JSON 中看到这一点。

heroForm value: { "name": "Magneta", "secretLairs": [] }

点击“Add a Secret Lair”按钮，一个新的地址区就出现了，干得好!

移除一个小屋

这个例子可以**添加**地址，但是还不能**移除**它们。 作为练习，你可以自己写一个 `removeLair` 方法，并且把它关联到地址 HTML 模板的一个按钮上。

监视控件的变化

每当用户在父组件 `HeroListComponent` 中选取了一个英雄，Angular 就会调用一次 `ngOnChanges`。 选取英雄会修改输入属性 `HeroDetailComponent.hero()`。

当用户修改英雄的**名字**或**秘密小屋**时，Angular**并不会**调用 `ngOnChanges()`。 幸运的是，你可以通过订阅表单控件的属性之一来了解这些变化，此属性会发出变更通知。

有一些属性，比如 `valueChanges`，可以返回一个 RxJS 的 `Observable` 对象。 要监听控件值的变化，你并不需要对 RxJS 的 `Observable` 了解更多。

添加下列方法，以监听 `name` 这个 `FormControl` 中值的变化。

src/app/hero-detail/hero-detail.component.ts (logNameChange)

```
nameChangeLog: string[] = [];
logNameChange() {
  const nameControl = this.heroForm.get('name');
  nameControl.valueChanges.forEach(
    (value: string) => this.nameChangeLog.push(value)
  );
}
```

在构造函数中调用它，就在 `createForm()` 之后。

src/app/hero-detail/hero-detail.component.ts

```
constructor(private fb: FormBuilder) {
  this.createForm();
  this.logNameChange();
}
```

`logNameChange()` 方法会把一条改名记录追加到 `nameChangeLog` 数组中。 用 `*ngFor` 绑定在组件模板的底部显示这个数组：

src/app/hero-detail/hero-detail.component.html (Name change log)

```
<h4>Name change log</h4>
<div *ngFor="let name of nameChangeLog">{{name}}</div>
```

返回浏览器，选择一个英雄（比如“Magneta”），并开始在 `name` 输入框中键入。 你会看到，每次按键都会记录一个新名字。

什么时候用它

插值表达式绑定时显示姓名变化比较简单的方式。 在组件类中订阅表单控件属性变化的可观察对象以触发应用逻辑则是比较难的方式。

保存表单数据

`HeroDetailComponent` 捕获了用户输入，但没有用它做任何事。 在真实的应用中，你可能要保存这些英雄的变化。 在真实的应用中，你还要能丢弃未保存的变更，然后继续编辑。 在实现完本节的这些特性之后，表单是这样的：

保存

当用户提交表单时， `HeroDetailComponent` 会把英雄实例的**数据模型**传给所注入进来的 `HeroService` 的一个方法来进行保存。 在 `HeroDetailComponent` 中添加如下内容：

src/app/hero-detail/hero-detail.component.ts (onSubmit)

```
onSubmit() {
  this.hero = this.prepareSaveHero();
  this.heroService.updateHero(this.hero).subscribe(/* error handling */);
  this.rebuildForm();
}
```

原始的 `hero` 中有一些保存之前的值，用户的修改仍然是在**表单模型**中。 所以你要根据原始英雄（根据 `hero.id` 找到它）的值组合出一个新的 `hero` 对象，并用 `prepareSaveHero()` 助手来深层复制变化后的模型值。

src/app/hero-detail/hero-detail.component.ts (prepareSaveHero)

```
prepareSaveHero(): Hero {
  const formModel = this.heroForm.value;

  // deep copy of form model lairs
  const secretLairsDeepCopy: Address[] = formModel.secretLairs.map(
    (address: Address) => Object.assign({}, address)
  );

  // return new `Hero` object containing a combination of original hero value(s)
  // and deep copies of changed form model values
  const saveHero: Hero = {
    id: this.hero.id,
    name: formModel.name as string,
    // addresses: formModel.secretLairs // <-- bad!
    addresses: secretLairsDeepCopy
  };
  return saveHero;
}
```

确保导入了 `HeroService` 并把它加入了构造函数中：

src/app/hero-detail/hero-detail.component.ts (prepareSaveHero)

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

src/app/hero-detail/hero-detail.component.ts (prepareSaveHero)

```
constructor(
  private fb: FormBuilder,
  private heroService: HeroService) {

  this.createForm();
  this.logNameChange();
}
```

> 地址的深层复制
>
> 你已经把 `formModel.secretLairs` 赋值给了 `saveHero.addresses`（参见注释掉的部分），`saveHero.addresses` 数组中的地址和 `formModel.secretLairs` 中的会是同一个对象。 用户随后对小屋

> 所在街道的修改将会改变 `saveHero` 中的街道地址。
>
> 但 `prepareSaveHero` 方法会制作表单模型中的 `secretLairs` 对象的复本，因此实际上并没有修改原有对象。

丢弃（撤销修改）

用户可以撤销修改，并通过点击 **Revert** 按钮来把表单恢复到原始状态。

丢弃很容易。只要重新执行 `rebuildForm()` 方法就可以根据原始的、未修改过的 `hero` 数据模型重新构建出表单模型。

src/app/hero-detail/hero-detail.component.ts (revert)

```
revert() { this.rebuildForm(); }
```

按钮

把“Save”和“Revert”按钮添加到组件模板的顶部：

src/app/hero-detail/hero-detail.component.html (Save and Revert buttons)

```
<form [formGroup]="heroForm" (ngSubmit)="onSubmit()">
  <div style="margin-bottom: 1em">
    <button type="submit"
            [disabled]="heroForm.pristine" class="btn btn-success">Save</button>
 
    <button type="button" (click)="revert()"
            [disabled]="heroForm.pristine" class="btn btn-danger">Revert</button>
  </div>

  <!-- Hero Detail Controls -->
  <div class="form-group radio">
    <h4>Super power:</h4>
    <label class="center-block"><input type="radio" formControlName="power"
value="flight">Flight</label>
    <label class="center-block"><input type="radio" formControlName="power"
value="x-ray vision">X-ray vision</label>
    <label class="center-block"><input type="radio" formControlName="power"
value="strength">Strength</label>
  </div>
  <div class="checkbox">
    <label class="center-block">
      <input type="checkbox" formControlName="sidekick">I have a sidekick.
    </label>
  </div>
</form>
```

这些按钮默认是禁用的，直到用户通过修改任何一个表单控件的值“弄脏”了表单中的数据（即 `heroForm.dirty`）。

点击一个类型为 `"submit"` 的按钮会触发 `ngSubmit` 事件，而它会调用组件的 `onSubmit` 方法。 点击“Revert”按钮则会调用组件的 `revert` 方法。 现在，用户可以保存或放弃修改了。

试试 Reactive Forms (final) in Stackblitz / 下载范例.

最终版中的核心文件如下：

src/app/app.component.html | src/app/app.component.ts | src/app/app.mo

```
<div class="container">
  <h1>Reactive Forms</h1>
```

```
    <app-hero-list></app-hero-list>
  </div>
```

你可以到Reactive Forms Demo in Stackblitz / 下载范例中下载本章所有步骤的完整代码。

动态表单

有时候手动编写和维护表单所需工作量和时间会过大。特别是在需要编写大量表单时。表单都很相似，而且随着业务和监管需求的迅速变化，表单也要随之变化，这样维护的成本过高。

基于业务对象模型的元数据，动态创建表单可能会更划算。

本文会展示如何利用 `formGroup` 来动态渲染一个简单的表单，包括各种控件类型和验证规则。 这个起点很简陋，但可以在这个基础上添加丰富多彩的问卷问题、更优美的渲染以及更卓越的用户体验。

这个例子要为正在找工作的英雄们创建一个在线申请表的动态表单。英雄管理局会不断修改申请流程，你要在**不修改应用代码**的情况下，动态创建这些表单。

参见在线例子 / 下载范例。

启动/引导 (bootstrap)

从创建一个名叫 `AppModule` 的 `NgModule` 开始。

这个烹饪书使用响应式表单。

响应式表单属于另外一个叫做 `ReactiveFormsModule` 的 `NgModule`，所以，为了使用响应式表单类的指令，你得从 `@angular/forms` 库中引入 `ReactiveFormsModule` 模块。

在 `main.ts` 中启动 `AppModule`。

app.module.ts *main.ts*

```
1. import { BrowserModule }                from '@angular/platform-browser';
2. import { ReactiveFormsModule }          from '@angular/forms';
3. import { NgModule }                     from '@angular/core';
4.
5. import { AppComponent }                 from './app.component';
6. import { DynamicFormComponent }         from './dynamic-form.component';
7. import { DynamicFormQuestionComponent } from './dynamic-form-
   question.component';
8.
9. @NgModule({
10.   imports: [ BrowserModule, ReactiveFormsModule ],
```

```
11.   declarations: [ AppComponent, DynamicFormComponent,
    DynamicFormQuestionComponent ],
12.   bootstrap: [ AppComponent ]
13. })
14. export class AppModule {
15.   constructor() {
16.   }
17. }
```

问卷问题模型

第一步是定义一个对象模型，用来描述所有表单功能需要的场景。英雄的申请流程涉及到一个包含很多问卷问题的表单。问卷问题是最基础的对象模型。

下面的 `QuestionBase` 是最基础的问卷问题基类。

src/app/question-base.ts

```
export class QuestionBase<T> {
  value: T;
  key: string;
  label: string;
  required: boolean;
  order: number;
  controlType: string;

  constructor(options: {
      value?: T,
      key?: string,
      label?: string,
      required?: boolean,
      order?: number,
      controlType?: string
    } = {}) {
    this.value = options.value;
    this.key = options.key || '';
    this.label = options.label || '';
    this.required = !!options.required;
    this.order = options.order === undefined ? 1 : options.order;
    this.controlType = options.controlType || '';
  }
}
```

在这个基础上，你派生出两个新类 `TextboxQuestion` 和 `DropdownQuestion`，分别代表文本框和下拉框。这么做的初衷是，表单能动态绑定到特定的问卷问题类型，并动态渲染出合适的控件。

`TextboxQuestion` 可以通过 `type` 属性来支持多种 HTML5 元素类型，比如文本、邮件、网址等。

src/app/question-textbox.ts

```
import { QuestionBase } from './question-base';

export class TextboxQuestion extends QuestionBase<string> {
  controlType = 'textbox';
  type: string;

  constructor(options: {} = {}) {
    super(options);
    this.type = options['type'] || '';
  }
}
```

`DropdownQuestion` 表示一个带可选项列表的选择框。

src/app/question-dropdown.ts

```
import { QuestionBase } from './question-base';

export class DropdownQuestion extends QuestionBase<string> {
  controlType = 'dropdown';
  options: {key: string, value: string}[] = [];

  constructor(options: {} = {}) {
    super(options);
    this.options = options['options'] || [];
  }
}
```

接下来定义了 `QuestionControlService`，一个可以把问卷问题转换为 `FormGroup` 的服务。 简而言之，这个 `FormGroup` 使用问卷模型的元数据，并允许你指定默认值和验证规则。

src/app/question-control.service.ts

```
import { Injectable }   from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';

import { QuestionBase } from './question-base';

@Injectable()
export class QuestionControlService {
  constructor() { }

  toFormGroup(questions: QuestionBase<any>[] ) {
    let group: any = {};

    questions.forEach(question => {
      group[question.key] = question.required ? new FormControl(question.value ||
'', Validators.required)
                                              : new FormControl(question.value ||
'');
    });
    return new FormGroup(group);
  }
}
```

问卷表单组件

现在你已经有一个定义好的完整模型了，接着就可以开始创建一个展现动态表单的组件。

`DynamicFormComponent` 是表单的主要容器和入口点。

dynamic-form.component.html　　*dynamic-form.component.ts*

```
1. <div>
2.   <form (ngSubmit)="onSubmit()" [formGroup]="form">
3.
4.     <div *ngFor="let question of questions" class="form-row">
5.       <app-question [question]="question" [form]="form"></app-question>
6.     </div>
7.
8.     <div class="form-row">
9.       <button type="submit" [disabled]="!form.valid">Save</button>
```

```
10.     </div>
11.   </form>
12.
13.   <div *ngIf="payLoad" class="form-row">
14.     <strong>Saved the following values</strong><br>{{payLoad}}
15.   </div>
16. </div>
```

它代表了问卷问题列表，每个问题都被绑定到一个 `<app-question>` 组件元素。 `<app-question>` 标签匹配到的是组件 `DynamicFormQuestionComponent`，该组件的职责是根据各个问卷问题对象的值来动态渲染表单控件。

dynamic-form-question.component.html *dynamic-form-question.component.ts*

```
1. <div [formGroup]="form">
2.   <label [attr.for]="question.key">{{question.label}}</label>
3.
4.   <div [ngSwitch]="question.controlType">
5.
6.     <input *ngSwitchCase="'textbox'" [formControlName]="question.key"
7.             [id]="question.key" [type]="question.type">
8.
9.     <select [id]="question.key" *ngSwitchCase="'dropdown'"
   [formControlName]="question.key">
10.       <option *ngFor="let opt of question.options" [value]="opt.key">
   {{opt.value}}</option>
11.     </select>
12.
13.   </div>
14.
15.   <div class="errorMessage" *ngIf="!isValid">{{question.label}} is
   required</div>
16. </div>
```

请注意，这个组件能代表模型里的任何问题类型。目前，还只有两种问题类型，但可以添加更多类型。可以用 `ngSwitch` 决定显示哪种类型的问题。

在这两个组件中，你依赖 Angular 的 formGroup 来把模板 HTML 和底层控件对象连接起来，该对象从问卷问题模型里获取渲染和验证规则。

`formControlName` 和 `formGroup` 是在 `ReactiveFormsModule` 中定义的指令。这个模板之所以能使用它们，是因为你曾从 `AppModule` 中导入了 `ReactiveFormsModule`。

问卷数据

`DynamicForm` 期望得到一个问题列表，该列表被绑定到 `@Input() questions` 属性。

`QuestionService` 会返回为工作申请表定义的那组问题列表。在真实的应用程序环境中，你会从数据库里获得这些问题列表。

关键是，你完全根据 `QuestionService` 返回的对象来控制英雄的工作申请表。 要维护这份问卷，只要非常简单的添加、修改和删除 `questions` 数组中的对象就可以了。

src/app/question.service.ts

```
1. import { Injectable }       from '@angular/core';
2.
3. import { DropdownQuestion } from './question-dropdown';
4. import { QuestionBase }     from './question-base';
5. import { TextboxQuestion }  from './question-textbox';
6.
7. @Injectable()
8. export class QuestionService {
9.
10.   // TODO: get from a remote source of question metadata
11.   // TODO: make asynchronous
12.   getQuestions() {
13.
14.     let questions: QuestionBase<any>[] = [
15.
16.       new DropdownQuestion({
17.         key: 'brave',
18.         label: 'Bravery Rating',
19.         options: [
20.           {key: 'solid',  value: 'Solid'},
21.           {key: 'great',  value: 'Great'},
22.           {key: 'good',   value: 'Good'},
23.           {key: 'unproven', value: 'Unproven'}
24.         ],
25.         order: 3
26.       }),
27.
28.       new TextboxQuestion({
29.         key: 'firstName',
30.         label: 'First name',
31.         value: 'Bombasto',
32.         required: true,
33.         order: 1
34.       }),
35.
36.       new TextboxQuestion({
37.         key: 'emailAddress',
38.         label: 'Email',
39.         type: 'email',
40.         order: 2
41.       })
```

```
42.     ];
43.
44.     return questions.sort((a, b) => a.order - b.order);
45.   }
46. }
```

最后，在 `AppComponent` 里显示出表单。

app.component.ts

```
1. import { Component }       from '@angular/core';
2.
3. import { QuestionService } from './question.service';
4.
5. @Component({
6.   selector: 'app-root',
7.   template: `
8.     <div>
9.       <h2>Job Application for Heroes</h2>
10.      <app-dynamic-form [questions]="questions"></app-dynamic-form>
11.    </div>
12.  `,
13.  providers:  [QuestionService]
14. })
15. export class AppComponent {
16.   questions: any[];
17.
18.   constructor(service: QuestionService) {
19.     this.questions = service.getQuestions();
20.   }
21. }
```

动态模板

在这个例子中，虽然你是在为英雄的工作申请表建模，但是除了 `QuestionService` 返回的那些对象外，没有其它任何地方是与英雄有关的。

这点非常重要，因为只要与**问卷**对象模型兼容，就可以在任何类型的调查问卷中复用这些组件。 这里的关键是用到元数据的动态数据绑定来渲染表单，对问卷问题没有任何硬性的假设。除控件的元数据外，还可以动态添加验证规则。

表单验证通过之前，**保存**按钮是禁用的。验证通过后，就可以点击**保存**按钮，程序会把当前值渲染成 JSON 显示出来。 这表明任何用户输入都被传到了数据模型里。至于如何储存和提取数据则是另一话题了。

完整的表单是这样的：

Job Application for Heroes

First name

Bombasto

Email

Bravery Rating

Solid

Save

回到顶部

可观察对象（Observable）

可观察对象支持在应用中的发布者和订阅者之间传递消息。 在需要进行事件处理、异步编程和处理多个值的时候，可观察对象相对其它技术有着显著的优点。

可观察对象是声明式的 —— 也就是说，虽然你定义了一个用于发布值的函数，但是在有消费者订阅它之前，这个函数并不会实际执行。 订阅之后，当这个函数执行完或取消订阅时，订阅者就会收到通知。

可观察对象可以发送多个任意类型的值 —— 字面量、消息、事件。无论这些值是同步发送的还是异步发送的，接收这些值的 API 都是一样的。 由于准备（setup）和清场（teardown）的逻辑都是由可观察对象自己处理的，因此你的应用代码只管订阅并消费这些值就可以了，做完之后，取消订阅。无论这个流是击键流、HTTP 响应流还是定时器，对这些值进行监听和停止监听的接口都是一样的。

由于这些优点，可观察对象在 Angular 中得到广泛使用，也同样建议应用开发者好好使用它。

基本用法和词汇

作为发布者，你创建一个 `Observable` 的实例，其中定义了一个**订阅者（subscriber）**函数。 当有消费者调用 `subscribe()` 方法时，这个函数就会执行。 订阅者函数用于定义“如何获取或生成那些要发布的值或消息”。

要执行所创建的可观察对象，并开始从中接收通知，你就要调用它的 `subscribe()` 方法，并传入一个**观察者（observer）**。 这是一个 JavaScript 对象，它定义了你收到的这些消息的处理器（handler）。 `subscribe()` 调用会返回一个 `Subscription` 对象，该对象具有一个 `unsubscribe()` 方法。 当调用该方法时，你就会停止接收通知。

下面这个例子中示范了这种基本用法，它展示了如何使用可观察对象来对当前地理位置进行更新。

Observe geolocation updates

```
// Create an Observable that will start listening to geolocation updates
// when a consumer subscribes.
const locations = new Observable((observer) => {
  // Get the next and error callbacks. These will be passed in when
  // the consumer subscribes.
  const {next, error} = observer;
  let watchId;

  // Simple geolocation API check provides values to publish
  if ('geolocation' in navigator) {
    watchId = navigator.geolocation.watchPosition(next, error);
  } else {
    error('Geolocation not available');
  }

  // When the consumer unsubscribes, clean up data ready for next subscription.
  return {unsubscribe() { navigator.geolocation.clearWatch(watchId); }};
});

// Call subscribe() to start listening for updates.
const locationsSubscription = locations.subscribe({
  next(position) { console.log('Current Position: ', position); },
  error(msg) { console.log('Error Getting Location: ', msg); }
});

// Stop listening for location after 10 seconds
setTimeout(() => { locationsSubscription.unsubscribe(); }, 10000);
```

定义观察者

用于接收可观察对象通知的处理器要实现 `Observer` 接口。这个对象定义了一些回调函数来处理可观察对象可能会发来的三种通知：

通知类型	说明
`next`	必要。用来处理每个送达值。在开始执行后可能执行零次或多次。
`error`	可选。用来处理错误通知。错误会中断这个可观察对象实例的执行过程。
`complete`	可选。用来处理执行完毕（complete）通知。当执行完毕后，这些值就会继续传给下一个处理器。

观察者对象可以定义这三种处理器的任意组合。如果你不为某种通知类型提供处理器，这个观察者就会忽略相应类型的通知。

订阅

只有当有人订阅 `Observable` 的实例时，它才会开始发布值。 订阅时要先调用该实例的 `subscribe()` 方法，并把一个观察者对象传给它，用来接收通知。

> 为了展示订阅的原理，我们需要创建新的可观察对象。它有一个构造函数可以用来创建新实例，但是为了更简明，也可以使用 `Observable` 上定义的一些静态方法来创建一些常用的简单可观察对象：
>
> - `Observable.of(...items)` —— 返回一个 `Observable` 实例，它用同步的方式把参数中提供的这些值发送出来。
> - `Observable.from(iterable)` —— 把它的参数转换成一个 `Observable` 实例。 该方法通常用于把一个数组转换成一个（发送多个值的）可观察对象。

下面的例子会创建并订阅一个简单的可观察对象，它的观察者会把接收到的消息记录到控制台中：

Subscribe using observer

```
// Create simple observable that emits three values
const myObservable = Observable.of(1, 2, 3);

// Create observer object
const myObserver = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};

// Execute with the observer object
myObservable.subscribe(myObserver);
// Logs:
// Observer got a next value: 1
// Observer got a next value: 2
// Observer got a next value: 3
// Observer got a complete notification
```

另外，`subscribe()` 方法还可以接收定义在同一行中的回调函数，无论 `next`、`error` 还是 `complete` 处理器。比如，下面的 `subscribe()` 调用和前面指定预定义观察者的例子是等价的。

Subscribe with positional arguments

```
myObservable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);
```

无论哪种情况，`next` 处理器都是必要的，而 `error` 和 `complete` 处理器是可选的。

注意，`next()` 函数可以接受消息字符串、事件对象、数字值或各种结构，具体类型取决于上下文。 为了更通用一点，我们把由可观察对象发布出来的数据统称为**流**。任何类型的值都可以表示为可观察对象，而这些值会被发布为一个流。

创建可观察对象

使用 `Observable` 构造函数可以创建任何类型的可观察流。 当执行可观察对象的 `subscribe()` 方法时，这个构造函数就会把它接收到的参数作为订阅函数来运行。 订阅函数会接收一个 `Observer` 对象，并把值发布给观察者的 `next()` 方法。

比如，要创建一个与前面的 `Observable.of(1, 2, 3)` 等价的可观察对象，你可以这样做：

Create observable with constructor

```
// This function runs when subscribe() is called
function sequenceSubscriber(observer) {
  // synchronously deliver 1, 2, and 3, then complete
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();

  // unsubscribe function doesn't need to do anything in this
  // because values are delivered synchronously
  return {unsubscribe() {}};
}

// Create a new Observable that will deliver the above sequence
const sequence = new Observable(sequenceSubscriber);

// execute the Observable and print the result of each notification
sequence.subscribe({
  next(num) { console.log(num); },
  complete() { console.log('Finished sequence'); }
});

// Logs:
// 1
// 2
// 3
// Finished sequence
```

如果要略微加强这个例子，我们可以创建一个用来发布事件的可观察对象。在这个例子中，订阅函数是用内联方式定义的。

Create with custom fromEvent function

```
1. function fromEvent(target, eventName) {
2.   return new Observable((observer) => {
3.     const handler = (e) => observer.next(e);
4.
5.     // Add the event handler to the target
6.     target.addEventListener(eventName, handler);
7.
8.     return () => {
9.       // Detach the event handler from the target
10.       target.removeEventListener(eventName, handler);
11.     };
12.   });
13. }
```

现在，你就可以使用这个函数来创建可发布 `keydown` 事件的可观察对象了：

Use custom fromEvent function

```
const ESC_KEY = 27;
const nameInput = document.getElementById('name') as HTMLInputElement;

const subscription = fromEvent(nameInput, 'keydown')
  .subscribe((e: KeyboardEvent) => {
    if (e.keyCode === ESC_KEY) {
      nameInput.value = '';
    }
  });
```

多播

典型的可观察对象会为每一个观察者创建一次新的、独立的执行。 当观察者进行订阅时，该可观察对象会连上一个事件处理器，并且向那个观察者发送一些值。当第二个观察者订阅时，这个可观察对象就会连上一个新的事件处理器，并独立执行一次，把这些值发送给第二个可观察对象。

有时候，不应该对每一个订阅者都独立执行一次，你可能会希望每次订阅都得到同一批值 —— 即使是那些你已经发送过的。这在某些情况下有用，比如用来发送 `document` 上的点击事件的可观察对象。

多播用来让可观察对象在一次执行中同时广播给多个订阅者。借助支持多播的可观察对象，你不必注册多个监听器，而是复用第一个（`next`）监听器，并且把值发送给各个订阅者。

当创建可观察对象时，你要决定你希望别人怎么用这个对象以及是否对它的值进行多播。

来看一个从 1 到 3 进行计数的例子，它每发出一个数字就会等待 1 秒。

Create a delayed sequence

```
function sequenceSubscriber(observer) {
  const seq = [1, 2, 3];
  let timeoutId;

  // Will run through an array of numbers, emitting one value
  // per second until it gets to the end of the array.
  function doSequence(arr, idx) {
    timeoutId = setTimeout(() => {
      observer.next(arr[idx]);
      if (idx === arr.length - 1) {
        observer.complete();
      } else {
        doSequence(arr, idx++);
      }
    }, 1000);
  }

  doSequence(seq, 0);

  // Unsubscribe should clear the timeout to stop execution
  return {unsubscribe() {
    clearTimeout(timeoutId);
  }};
}

// Create a new Observable that will deliver the above sequence
const sequence = new Observable(sequenceSubscriber);

sequence.subscribe({
  next(num) { console.log(num); },
  complete() { console.log('Finished sequence'); }
});

// Logs:
// (at 1 second): 1
// (at 2 seconds): 2
// (at 3 seconds): 3
// (at 3 seconds): Finished sequence
```

注意，如果你订阅了两次，就会有两个独立的流，每个流都会每秒发出一个数字。代码如下：

Two subscriptions

```
// Subscribe starts the clock, and will emit after 1 second
sequence.subscribe({
  next(num) { console.log('1st subscribe: ' + num); },
  complete() { console.log('1st sequence finished.'); }
});

// After 1/2 second, subscribe again.
setTimeout(() => {
  sequence.subscribe({
    next(num) { console.log('2nd subscribe: ' + num); },
    complete() { console.log('2nd sequence finished.'); }
  });
}, 500);

// Logs:
// (at 1 second): 1st subscribe: 1
// (at 1.5 seconds): 2nd subscribe: 1
// (at 2 seconds): 1st subscribe: 2
// (at 2.5 seconds): 2nd subscribe: 2
// (at 3 seconds): 1st subscribe: 3
// (at 3 seconds): 1st sequence finished
// (at 3.5 seconds): 2nd subscribe: 3
// (at 3.5 seconds): 2nd sequence finished
```

修改这个可观察对象以支持多播，代码如下：

Create a multicast subscriber

```
function multicastSequenceSubscriber() {
  const seq = [1, 2, 3];
  // Keep track of each observer (one for every active subscription)
  const observers = [];
  // Still a single timeoutId because there will only ever be one
  // set of values being generated, multicasted to each subscriber
  let timeoutId;

  // Return the subscriber function (runs when subscribe()
  // function is invoked)
  return (observer) => {
    observers.push(observer);
    // When this is the first subscription, start the sequence
    if (observers.length === 1) {
      timeoutId = doSequence({
        next(val) {
          // Iterate through observers and notify all subscriptions
          observers.forEach(obs => obs.next(val));
        },
        complete() {
          // Notify all complete callbacks
          observers.forEach(obs => obs.complete());
        }
      }, seq, 0);
    }

    return {
      unsubscribe() {
        // Remove from the observers array so it's no longer notified
        observers.splice(observers.indexOf(observer), 1);
        // If there's no more listeners, do cleanup
        if (observers.length === 0) {
          clearTimeout(timeoutId);
        }
      }
    };
  };
}

// Run through an array of numbers, emitting one value
// per second until it gets to the end of the array.
```

```
42. function doSequence(observer, arr, idx) {
43.   return setTimeout(() => {
44.     observer.next(arr[idx]);
45.     if (idx === arr.length - 1) {
46.       observer.complete();
47.     } else {
48.       doSequence(observer, arr, idx++);
49.     }
50.   }, 1000);
51. }
52.
53. // Create a new Observable that will deliver the above sequence
54. const multicastSequence = new Observable(multicastSequenceSubscriber);
55.
56. // Subscribe starts the clock, and begins to emit after 1 second
57. multicastSequence.subscribe({
58.   next(num) { console.log('1st subscribe: ' + num); },
59.   complete() { console.log('1st sequence finished.'); }
60. });
61.
62. // After 1 1/2 seconds, subscribe again (should "miss" the first value).
63. setTimeout(() => {
64.   multicastSequence.subscribe({
65.     next(num) { console.log('2nd subscribe: ' + num); },
66.     complete() { console.log('2nd sequence finished.'); }
67.   });
68. }, 1500);
69.
70. // Logs:
71. // (at 1 second): 1st subscribe: 1
72. // (at 2 seconds): 1st subscribe: 2
73. // (at 2 seconds): 2nd subscribe: 2
74. // (at 3 seconds): 1st subscribe: 3
75. // (at 3 seconds): 1st sequence finished
76. // (at 3 seconds): 2nd subscribe: 3
77. // (at 3 seconds): 2nd sequence finished
```

> 虽然支持多播的可观察对象需要做更多的准备工作，但对某些应用来说，这非常有用。稍后我们会介绍一些简化多播的工具，它们让你能接收任何可观察对象，并把它变成支持多播的。

错误处理

由于可观察对象会异步生成值，所以用 `try/catch` 是无法捕获错误的。你应该在观察者中指定一个 `error` 回调来处理错误。发生错误时还会导致可观察对象清理现有的订阅，并且停止生成值。可观察对象可以生成值（调用 `next` 回调），也可以调用 `complete` 或 `error` 回调来主动结束。

```
myObservable.subscribe({
  next(num) { console.log('Next num: ' + num)},
  error(err) { console.log('Received an errror: ' + err)}
});
```

在稍后的小节中会对错误处理（特别是从错误中的恢复）做更详细的讲解。

RxJS 库

响应式编程是一种面向数据流和变更传播的异步编程范式（Wikipedia）。RxJS（响应式扩展的 JavaScript 版）是一个使用可观察对象进行响应式编程的库，它让组合异步代码和基于回调的代码变得更简单 (RxJS Docs)。

RxJS 提供了一种对 `Observable` 类型的实现，直到 `Observable` 成为了 JavaScript 语言的一部分并且浏览器支持它之前，它都是必要的。这个库还提供了一些工具函数，用于创建和使用可观察对象。这些工具函数可用于：

- 把现有的异步代码转换成可观察对象
- 迭代流中的各个值
- 把这些值映射成其它类型
- 对流进行过滤
- 组合多个流

创建可观察对象的函数

RxJS 提供了一些用来创建可观察对象的函数。这些函数可以简化根据某些东西创建可观察对象的过程，比如事件、定时器、承诺等等。比如：

Create an observable from a promise

```
import { fromPromise } from 'rxjs';

// Create an Observable out of a promise
const data = fromPromise(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe({
 next(response) { console.log(response); },
 error(err) { console.error('Error: ' + err); },
 complete() { console.log('Completed'); }
});
```

Create an observable from a counter

```
import { interval } from 'rxjs';

// Create an Observable that will publish a value on an interval
const secondsCounter = interval(1000);
// Subscribe to begin publishing values
secondsCounter.subscribe(n =>
  console.log(`It's been ${n} seconds since subscribing!`));
```

Create an observable from an event

```
1. import { fromEvent } from 'rxjs';
2.
3. const el = document.getElementById('my-element');
4.
5. // Create an Observable that will publish mouse movements
6. const mouseMoves = fromEvent(el, 'mousemove');
7.
8. // Subscribe to start listening for mouse-move events
9. const subscription = mouseMoves.subscribe((evt: MouseEvent) => {
10.   // Log coords of mouse movements
11.   console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);
12.
13.   // When the mouse is over the upper-left of the screen,
14.   // unsubscribe to stop listening for mouse movements
15.   if (evt.clientX < 40 && evt.clientY < 40) {
16.     subscription.unsubscribe();
17.   }
18. });
```

Create an observable that creates an AJAX request

```
import { ajax } from 'rxjs/ajax';

// Create an Observable that will create an AJAX request
const apiData = ajax('/api/data');
// Subscribe to create the request
apiData.subscribe(res => console.log(res.status, res.response));
```

操作符

操作符是基于可观察对象构建的一些对集合进行复杂操作的函数。RxJS 定义了一些操作符，比如 `map()`、`filter()`、`concat()` 和 `flatMap()`。

操作符接受一些配置项，然后返回一个以来源可观察对象为参数的函数。当执行这个返回的函数时，这个操作符会观察来源可观察对象中发出的值，转换它们，并返回由转换后的值组成的新的可观察对象。下面是一个简单的例子：

Map operator

```
import { map } from 'rxjs/operators';

const nums = of(1, 2, 3);

const squareValues = map((val: number) => val * val);
const squaredNums = squareValues(nums);

squaredNums.subscribe(x => console.log(x));

// Logs
// 1
// 4
// 9
```

你可以使用**管道**来把这些操作符链接起来。管道让你可以把多个由操作符返回的函数组合成一个。`pipe()` 函数以你要组合的这些函数作为参数，并且返回一个新的函数，当执行这个新函数时，就会顺序执行那些被组合进去的函数。

应用于某个可观察对象上的一组操作符就像一个菜谱 —— 也就是说，对你感兴趣的这些值进行处理的一组操作步骤。这个菜谱本身不会做任何事。你需要调用 `subscribe()` 来通过这个菜谱生成一个结果。

例子如下：

Standalone pipe function

```
import { filter, map } from 'rxjs/operators';

const nums = of(1, 2, 3, 4, 5);

// Create a function that accepts an Observable.
const squareOddVals = pipe(
  filter(n => n % 2),
  map(n => n * n)
);

// Create an Observable that will run the filter and map functions
const squareOdd = squareOddVals(nums);

// Suscribe to run the combined functions
squareOdd.subscribe(x => console.log(x));
```

`pipe()` 函数也同时是 RxJS 的 `Observable` 上的一个方法，所以你可以用下列简写形式来达到同样的效果：

Observable.pipe function

```
import { filter, map } from 'rxjs/operators';

const squareOdd = of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );

// Subscribe to get values
squareOdd.subscribe(x => console.log(x));
```

常用操作符

RxJS 提供了很多操作符（超过 150 个），不过只有少数是常用的。 下面是一个常用操作符的列表，要查看用法范例，参见 RxJS 文档中的 RxJS 5 操作符范例。

> 注意，对于 Angular 应用来说，我们提倡使用管道来组合操作符，而不是使用链式写法。链式写法仍然在很多 RxJS 中使用着。

类别	操作
创建	`from` , `fromPromise` , `fromEvent` , `of`
组合	`combineLatest` , `concat` , `merge` , `startWith` , `withLatestFrom` , `zip`
过滤	`debounceTime` , `distinctUntilChanged` , `filter` , `take` , `takeUntil`
转换	`bufferTime` , `concatMap` , `map` , `mergeMap` , `scan` , `switchMap`
工具	`tap`
多播	`share`

错误处理

除了可以在订阅时提供 `error()` 处理器外，RxJS 还提供了 `catchError` 操作符，它允许你在管道中处理已知错误。

假设你有一个可观察对象，它发起 API 请求，然后对服务器返回的响应进行映射。如果服务器返回了错误或值不存在，就会生成一个错误。如果你捕获这个错误并提供了一个默认值，流就会继续处理这些值，而不会报错。

下面是使用 `catchError` 操作符实现这种效果的例子：

catchError operator

```
import { ajax } from 'rxjs/ajax';
import { map, catchError } from 'rxjs/operators';
// Return "response" from the API. If an error happens,
// return an empty array.
const apiData = ajax('/api/data').pipe(
  map(res => {
    if (!res.response) {
      throw new Error('Value expected!');
    }
    return res.response;
  }),
  catchError(err => of([]))
);

apiData.subscribe({
  next(x) { console.log('data: ', x); },
  error(err) { console.log('errors already caught... will not run'); }
});
```

重试失败的可观察对象

`catchError` 提供了一种简单的方式进行恢复，而 `retry` 操作符让你可以尝试失败的请求。

可以在 `catchError` 之前使用 `retry` 操作符。它会订阅到原始的来源可观察对象，它可以重新运行导致结果出错的动作序列。如果其中包含 HTTP 请求，它就会重新发起那个 HTTP 请求。

下列代码为前面的例子加上了捕获错误前重发请求的逻辑：

retry operator

```
import { ajax } from 'rxjs/ajax';
import { map, retry, catchError } from 'rxjs/operators';

const apiData = ajax('/api/data').pipe(
  retry(3), // Retry up to 3 times before failing
  map(res => {
    if (!res.response) {
      throw new Error('Value expected!');
    }
    return res.response;
  }),
  catchError(err => of([]))
);

apiData.subscribe({
  next(x) { console.log('data: ', x); },
  error(err) { console.log('errors already caught... will not run'); }
});
```

> 不要重试登录认证请求，这些请求只应该由用户操作触发。我们肯定不会希望自动重复发送登录请求导致用户的账号被锁定。

可观察对象的命名约定

由于 Angular 的应用几乎都是用 TypeScript 写的，你通常会希望知道某个变量是否可观察对象。虽然 Angular 框架并没有针对可观察对象的强制性命名约定，不过你经常会看到可观察对象的名字以“$”符号结尾。

这在快速浏览代码并查找可观察对象值时会非常有用。同样的，如果你希望用某个属性来存储来自可观察对象的最近一个值，它的命名惯例是与可观察对象同名，但不带“$”后缀。

比如：

Naming observables

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-stopwatch',
  templateUrl: './stopwatch.component.html'
})
export class StopwatchComponent {

  stopwatchValue: number;
  stopwatchValue$: Observable<number>;

  start() {
    this.stopwatchValue$.subscribe(num =>
      this.stopwatchValue = num
    );
  }
}
```

Angular 中的可观察对象

Angular 使用可观察对象作为处理各种常用异步操作的接口。比如：

- `EventEmitter` 类派生自 `Observable`。
- HTTP 模块使用可观察对象来处理 AJAX 请求和响应。
- 路由器和表单模块使用可观察对象来监听对用户输入事件的响应。

事件发送器 `EventEmitter`

Angular 提供了一个 `EventEmitter` 类，它用来从组件的 `@Output()` 属性中发布一些值。`EventEmitter` 扩展了 `Observable`，并添加了一个 `emit()` 方法，这样它就可以发送任意值了。当你调用 `emit()` 时，就会把所发送的值传给订阅上来的观察者的 `next()` 方法。

这种用法的例子参见 EventEmitter 文档。下面这个范例组件监听了 `open` 和 `close` 事件：

`<zippy (open)="onOpen($event)" (close)="onClose($event)"></zippy>`

组件的定义如下：

EventEmitter

```
@Component({
  selector: 'zippy',
  template: `
  <div class="zippy">
    <div (click)="toggle()">Toggle</div>
    <div [hidden]="!visible">
      <ng-content></ng-content>
    </div>
  </div>`})

export class ZippyComponent {
  visible = true;
  @Output() open = new EventEmitter<any>();
  @Output() close = new EventEmitter<any>();

  toggle() {
    this.visible = !this.visible;
    if (this.visible) {
      this.open.emit(null);
    } else {
      this.close.emit(null);
    }
  }
}
```

HTTP

Angular 的 `HttpClient` 从 HTTP 方法调用中返回了可观察对象。例如，`http.get('/api')` 就会返回可观察对象。相对于基于承诺（Promise）的 HTTP API，它有一系列优点：

- 可观察对象不会修改服务器的响应（和在承诺上串联起来的 `.then()` 调用一样）。反之，你可以使用一系列操作符来按需转换这些值。
- HTTP 请求是可以通过 `unsubscribe()` 方法来取消的。
- 请求可以进行配置，以获取进度事件的变化。
- 失败的请求很容易重试。

Async 管道

AsyncPipe 会订阅一个可观察对象或承诺，并返回其发出的最后一个值。当发出新值时，该管道就会把这个组件标记为需要进行变更检查的（译注：因此可能导致刷新界面）。

下面的例子把 `time` 这个可观察对象绑定到了组件的视图中。这个可观察对象会不断使用当前时间更新组件的视图。

Using async pipe

```
@Component({
  selector: 'async-observable-pipe',
  template: `<div><code>observable|async</code>:
       Time: {{ time | async }}</div>`
})
export class AsyncObservablePipeComponent {
  time = new Observable(observer =>
    setInterval(() => observer.next(new Date().toString()), 1000)
  );
}
```

路由器 (router)

`Router.events` 以可观察对象的形式提供了其事件。 你可以使用 RxJS 中的 `filter()` 操作符来找到感兴趣的事件，并且订阅它们，以便根据浏览过程中产生的事件序列作出决定。 例子如下：

Router events

```
import { Router, NavigationStart } from '@angular/router';
import { filter } from 'rxjs/operators';

@Component({
  selector: 'app-routable',
  templateUrl: './routable.component.html',
  styleUrls: ['./routable.component.css']
})
export class Routable1Component implements OnInit {

  navStart: Observable<NavigationStart>;

  constructor(private router: Router) {
    // Create a new Observable the publishes only the NavigationStart event
    this.navStart = router.events.pipe(
      filter(evt => evt instanceof NavigationStart)
    ) as Observable<NavigationStart>;
  }

  ngOnInit() {
    this.navStart.subscribe(evt => console.log('Navigation Started!'));
  }
}
```

ActivatedRoute 是一个可注入的路由器服务，它使用可观察对象来获取关于路由路径和路由参数的信息。比如，`ActivateRoute.url` 包含一个用于汇报路由路径的可观察对象。例子如下：

ActivatedRoute

```
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-routable',
  templateUrl: './routable.component.html',
  styleUrls: ['./routable.component.css']
})
export class Routable2Component implements OnInit {
  constructor(private activatedRoute: ActivatedRoute) {}

  ngOnInit() {
    this.activatedRoute.url
      .subscribe(url => console.log('The URL changed to: ' + url));
  }
}
```

响应式表单 (reactive forms)

响应式表单具有一些属性，它们使用可观察对象来监听表单控件的值。 `FormControl` 的 `valueChanges` 属性和 `statusChanges` 属性包含了会发出变更事件的可观察对象。订阅可观察的表单控件属性是在组件类中触发应用逻辑的途径之一。比如：

Reactive forms

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

@Component({
  selector: 'my-component',
  template: 'MyComponent Template'
})
export class MyComponent implements OnInit {
  nameChangeLog: string[] = [];
  heroForm: FormGroup;

  ngOnInit() {
    this.logNameChange();
  }
  logNameChange() {
    const nameControl = this.heroForm.get('name');
    nameControl.valueChanges.forEach(
      (value: string) => this.nameChangeLog.push(value)
    );
  }
}
```

可观察对象用法实战

这里示范了一些在某种领域中可观察对象会特别有用的例子。

输入提示（type-ahead）建议

可观察对象可以简化输入提示建议的实现方式。典型的输入提示要完成一系列独立的任务：

- 从输入中监听数据。
- 移除输入值前后的空白字符，并确认它达到了最小长度。
- 防抖（这样才能防止连续按键时每次按键都发起 API 请求，而应该等到按键出现停顿时才发起）
- 如果输入值没有变化，则不要发起请求（比如按某个字符，然后快速按退格）。
- 如果已发出的 AJAX 请求的结果会因为后续的修改而变得无效，那就取消它。

完全用 JavaScript 的传统写法实现这个功能可能需要大量的工作。使用可观察对象，你可以使用这样一个 RxJS 操作符的简单序列：

Typeahead

```
import { fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { map, filter, debounceTime, distinctUntilChanged, switchMap } from 'rxjs/operators';

const searchBox = document.getElementById('search-box');

const typeahead = fromEvent(searchBox, 'input').pipe(
  map((e: KeyboardEvent) => e.target.value),
  filter(text => text.length > 2),
  debounceTime(10),
  distinctUntilChanged(),
  switchMap(() => ajax('/api/endpoint'))
);

typeahead.subscribe(data => {
 // Handle the data from the API
});
```

指数化退避

指数化退避是一种失败后重试 API 的技巧，它会在每次连续的失败之后让重试时间逐渐变长，超过最大重试次数之后就会彻底放弃。 如果使用承诺和其它跟踪 AJAX 调用的方法会非常复杂，而使用可观察对象，这非常简单：

Exponential backoff

```
import { pipe, range, timer, zip } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { retryWhen, map, mergeMap } from 'rxjs/operators';

function backoff(maxTries, ms) {
 return pipe(
   retryWhen(attempts => range(1, maxTries)
     .pipe(
       zip(attempts, (i) => i),
       map(i => i * i),
       mergeMap(i =>  timer(i * ms))
     )
   )
 );
}

ajax('/api/endpoint')
  .pipe(backoff(3, 250))
  .subscribe(data => handleData(data));

function handleData(data) {
  // ...
}
```

可观察对象与其它技术的比较

你可以经常使用可观察对象（Observable）而不是承诺（Promise）来异步传递值。 类似的，可观察对象也可以取代事件处理器的位置。最后，由于可观察对象传递多个值，所以你可以在任何可能构建和操作数组的地方使用可观察对象。

在这些情况下，可观察对象的行为与其替代技术有一些差异，不过也提供了一些显著的优势。下面是对这些差异的详细比较。

可观察对象 vs. 承诺

可观察对象经常拿来和承诺进行对比。有一些关键的不同点：

- 可观察对象是声明式的，在被订阅之前，它不会开始执行。承诺是在创建时就立即执行的。这让可观察对象可用于定义那些应该按需执行的菜谱。
- 可观察对象能提供多个值。承诺只提供一个。这让可观察对象可用于随着时间的推移获取多个值。
- 可观察对象会区分串联处理和订阅语句。承诺只有 `.then()` 语句。这让可观察对象可用于创建供系统的其它部分使用而不希望立即执行的复杂菜谱。
- 可观察对象的 `subscribe()` 会负责处理错误。承诺会把错误推送给它的子承诺。这让可观察对象可用于进行集中式、可预测的错误处理。

创建与订阅

- 在有消费者订阅之前，可观察对象不会执行。`subscribe()` 会执行一次定义好的行为，并且可以再次调用它。每次订阅都是单独计算的。重新订阅会导致重新计算这些值。

```
// declare a publishing operation
new Observable((observer) => { subscriber_fn });
// initiate execution
observable.subscribe(() => {
      // observer handles notifications
    });
```

- 承诺会立即执行，并且只执行一次。当承诺创建时，会立即计算出结果。没有办法重新做一次。所有的 `then` 语句（订阅）都会共享同一次计算。

```
// initiate execution
new Promise((resolve, reject) => { executer_fn });
// handle return value
promise.then((value) => {
      // handle result here
    });
```

串联

- 可观察对象会区分各种转换函数，比如映射和订阅。只有订阅才会激活订阅者函数，以开始计算那些值。

```
observable.map((v) => 2*v);
```

- 承诺并不区分最后的 `.then()` 语句（等价于订阅）和中间的 `.then()` 语句（等价于映射）。

```
promise.then((v) => 2*v);
```

可取消

- 可观察对象的订阅是可取消的。取消订阅会移除监听器，使其不再接受将来的值，并通知订阅者函数取消正在进行的工作。

```
const sub = obs.subscribe(...);
sub.unsubscribe();
```

- 承诺是不可取消的。

错误处理

- 可观察对象的错误处理是交给订阅者的错误处理器的，并且该订阅者会自动取消对这个可观察对象的订阅。

```
obs.subscribe(() => {
  throw Error('my error');
});
```

- 承诺会把错误推给其子承诺。

```
promise.then(() => {
      throw Error('my error');
});
```

速查表

下列代码片段揭示了同样的操作要如何分别使用可观察对象和承诺进行实现。

操作	可观察对象	承诺

创建	`new Observable((observer) => {`
	`  observer.next(123);`
	`});`
		`new Promise((resolve, reject) => {`
		`  resolve(123);`
		`});`

转换	`obs.map((value) => value * 2 );`	`promise.then((value) => value * 2);`

订阅	`sub = obs.subscribe((value) => {`
	`  console.log(value)`
	`});`
		`promise.then((value) => {`
		`  console.log(value);`
		`});`

取消订阅	`sub.unsubscribe();`	承诺被解析时隐式完成。

可观察对象 vs. 事件 API

可观察对象和事件 API 中的事件处理器很像。这两种技术都会定义通知处理器，并使用它们来处理一段时间内传递的多个值。订阅可观察对象与添加事件处理器是等价的。一个显著的不同是你可以配置可观察对象，使其在把事件传给事件处理器之间先进行转换。

使用可观察对象来处理错误和异步操作在 HTTP 请求这样的场景下更加具有一致性。

下列代码片段揭示了同样的操作要如何分别使用可观察对象和事件 API 进行实现。

	可观察对象	事件 API

创建与取消	`// Setup`
	`let clicks$ = fromEvent(buttonEl, ‘click’);`
	`// Begin listening`
	`let subscription = clicks$`
	`  .subscribe(e => console.log(‘Clicked’, e))`
	`// Stop listening`
	`subscription.unsubscribe();`
		`function handler(e) {`
		`  console.log(‘Clicked’, e);`
		`}`
		
		`// Setup & begin listening`
		`button.addEventListener(‘click’, ha`
		`// Stop listening`
		`button.removeEventListener(‘click’,`

订阅	`observable.subscribe(() => {`
	`  // notification handlers here`
	`});`
		`element.addEventListener(eventName, (`
		`  // notification handler here`
		`});`

配置	监听按键，提供一个流来表示这些输入的值。
	
	`fromEvent(inputEl, 'keydown').pipe(`
	`  map(e => e.target.value)`
	`);`
		不支持配置。
		
		`element.addEventListener(eventName, (`
		`  // Cannot change the passed Event i`
		`  // value before it gets to the hand`
		`});`

可观察对象 vs. 数组

可观察对象会随时间生成值。数组是用一组静态的值创建的。某种意义上，可观察对象是异步的，而数组是同步的。 在下列例子中，➞ 符号表示异步传递值。

	可观察对象	数组
给出值	obs: →1→2→3→5→7 obsB: →’a’→’b’→’c’	arr: [1, 2, 3, 5, 7] arrB: [’a’, ’b’, ’c’]
concat()	obs.concat(obsB) →1→2→3→5→7→’a’→’b’→’c’	arr.concat(arrB) [1,2,3,5,7,’a’,’b’,’c’]
filter()	obs.filter((v) => v>3) →5→7	arr.filter((v) => v>3) [5, 7]
find()	obs.find((v) => v>3) →5	arr.find((v) => v>3) 5
findIndex()	obs.findIndex((v) => v>3) →3	arr.findIndex((v) => v>3) 3

forEach()	obs.forEach((v) => {
	  console.log(v);
	})
	1
	2
	3
	5
	7
		arr.forEach((v) => {
		  console.log(v);
		})
		1
		2
		3
		5
		7

map()	obs.map((v) => -v) ➞-1➞-2➞-3➞-5➞-7	arr.map((v) => -v) [-1, -2, -3, -5, -7]
reduce()	obs.scan((s,v)=> s+v, 0) ➞1➞3➞6➞11➞18	arr.reduce((s,v) => s+v, 0) 18

启动过程

前提条件

对下列知识有基本的了解：

- JavaScript 模块与 NgModules。

NgModule 用于描述应用的各个部分如何组织在一起。 每个应用又至少一个 Angular 模块，**根**模块就是你用来启动此应用的模块。 按照惯例，它通常命名为 `AppModule`。

如果你使用 CLI 来生成一个应用，其默认的 `AppModule` 是这样的：

```
/* JavaScript imports */
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

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

/* the AppModule class with the @NgModule decorator */
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

在 `import` 语句之后，是一个带有 `@NgModule` 装饰器的类。

`@NgModule` 装饰器表明 `AppModule` 是一个 `NgModule` 类。 `@NgModule` 获取一个元数据对象，它会告诉 Angular 如何编译和启动本应用。

- **declarations** —— 该应用所拥有的组件。
- **imports** —— 导入 `BrowserModule` 以获取浏览器特有的服务，比如 DOM 渲染、无害化处理和位置（location）。
- **providers** —— 各种服务提供商。
- **bootstrap** —— **根**组件，Angular 创建它并插入 `index.html` 宿主页面。

默认的 CLI 应用只有一个组件 `AppComponent`，所以它会同时出现在 `declarations` 和 `bootstrap` 数组中。

`declarations` 数组

该模块的 `declarations` 数组告诉 Angular 哪些组件属于该模块。 当你创建更多组件时，也要把它们添加到 `declarations` 中。

每个组件都应该（且只能）声明（declare）在一个 `NgModule` 类中。 如果你使用了未声明过的组件，Angular 就会报错。

`declarations` 数组只能接受可声明对象。可声明对象包括组件、指令和管道。 一个模块的所有可声明对象都必须放在 `declarations` 数组中。 可声明对象必须只能属于一个模块，如果同一个类被声明在了多个模块中，编译器就会报错。

这些可声明的类在当前模块中是可见的，但是对其它模块中的组件是不可见的 —— 除非把它们从当前模块导出， 并让对方模块导入本模块。

下面是哪些类可以添加到 `declarations` 数组中的例子：

```
declarations: [
  YourComponent,
  YourPipe,
  YourDirective
],
```

每个可声明对象都只能属于一个模块，所以只能把它声明在一个 `@NgModule` 中。当你需要在其它模块中使用它时，就要在那里导入包含这个可声明对象的模块。

只有 `@NgModule` 可以出现在 `imports` 数组中。

通过 `@NgModule` 使用指令

使用 `declarations` 数组声明指令。在模块中使用指令、组件或管道的步骤如下：

1. 从你编写它的文件中导出它。
2. 把它导入到适当的模块中。
3. 在 `@NgModule` 的 `declarations` 数组中声明它。

这三步的结果如下所示。在你创建指令的文件中导出它。 下面的例子中，`item.directive.ts` 中的 `ItemDirective` 是 CLI 自动生成的默认指令结构。

src/app/item.directive.ts

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

@Directive({
  selector: '[appItem]'
})
export class ItemDirective {
// code goes here
  constructor() { }

}
```

重点在于你要先在这里导出它才能在别处导入它。接下来，使用 JavaScript 的 `import` 语句把它导入到 NgModule 中（这里是 `app.module.ts`）。

src/app/app.module.ts

```
import { ItemDirective } from './item.directive';
```

同样在这个文件中，把它添加到 `@NgModule` 的 `declarations` 数组中：

src/app/app.module.ts

```
declarations: [
  AppComponent,
  ItemDirective
],
```

现在，你就可以在组件中使用 `ItemDirective` 了。这个例子中使用的是 `AppModule`，但是在特性模块中你也可以这么做。 要进一步了解指令，参见属性型指令和结构型指令。 这些也同样适用于管道和组件。

记住：组件、指令和管道都只能属于一个模块。你在应用中也只需要声明它们一次，因为你还可以通过导入必要的模块来使用它们。这能节省你的时间，并且帮助你的应用保持精简。

`imports` 数组

模块的 `imports` 数组只会出现在 `@NgModule` 元数据对象中。 它告诉 Angular 该模块想要正常工作，还需要哪些模块。

列表中的模块导出了本模块中的各个组件模板中所引用的各个组件、指令或管道。在这个例子中，当前组件是 `AppComponent`，它引用了导出自 `BrowserModule`、`FormsModule` 或 `HttpModule` 的组件、指令或管道。总之，组件的模板中可以引用在当前模块中声明的或从其它模块中导入的组件、指令、管道。

你还没有提供任何服务，但是很快就会创建一些，而且可能也会选择在这里提供它们。

`providers` 数组

`providers` 数组中列出了该应用所需的服务。当直接把服务列在这里时，它们是全应用范围的。 当你使用特性模块和惰性加载时，它们是范围化的。要了解更多，参见服务提供商。

`bootstrap` 数组

应用是通过引导根模块 `AppModule` 来启动的，根模块还引用了 `entryComponent`。 此外，引导过程还会创建 `bootstrap` 数组中列出的组件，并把它们逐个插入到浏览器的 DOM 中。

每个被引导的组件都是它自己的组件树的根。 插入一个被引导的组件通常触发一系列组件的创建并形成组件树。

虽然也可以在宿主页面中放多个组件，但是大多数应用只有一个组件树，并且只从一个根组件开始引导。

这个根组件通常叫做 `AppComponent`，并且位于根模块的 `bootstrap` 数组中。

关于 Angular 模块的更多知识

要进一步了解常见的 NgModules 知识，参见 关于模块的常见问题。

Angular 模块 (NgModule)

前提条件

对下列概念有基本的理解：

- 引导启动。
- JavaScript 模块与 NgModules。

NgModules 用于配置注入器和编译器，并帮你把那些相关的东西组织在一起。

NgModule 是一个带有 `@NgModule` 装饰器的类。 `@NgModule` 的参数是一个元数据对象，用于描述如何编译组件的模板，以及如何在运行时创建注入器。 它会标出该模块自己的组件、指令和管道，通过 `exports` 属性公开其中的一部分，以便外部组件使用它们。 `NgModule` 还能把一些服务提供商添加到应用的依赖注入器中。

要想找一个涉及本章所讲的全部技术的范例，参见 在线例子 / 下载范例。 要想得到针对单项技术的一些讲解，参见本目录下的相关页面。

Angular 模块化

模块是组织应用和使用外部库扩展应用的最佳途径。

Angular 自己的库都是 NgModule，比如 `FormsModule`、`HttpClientModule` 和 `RouterModule`。 很多第三方库也是 NgModule，比如 Material Design、 Ionic 和 AngularFire2。

Angular 模块把组件、指令和管道打包成内聚的功能块，每个模块聚焦于一个特性区域、业务领域、工作流或通用工具。

模块还可以把服务加到应用中。 这些服务可能是内部开发的（比如你自己写的），或者来自外部的（比如 Angular 的路由和 HTTP 客户端）。

模块可以在应用启动时立即加载，也可以由路由器进行异步的惰性加载。

NgModule 的元数据会做这些：

- 声明某些组件、指令和管道属于这个模块。
- 公开其中的部分组件、指令和管道，以便其它模块中的组件模板中可以使用它们。
- 导入其它带有组件、指令和管道的模块，这些模块中的元件都是本模块所需的。
- 提供一些供应用中的其它组件使用的服务。

每个 Angular 应用都至少有一个模块，也就是根模块。 你可以引导那个模块，以启动该应用。

对于那些只有少量组件的简单应用，根模块就是你所需的一切。 随着应用的成长，你要把这个根模块重构成一些特性模块，它们代表一组密切相关的功能集。 然后你再把这些模块导入到根模块中。

基本的模块

CLI 在创建新应用时会生成下列基本的应用模块。

src/app/app.module.ts

```
// imports
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';
import { ItemDirective } from './item.directive';


// @NgModule decorator with its metadata
@NgModule({
  declarations: [
    AppComponent,
    ItemDirective
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

文件的顶部是一些导入语句。接下来是你配置 `NgModule` 的地方，用于规定哪些组件和指令属于它（`declarations`），以及它使用了哪些其它模块（`imports`）。 本章是基于引导一章的，那里详细讲了 NgModule 的结构。如果要进一步了解 `@NgModule` 的结构，参见引导。

关于 NgModule 的更多知识

你可能还对下列内容感兴趣：

- 特性模块
- 入口组件
- 服务提供商。
- NgModule 的分类.

JavaScript 模块 vs. NgModule

前提条件

对 JavaScript/ECMAScript 模块 有基本的了解。

JavaScript 和 Angular 都使用模块来组织代码，虽然它们的组织形式不同，但 Angular 的应用会同时依赖两者。

JavaScript 模块

在 JavaScript 中，模块是内含 JavaScript 代码的独立文件。要让其中的东西可用，你要写一个导出语句，通常会放在相应的代码之后，类似这样：

```
export class AppComponent { ... }
```

然后，当你在其它文件中需要这个文件的代码时，要像这样导入它：

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

JavaScript 模块让你能为代码加上命名空间，防止因为全局变量而引起意外。

NgModules

NgModule 是一些带有 `@NgModule` 装饰器的类。`@NgModule` 装饰器的 `imports` 数组会告诉 Angular 哪些其它的 NgModule 是当前模块所需的。 `imports` 数组中的这些模块与 JavaScript 模块不同，它们都是 NgModule 而不是常规的 JavaScript 模块。 带有 `@NgModule` 装饰器的类通常会习惯性地放在单独的文件中，但单独的文件并不像 JavaScript 模块那样作为必要条件，而是因为它带有 `@NgModule` 装饰器及其元数据。

Angular CLI 生成的 `AppModule` 实际演示了这两种模块：

```
/* These are JavaScript import statements. Angular doesn’t know anything about these. */
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

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

/* The @NgModule decorator lets Angular know that this is an NgModule. */
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [     /* These are NgModule imports. */
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

NgModule 类与 JavaScript 模块有下列关键性的不同：

- Angular 模块只绑定了**可声明的类**，这些可声明的类只是供Angular 编译器用的。
- 与 JavaScript 类把它所有的成员类都放在一个巨型文件中不同，你要把该模块的类列在它的 `@NgModule.declarations` 列表中。
- Angular 模块只能导出**可声明的类**。这可能是它自己拥有的也可能是从其它模块中导入的。它不会声明或导出任何其它类型的类。
- 与 JavaScript 模块不同，NgModule 可以通过把服务提供商加到 `@NgModule.providers` 列表中，来用服务扩展**整个**应用。

关于 NgModule 的更多知识

要了解关于 NgModule 的更多知识，参见

- 引导启动。
- 常用模块。
- 服务提供商。

常用模块

前提条件

对引导有基本的了解。

Angular 应用需要不止一个模块，它们都为根模块服务。 如果你要把某些特性添加到应用中，可以通过添加模块来实现。 下列是一些常用的 Angular 模块，其中带有一些其内容物的例子：

NgModule	导入自	为何使用
`BrowserModule`	`@angular/platform-browser`	当你想要在浏览器中运行应用时
`CommonModule`	`@angular/common`	当你想要使用 `NgIf` 和 `NgFor` 时
`FormsModule`	`@angular/forms`	当要构建模板驱动表单时（它包含 `NgModel` ）
`ReactiveFormsModule`	`@angular/forms`	当要构建响应式表单时
`RouterModule`	`@angular/router`	要使用路由功能，并且你要用到 `RouterLink`,`.forRoot()` 和 `.forChild()` 时
`HttpClientModule`	`@angular/common/http`	当你要和服务器对话时

导入模块

当你使用这些 Angular 模块时，在 `AppModule`（或适当的特性模块）中导入它们，并把它们列在当前 `@NgModule` 的 `imports` 数组中。比如，在 CLI 生成的基本应用中，`BrowserModule` 会在 `app.module.ts` 中 `AppModule` 的顶部最先导入。

```
/* import modules so that AppModule can access them */
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [ /* add modules here so Angular knows to use them */
    BrowserModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

文件顶部的这些导入是 JavaScript 的导入语句，而 `@NgModule` 中的 `imports` 数组则是 Angular 特有的。 要了解更多的不同点，参见 JavaScript 模块 vs. NgModule。

`BrowserModule` 和 `CommonModule`

`BrowserModule` 导入了 `CommonModule`，它贡献了很多通用的指令，比如 `ngIf` 和 `ngFor`。 另外，`BrowserModule` 重新导出了 `CommonModule`，以便它所有的指令在任何导入了 `BrowserModule` 的 Angular 模块中都可以使用。

对于运行在浏览器中的应用来说，都必须在根模块中 `AppModule` 导入 `BrowserModule` ，因为它提供了启动和运行浏览器应用时某些必须的服务。`BrowserModule` 的提供商是面向整个应用的，所以它只能在根模块中使用，而不是特性模块。 特性模块只需要 `CommonModule` 中的常用指令，它们不需要重新安装所有全应用级的服务。

如果你把 `BrowserModule` 导入了惰性加载的特性模块中，Angular 就会返回一个错误，并告诉你应该改用 `CommonModule`。

```
▶EXCEPTION: Uncaught (in promise): Error: BrowserModule has already been loaded. If you need access to common    error_handler.js:54
directives such as NgIf and NgFor from a lazy loaded module, import CommonModule instead.
Error: BrowserModule has already been loaded. If you need access to common directives such as NgIf and NgFor from a lazy loaded
module, import CommonModule instead.
```

关于 NgModule 的更多知识

你可能还对下列内容感兴趣：

- 引导启动。
- Angular 模块.
- JavaScript 模块与 NgModules。

特性模块的分类

前提条件

对下列概念有基本的理解：

- 特性模块
- JavaScript 模块与 NgModules。
- 常用模块。

下面是特性模块的五个常用分类，包括五组：

- 领域特性模块。
- 带路由的特性模块。
- 路由模块。
- 服务特性模块
- 可视部件特性模块。

虽然下面的指南中描述了每种类型的使用及其典型特征，但在实际的应用中，你还可能看到它们的混合体。

特性模块	指导原则
领域	领域特性模块用来给用户提供应用程序领域中特有的用户体验，比如编辑客户信息或下订单等。它们通常会有一个顶级组件来充当该特性的根组件，并且通常是私有的。用来支持它的各级子组件。 领域特性模块大部分由 `declarations` 组成，只有顶级组件会被导出。 领域特性模块很少会有服务提供商。如果有，那么这些服务的生命周期必须和该模块的生命周期完全相同。 领域特性模块通常会由更高一级的特性模块导出且只导出一次。 对于缺少路由的小型应用，它们可能只会被根模块 `AppModule` 导入一次。
路由	带路由的特性模块是一种特殊的领域特性模块，但它的顶层组件会作为路由导航时的目标组件。根据这个定义，所有惰性加载的模块都是路由特性模块。 带路由的特性模块不会导出任何东西，因为它们的组件永远不会出现在外部组件的模板中。 惰性加载的路由特性模块不应该被任何模块导入。如果那样做就会导致它被立即加载，破坏了惰性加载的设计用途。 也就是说你应该永远不会看到它们在 `AppModule` 的 `imports` 中被引用。 立即加载的路由特性模块必须被其它模块导入，以便编译器能了解它所包含的组件。 路由特性模块很少会有服务提供商，原因参见惰性加载的特性模块中的解释。如果那样做，那么它所提供的服务的生命周期必须与该模块的生命周期完全相同。不要在路由特性模块或被路由特性模块所导入的模块中提供全应用级的单例服务。

路由	路由模块为其它模块提供路由配置，并且把路由这个关注点从它的配套模块中分离出来。 路由模块通常会做这些： • 定义路由。 • 把路由配置添加到该模块的 `imports` 中。 • 把路由守卫和解析器的服务提供商添加到该模块的 `providers` 中。 • 路由模块应该与其配套模块同名，但是加上“Routing”后缀。比如，`foo.module.ts` 中的 `FooModule` 就有一个位于 `foo-routing.module.ts` 文件中的 `FooRoutingModule` 路由模块。 如果其配套模块是根模块 `AppModule`，`AppRoutingModule` 就要使用 `RouterModule.forRoot(routes)` 来把路由器配置添加到它的 `imports` 中。 所有其它路由模块都是子模块，要使用 `RouterModule.forChild(routes)`。 • 按照惯例，路由模块会重新导出这个 `RouterModule`，以便其配套模块中的组件可以访问路由器指令，比如 `RouterLink` 和 `RouterOutlet`。 • 路由模块没有自己的可声明对象。组件、指令和管道都是特性模块的职责，而不是路由模块的。 路由模块只应该被它的配套模块导入。
服务	服务模块提供了一些工具服务，比如数据访问和消息。理论上，它们应该是完全由服务提供商组成的，不应该有可声明对象。Angular 的 `HttpClientModule` 就是一个服务模块的好例子。 根模块 `AppModule` 是唯一的可以导入服务模块的模块。
窗口部件	窗口部件模块为外部模块提供组件、指令和管道。很多第三方 UI 组件库都是窗口部件模块。 窗口部件模块应该完全由可声明对象组成，它们中的大部分都应该被导出。 窗口部件模块很少会有服务提供商。 如果任何模块的组件模板中需要用到这些窗口部件，就请导入相应的窗口部件模块。

下表中汇总了各种特性模块类型的关键特征。

特性模块	声明 `declarations`	提供商 `providers`	导出什么	被谁导入
领域	有	罕见	顶级组件	特性模块，AppModule
路由	有	罕见	无	无
路由	无	是（守卫）	RouterModule	特性（供路由使用）
服务	无	有	无	AppModule
窗口部件	有	罕见	有	特性

关于 NgModule 的更多知识

你可能还对下列内容感兴趣：

- 使用 Angular 路由器惰性加载模块。
- 服务提供商。

入口组件

前提条件：

对下列概念有基本的理解：

- 引导启动。

从分类上说，入口组件是 Angular 命令式加载的任意组件（也就是说你没有在模板中引用过它）， 你可以在 NgModule 中引导它，或把它包含在路由定义中来指定入口组件。

> 对比一下这两种组件类型：有一类组件被包含在模板中，它们是声明式加载的；另一类组件你会命令式加载它，这就是入口组件。

入口组件有两种主要的类型：

- 引导用的根组件。
- 在路由定义中指定的组件。

引导用的入口组件

下面这个例子中指定了一个引导用组件 `AppComponent`，位于基本的 `app.module.ts` 中：

```
1. @NgModule({
2.   declarations: [
3.     AppComponent
4.   ],
5.   imports: [
6.     BrowserModule,
7.     FormsModule,
8.     HttpModule,
9.     AppRoutingModule
10.  ],
11.  providers: [],
12.  bootstrap: [AppComponent] // bootstrapped entry component
13. })
```

可引导组件是一个入口组件，Angular 会在引导过程中把它加载到 DOM 中。 其它入口组件是在其它时机动态加载的，比如用路由器。

Angular 会动态加载根组件 `AppComponent`，是因为它的类型作为参数传给了 `@NgModule.bootstrap` 函数。

> 组件也可以在该模块的 `ngDoBootstrap()` 方法中进行命令式引导。 `@NgModule.bootstrap` 属性告诉编译器，这里是一个入口组件，它应该生成代码，来使用这个组件引导该应用。

引导用的组件必须是入口组件，因为引导过程是命令式的，所以它需要一个入口组件。

路由到的入口组件

入口组件的第二种类型出现在路由定义中，就像这样：

```
const routes: Routes = [
  {
    path: '',
    component: CustomerListComponent
  }
];
```

路由定义使用组件类型引用了一个组件：`component: CustomerListComponent`。

所有路由组件都必须是入口组件。这需要你把同一个组件添加到两个地方（路由中和 `entryComponents` 中），但编译器足够聪明，可以识别出这里是一个路由定义，因此它会自动把这些路由组件添加到 `entryComponents` 中。

`entryComponents` 数组

虽然 `@NgModule` 装饰器具有一个 `entryComponents` 数组，但大多数情况下你不用显式设置入口组件，因为 Angular 会自动把 `@NgModule.bootstrap` 中的组件以及路由定义中的组件添加到入口组件中。 虽然这两种机制足够自动添加大多数入口组件，但如果你要用其它方式根据类型来命令式的引导或动态加载某个组件，你就必须把它们显式添加到 `entryComponents` 中了。

`entryComponents` 和编译器

对于生产环境的应用，你总是希望加载尽可能小的代码。 这些代码应该只包含你实际使用到的类，并且排除那些从未用到的组件。因此，Angular 编译器只会为那些可以从 `entryComponents` 中直接或间接访问到的组

件生成代码。 这意味着，仅仅往 `@NgModule.declarations` 中添加更多引用，并不能表达出它们在最终的代码包中是必要的。

实际上，很多库声明和导出的组件都是你从未用过的。 比如，Material Design 库会导出其中的所有组件，因为它不知道你会用哪一个。然而，显然你也不打算全都用上。 对于那些你没有引用过的，摇树优化工具就会把这些组件从最终的代码包中摘出去。

如果一个组件既不是**入口组件**也不没有在模板中使用过，摇树优化工具就会把它扔出去。 所以，最好只添加那些真正的入口组件，以便让应用尽可能保持精简。

关于 Angular 模块的更多知识

你可能还对下列内容感兴趣：

- NgModule 的分类.
- 使用 Angular 路由器惰性加载模块。
- 服务提供商。
- NgModule 常见问题.

特性模块

特性模块是用来对代码进行组织的模块。

前提条件

对下列知识有基本的了解：

- 引导启动。
- JavaScript 模块与 NgModules。
- 常用模块。

要想查看本页提到的这个带有特性模块的范例应用，参见 在线例子 / 下载范例。

随着应用的增长，你可能需要组织与特定应用有关的代码。 这将帮你把特性划出清晰的边界。使用特性模块，你可以把与特定的功能或特性有关的代码从其它代码中分离出来。 为应用勾勒出清晰的边界，有助于开发人员之间、小组之间的协作，有助于分离各个指令，并帮助管理根模块的大小。

特性模块 vs. 根模块

与核心的 Angular API 的概念相反，特性模块是最佳的组织方式。特性模块提供了聚焦于特定应用需求的一组功能，比如用户工作流、路由或表单。 虽然你也可以用根模块做完所有事情，不过特性模块可以帮助你把应用划分成一些聚焦的功能区。特性模块通过它提供的服务以及共享出的组件、指令和管道来与根模块和其它模块合作。

如何制作特性模块

如果你已经有了 CLI 生成的应用，可以在项目的根目录下输入下面的命令来创建特性模块。把这里的 `CustomerDashboard` 替换成你的模块名。你可以从名字中省略掉“Module”后缀，因为 CLI 会自动追加上它：

```
ng generate module CustomerDashboard
```

这会让 CLI 创建一个名叫 `customer-dashboard` 的文件夹，其中有一个名叫 `customer-dashboard.module.ts`，内容如下：

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class CustomerDashboardModule { }
```

无论根模块还是特性模块，其 NgModule 结构都是一样的。在 CLI 生成的特性模块中，在文件顶部有两个 JavaScript 的导入语句：第一个导入了 `NgModule`，它像根模块中一样让你能使用 `@NgModule` 装饰器；第二个导入了 `CommonModule`，它提供了很多像 `ngIf` 和 `ngFor` 这样的常用指令。 特性模块导入 `CommonModule`，而不是 `BrowserModule`，后者只应该在根模块中导入一次。 `CommonModule` 只包含常用指令的信息，比如 `ngIf` 和 `ngFor`，它们在大多数模板中都要用到，而 `BrowserModule` 为浏览器所做的应用配置只会使用一次。

`declarations` 数组让你能添加专属于这个模块的可声明对象（组件、指令和管道）。 要添加组件，就在命令行中输入如下命令，这里的 `customer-dashboard` 是一个目录，CLI 会把特性模块生成在这里，而 `CustomerDashboard` 就是该组件的名字：

```
ng generate component customer-dashboard/CustomerDashboard
```

这会在 `customer-dashboard` 中为新组件生成一个目录，并使用 `CustomerDashboardComponent` 的信息修改这个特性模块：

src/app/customer-dashboard/customer-dashboard.module.ts

```
// import the new component
import { CustomerDashboardComponent } from './customer-dashboard/customer-dashboard.component';
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    CustomerDashboardComponent
  ],
})
```

`CustomerDashboardComponent` 出现在了顶部的 JavaScript 导入列表里，并且被添加到了 `declarations` 数组中，它会让 Angular 把新组件和这个特性模块联系起来。

导入特性模块

要想把这个特性模块包含进应用中，你还得让根模块 `app.module.ts` 知道它。注意，在 `customer-dashboard.module.ts` 的底部导出了 `CustomerDashboardModule`。这样就把它暴露出来，以便其它模块可以拿到它。要想把它导入到 `AppModule` 中，就把它加入 `app.module.ts` 的导入表中，并将其加入 `imports` 数组：

src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';
// import the feature module here so you can add it to the imports array below
import { CustomerDashboardModule } from './customer-dashboard/customer-
dashboard.module';


@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    CustomerDashboardModule // add the feature module here
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

现在 `AppModule` 知道这个特性模块了。如果你往该特性模块中加入过任何服务提供商，`AppModule` 也同样会知道它，其它模块中也一样。不过，NgModule 并不会暴露出它们的组件。

渲染特性模块的组件模板

当 CLI 为这个特性模块生成 `CustomerDashboardComponent` 时，还包含一个模板 `customer-dashboard.component.html`，它带有如下页面脚本：

src/app/customer-dashboard/customer-dashboard/customer-dashboard.component.html

```
<p>
  customer-dashboard works!
</p>
```

要想在 `AppComponent` 中查看这些 HTML，你首先要在 `CustomerDashboardModule` 中导出 `CustomerDashboardComponent`。 在 `customer-dashboard.module.ts` 中，`declarations` 数组的紧下方，加入一个包含 `CustomerDashboardModule` 的 `exports` 数组：

src/app/customer-dashboard/customer-dashboard.module.ts

```
exports: [
  CustomerDashboardComponent
]
```

然后，在 `AppComponent` 的 `app.component.html` 中，加入标签 `<app-customer-dashboard>`：

src/app/app.component.html

```
<h1>
  {{title}}
</h1>

<!-- add the selector from the CustomerDashboardComponent -->
<app-customer-dashboard></app-customer-dashboard>
```

现在，除了默认渲染出的标题外，还渲染出了 `CustomerDashboardComponent` 的模板：

localhost:4200

app works!

customer-dashboard works!

关于 NgModule 的更多知识

你可能还对下列内容感兴趣：

- 使用 Angular 路由器惰性加载模块。
- 服务提供商。
- 特性模块的分类。

服务提供商

前提条件：

- 对引导有基本的了解。
- 熟悉常用模块。

要想查看本页提到的这个带有特性模块的范例应用，参见 在线例子 / 下载范例。

提供商就相当于说明书，用来指导 DI 系统该如何获取某个依赖的值。 大多数情况下，这些依赖就是你要创建和提供的那些服务。

提供服务

如果你是用 CLI 创建的应用，那么可以使用下列 CLI 命令在项目根目录下创建一个服务。把其中的 `User` 替换成你的服务名。

```
ng generate service User
```

该命令会创建下列 `UserService` 骨架：

src/app/user.service.0.ts

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

@Injectable({
  providedIn: 'root',
})
export class UserService {
}
```

现在，你就可以在应用中到处注入 `UserService` 了。

该服务本身是 CLI 创建的一个类，并且加上了 `@Injectable` 装饰器。默认情况下，该装饰器是用 `providedIn` 属性进行配置的，它会为该服务创建一个提供商。在这个例子中，`providedIn: 'root'` 指定该服务应该在根注入器中提供。

提供商的作用域

当你把服务提供商添加到应用的根注入器中时，它就在整个应用程序中可用了。 另外，这些服务提供商也同样对整个应用中的类是可用的 —— 只要它们有供查找用的服务令牌。

你应该始终在根注入器中提供这些服务 —— 除非你希望该服务只有在消费方要导入特定的 `@NgModule` 时才生效。

`providedIn` 与 NgModule

也可以规定某个服务只有在特定的 `@NgModule` 中提供。比如，如果你你希望只有当消费方导入了你创建的 `UserModule` 时才让 `UserService` 在应用中生效，那就可以指定该服务要在该模块中提供：

src/app/user.service.1.ts

```
import { Injectable } from '@angular/core';
import { UserModule } from './user.module';

@Injectable({
  providedIn: UserModule,
})
export class UserService {
}
```

上面的例子展示的就是在模块中提供服务的首选方式。之所以推荐该方式，是因为当没有人注入它时，该服务就可以被摇树优化掉。如果没办法指定哪个模块该提供这个服务，你也可以在那个模块中为该服务声明一个提供商：

src/app/user.module.ts

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

import { UserService } from './user.service';

@NgModule({
  providers: [UserService],
})
export class UserModule {
}
```

使用惰性加载模块限制提供商的作用域

在 CLI 生成的基本应用中，模块是立即加载的，这意味着它们都是由本应用启动的，Angular 会使用一个依赖注入体系来让一切服务都在模块间有效。对于立即加载式应用，应用中的根注入器会让所有服务提供商都对整个应用有效。

当使用惰性加载时，这种行为需要进行改变。惰性加载就是只有当需要时才加载模块，比如路由中。它们没办法像立即加载模块那样进行加载。这意味着，在它们的 `providers` 数组中列出的服务都是不可用的，因为根注入器并不知道这些模块。

当 Angular 的路由器惰性加载一个模块时，它会创建一个新的注入器。这个注入器是应用的根注入器的一个子注入器。想象一棵注入器树，它有唯一的根注入器，而每一个惰性加载模块都有一个自己的子注入器。路由器会把根注入器中的所有提供商添加到子注入器中。如果路由器在惰性加载时创建组件， Angular 会更倾向于使用从这些提供商中创建的服务实例，而不是来自应用的根注入器的服务实例。

任何在惰性加载模块的上下文中创建的组件（比如路由导航），都会获取该服务的局部实例，而不是应用的根注入器中的实例。而外部模块中的组件，仍然会收到来自于应用的根注入器创建的实例。

虽然你可以使用惰性加载模块来提供实例，但不是所有的服务都能惰性加载。比如，像路由之类的模块只能在根模块中使用。路由器需要使用浏览器中的全局对象 `location` 进行工作。

使用组件限定服务提供商的作用域

另一种限定提供商作用域的方式是把要限定的服务添加到组件的 `providers` 数组中。组件中的提供商和 NgModule 中的提供商是彼此独立的。 当你要立即加载一个自带了全部所需服务的模块时，这种方式是有帮助的。 在组件中提供服务，会限定该服务只能在该组件中有效（同一模块中的其它组件不能访问它）。

src/app/app.component.ts

```
@Component({
/* . . . */
  providers: [UserService]
})
```

在模块中提供服务还是在组件中?

通常，要在根模块中提供整个应用都需要的服务，在惰性加载模块中提供限定范围的服务。

路由器工作在根级，所以如果你把服务提供商放进组件（即使是 `AppComponent`）中，那些依赖于路由器的惰性加载模块，将无法看到它们。

当你必须把一个服务实例的作用域限定到组件及其组件树中时，可以使用组件注册一个服务提供商。 比如，用户编辑组件 `UserEditorComponent`，它需要一个缓存 `UserService` 实例，那就应该把 `UserService` 注册进 `UserEditorComponent` 中。 然后，每个 `UserEditorComponent` 的实例都会获取它自己的缓存服务实例。

关于 NgModule 的更多知识

你还可能对下列内容感兴趣：

- 单例服务详细解释了本页包含的那些概念。
- 惰性加载模块。
- 可摇树优化的服务提供商。
- NgModule 常见问题。

单例应用

前提条件：

- 对引导有基本的了解。
- 熟悉服务提供商。

本页中描述的这种全应用级单例服务的例子位于在线例子 / 下载范例，它示范了 NgModule 的所有已文档化的特性。

提供单例服务

在 Angular 中有两种方式来生成单例服务：

- 声明该服务应该在应用的根上提供。
- 把该服务包含在 `AppModule` 或某个只会被 `AppModule` 导入的模块中。

从 Angular 6.0 开始，创建单例服务的首选方式是在那个服务类上指定它应该在应用的根上提供。只要在该服务的 `@Injectable` 装饰器上把 `providedIn` 设置为 `root` 就可以了：

src/app/user.service.0.ts

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

@Injectable({
  providedIn: 'root',
})
export class UserService {
}
```

要想深入了解关于服务的信息，参见《英雄指南》教程中的服务一章。

`forRoot()`

如果某个模块同时提供了服务提供商和可声明对象（组件、指令、管道），那么当在某个子注入器中加载它的时候（比如路由），就会生成多个该服务提供商的实例。 而存在多个实例会导致一些问题，因为这些实例

会屏蔽掉根注入器中该服务提供商的实例，而它的本意可能是作为单例对象使用的。 因此，Angular 提供了一种方式来把服务提供商从该模块中分离出来，以便该模块既可以带着 `providers` 被根模块导入，也可以不带 `providers` 被子模块导入。

1. 在该模块上创建一个静态方法 `forRoot()`（习惯名称）。
2. 把那些服务提供商放进 `forRoot` 方法中，参见下面的例子。

以 `RouterModule` 为例具体说说。`RouterModule` 要提供 `Router` 服务，还要提供 `RouterOutlet` 指令。`RouterModule` 要由根应用模块导入，以便该应用拥有一个路由器，而且它还需要至少一个 `RouterOutlet`。`RouterModule` 还必须由各个独立的路由组件导入，让它们能在自己的模板中使用 `RouterOutlet` 指令来支持其子路由。

如果 `RouterModule` 没有 `forRoot()`，那么每个路由组件都会创建一个新的 `Router` 实例。这将会破坏整个应用，因为应用中只能有一个 `Router`。 `RouterModule` 拥有 `RouterOutlet` 指令，它应该随处可用，但是 `Router` 只能有一个，它应该在 `forRoot()` 中提供。 最终的结果就是，应用的根模块导入了 `RouterModule.forRoot(...)` 以获取 `Router`，而所有路由组件都导入了 `RouterModule`，它不包括这个 `Router` 服务。

如果你有一个同时提供服务提供商和可声明对象的模块，请使用下面的模式把它们分离开。

那些需要把服务提供商加到应用中的模块可以通过某种类似 `forRoot()` 方法的方式配置那些服务提供商。

`forRoot()` 接收一个服务配置对象，然后返回一个 ModuleWithProviders ， 它是一个带有下列属性的简单对象：

- `ngModule`： 在这个例子中就是 `CoreModule` 类
- `providers` - 配置好的服务提供商

在这个在线例子 / 下载范例中，根 `AppModule` 导入了 `CoreModule`，并把它的 `providers` 添加到了 `AppModule` 的服务提供商中。 特别是，Angular 会在 `@NgModule.providers` 前面添加这些导入的服务提供商。 这种顺序保证了 `AppModule` 中的服务提供商总是会优先于那些从其它模块中导入的服务提供商。

应该只在 `AppModule` 中导入 `CoreModule` 并只使用一次 `forRoot()` 方法，因为该方法中会注册服务，而你希望那些服务在该应用中只注册一次。 如果你多次注册它们，就可能会得到该服务的多个实例，并导致运行时错误。

你还可以在 `CoreModule` 中添加一个用于配置 `UserService` 的 `forRoot()` 方法。

在下面的例子中，可选的注入 `UserServiceConfig` 扩展了 `Core` 模块中的 `UserService`。如果 `UserServiceConfig` 存在，就从这个配置中设置用户名。

src/app/core/user.service.ts (constructor)

```
constructor(@Optional() config: UserServiceConfig) {
  if (config) { this._userName = config.userName; }
}
```

下面是一个接受 `UserServiceConfig` 参数的 `forRoot()` 方法：

src/app/core/core.module.ts (forRoot)

```
static forRoot(config: UserServiceConfig): ModuleWithProviders {
  return {
    ngModule: CoreModule,
    providers: [
      {provide: UserServiceConfig, useValue: config }
    ]
  };
}
```

最后，在 `AppModule` 的 `imports`**列表**中调用它。

src/app/app.module.ts (imports)

```
import { CoreModule } from './core/core.module';
/* . . . */
@NgModule({
  imports: [
    BrowserModule,
    ContactModule,
    CoreModule.forRoot({userName: 'Miss Marple'}),
    AppRoutingModule
  ],
/* . . . */
})
export class AppModule { }
```

该应用不再显示默认的 “Sherlock Holmes”，而是用 “Miss Marple” 作为用户名称。

记住，在文件顶部使用 JavaScript 的 `import` 语句**导入** `CoreModule`，但不要在多于一个 `@NgModule` 的 `imports` 列表中添加它。

防止重复导入 `CoreModule`

只有根模块 `AppModule` 才能导入 `CoreModule`。如果一个惰性加载模块也导入了它， 该应用就会为服务生成多个实例。

要想防止惰性加载模块重复导入 `CoreModule`，可以添加如下的 `CoreModule` 构造函数。

src/app/core/core.module.ts

```
constructor (@Optional() @SkipSelf() parentModule: CoreModule) {
  if (parentModule) {
    throw new Error(
      'CoreModule is already loaded. Import it in the AppModule only');
  }
}
```

这个构造函数要求 Angular 把 `CoreModule` 注入到它自己。 如果 Angular 在**当前**注入器中查找 `CoreModule`，这个注入过程就会陷入死循环。 而 `@SkipSelf` 装饰器表示 "在注入器树中那些高于我的祖先注入器中查找 `CoreModule`"。

如果构造函数在 `AppModule` 中执行，那就没有祖先注入器能提供 `CoreModule` 的实例，于是注入器就会放弃查找。

默认情况下，当注入器找不到想找的提供商时，会抛出一个错误。 但 `@Optional` 装饰器表示找不到该服务也无所谓。 于是注入器会返回 `null`，`parentModule` 参数也就被赋成了空值，而构造函数没有任何异常。

但如果你把 `CoreModule` 导入到像 `CustomerModule` 这样的惰性加载模块中，事情就不一样了。

Angular 会创建一个惰性加载模块，它具有自己的注入器，它是根注入器的**子注入器**。 `@SkipSelf` 让 Angular 在其父注入器中查找 `CoreModule`，这次，它的父注入器却是根注入器了（而上次的父注入器是空）。 当然，这次它找到了由根模块 `AppModule` 导入的实例。 该构造函数检测到存在 `parentModule`，于是抛出一个错误。

以下这两个文件仅供参考：

app.module.ts *core.module.ts*

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

/* App Root */
import { AppComponent } from './app.component';

/* Feature Modules */
import { ContactModule } from './contact/contact.module';
import { CoreModule } from './core/core.module';

/* Routing Module */
import { AppRoutingModule } from './app-routing.module';
```

```
@NgModule({
  imports: [
    BrowserModule,
    ContactModule,
    CoreModule.forRoot({userName: 'Miss Marple'}),
    AppRoutingModule
  ],
  providers: [],
  declarations: [
    AppComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

关于 NgModule 的更多知识

你还可能对下列内容感兴趣:

- 共享模块解释了本页中涉及的这些概念。
- 惰性加载模块。
- NgModule 常见问题。

惰性加载的特性模块

前提条件

对下列知识有基本的了解：

- 特性模块
- JavaScript 模块与 NgModules。
- 常用模块。
- 特性模块的分类。
- 路由与导航。

如果需要本页描述的具有两个惰性加载模块的范例应用，参见在线例子 / 下载范例。

高层视角

要想建立一个惰性加载的特性模块，有三个主要步骤：

1. 创建该特性模块。
2. 创建该特性模块的路由模块。
3. 配置相关路由。

建立应用

如果你还没有应用，可以遵循下面的步骤使用 CLI 创建一个。如果已经有了，可以直接跳到 配置路由部分。输入下列命令，其中的 `customer-app` 表示你的应用名称：

```
ng new customer-app --routing
```

这会创建一个名叫 `customer-app` 的应用，而 `--routing` 标识生成了一个名叫 `app-routing.module.ts` 的文件，它是你建立惰性加载的特性模块时所必须的。 输入命令 `cd customer-app` 进入该项目。

创建一个带路由的特性模块

接下来，你需要一个要路由到的特性模块。要生成一个，请输入下列命令，其中的 `customers` 是该模块的名字：

```
ng generate module customers --routing
```

这会创建一个 `customers` 目录，其中有两个文件：`CustomersModule` 和 `CustomersRoutingModule`。`CustomersModule` 扮演的是与客户紧密相关的所有事物的管理员。`CustomersRoutingModule` 则会处理任何与客户有关的路由。 这样就可以在应用不断成长时保持应用的良好结构，并且当复用本模块时，你可以轻松的让其路由保持完好。

CLI 会把 `CustomersRoutingModule` 自动导入到 `CustomersModule`。它会在文件的顶部添加一条 JavaScript 的 `import` 语句，并把 `CustomersRoutingModule` 添加到 `@NgModule` 的 `imports` 数组中。

向特性模块中添加组件

要想在浏览器中看出该模块惰性加载成功了，就创建一个组件用来在应用加载 `CustomersModule` 之后渲染出一些 HTML。在命令行中输入如下命令：

```
ng generate component customers/customer-list
```

这会在 `customers` 目录中创建一个名叫 `customer-list` 的文件夹，其中包含该组件的四个文件。

就像路由模块一样，CLI 也自动把 `CustomerListComponent` 导入了 `CustomersModule`。

再添加一个特性模块

为了提供另一个可路由到的地点，再创建第二个带路由的特性模块：

```
ng generate module orders --routing
```

这会创建一个名叫 `orders` 的新文件夹，其中包含 `OrdersModule` 和 `OrdersRoutingModule`。

现在，像 `CustomersModule` 一样，给它添加一些内容：

```
ng generate component orders/order-list
```

建立 UI

虽然你也可以在地址栏中输入 URL，不过导航菜单会更好用，而且更常见。 把 `app.component.html` 中的占位脚本替换成一个自定义的导航，以便你在浏览器中能轻松地在模块之间导航。

src/app/app.component.html

```html
<h1>
  {{title}}
</h1>

<button routerLink="/customers">Customers</button>
<button routerLink="/orders">Orders</button>
<button routerLink="">Home</button>

<router-outlet></router-outlet>
```

要想在浏览器中看到你的应用，就在终端窗口中输入下列命令：

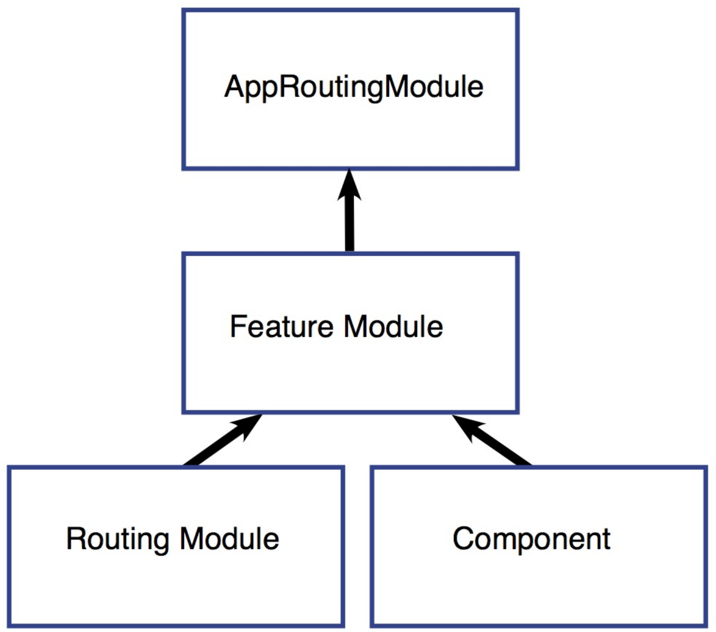
```
ng serve
```

然后，跳转到 `localhost:4200`，这时你应该看到 “app works!” 和三个按钮。

要想让这些按钮生效，你需要配置一下这些路由模块。

配置路由

这两个特性模块（`OrdersModule` 和 `CustomersModule`）应该挂接到 `AppRoutingModule` 中，来让路由器知道它们。其结构如下：



每个特性模块都是路由器中的一个“门口”。在 `AppRoutingModule` 中，你配置了一些路由指向这些特性模块（即 `OrderModule` 和 `CustomersModule`）。通过这种方式，路由器就知道了如何跳转到特性模块。然后，特性模块就把 `AppRoutingModule` 和 `CustomersRoutingModule` 或 `OrdersRoutingModule` 连接到一起。这些路由模块会告诉路由器该到哪里去加载相应的组件。

顶层的路由

在 `AppRoutingModule` 中，把 `routes` 数组修改成这样：

src/app/app-routing.module.ts

```
const routes: Routes = [
  {
    path: 'customers',
    loadChildren: 'app/customers/customers.module#CustomersModule'
  },
  {
    path: 'orders',
    loadChildren: 'app/orders/orders.module#OrdersModule'
  },
  {
    path: '',
    redirectTo: '',
    pathMatch: 'full'
  }
];
```

这些 `import` 语句没有变化。前两个路径分别路由到了 `CustomersModule` 和 `OrdersModule`。注意看惰性加载的语法：`loadChildren` 后面紧跟着一个字符串，它指向模块路径，然后是一个 `#`，然后是该模块的类名。

特性模块内部

接下来看看 `customers.module.ts`。如果你使用的是 CLI，并遵循本页面中给出的步骤，那么在这里你不必做任何事。 特性模块就像是 `AppRoutingModule` 和该特性自己的路由模块之间的连接器。 `AppRoutingModule` 导入了特性模块 `CustomersModule`，而 `CustomersModule` 又导入了 `CustomersRoutingModule`。

src/app/customers/customers.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CustomersRoutingModule } from './customers-routing.module';
import { CustomerListComponent } from './customer-list/customer-list.component';

@NgModule({
  imports: [
    CommonModule,
    CustomersRoutingModule
  ],
  declarations: [CustomerListComponent]
})
export class CustomersModule { }
```

`customers.module.ts` 文件导入了 `CustomersRoutingModule` 和 `CustomerListComponent`，所以 `CustomersModule` 类可以访问它们。 接着 `CustomersRoutingModule` 出现在了 `@NgModule` 的 `imports` 数组中，这让 `CustomersModule` 可以访问它的路由模块。而 `CustomerListComponent` 出现在了 `declarations` 数组中，这表示 `CustomerListComponent` 属于 `CustomersModule`。

配置该特性模块的路由

接下来的步骤位于 `customers-routing.module.ts` 中。首先，在文件的顶部使用 JS 的 `import` 语句导入该组件。然后添加指向 `CustomerListComponent` 的路由。

src/app/customers/customers-routing.module.ts

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

import { CustomerListComponent } from './customer-list/customer-list.component';


const routes: Routes = [
  {
    path: '',
    component: CustomerListComponent
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class CustomersRoutingModule { }
```

注意，`path` 被设置成了空字符串。这是因为 `AppRoutingModule` 中的路径已经设置为了 `customers`，所以 `CustomersRoutingModule` 中的这个路由定义已经位于 `customers` 这个上下文中了。也就是说这个路由模块中的每个路由其实都是子路由。

重复这个步骤以导入 `OrdersListComponent`，并为 `orders-routing.module.ts` 配置路由树组：

src/app/orders/orders-routing.module.ts (excerpt)

```
import { OrderListComponent } from './order-list/order-list.component';

const routes: Routes = [
  {
    path: '',
    component: OrderListComponent
  }
];
```

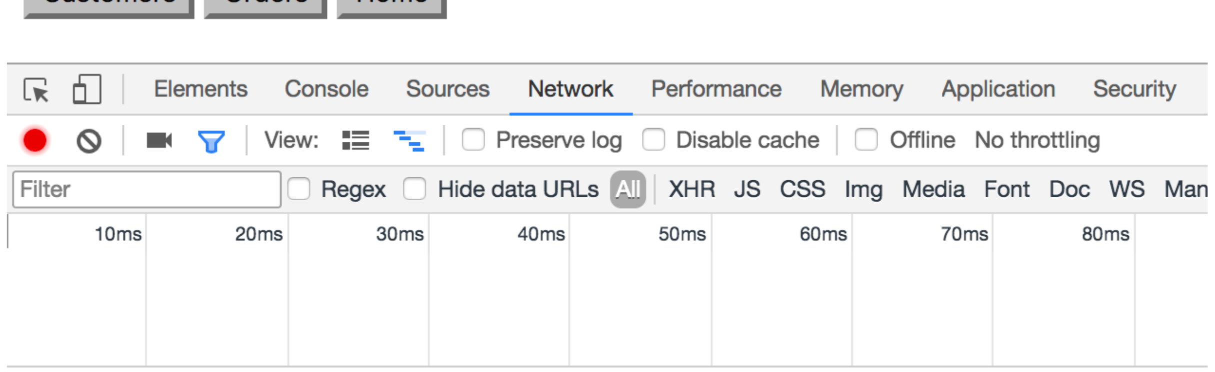
现在，如果你在浏览器中查看该应用，这三个按钮会把你带到每个模块去。

确认它工作正常

你可以使用 Chrome 开发者工具来确认一下这些模块真的是惰性加载的。 在 Chrome 中，按 `Cmd+Option+i`（Mac）或 `Ctrl+Alt+i`（PC），并选中 `Network` 页标签。

app works!

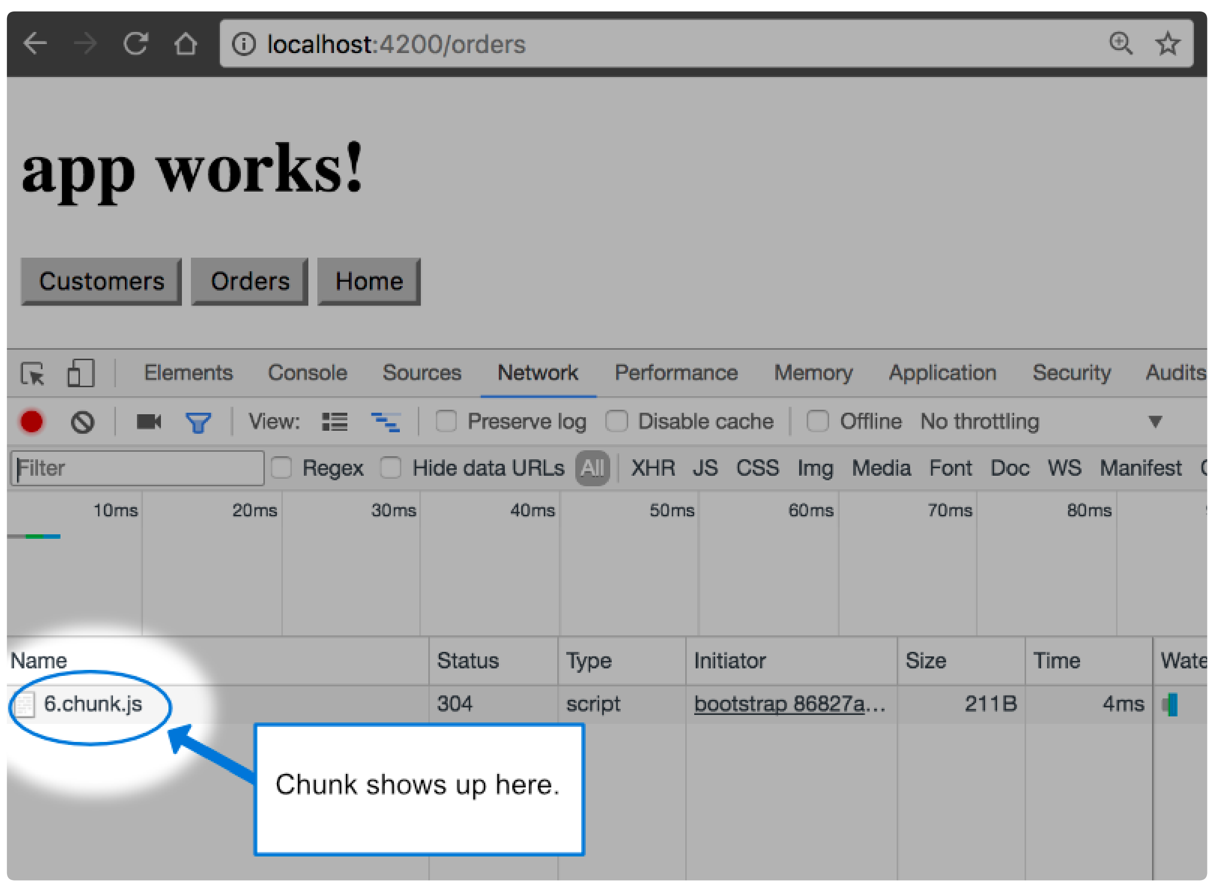
Customers Orders Home



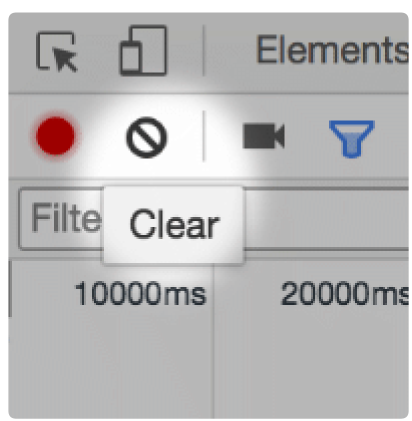
Recording network activity…

Perform a request or hit ⌘ R to record the reload.

点击 Orders 或 Customers 按钮。如果你看到某个 chunk 文件出现了，就表示你已经惰性加载并接入了这个特性模块。Orders 和 Customers 都应该出现一次 chunk，并且它们各自只应该出现一次。



要想再次查看它或测试本项目后面的行为，只要点击 Network 页左上放的“清除”图标即可。



然后，使用 `Cmd+r`（Mac）或 `Ctrl+r`（PC）重新加载页面。

`forRoot()` 与 `forChild()`

你可能已经注意到了，CLI 会把 `RouterModule.forRoot(routes)` 添加到 `app-routing.module.ts` 的 `imports` 数组中。这会让 Angular 知道 `AppRoutingModule` 是一个路由模块，而 `forRoot()` 表示这是一个根路由模

块。 它会配置你传入的所有路由、让你能访问路由器指令并注册 `RouterService`。 在 `AppRoutingModule` 中使用 `forRoot()`，在本应用中这只会在顶层模块中写一次。

CLI 还会把 `RouterModule.forChild(routes)` 添加到各个特性模块中。这种方式下 Angular 就会知道这个路由列表只负责提供额外的路由并且其设计意图是作为特性模块使用。你可以在多个模块中使用 `forChild()`。

`forRoot()` 包含的注入器配置是全局性的，比如对路由器的配置。`forChild()` 中没有注入器配置，只有像 `RouterOutlet` 和 `RouterLink` 这样的指令。

更多关于 NgModule 和路由的知识

你可能还对下列内容感兴趣：

- 路由与导航。
- 服务提供商。
- 特性模块的分类。

共享特性模块

前提条件

对下列知识有基本的了解：

- 特性模块
- JavaScript 模块与 NgModules。
- 常用模块。
- 路由与导航。
- 惰性加载模块。

创建共享模块能让你更好地组织和梳理代码。你可以把常用的指令、管道和组件放进一个模块中，然后在应用中其它需要这些的地方导入该模块。

想象某个应用有下列模块：

```
1. import { CommonModule } from '@angular/common';
2. import { NgModule } from '@angular/core';
3. import { FormsModule } from '@angular/forms';
4. import { CustomerComponent } from './customer.component';
5. import { NewItemDirective } from './new-item.directive';
6. import { OrdersPipe } from './orders.pipe';
7.
8. @NgModule({
9.  imports:      [ CommonModule ],
10.  declarations: [ CustomerComponent, NewItemDirective, OrdersPipe ],
11.  exports:      [ CustomerComponent, NewItemDirective, OrdersPipe,
12.                  CommonModule, FormsModule ]
13. })
14. export class SharedModule { }
```

请注意以下几点：

- 它导入了 `CommonModule`，因为该模块需要一些常用指令。
- 它声明并导出了一些工具性的管道、指令和组件类。
- 它重新导出了 `CommonModule` 和 `FormsModule`

通过重新导出 `CommonModule` 和 `FormsModule`，任何导入了这个 `SharedModule` 的其它模块，就都可以访问来自 `CommonModule` 的 `NgIf` 和 `NgFor` 等指令了，也可以绑定到来自 `FormsModule` 中的 `[(ngModel)]` 的属性

了。

即使 `SharedModule` 中声明的组件没有绑定过 `[(ngModel)]`，而且`SharedModule` 也不需要导入 `FormsModule`，`SharedModule` 仍然可以导出 `FormsModule`，而不必把它列在 `imports` 中。 这种方式下，你可以让其它模块也能访问 `FormsModule`，而不用直接在自己的 `@NgModule` 装饰器中导入它。

使用来自其它模块的组件和服务

在使用来自其它模块的组件和来自其它模块的服务时，有一个很重要的区别。 当你要使用指令、管道和组件时，导入那些模块就可以了。而导入带有服务的模块意味着你会拥有那个服务的一个新实例，这通常不会是你想要的结果（你通常会想取到现存的服务）。使用模块导入来控制服务的实例化。

获取共享服务的最常见方式是通过 Angular 的依赖注入系统，而不是模块系统（导入模块将导致创建新的服务实例，那不是典型的用法）。

要进一步了解共享服务，参见服务提供商。

关于 NgModule 的更多知识

你可能还对下列内容感兴趣：

- 服务提供商。
- 特性模块的分类。

NgModule API

前提条件

对下列概念有基本的理解：

- 引导启动。
- JavaScript 模块与 NgModules。

`@NgModule` 的设计意图

宏观来讲，NgModule 是组织 Angular 应用的一种方式，它们通过 `@NgModule` 装饰器中的元数据来实现这一点。 这些元数据可以分成三类：

- 静态的：编译器配置，用于告诉编译器指令的选择器并通过选择器匹配的方式决定要把该指令应用到模板中的什么位置。它是通过 `declarations` 数组来配置的。
- 运行时：通过 `providers` 数组提供给注入器的配置。
- 组合/分组：通过 `imports` 和 `exports` 数组来把多个 NgModule 放在一起，并彼此可用。

```
@NgModule({
  // Static, that is compiler configuration
  declarations: [], // Configure the selectors
  entryComponents: [], // Generate the host factory

  // Runtime, or injector configuration
  providers: [], // Runtime injector configuration

  // Composability / Grouping
  imports: [], // composing NgModules together
  exports: [] // making NgModules available to other parts of the app
})
```

`@NgModule` 元数据

下面是 `@NgModule` 元数据中属性的汇总表：

属性	说明

`declarations`	**属于该模块的**可声明对象（**组件、指令**和**管道**）的列表。
	1. 当编译模板时，你需要确定一组选择器，它们将用于触发相应的指令。
	2. 该模板在 NgModule 环境中编译 —— 模板的组件是在该 NgModule 内部声明的，它会使用如下规则来确定这组选择器：
	    ◦ 列在 `declarations` 中的所有指令选择器。
	    ◦ 从所导入的 NgModule 中导出的那些指令的选择器。
	
	组件、指令和管道**只能**属于一个模块。 如果尝试把同一个类声明在多个模块中，编译器就会报告一个错误。
	不要重复声明从其它模块中导入的类。

`providers`	依赖注入提供商的列表。 Angular 会使用该模块的注入器注册这些提供商。 如果该模块是启动模块，那就会使用根注入器。 当需要注入到任何组件、指令、管道或服务时，这些服务对于本注入器的子注入器都是可用的。 惰性加载模块有自己的注入器，它通常是应用的根注入器的子注入器。 惰性加载的服务是局限于这个惰性加载模块的注入器中的。 如果惰性加载模块也提供了 `UserService`，那么在这个模块的上下文中创建的任何组件（比如在路由器导航时），都会获得这个服务的本模块内实例，而不是来自应用的根注入器的实例。 其它外部模块中的组件也会使用它们自己的注入器提供的服务实例。 要深入了解关于多级注入器及其作用域，参见服务提供商。
`imports`	要折叠（Folded）进本模块中的其它模块。折叠的意思是从被导入的模块中导出的那些软件资产同样会被声明在这里。 特别是，这里列出的模块，其导出的组件、指令或管道，当在组件模板中被引用时，和本模块自己声明的那些是等价的。 组件模板可以引用其它组件、指令或管道，不管它们是在本模块中声明的，还是从导入的模块中导出的。 比如，只有当该模块导入了 Angular 的 `CommonModule（也可能从`BrowserModule`中间接导入）时，组件才能使用`NgIf`和`NgFor` 指令。 你可以从 `CommonModule` 中导入很多标准指令，不过也有些常用的指令属于其它模块。 比如，你只有导入了 Angular 的 `FormsModule` 时才能使用 `[(ngModel)]`。

`exports`	可供导入了自己的模块使用的可声明对象（组件、指令、管道类）的列表。 导出的可声明对象就是本模块的**公共 API**。 只有当其它模块导入了本模块，并且本模块导出了 `UserComponent` 时，其它模块中的组件才能使用**本**模块中的 `UserComponent`。 默认情况下这些可声明对象都是私有的。 如果本模块**没有**导出 `UserComponent`，那么就只有**本**模块中的组件才能使用 `UserComponent`。 导入某个模块**并不会**自动重新导出被导入模块的那些导入。 模块 B 不会因为它导入了模块 A，而模块 A 导入了 `CommonModule` 而能够使用 `ngIf`。 模块 B 必须自己导入 `CommonModule`。 一个模块可以把另一个模块加入自己的 `exports` 列表中，这时，另一个模块的所有公共组件、指令和管道都会被导出。 重新导出可以让模块被显式传递。 如果模块 A 重新导出了 `CommonModule`，而模块 B 导入了模块 A，那么模块 B 就可以使用 `ngIf` 了 —— 即使它自己没有导入 `CommonModule`。
`bootstrap`	要自动启动的组件列表。 通常，在这个列表中只有一个组件，也就是应用的**根组件**。 Angular 也可以引导多个引导组件，它们每一个都在宿主页面中有自己的位置。 启动组件会自动添加到 `entryComponents` 中。
`entryComponents`	那些可以动态加载进视图的组件列表。 默认情况下，Angular 应用至少有一个入口组件，也就是根组件 `AppComponent`。 它用作进入该应用的入口点，也就是说你通过引导它来启动本应用。 路由组件也是**入口组件**，因为你需要动态加载它们。 路由器创建它们，并把它们扔到 DOM 中的 `<router-outlet>` 附近。 虽然引导组件和路由组件都是**入口组件**，不过你不用自己把它们加到模块的 `entryComponents` 列表中，因为它们会被隐式添加进去。 Angular 会自动把模块的 `bootstrap` 中的组件和路由定义中的组件添加到 `entryComponents` 列表。 而那些使用不易察觉的`ViewComponentRef.createComponent()`的方式进行命令式引导的组件仍然需要添加。 动态组件加载在除路由器之外的大多数应用中都不太常见。如果你需要动态加载组件，就必须自己把那些组件添加到 `entryComponents` 列表中。 要了解更多，参见入口组件一章。

关于 NgModule 的更多知识

你可能还对下列内容感兴趣：

- 特性模块
- 入口组件
- 服务提供商。
- 特性模块的分类。

Angular 模块常见问题

前提条件:

对下列概念有基本的理解:

- Angular 模块.

NgModules 可以帮你把应用组织成一些紧密相关的代码块。

这里回答的是开发者常问起的关于 Angular 模块的设计与实现问题。

我应该把哪些类加到 **declarations** 中?

把可声明的类（组件、指令和管道）添加到 `declarations` 列表中。

这些类只能在应用程序的**一个并且只有一个**模块中声明。 只有当它们**从属于**某个模块时，才能把在**此**模块中声明它们。

什么是**可声明的**?

可声明的就是组件、指令和管道等可以被加到模块的 `declarations` 列表中的类。它们也是**所有**能被加到 `declarations` 中的类。

哪些类**不**应该加到 `declarations` 中?

只有可声明的类才能加到模块的 `declarations` 列表中。

不要声明:

- 已经在其它模块中声明过的类。无论它来自应用自己的模块（@NgModule）还是第三方模块。
- 从其它模块中导入的指令。例如，不要声明来自 `@angular/forms` 的 FORMS_DIRECTIVES，因为 `FormsModule` 已经声明过它们了。
- 模块类。
- 服务类
- 非 Angular 的类和对象，比如：字符串、数字、函数、实体模型、配置、业务逻辑和辅助类。

为什么要把同一个组件声明在不同的 NgModule 属性中？

`AppComponent` 经常被同时列在 `declarations` 和 `bootstrap` 中。 另外你还可能看到 `HeroComponent` 被同时列在 `declarations`、`exports` 和 `entryComponent` 中。

这**看起来**是多余的，不过这些函数具有不同的功能，从它出现在一个列表中无法推断出它也应该在另一个列表中。

- `AppComponent` 可能被声明在此模块中，但可能不是引导组件。
- `AppComponent` 可能在此模块中引导，但可能是由另一个特性模块声明的。
- `HeroComponent` 可能是从另一个应用模块中导入的（所以你没法声明它）并且被当前模块重新导出。
- `HeroComponent` 可能被导入，以便用在外部组件的模板中，但也可能同时被一个弹出式对话框加载。

"Can't bind to 'x' since it isn't a known property of 'y'"是什么意思？

这个错误通常意味着你或者忘了声明指令“x”，或者你没有导入“x”所属的模块。

> 如果“x”其实不是属性，或者是组件的私有属性（比如它不带 `@Input` 或 `@Output` 装饰器），那么你也同样会遇到这个错误。

我应该导入什么？

导入你需要在当前模块的组件模板中使用的那些公开的（被导出的）可声明类。

这意味着要从 `@angular/common` 中导入 `CommonModule` 才能访问 Angular 的内置指令，比如 `NgIf` 和 `NgFor`。 你可以直接导入它或者从重新导出过该模块的其它模块中导入它。

如果你的组件有 `[(ngModel)]` 双向绑定表达式，就要从 `@angular/forms` 中导入 `FormsModule`。

如果当前模块中的组件包含了**共享**模块和**特性**模块中的组件、指令和管道，就导入这些模块。

只能在根模块 `AppModule` 中导入**BrowserModule**。

我应该导入 BrowserModule 还是 CommonModule？

几乎所有要在浏览器中使用的应用的根模块（`AppModule`）都应该从 `@angular/platform-browser` 中导入 `BrowserModule`。

`BrowserModule` 提供了启动和运行浏览器应用的那些基本的服务提供商。

`BrowserModule` 还从 `@angular/common` 中重新导出了 `CommonModule`，这意味着 `AppModule` 中的组件也同样可以访问那些每个应用都需要的 Angular 指令，如 `NgIf` 和 `NgFor`。

在其它任何模块中都**不要导入**`BrowserModule`。 **特性模块**和**惰性加载模块**应该改成导入 `CommonModule`。 它们需要通用的指令。它们不需要重新初始化全应用级的提供商。

特性模块中导入 `CommonModule` 可以让它能用在任何目标平台上，不仅是浏览器。那些跨平台库的作者应该喜欢这种方式的。

如果我两次导入同一个模块会怎么样?

没有任何问题。当三个模块全都导入模块'A'时，Angular 只会首次遇到时加载一次模块'A'，之后就不会这么做了。

无论 `A` 出现在所导入模块的哪个层级，都会如此。 如果模块'B'导入模块'A'、模块'C'导入模块'B'，模块'D'导入 `[C, B, A]`，那么'D'会触发模块'C'的加载，'C'会触发'B'的加载，而'B'会加载'A'。 当 Angular 在'D'中想要获取'B'和'A'时，这两个模块已经被缓存过了，可以立即使用。

Angular 不允许模块之间出现循环依赖，所以不要让模块'A'导入模块'B'，而模块'B'又导入模块'A'。

我应该导出什么?

导出那些**其它模块**希望在自己的模板中引用的可声明类。这些也是你的**公共**类。 如果你不导出某个类，它就是**私有的**，只对当前模块中声明的其它组件可见。

你**可以**导出任何可声明类（组件、指令和管道），而不用管它是声明在当前模块中还是某个导入的模块中。

你**可以**重新导出整个导入过的模块，这将导致重新导出它们导出的所有类。重新导出的模块甚至不用先导入。

我**不应该**导出什么?

不要导出：

- 那些你只想在当前模块中声明的那些组件中使用的私有组件、指令和管道。如果你不希望任何模块看到它，就不要导出。
- 不可声明的对象，比如服务、函数、配置、实体模型等。
- 那些只被路由器或引导函数动态加载的组件。 比如入口组件可能从来不会在其它组件的模板中出现。导出它们没有坏处，但也没有好处。
- 纯服务模块没有公开（导出）的声明。 例如，没必要重新导出 `HttpClientModule`，因为它不导出任何东西。 它唯一的用途是一起把 http 的那些服务提供商添加到应用中。

我可以重新导出类和模块吗?

毫无疑问!

模块是从其它模块中选取类并把它们重新导出成统一、便利的新模块的最佳方式。

模块可以重新导出其它模块，这会导致重新导出它们导出的所有类。 Angular 自己的 `BrowserModule` 就重新导出了一组模块，例如:

```
exports: [CommonModule, ApplicationModule]
```

模块还能导出一个组合，它可以包含自己的声明、某些导入的类以及导入的模块。

不要费心去导出纯服务类。 纯服务类的模块不会导出任何可供其它模块使用的可声明类。 例如，不用重新导出 `HttpClientModule`，因为它没有导出任何东西。 它唯一的用途是把那些 http 服务提供商一起添加到应用中。

forRoot()方法是什么?

静态方法 `forRoot()` 是一个约定，它可以让开发人员更轻松的配置模块的想要单例使用的服务及其提供商。 `RouterModule.forRoot()` 就是一个很好的例子。

应用把一个 `Routes` 对象传给 `RouterModule.forRoot()`，为的就是使用路由配置全应用级的 `Router` 服务。 `RouterModule.forRoot()` 返回一个ModuleWithProviders对象。 你把这个结果添加到根模块 `AppModule` 的 `imports` 列表中。

只能在应用的根模块 `AppModule` 中调用并导入 `.forRoot()` 的结果。 在其它模块中导入它，特别是惰性加载模块中，是违反设计目标的并会导致一个运行时错误。 要了解更多，参见单例服务。

对于服务来说，除了可以使用 `forRoot()`外，更好的方式是在该服务的 `@Injectable()` 装饰器中指定 `providedIn: 'root'`，它让该服务自动在全应用级可用，这样它也就默认是单例的。

`RouterModule` 也提供了静态方法 `forChild`，用于配置惰性加载模块的路由。

`forRoot()` 和 `forChild()` 都是约定俗成的方法名，它们分别用于在根模块和特性模块中配置服务。

Angular 并不识别这些名字，但是 Angular 的开发人员可以。 当你写类似的需要可配置的服务提供商时，请遵循这个约定。

为什么服务提供商在特性模块中的任何地方都是可见的？

列在引导模块的 `@NgModule.providers` 中的服务提供商具有全应用级作用域。 往 `NgModule.providers` 中添加服务提供商将导致该服务被发布到整个应用中。

当你导入一个模块时，Angular 就会把该模块的服务提供商（也就是它的 `providers` 列表中的内容）加入该应用的**根注入器**中。

这会让该提供商对应用中所有知道该提供商令牌（token）的类都可见。

Angular 就是如此设计的。 通过模块导入来实现可扩展性是 Angular 模块系统的主要设计目标。 把模块的提供商并入应用程序的注入器可以让库模块使用新的服务来强化应用程序变得更容易。 只要添加一次 `HttpClientModule`，那么应用中的每个组件就都可以发起 Http 请求了。

不过，如果你期望模块的服务只对那个特性模块内部声明的组件可见，那么这可能会带来一些不受欢迎的意外。 如果 `HeroModule` 提供了一个 `HeroService`，并且根模块 `AppModule` 导入了 `HeroModule`，那么任何知道 `HeroService`**类型**的类都可能注入该服务，而不仅是在 `HeroModule` 中声明的那些类。

为什么在惰性加载模块中声明的服务提供商只对该模块自身可见？

和启动时就加载的模块中的提供商不同，惰性加载模块中的提供商是**局限于模块**的。

当 Angular 路由器惰性加载一个模块时，它创建了一个新的运行环境。 那个环境拥有自己的注入器，它是应用注入器的直属子级。

路由器把该惰性加载模块的提供商和它导入的模块的提供商添加到这个子注入器中。

这些提供商不会被拥有相同令牌的应用级别提供商的变化所影响。 当路由器在惰性加载环境中创建组件时，Angular 优先使用惰性加载模块中的服务实例，而不是来自应用的根注入器的。

如果两个模块提供了**同一个**服务会怎么样？

当同时加载了两个导入的模块，它们都列出了使用同一个令牌的提供商时，后导入的模块会“获胜”，这是因为这两个提供商都被添加到了同一个注入器中。

当 Angular 尝试根据令牌注入服务时，它使用第二个提供商来创建并交付服务实例。

每个注入了该服务的类获得的都是由第二个提供商创建的实例。 即使是声明在第一个模块中的类，它取得的实例也是来自第二个提供商的。

如果模块 A 提供了一个使用令牌'X'的服务，并且导入的模块 B 也用令牌'X'提供了一个服务，那么模块 A 中定义的服务“获胜”了。

由根 `AppModule` 提供的服务相对于所导入模块中提供的服务有优先权。换句话说：`AppModule` 总会获胜。

我应该如何把服务的范围限制到模块中?

如果一个模块在应用程序启动时就加载，它的 `@NgModule.providers` 具有**全应用级作用域**。 它们也可用于整个应用的注入中。

导入的提供商很容易被由其它导入模块中的提供商替换掉。 这虽然是故意这样设计的，但是也可能引起意料之外的结果。

作为一个通用的规则，应该**只导入一次**带提供商的模块，最好在应用的**根模块**中。 那里也是配置、包装和改写这些服务的最佳位置。

假设模块需要一个定制过的 `HttpBackend`，它为所有的 Http 请求添加一个特别的请求头。 如果应用中其它地方的另一个模块也定制了 `HttpBackend` 或仅仅导入了 `HttpClientModule`，它就会改写当前模块的 `HttpBackend` 提供商，丢掉了这个特别的请求头。 这样服务器就会拒绝来自该模块的请求。

要消除这个问题，就只能在应用的根模块 `AppModule` 中导入 `HttpClientModule`。

如果你必须防范这种“提供商腐化”现象，那就**不要依赖于“启动时加载”模块的 `providers`**。

只要可能，就让模块惰性加载。 Angular 给了惰性加载模块自己的子注入器。 该模块中的提供商只对由该注入器创建的组件树可见。

如果你必须在应用程序启动时主动加载该模块，**就改成在组件中提供该服务**。

继续看这个例子，假设某个模块的组件真的需要一个私有的、自定义的 `HttpBackend`。

那就创建一个“顶级组件”来扮演该模块中所有组件的根。 把这个自定义的 `HttpBackend` 提供商添加到这个顶级组件的 `providers` 列表中，而不是该模块的 `providers` 中。 回忆一下，Angular 会为每个组件实例创建一个子注入器，并使用组件自己的 `providers` 来配置这个注入器。

当该组件的子组件**想要**一个 `HttpBackend` 服务时，Angular 会提供一个局部的 `HttpBackend` 服务，而不是应用的根注入器创建的那个。 子组件将正确发起 http 请求，而不管其它模块对 `HttpBackend` 做了什么。

确保把模块中的组件都创建成这个顶级组件的子组件。

你可以把这些子组件都嵌在顶级组件的模板中。或者，给顶级组件一个 `<router-outlet>`，让它作为路由的宿主。 定义子路由，并让路由器把模块中的组件加载进该路由出口（outlet）中。

我应该把全应用级提供商添加到根模块 `AppModule` 中还是根组件 `AppComponent` 中?

通过在服务的 `@Injectable()` 装饰器中（例如服务）指定 `providedIn: 'root'` 来定义全应用级提供商，或者 `InjectionToken` 的构造器（例如提供令牌的地方），都可以定义全应用级提供商。 通过这种方式创建的服务提供商会自动在整个应用中可用，而不用把它列在任何模块中。

如果某个提供商不能用这种方式配置（可能因为它没有有意义的默认值），那就在根模块 `AppModule` 中注册这些全应用级服务，而不是在 `AppComponent` 中。

惰性加载模块及其组件可以注入 `AppModule` 中的服务，却不能注入 `AppComponent` 中的。

只有当该服务必须对 `AppComponent` 组件树之外的组件不可见时，才应该把服务注册进 `AppComponent` 的 `providers` 中。 这是一个非常罕见的异常用法。

更一般地说，优先把提供商注册进模块中，而不是组件中。

讨论

Angular 把所有启动期模块的提供商都注册进了应用的根注入器中。 这些服务是由根注入器中的提供商创建的，并且在整个应用中都可用。 它们具有**应用级作用域**。

某些服务（比如 `Router`）只有当注册进应用的根注入器时才能正常工作。

相反，Angular 使用 `AppComponent` 自己的注入器注册了 `AppComponent` 的提供商。 `AppComponent` 服务只在该组件及其子组件树中才能使用。 它们具有**组件级作用域**。

`AppComponent` 的注入器是根注入器的**子级**，注入器层次中的下一级。 这对于没有路由器的应用来说**几乎是**整个应用了。 但对那些带路由的应用，路由操作位于顶层，那里不存在 `AppComponent` 服务。这意味着惰性加载模块不能使用它们。

我应该把其它提供商注册到模块中还是组件中?

提供商应该使用 `@Injectable` 语法进行配置。只要可能，就应该把它们在应用的根注入器中提供（`providedIn: 'root'`）。 如果它们只被惰性加载的上下文中使用，那么这种方式配置的服务就是惰性加载的。

如果要由消费方来决定是否把它作为全应用级提供商，那么就要在模块中（`@NgModule.providers`）注册提供商，而不是组件中（`@Component.providers`）。

当你**必须**把服务实例的范围限制到某个组件及其子组件树时，就把提供商注册到该组件中。 指令的提供商也同样照此处理。

例如，如果英雄编辑组件需要自己私有的缓存英雄服务实例，那就应该把 `HeroService` 注册进 `HeroEditorComponent` 中。 这样，每个新的 `HeroEditorComponent` 的实例都会得到一份自己的缓存服务实例。 编辑器的改动只会作用于它自己的服务，而不会影响到应用中其它地方的英雄实例。

总是在根模块 `AppModule` 中注册**全应用级**服务，而不要在根组件 `AppComponent` 中。

为什么在共享模块中为惰性加载模块提供服务是个馊主意?

立即加载的场景

当立即加载的模块提供了服务时，比如 `UserService`，该服务是在全应用级可用的。如果根模块提供了 `UserService`，并导入了另一个也提供了同一个 `UserService` 的模块，Angular 就会把它们中的一个注册进应用的根注入器中（参见如果两次导入了同一个模块会怎样？）。

然后，当某些组件注入 `UserService` 时，Angular 就会发现它已经在应用的根注入器中了，并交付这个全应用级的单例服务。这样不会出现问题。

惰性加载场景

现在，该考虑 `HeroModule` 了，**它是惰性加载的！**。

当路由器准备惰性加载 `HeroModule` 的时候，它会创建一个子注入器，并且把 `UserService` 的提供商注册到那个子注入器中。子注入器和根注入器是**不同**的。

当 Angular 创建一个惰性加载的 `HeroComponent` 时，它必须注入一个 `UserService`。 这次，它会从惰性加载模块的**子注入器**中查找 `UserService` 的提供商，并用它创建一个 `UserService` 的新实例。 这个 `UserService` 实例与 Angular 在主动加载的组件中注入的那个全应用级单例对象截然不同。

这个场景导致你的应用每次都创建一个新的服务实例，而不是使用单例的服务。

为什么惰性加载模块会创建一个子注入器?

Angular 会把 `@NgModule.providers` 中的提供商添加到应用的根注入器中…… 除非该模块是惰性加载的，这种情况下，它会创建一**子注入器**，并且把该模块的提供商添加到这个子注入器中。

这意味着模块的行为将取决于它是在应用启动期间加载的还是后来惰性加载的。如果疏忽了这一点，可能导致严重后果。

为什么 Angular 不能像主动加载模块那样把惰性加载模块的提供商也添加到应用程序的根注入器中呢？为什么会出现这种不一致？

归根结底，这来自于 Angular 依赖注入系统的一个基本特征： 在注入器还没有被第一次使用之前，可以不断为其添加提供商。 一旦注入器已经创建和开始交付服务，它的提供商列表就被冻结了，不再接受新的提供商。

当应用启动时，Angular 会首先使用所有主动加载模块中的提供商来配置根注入器，这发生在它创建第一个组件以及注入任何服务之前。 一旦应用开始工作，应用的根注入器就不再接受新的提供商了。

之后，应用逻辑开始惰性加载某个模块。 Angular 必须把这个惰性加载模块中的提供商添加到**某个**注入器中。 但是它无法将它们添加到应用的根注入器中，因为根注入器已经不再接受新的提供商了。 于是，

Angular 在惰性加载模块的上下文中创建了一个新的子注入器。

我要如何知道一个模块或服务是否已经加载过了?

某些模块及其服务只能被根模块 `AppModule` 加载一次。 在惰性加载模块中再次导入这个模块会导致错误的行为，这个错误可能非常难于检测和诊断。

为了防范这种风险，可以写一个构造函数，它会尝试从应用的根注入器中注入该模块或服务。如果这种注入成功了，那就说明这个类是被第二次加载的，你就可以抛出一个错误，或者采取其它挽救措施。

某些 Angular 模块（例如 `BrowserModule`）就实现了一个像 Angular 模块那一章的 `CoreModule` 构造函数那样的守卫。

src/app/core/core.module.ts (Constructor)

```
constructor (@Optional() @SkipSelf() parentModule: CoreModule) {
  if (parentModule) {
    throw new Error(
      'CoreModule is already loaded. Import it in the AppModule only');
  }
}
```

什么是**入口组件**?

Angular 根据其类型**不可避免地**加载的组件是**入口组件**,

而通过组件选择器**声明式**加载的组件则**不是**入口组件。

大多数应用组件都是声明式加载的。 Angular 使用该组件的选择器在模板中定位元素，然后创建表现该组件的 HTML，并把它插入 DOM 中所选元素的内部。它们不是入口组件。

而用于引导的根 `AppComponent` 则是一个**入口组件**。 虽然它的选择器匹配了 `index.html` 中的一个元素，但是 `index.html` 并不是组件模板，而且 `AppComponent` 选择器也不会在任何组件模板中出现。

在路由定义中用到的组件也同样是**入口组件**。 路由定义根据**类型**来引用组件。 路由器会忽略路由组件的选择器（即使它有选择器），并且把该组件动态加载到 `RouterOutlet` 中。

要了解更多，参见入口组件一章。

引导组件和**入口组件**有什么不同?

引导组件是入口组件的一种。 它是被 Angular 的引导（应用启动）过程加载到 DOM 中的入口组件。 其它入口组件则是被其它方式动态加载的，比如被路由器加载。

`@NgModule.bootstrap` 属性告诉编译器这是一个入口组件，同时它应该生成一些代码来用该组件引导此应用。

不需要把组件同时列在 `bootstrap` 和 `entryComponent` 列表中 —— 虽然这样做也没坏处。

要了解更多，参见入口组件一章。

什么时候我应该把组件加到 `entryComponents` 中?

大多数应用开发者都不需要把组件添加到 `entryComponents` 中。

Angular 会自动把恰当的组件添加到**入口组件**中。 列在 `@NgModule.bootstrap` 中的组件会自动加入。 由路由配置引用到的组件会被自动加入。 用这两种机制添加的组件在入口组件中占了绝大多数。

如果你的应用要用其它手段来**根据类型**引导或动态加载组件，那就得把它显式添加到 `entryComponents` 中。

虽然把组件加到这个列表中也没什么坏处，不过最好还是只添加真正的**入口组件**。 不要添加那些被其它组件的模板引用过的组件。

要了解更多，参见入口组件一章。

为什么 Angular 需要**入口组件**?

原因在于**摇树优化**。对于产品化应用，你会希望加载尽可能小而快的代码。 代码中应该仅仅包括那些实际用到的类。 它应该排除那些从未用过的组件，无论该组件是否被声明过。

事实上，大多数库中声明和导出的组件你都用不到。 如果你从未引用它们，那么**摇树优化器**就会从最终的代码包中把这些组件砍掉。

如果Angular 编译器为每个声明的组件都生成了代码，那么摇树优化器的作用就没有了。

所以，编译器转而采用一种递归策略，它只为你用到的那些组件生成代码。

编译器从入口组件开始工作，为它在入口组件的模板中找到的那些组件生成代码，然后又为在这些组件中的模板中发现的组件生成代码，以此类推。 当这个过程结束时，它就已经为每个入口组件以及从入口组件可以抵达的每个组件生成了代码。

如果该组件不是**入口组件**或者没有在任何模板中发现过，编译器就会忽略它。

有哪些类型的模块？我应该如何使用它们?

每个应用都不一样。根据不同程度的经验，开发者会做出不同的选择。下列建议和指导原则广受欢迎。

`SharedModule`

为那些可能会在应用中到处使用的组件、指令和管道创建 `SharedModule`。 这种模块应该只包含 `declarations`，并且应该导出几乎所有 `declarations` 里面的声明。

`SharedModule` 可以重新导出其它小部件模块，比如 `CommonModule`、`FormsModule` 和提供你广泛使用的 UI 控件的那些模块。

`SharedModule`**不应该**带有 `providers`，原因在前面解释过了。 它的导入或重新导出的模块中也不应该有 `providers`。 如果你要违背这条指导原则，请务必想清楚你在做什么，并要有充分的理由。

在任何特性模块中（无论是你在应用启动时主动加载的模块还是之后惰性加载的模块），你都可以随意导入这个 `SharedModule`。

`CoreModule`

为你要在应用启动时加载的那些服务创建一个带 `providers` 的 `CoreModule`。

只能在根模块 `AppModule` 中导入 `CoreModule`。 永远不要在除根模块 `AppModule` 之外的任何模块中导入 `CoreModule`。

考虑把 `CoreModule` 做成一个没有 `declarations` 的纯服务模块。

要了解更多，参见共享模块和单例服务。

特性模块

特性模块是你围绕特定的应用业务领域创建的模块，比如用户工作流、小工具集等。它们包含指定的特性，并为你的应用提供支持，比如路由、服务、窗口部件等。 要对你的应用中可能会有哪些特性模块有个概念，考虑如果你要把与特定功能（比如搜索）有关的文件放进一个目录下，该目录的内容就可能是一个名叫 `SearchModule` 的特性模块。 它将会包含构成搜索功能的全部组件、路由和模板。

要了解更多，参见特性模块和模块的分类。

在 NgModule 和 JavaScript 模块之间有什么不同?

在 Angular 应用中，NgModule 会和 JavaScript 的模块一起工作。

在现代 JavaScript 中，每个文件都是模块（参见模块）。 在每个文件中，你要写一个 `export` 语句将模块的一部分公开。

Angular 模块是一个带有 `@NgModule` 装饰器的类，而 JavaScript 模块则没有。 Angular 的 `NgModule` 有自己的 `imports` 和 `exports` 来达到类似的目的。

你可以**导入**其它 Angular 模块，以便在当前模块的组件模板中使用它们导出的类。 你可以**导出**当前 Angular 模块中的类，以便其它模块可以导入它们，并用在自己的组件模板中。

要了解更多，参见 JavaScript 模块 vs. NgModules 一章

Angular 如何查找模板中的组件、指令和管道？什么是 **模板引用** ？

Angular 编译器在组件模板内查找其它组件、指令和管道。一旦找到了，那就是一个“模板引用”。

Angular 编译器通过在一个模板的 HTML 中匹配组件或指令的选择器（selector），来查找组件或指令。

编译器通过分析模板 HTML 中的管道语法中是否出现了特定的管道名来查找对应的管道。

Angular 只查询两种组件、指令或管道：1）那些在当前模块中声明过的，以及 2）那些被当前模块导入的模块所导出的。

什么是 Angular 编译器？

Angular 编译器会把你所编写的应用代码转换成高性能的 JavaScript 代码。 在编译过程中，`@NgModule` 的元数据扮演了很重要的角色。

你写的代码是无法直接执行的。 比如组件。 组件有一个模板，其中包含了自定义元素、属性型指令、Angular 绑定声明和一些显然不属于原生 HTML 的古怪语法。

Angular 编译器读取模板的 HTML，把它和相应的组件类代码组合在一起，并产出**组件工厂**。

组件工厂为组件创建纯粹的、100% JavaScript 的表示形式，它包含了 `@Component` 元数据中描述的一切：HTML、绑定指令、附属的样式等……

由于指令和管道都出现在组件模板中，*Angular 编译器**也同样会把它们组合到编译成的组件代码中。

`@NgModule` 元数据告诉**Angular 编译器**要为当前模块编译哪些组件，以及如何把当前模块和其它模块链接起来。

依赖注入（Dependency injection）模式

依赖注入是一个很重要的设计模式。 它使用得非常广泛，以至于几乎每个人都把它简称为 **DI** 。

Angular 有自己的依赖注入框架，离开它，你几乎没办法构建出 Angular 应用。

本页会告诉你 DI 是什么，以及为什么它很有用。

当你学会了这种通用的模式之后，就可以转到 Angular 依赖注入 中去看看它在 Angular 应用中的工作原理了。

为什么需要依赖注入？

要理解为什么依赖注入这么重要，不妨先考虑不使用它的一个例子。想象下列代码：

src/app/car/car.ts (without DI)

```
export class Car {

  public engine: Engine;
  public tires: Tires;
  public description = 'No DI';

  constructor() {
    this.engine = new Engine();
    this.tires = new Tires();
  }

  // Method using the engine and tires
  drive() {
    return `${this.description} car with ` +
      `${this.engine.cylinders} cylinders and ${this.tires.make} tires.`;
  }
}
```

`Car` 类在自己的构造函数中创建了它所需的一切。 这样做有什么问题？ 问题在于 `Car` 类是脆弱、不灵活以及难于测试的。

`Car` 类需要一个引擎 (engine) 和一些轮胎 (tire)，它没有去请求现成的实例， 而是在构造函数中用具体的 `Engine` 和 `Tires` 类实例化出自己的副本。

如果 `Engine` 类升级了，它的构造函数要求传入一个参数，这该怎么办？ 这个 `Car` 类就被破坏了，在把创建引擎的代码重写为 `this.engine = new Engine(theNewParameter)` 之前，它都是坏的。 当第一次写 `Car` 类时，你不关心 `Engine` 构造函数的参数，现在也不想关心。 但是，当 `Engine` 类的定义发生变化时，就不得不在乎了，`Car` 类也不得不跟着改变。 这就会让 `Car` 类过于脆弱。

如果想在 `Car` 上使用不同品牌的轮胎会怎样？太糟了。 你被锁定在 `Tires` 类创建时使用的那个品牌上。这让 `Car` 类缺乏弹性。

现在，每辆车都有它自己的引擎。它不能和其它车辆共享引擎。 虽然这对于汽车来说还算可以理解，但是设想一下那些应该被共享的依赖，比如用来联系厂家服务中心的车载无线电。 这种车缺乏必要的弹性，无法共享当初给其它消费者创建的车载无线电。

当给 `Car` 类写测试的时候，你就会受制于它背后的那些依赖。 能在测试环境中成功创建新的 `Engine` 吗？ `Engine` 自己又依赖什么？那些依赖本身又依赖什么？ `Engine` 的新实例会发起到服务器的异步调用吗？ 你当然不想在测试期间这么一层层追下去。

如果 `Car` 应该在轮胎气压低的时候闪动警示灯该怎么办？ 如果没法在测试期间换上一个低气压的轮胎，那该如何确认它能正确的闪警示灯？

你没法控制这辆车背后隐藏的依赖。 当不能控制依赖时，类就会变得难以测试。

该如何让 `Car` 更强壮、有弹性以及可测试？

答案非常简单。把 `Car` 的构造函数改造成使用 DI 的版本：

src/app/car/car.ts (excerpt with DI) *src/app/car/car.ts (excerpt without DI)*

```
public description = 'DI';

constructor(public engine: Engine, public tires: Tires) { }
```

发生了什么？现在依赖的定义移到了构造函数中。 `Car` 类不再创建引擎 `engine` 或者轮胎 `tires`。 它仅仅“消费”它们。

> 这个例子又一次借助 TypeScript 的构造器语法来同时定义参数和属性。

现在，通过往构造函数中传入引擎和轮胎来创建一辆车。

```
// Simple car with 4 cylinders and Flintstone tires.
let car = new Car(new Engine(), new Tires());
```

酷！引擎和轮胎这两个依赖的定义与 `Car` 类本身解耦了。 只要喜欢，可以传入任何类型的引擎或轮胎，只要它们能满足引擎或轮胎的通用 API 需求。

这样一来，如果有人扩展了 `Engine` 类，那就不再是 `Car` 类的烦恼了。

> `Car` 的**消费者**也有这个问题。消费者必须修改创建这辆车的代码，就像这样：
>
> ```
> class Engine2 {
>   constructor(public cylinders: number) { }
> }
> // Super car with 12 cylinders and Flintstone tires.
> let bigCylinders = 12;
> let car = new Car(new Engine2(bigCylinders), new Tires());
> ```
>
> 这里的要点是：`Car` 本身不必变化。下面就来解决消费者的问题。

`Car` 类非常容易测试，因为现在你对它的依赖有了完全的控制权。 在每个测试期间，你可以往构造函数中传入 mock 对象，做想让它们做的事：

```
class MockEngine extends Engine { cylinders = 8; }
class MockTires  extends Tires  { make = 'YokoGoodStone'; }

// Test car with 8 cylinders and YokoGoodStone tires.
let car = new Car(new MockEngine(), new MockTires());
```

刚刚学习了什么是依赖注入

它是一种编程模式，可以让类从外部源中获得它的依赖，而不必亲自创建它们。

酷！但是，可怜的消费者怎么办？ 那些希望得到一个 `Car` 的人们现在必须创建所有这三部分了：`Car`、`Engine` 和 `Tires`。 `Car` 类把它的快乐建立在了消费者的痛苦之上。 需要某种机制为你把这三个部分装配好。

可以写一个巨型类来做这件事：

src/app/car/car-factory.ts

```
import { Engine, Tires, Car } from './car';

// BAD pattern!
export class CarFactory {
  createCar() {
    let car = new Car(this.createEngine(), this.createTires());
    car.description = 'Factory';
    return car;
  }

  createEngine() {
    return new Engine();
  }

  createTires() {
    return new Tires();
  }
}
```

现在只需要三个创建方法，这还不算太坏。 但是当应用规模变大之后，维护它将变得惊险重重。 这个工厂类将变成由相互依赖的工厂方法构成的巨型蜘蛛网。

如果能简单的列出想建造的东西，而不用定义该把哪些依赖注入到哪些对象中，那该多好！

到了依赖注入框架一展身手的时候了！ 想象框架中有一个叫做**注入器 (injector)** 的东西。 用这个注入器注册一些类，它会弄明白如何创建它们。

当需要一个 `Car` 时，就简单的找注入器取车就可以了。

src/app/car/car-injector.ts

```
let car = injector.get(Car);
```

皆大欢喜。 `Car` 不需要知道如何创建 `Engine` 和 `Tires`。 消费者不需要知道如何创建 `Car`。 开发人员不需要维护巨大的工厂类。 `Car` 和消费者只要简单地请求想要什么，注入器就会交付它们。

这就是“依赖注入框架”存在的原因。

现在，你知道什么是依赖注入以及它有什么优点了吧？那就请到 Angular 依赖注入 中去看看它在 Angular 中是如何实现的。

Angular 依赖注入

依赖注入（DI）是用来创建对象及其依赖的其它对象的一种方式。 当依赖注入系统创建某个对象实例时，会负责提供该对象所依赖的对象（称为该对象的**依赖**）。

依赖注入模式中讲解了这种通用的方法。 **在这里**

DI 的例子

在这篇指南中，你将会通过对一个范例应用的讨论来学习 Angular 的依赖注入技术。

先从《英雄指南》中**英雄**特性区的一个简化版本开始。

src/app/heroes/heroes.component.ts *src/app/heroes/hero-list.component.ts*

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-heroes',
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent { }
```

`HeroesComponent` 是位于顶级的组件。 它唯一的用途是显示 `HeroListComponent`，它显示一个英雄名字的列表。

这个版本的 `HeroListComponent` 从 `HEROES` 数组（定义在 `mock-heroes` 文件中的内存数组）中获取 `heroes`。

src/app/heroes/hero-list.component.ts (class)

```
export class HeroListComponent {
  heroes = HEROES;
}
```

在开发的早期阶段，这就够用了，不过还很不理想。 当要测试这个组件或者要从远端服务器获取英雄数据时，你就不得不去修改 `HeroesListComponent` 的实现，并要替换所有使用了 `HEROES` 模拟数据的地方。

最好隐藏服务类的这些内部实现细节，那就先把它定义在自己的文件中。

创建一个可注入的 `HeroService`

Angular CLI 可以使用下列命令在 `src/app/heroes` 目录下新建一个 `HeroService` 类。

```
ng generate service heroes/hero
```

上述命令会创建如下的 `HeroService` 骨架代码：

src/app/heroes/hero.service.ts (CLI-generated)

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

@Injectable({
  providedIn: 'root',
})
export class HeroService {
  constructor() { }
}
```

`@Injectable` 装饰器是定义每个 Angular 服务时的必备部分。 把该类的其它部分改写为暴露一个返回和以前一样的 mock 数据的 `getHeroes` 方法。

src/app/heroes/hero.service.3.ts

```
1. import { Injectable } from '@angular/core';
2. import { HEROES }     from './mock-heroes';
3.
4. @Injectable({
5.   // we declare that this service should be created
6.   // by the root application injector.
7.
8.   providedIn: 'root',
9. })
10. export class HeroService {
11.   getHeroes() { return HEROES; }
12. }
```

当然，这还不是真正的数据服务。 如果该应用真的从远端服务器获取数据，那么 `getHeroes` 的方法签名就应该是异步形式的。

我们可以放心地忽略这个问题，因为这里的焦点在于**把服务注入**到 `HeroListComponent` 组件中。

注入器

在你把 Angular 中的**服务类**（比如 `HeroService` ）注册进依赖注入器（injector）之前，它只是个普通类而已。

Angular 的依赖注入器负责创建服务的实例，并把它们注入到像 `HeroListComponent` 这样的类中。

你很少需要自己创建 Angular 的依赖注入器。 当 Angular 运行本应用时，它会为你创建这些注入器，首先会在引导过程中创建一个**根注入器**。

Angular 本身没法自动判断你是打算自行创建服务类的实例，还是等注入器来创建它。你必须通过为每个服务指定服务提供商来配置它。

提供商会告诉注入器**如何创建该服务**。 如果没有提供商，注入器既不知道它该负责创建该服务，也不知道如何创建该服务。

> 你可以在稍后的部分学到更多关于**提供商**的知识。 不过目前，你只要知道它们是用来配置服务应该在哪里创建以及如何创建的就够了。

有很多方式可以为注入器注册服务提供商。本节会展示为你的服务配置提供商最常见的途径。

@Injectable 的 providers 数组

`@Injectable` 装饰器会指出这些服务或其它类是用来注入的。它还能用于为这些服务提供配置项。

这里我们使用类上的 `@Injectable` 装饰器来为 `HeroService` 配置了一个提供商。

src/app/heroes/heroes.service.ts

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

@Injectable({
  providedIn: 'root',
})
export class HeroService {
  constructor() { }
}
```

`providedIn` 告诉 Angular，它的根注入器要负责调用 `HeroService` 类的构造函数来创建一个实例，并让它在整个应用中都是可用的。在使用 CLI 生成新服务时，会默认帮你设置为这种提供商。

有时，你不希望只在应用的根注入器中提供服务。有可能用户希望显式选择要使用的服务，或者应该在一个惰性加载的环境下提供该服务。这种情况下，服务提供商应该关联到一个特定的 `@NgModule` 类，而且应该用于该模块包含的任何一个注入器中。

下面这段代码中，`@Injectable` 装饰器用来配置一个服务提供商，它可以用在任何包含了 HeroModule 的注入器中。

src/app/heroes/hero.service.ts

```
import { Injectable } from '@angular/core';
import { HeroModule } from './hero.module';
import { HEROES }     from './mock-heroes';

@Injectable({
  // we declare that this service should be created
  // by any injector that includes HeroModule.

  providedIn: HeroModule,
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

`@NgModule` 中的 `providers`

在下面的代码片段中，根模块 `AppModule` 在自己的 `providers` 数组中注册了两个提供商。

src/app/app.module.ts (providers)

```
providers: [
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
```

第一条使用 `UserService` 这个**注入令牌（injection token）**注册了 `UserService` 类（代码中未显示）。 第二条使用 `APP_CONFIG` 这个注入令牌注册了一个值（`HERO_DI_CONFIG`）。

借助这些注册语句，Angular 现在可以向它创建的任何类中注册 `UserService` 或 `HERO_DI_CONFIG` 值了。

> 稍后你就会学到关于**注入令牌**和服务提供商语法的知识。

在组件中注册提供商

除了提供给全应用级或特定的 `@NgModule` 中之外，服务还可以提供给指定的组件。在组件级提供的服务职能在该组件及其子组件的注入器中使用。

下面的例子展示了一个修改过的 `HeroesComponent`，它在自己的 `providers` 数组中注册了 `HeroService`。

src/app/heroes/heroes.component.ts

```
import { Component } from '@angular/core';
import { HeroService } from './hero.service';

@Component({
  selector: 'app-heroes',
  providers: [ HeroService ],
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent { }
```

@Injectable、**@NgModule** 还是 **@Component**？

你该使用 `@Injectable` 装饰器、`@NgModule`还是 `@Component` 来提供服务呢？ 这几个选择的差别在于最终的打包体积、服务的**范围**和服务的**生命周期**。

当你在服务本身的 @Injectable 装饰器中注册提供商时，优化工具（比如 CLI 产品模式构建时所用的）可以执行摇树优化，这会移除所有没在应用中使用过的服务。摇树优化会导致更小的打包体积。

Angular 模块中的 `providers` （`@NgModule.providers`）是注册在应用的根注入器下的。 因此，Angular 可以往它所创建的任何类中注入相应的服务。 一旦创建，服务的实例就会存在于该应用的全部生存期中，Angular 会把这一个服务实例注入到需求它的每个类中。

你可能想要把这个 `UserService` 注入到应用中的很多地方，并期望每次注入的都是同一个服务实例。 这时候如果不能用 `@Injectable`，那么就可以在 Angular 的模块中提供 `UserService`。

> 严格来说，Angular 模块中的服务提供商会注册到根注入器上，但是，惰性加载的模块是例外。 在这个例子中，所有模块都是在应用启动时**立即加载**的，因此模块上的所有服务提供商都注册到了应用的根注入器上。

组件的提供商（`@Component.providers`）会注册到每个组件实例自己的注入器上。

因此 Angular 只能在该组件及其各级子组件的实例上注入这个服务实例，而不能在其它地方注入这个服务实例。

注意，由组件提供的服务，也同样具有有限的生命周期。组件的每个实例都会有它自己的服务实例，并且，当组件实例被销毁的时候，服务的实例也同样会被销毁。

在这个范例应用中，`HeroComponent` 会在应用启动时创建，并且它从未销毁，因此，由 `HeroComponent` 创建的 `HeroService` 也同样会活在应用的整个生命周期中。

如果你要把 `HeroService` 的访问权限定在 `HeroesComponent` 及其嵌套的 `HeroListComponent` 中，那么在 `HeroesComponent` 中提供这个 `HeroService` 就是一个好选择。

> 由组件提供的服务，其范围和生命周期是 Angular 如何创建组件实例 的必然结果。

服务提供商们

服务提供商**提供**依赖值的一个具体的、运行时的版本。 注入器依靠提供商来创建服务的实例，注入器再将服务的实例注入组件、管道或其它服务。

必须为注入器注册一个服务的**提供商**，否则它就不知道该如何创建该服务。

在下面的几节中会解释指定提供商的多种方式。

把类作为它自己的提供商

有很多方式可以**提供**一些实现 `Logger` 类的东西。 `Logger` 类本身是一个显而易见而且自然而然的提供商。

```
providers: [Logger]
```

但它不是唯一的途径。

可以用其它备选提供商来配置注入器，只要它们能交付一个行为类似于 `Logger` 的对象就可以了。 可以提供一个替代类。你可以提供一个类似日志的对象。 可以给它一个提供商，让它调用可以创建日志服务的工厂函数。 所有这些方法，只要用在正确的场合，都可能是一个好的选择。

重点是，当注入器需要一个 `Logger` 时，它得先有一个提供商。

provide 对象字面量

下面是类提供商的另一种语法。

```
providers: [Logger]
```

这其实是用于注册提供商的简写表达式。 使用的是一个带有两个属性的**提供商**对象字面量：

```
[{ provide: Logger, useClass: Logger }]
```

`provide` 属性保存的是令牌 (token)，它作为键值 (key) 使用，用于定位依赖值和注册提供商。

第二个是一个提供商定义对象。 可以把它看做是指导如何创建依赖值的**配方**。 有很多方式创建依赖值…… 也有很多方式可以写配方。

备选的类提供商

某些时候，你会请求一个不同的类来提供服务。 下列代码告诉注入器，当有人请求 `Logger` 时，返回 `BetterLogger`。

```
[{ provide: Logger, useClass: BetterLogger }]
```

带依赖的类提供商

假设 `EvenBetterLogger` 可以在日志消息中显示用户名。 这个日志服务从注入的 `UserService` 中取得用户， `UserService` 通常也会在应用级注入。

```
@Injectable()
export class EvenBetterLogger extends Logger {
  constructor(private userService: UserService) { super(); }

  log(message: string) {
    let name = this.userService.user.name;
    super.log(`Message to ${name}: ${message}`);
  }
}
```

就像之前在 `BetterLogger` 中那样配置它。

```
[ UserService,
  { provide: Logger, useClass: EvenBetterLogger }]
```

别名类提供商

假设某个旧组件依赖一个 `OldLogger` 类。 `OldLogger` 和 `NewLogger` 具有相同的接口，但是由于某些原因， 你不能升级这个旧组件并使用它。

当**旧**组件想使用 `OldLogger` 记录消息时，你希望改用 `NewLogger` 的单例对象来记录。

不管组件请求的是新的还是旧的日志服务，依赖注入器注入的都应该是同一个单例对象。 也就是说， `OldLogger` 应该是 `NewLogger` 的别名。

你当然不会希望应用中有两个不同的 `NewLogger` 实例。 不幸的是，如果尝试通过 `useClass` 来把 `OldLogger` 作为 `NewLogger` 的别名，就会导致这样的后果。

```
[ NewLogger,
  // Not aliased! Creates two instances of `NewLogger`
  { provide: OldLogger, useClass: NewLogger}]
```

解决方案：使用 `useExisting` 选项指定别名。

```
[ NewLogger,
  // Alias OldLogger w/ reference to NewLogger
  { provide: OldLogger, useExisting: NewLogger}]
```

值提供商

有时，提供一个预先做好的对象会比请求注入器从类中创建它更容易。

```
// An object in the shape of the logger service
export function SilentLoggerFn() {}

const silentLogger = {
  logs: ['Silent logger says "Shhhhh!". Provided via "useValue"'],
  log: SilentLoggerFn
};
```

于是可以通过 `useValue` 选项来注册提供商，它会让这个对象直接扮演 logger 的角色。

```
[{ provide: Logger, useValue: silentLogger }]
```

查看更多 `useValue` 的例子，见非类依赖和 InjectionToken部分。

工厂提供商

有时，你需要动态创建这个依赖值，因为它所需要的信息直到最后一刻才能确定。 也许这个信息会在浏览器的会话中不停地变化。

还假设这个可注入的服务没法通过独立的源访问此信息。

这种情况下，请调用工厂提供商。

下面通过添加新的业务需求来说明这一点： `HeroService` 必须对普通用户隐藏掉**秘密**英雄。 只有授权用户才能看到秘密英雄。

就像 `EvenBetterLogger` 那样， `HeroService` 需要了解此用户的身份。 它需要知道，这个用户是否有权看到隐藏英雄。 这个授权可能在单一的应用会话中被改变，例如，改用另一个用户的身份登录时。

与 `EvenBetterLogger` 不同，不能把 `UserService` 注入到 `HeroService` 中。 `HeroService` 无权访问用户信息，来决定谁有授权谁没有授权。

让 `HeroService` 的构造函数带上一个布尔型的标志，来控制是否显示隐藏的英雄。

src/app/heroes/hero.service.ts (excerpt)

```
constructor(
  private logger: Logger,
  private isAuthorized: boolean) { }

getHeroes() {
  let auth = this.isAuthorized ? 'authorized ' : 'unauthorized';
  this.logger.log(`Getting heroes for ${auth} user.`);
  return HEROES.filter(hero => this.isAuthorized || !hero.isSecret);
}
```

你可以注入 `Logger`，但是不能注入逻辑型的 `isAuthorized`。 你不得不通过通过工厂提供商创建这个 `HeroService` 的新实例。

工厂提供商需要一个工厂方法：

src/app/heroes/hero.service.provider.ts (excerpt)

```
let heroServiceFactory = (logger: Logger, userService: UserService) => {
  return new HeroService(logger, userService.user.isAuthorized);
};
```

虽然 `HeroService` 不能访问 `UserService`，但是工厂方法可以。

同时把 `Logger` 和 `UserService` 注入到工厂提供商中，并且让注入器把它们传给工厂方法：

src/app/heroes/hero.service.provider.ts (excerpt)

```
export let heroServiceProvider =
  { provide: HeroService,
    useFactory: heroServiceFactory,
    deps: [Logger, UserService]
  };
```

> `useFactory` 字段告诉 Angular：这个提供商是一个工厂方法，它的实现是 `heroServiceFactory`。
>
> `deps` 属性是提供商令牌数组。 `Logger` 和 `UserService` 类作为它们自身类提供商的令牌。 注入器解析这些令牌，把相应的服务注入到工厂函数中相应的参数中去。

注意，你在一个导出的变量中捕获了这个工厂提供商：`heroServiceProvider`。 这个额外的步骤让工厂提供商可被复用。 无论哪里需要，都可以使用这个变量注册 `HeroService`。

在这个例子中，只在 `HeroesComponent` 中需要它， 这里，它代替了元数据 `providers` 数组中原来的 `HeroService` 注册。 对比一下新的和旧的实现：

src/app/heroes/heroes.component (v3) *src/app/heroes/heroes.component (v2)*

```
1. import { Component }           from '@angular/core';
2. import { heroServiceProvider } from './hero.service.provider';
3.
4. @Component({
5.   selector: 'app-heroes',
6.   providers: [ heroServiceProvider ],
7.   template: `
8.     <h2>Heroes</h2>
9.     <app-hero-list></app-hero-list>
10.   `
11. })
12. export class HeroesComponent { }
```

可以被摇树优化的提供商

摇树优化可以在最终打包时移除应用中从未引用过的代码。可摇树优化的提供商可以让 Angular 从结果中移除应用中那些从未使用过的服务。这可以显著减小打包体积。

理想情况下，如果应用没有注入过某个服务，它就不应该被包含在最终结果中。不过，问题在于 Angular 的编译器无法在构建期间识别出该服务是不是必要的。 因为总是可以使用 `injector.get(Service)` 的形式直接注入某个服务，而 Angular 不能从你的代码中识别出所有能够进行这种注入的地方。所以，Angular 别无选择，只能把这个服务包含到注入器中。因此，在模块中提供的服务也就无法进行摇树优化了。

来看一个 Angular 无法对提供商进行摇树优化的例子。

在这个例子中，为了在 Angular 中提供服务，你把它们都包含进了 `@NgModule` 中：

src/app/tree-shaking/service-and-modules.ts

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

@Injectable()
export class Service {
  doSomething(): void {
  }
}

@NgModule({
  providers: [Service],
})
export class ServiceModule {
}
```

接着，该模块可以导入到你的应用模块中，以便让该服务在整个应用中可用：

src/app/tree-shaking/app.modules.ts

```
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([]),
    ServiceModule,
  ],
})
export class AppModule {
}
```

当运行 `ngc` 时，它会把 AppModule 编译进一个模块工厂里，该工厂中含有它包含的所有子模块中声明过的所有提供商。在运行期间，该工厂会编程一个用于实例化这些服务的注入器。

在这种方式下，摇树优化无法工作，因为 Angular 无法根据该代码（服务类）是否被其它代码块使用过（比如该服务的提供商就定义在了模块工厂里）来排除它。 要让这些服务可以被摇树优化，所有关于如何构建该

服务的实例的信息（提供商定义）就应该是该服务类本身的一部分。

创建可摇树优化的服务提供商

要想创建可摇树优化的服务提供商，那些原本要通过模块来指定的信息就要改为在服务自身的 `@Injectable` 装饰器中提供。

下面的例子展示了一个与上面的 `ServiceModule` 范例等价的可摇树的优化版本：

src/app/tree-shaking/service.ts

```
@Injectable({
  providedIn: 'root',
})
export class Service {
}
```

上面这个例子中，`providedIn` 允许你声明要由哪个注入器来注入该服务。除了一些特殊情况外，这个值应该始终是 root（根注入器）。把该值设置为 root 可以确保该服务的范围是根注入器，而不是该注入器所在的那个特定模块。

该服务也可以通过配置一个工厂函数来实例化，例子如下：

src/app/tree-shaking/service.0.ts

```
@Injectable({
  providedIn: 'root',
  useFactory: () => new Service('dependency'),
})
export class Service {
  constructor(private dep: string) {
  }
}
```

> 要想改写（override）一个可摇树优化的提供商，可以在任何支持 `providers: []` 数组的 Angular 装饰器中注册该提供商。

注入某个服务

`HeroListComponent` 应该从 `HeroService` 中获取这些英雄数据。

该组件不应该使用 `new` 来创建 `HeroService`。 它应该要求注入 `HeroService`。

你可以通过在构造函数中添加一个带有该依赖类型的参数来要求 Angular 把这个依赖注入到组件的构造函数中。 下面是 `HeroListComponent` 的构造函数，它要求注入 `HeroService`。

src/app/heroes/hero-list.component (constructor signature)

```
constructor(heroService: HeroService)
```

当然，`HeroListComponent` 还应该使用注入的这个 `HeroService` 做点什么。 下面输出修改过的组件，改用注入的服务，与前一个版本对比一下。

hero-list.component (with DI)   *hero-list.component (without DI)*

```
1. import { Component }   from '@angular/core';
2. import { Hero }        from './hero';
3. import { HeroService } from './hero.service';
4.
5. @Component({
6.   selector: 'app-hero-list',
7.   template: `
8.     <div *ngFor="let hero of heroes">
9.       {{hero.id}} - {{hero.name}}
10.    </div>
11.  `
12. })
13. export class HeroListComponent {
14.   heroes: Hero[];
15.
16.   constructor(heroService: HeroService) {
17.     this.heroes = heroService.getHeroes();
18.   }
19. }
```

注意，`HeroListComponent` 并不知道 `HeroService` 来自哪里。 不过，你知道它来自其父组件 `HeroesComponent`。 如果你决定改为在 `AppModule` 中提供这个 `HeroService`，`HeroListComponent` 不需要做任何改动。 唯一需要关心的问题是，`HeroService` 是由某个父注入器提供的。

单例服务

服务**在每个注入器的范围内**是单例的。 在任何一个注入器中，最多只会有同一个服务的一个实例。

这里只有一个根注入器，而 `UserService` 就是在该注入器中注册的。 所以，在整个应用中只能有一个 `UserService` 实例，每个要求注入 `UserService` 的类都会得到这个服务实例。

不过，Angular DI 是一个 多级注入系统，这意味着各级注入器都可以创建它们自己的服务实例。 Angular 总会创建多级注入器。

组件的子注入器

组件注入器是彼此独立的，每一个都会为这些组件提供的服务创建单独的实例。

例如，当 Angular 创建一个带有 `@Component.providers` 的组件实例时，也会同时为这个实例创建一个新的**子注入器**。

当 Angular 销毁某个组件实例时，也会同时销毁该组件的注入器，以及该注入器中的服务实例。

由于是多层注入器，因此你仍然可以把全应用级的服务注入到这些组件中。 组件的注入器是其父组件注入器的孩子，也是其爷爷注入器的孙子，以此类推，直到该应用的根注入器。 Angular 可以注入这条线上的任何注入器所提供的服务。

比如，Angular 可以把由 `HeroComponent` 提供的 `HeroService` 和由 `AppModule` 提供的 `UserService` 注入到 `HeroService` 中。

测试组件

前面强调过，设计一个适合依赖注入的类，可以让这个类更容易测试。 要有效的测试应用中的一部分，只需要在构造函数的参数中列出依赖。

例如，新建的 `HeroListComponent` 实例使用一个模拟 (mock) 服务，以便可以在测试中操纵它：

src/app/test.component.ts

```
const expectedHeroes = [{name: 'A'}, {name: 'B'}]
const mockService = <HeroService> {getHeroes: () => expectedHeroes }

it('should have heroes when HeroListComponent created', () => {
  // Pass the mock to the constructor as the Angular injector would
  const component = new HeroListComponent(mockService);
  expect(component.heroes.length).toEqual(expectedHeroes.length);
});
```

> 要学习更多知识，参见测试一章。

当服务需要别的服务时

这个 `HeroService` 非常简单。它本身不需要任何依赖。

如果它也有依赖，该怎么办呢？例如，它需要通过日志服务来汇报自己的活动。 你同样用**构造函数注入**模式，来添加一个带有 `Logger` 参数的构造函数。

下面是修改后的 `HeroService`，它注入了 `Logger`，对比前后这两个版本：

src/app/heroes/hero.service (v2)   *src/app/heroes/hero.service (v1)*

```
1. import { Injectable } from '@angular/core';
2. import { HEROES }     from './mock-heroes';
3. import { Logger }     from '../logger.service';
4.
5. @Injectable({
6.   providedIn: 'root',
7. })
8. export class HeroService {
9.
10.   constructor(private logger: Logger) {  }
11.
12.   getHeroes() {
13.     this.logger.log('Getting heroes ...');
14.     return HEROES;
15.   }
16. }
```

这个构造函数要求注入一个 `Logger` 类的实例，并把它存到名为 `logger` 的私有字段中。 当请求英雄数据时，`getHeroes()` 中就会记录一个消息。

被依赖的 `Logger` 服务

这个范例应用的 `Logger` 服务非常简单：

src/app/logger.service.ts

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

@Injectable()
export class Logger {
  logs: string[] = []; // capture logs for testing

  log(message: string) {
    this.logs.push(message);
    console.log(message);
  }
}
```

如果该应用没有提供这个 `Logger` 服务，当 Angular 试图把 `Logger` 注入到 `HeroService` 中时，就会抛出一个异常。

```
ERROR Error: No provider for Logger!
```

因为 `Logger` 服务的单例应该随处可用，所以要在根模块 `AppModule` 中提供它。

src/app/app.module.ts (providers)

```
providers: [
  Logger,
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
```

依赖注入令牌

当向注入器注册提供商时，实际上是把这个提供商和一个 DI 令牌关联起来了。 注入器维护一个内部的**令牌-提供商**映射表，这个映射表会在请求依赖时被引用到。 令牌就是这个映射表中的键值。

在前面的所有例子中，依赖值都是一个类**实例**，并且类的**类型**作为它自己的查找键值。 在下面的代码中，`HeroService` 类型作为令牌，直接从注入器中获取 `HeroService` 实例：

src/app/injector.component.ts

```
heroService: HeroService;
```

编写需要基于类的依赖注入的构造函数对你来说是很幸运的。 只要定义一个 `HeroService` 类型的构造函数参数， Angular 就会知道把跟 `HeroService` 类令牌关联的服务注入进来：

src/app/heroes/hero-list.component.ts

```
constructor(heroService: HeroService)
```

这是一个特殊的规约，因为大多数依赖值都是以类的形式提供的。

非类依赖

如果依赖值不是一个类呢？有时候想要注入的东西是一个字符串，函数或者对象。

应用程序经常为很多很小的因素定义配置对象（例如应用程序的标题或网络 API 终点的地址）。 但是这些配置对象不总是类的实例，它们可能是对象，如下面这个：

src/app/app.config.ts (excerpt)

```
export const HERO_DI_CONFIG: AppConfig = {
  apiEndpoint: 'api.heroes.com',
  title: 'Dependency Injection'
};
```

如果想让这个配置对象在注入时可用该怎么办？你知道你可以用值提供商来注册一个对象。

但是，这种情况下用什么作令牌呢？ 你没办法找一个类来当作令牌，因为没有 `Config` 类。

> ### TypeScript 接口不是一个有效的令牌
>
> `HERO_DI_CONFIG` 常量有一个接口： `AppConfig`。不幸的是，不能把 TypeScript 接口用作令牌：
>
> ```
> // FAIL! Can't use interface as provider token
> [{ provide: AppConfig, useValue: HERO_DI_CONFIG })]
> ```
>
> ```
> // FAIL! Can't inject using the interface as the parameter type
> constructor(private config: AppConfig){ }
> ```
>
> 对于习惯于在强类型的语言中使用依赖注入的开发人员，这会看起来很奇怪， 因为在强类型语言中，接口是首选的用于查找依赖的主键。

> 这不是 Angular 的错。接口只是 TypeScript 设计时 (design-time) 的概念。JavaScript 没有接口。TypeScript 接口不会出现在生成的 JavaScript 代码中。 在运行期，没有接口类型信息可供 Angular 查找。

InjectionToken 值

解决方案是为非类依赖定义和使用**InjectionToken**作为提供商令牌。 定义方式是这样的：

```
import { InjectionToken } from '@angular/core';
export const TOKEN = new InjectionToken('desc');
```

你可以在创建 `InjectionToken` 时直接配置一个提供商。该提供商的配置会决定由哪个注入器来提供这个令牌，以及如何创建它的值。 这和 `@Injectable` 的用法很像，不过你没法用 `InjectionToken` 来定义标准提供商（比如 `useClass` 或 `useFactory`），而要指定一个工厂函数，该函数直接返回想要提供的值。

```
export const TOKEN =
  new InjectionToken('desc', { providedIn: 'root', factory: () => new AppConfig(),
})
```

现在，在 `@Inject` 装饰器的帮助下，这个配置对象可以注入到任何需要它的构造函数中：

```
constructor(@Inject(TOKEN));
```

如果工厂函数需要访问其它的 DI 令牌，它可以使用来自 `@angular/core` 中的 `inject` 函数来申请它的依赖。

```
const TOKEN =
  new InjectionToken('tree-shakeable token',
    { providedIn: 'root', factory: () =>
        new AppConfig(inject(Parameter1), inject(Paremeter2)), });
```

可选依赖

可以把构造函数的参数标记为 `null` 来告诉 Angular 该依赖是可选的：

```
constructor(@Inject(Token, null));
```

如果要使用可选依赖，你的代码就必须准备好处理空值。

小结

本章，你学习了 Angular 依赖注入的基础知识。 你可以注册很多种类的提供商，知道如何通过添加构造函数的参数来请求一个注入对象（例如一个服务）。

Angular 依赖注入比前面描述的更能干。 学习更多高级特性，如对嵌套注入器的支持，见多级依赖注入一章。

附录：直接使用注入器

这里的 `InjectorComponent` 直接使用了注入器， 但开发者很少直接使用它。

src/app/injector.component.ts

```
@Component({
  selector: 'app-injectors',
  template: `
  <h2>Other Injections</h2>
  <div id="car">{{car.drive()}}</div>
  <div id="hero">{{hero.name}}</div>
  <div id="rodent">{{rodent}}</div>
  `,
  providers: [Car, Engine, Tires, heroServiceProvider, Logger]
})
export class InjectorComponent implements OnInit {
  car: Car;

  heroService: HeroService;
  hero: Hero;

  constructor(private injector: Injector) { }

  ngOnInit() {
    this.car = this.injector.get(Car);
    this.heroService = this.injector.get(HeroService);
    this.hero = this.heroService.getHeroes()[0];
  }

  get rodent() {
    let rousDontExist = `R.O.U.S.'s? I don't think they exist!`;
    return this.injector.get(ROUS, rousDontExist);
  }
}
```

`Injector` 本身是可注入的服务。

在这个例子中，Angular 把组件自身的 `Injector` 注入到了组件的构造函数中。 然后，组件在 `ngOnInit()` 中向注入的注入器请求它所需的服务。

注意，这些服务本身没有注入到组件，它们是通过调用 `injector.get()` 获得的。

`get()` 方法如果不能解析所请求的服务，会抛出异常。 调用 `get()` 时，还可以使用第二个参数，一旦获取的服务没有在当前或任何祖先注入器中注册过， 就把它作为返回值。

> 刚描述的这项技术是服务定位器模式的一个范例。

> 要避免使用此技术，除非确实需要它。 它会鼓励鲁莽的方式，就像在这里看到的。 它难以解释、理解和测试。 仅通过阅读构造函数，没法知道这个类需要什么或者它将做什么。 它可以从任何祖先组件中获得服务，而不仅仅是它自己。 会迫使你深入它的实现，才可能明白它都做了啥。
>
> 框架开发人员必须采用通用的或者动态的方式获取服务时，可能采用这个方法。

附录：为什么建议每个文件只放一个类

在同一个文件中有多个类容易造成混淆，最好避免。 开发人员期望每个文件只放一个类。这会让它们开心点。

如果你把 `HeroService` 和 `HeroesComponent` 组合在同一个文件里， 就得把组件定义放在最后面！ 如果把组件定义在了服务的前面， 在运行时抛出空指针错误。

> 在 `forwardRef()` 方法的帮助下，实际上也可以先定义组件， 具体说明见这篇博客。
>
> 但是为什么要先给自己找麻烦呢？ 还是通过在独立的文件中定义组件和服务，完全避免此问题吧。

多级依赖注入器

在依赖注入一章中，你已经学过了 Angular 依赖注入的基础知识。

Angular 有一个**多级依赖注入系统**。 实际上，应用程序中有一个与组件树平行的注入器树（译注：平行是指结构完全相同且一一对应）。 你可以在组件树中的任何级别上重新配置注入器。

本文将浏览这个体系，并告诉你如何善用它。

试试在线例子 / 下载范例。

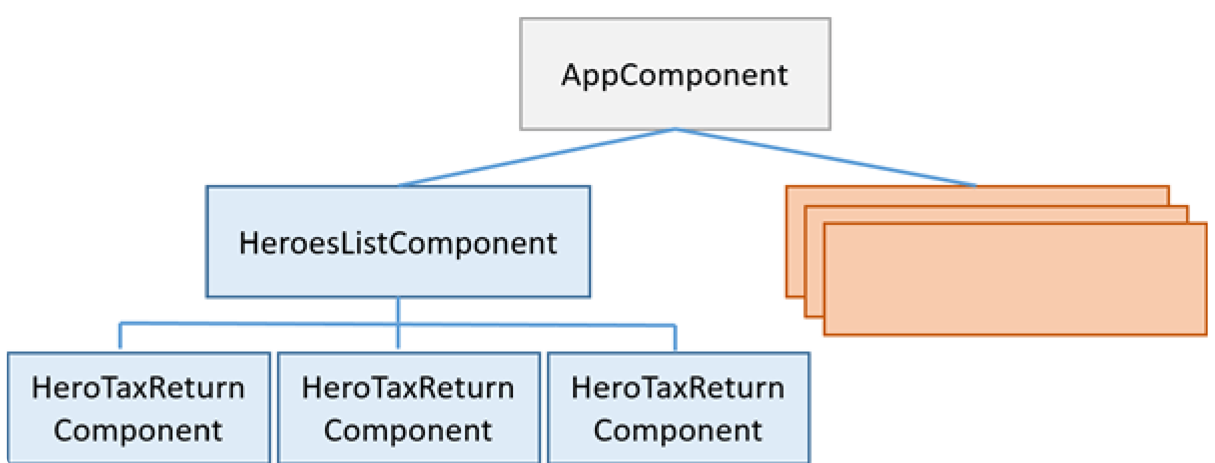
注入器树

在依赖注入一章中，你学过如何配置依赖注入器，以及如何在需要时用它获取依赖。

实际上，没有**那个（唯一的）**注入器这回事，一个应用中可能有多个注入器。 一个 Angular 应用是一个组件树。每个组件实例都有自己的注入器！ 组件的树与注入器的树平行。

> 组件的注入器可能是一个组件树中更高级的祖先注入器的**代理**。 但这只是提升效率的实现细节，你不用在乎这点差异，在你的脑海里只要想象成每个组件都有自己的注入器就可以了。

考虑《英雄指南》应用的一个简单变种。它的顶层是 `AppComponent` 组件，它有一些子组件。 `HeroesListComponent` 组件保存和管理着 `HeroTaxReturnComponent` 的多个实例。 下图展示了当 `HeroesCardComponent` 的三个 `HeroTaxReturnComponent` 实例同时展开时的三级组件树状态。



注入器冒泡

当一个组件申请获得一个依赖时，Angular 先尝试用该组件自己的注入器来满足它。 如果该组件的注入器没有找到对应的提供商，它就把这个申请转给它父组件的注入器来处理。 如果那个注入器也无法满足这个申请，它就继续转给**它的**父组件的注入器。 这个申请继续往上冒泡 —— 直到找到了一个能处理此申请的注入器或者超出了组件树中的祖先位置为止。 如果超出了组件树中的祖先还未找到，Angular 就会抛出一个错误。

> 你还可以“盖住”这次冒泡。一个中层的组件可以声称自己是“宿主”组件。 向上查找提供商的过程会截止于这个“宿主”组件。 这个问题先放一放，等改天再讨论它。

在不同层级再次提供同一个服务

你可以在注入器树中的多个层次上为指定的依赖令牌重新注册提供商。 但**并非必须**重新注册，事实上，虽然可以重新注册，但除非有很好的理由，否则不应该这么做。

服务解析逻辑会自下而上查找，碰到的第一个提供商会胜出。 因此，注入器树中间层注入器上的提供商，可以拦截来自底层的对特定服务的请求。 这导致它可以“重新配置”和者说“遮蔽”高层的注入器。

如果你只在顶级（通常是根模块 `AppModule`），这三个注入器看起来将是“平面”的。 所有的申请都会冒泡到根`NgModule`进行处理，也就是你在 `bootstrapModule` 方法中配置的那个。

组件注入器

在不同层次上重新配置一个或多个提供商的能力，开启了一些既有趣又有用的可能性。

场景：服务隔离

出于架构方面的考虑，可能会让你决定把一个服务限制到只能在它所属的特定领域中访问。

本章的范例中包括一个 `VillainsListComponent`，它显示一个反派的列表。

虽然你也**可以**在根模块 `AppModule` 中提供 `VillainsService`（就像 `HeroesService` 那样），不过那样一来就会导致在整个应用中到处都能访问到 `VillainsService`，包括在**英雄**工作流中。

如果你以后修改了 `VillainsService`，那就可能会破坏英雄组件中的某些部分。 这可不妙，但是在根模块 `AppModule` 中提供这个服务可能会导致这种风险。

可以换一种方案：在 `VillainsListComponent` 元数据的 `providers` 中提供 `VillainsService`，就像这样：

src/app/villains-list.component.ts (metadata)

```
@Component({
  selector: 'app-villains-list',
  templateUrl: './villains-list.component.html',
  providers: [ VillainsService ]
})
```

在 `VillainsListComponent` 的元数据中而不是其它地方提供 `VillainsService` 服务，该服务就会只在 `VillainsListComponent` 及其子组件树中可用。 它仍然是单例，但是这个单例只存在于**反派（villain）**这个领域中。

现在，你可以确信英雄组件不会访问它，因此减少了犯错误的机会。

场景：多重编辑会话

很多应用允许用户同时进行多个任务。 比如，在纳税申报应用中，申报人可以打开多个报税单，随时可能从一个切换到另一个。

本章要示范的场景仍然是基于《英雄指南》的。 想象一个外层的 `HeroListComponent`，它显示一个超级英雄的列表。

要打开一个英雄的报税单，申报者点击英雄名，它就会打开一个组件来编辑那个申报单。 每个选中的申报单都会在自己的组件中打开，并且可以同时打开多个申报单。

每个报税单组件都有下列特征：

- 属于它自己的报税单会话。
- 可以修改一个报税单，而不会影响另一个组件中的申报单。
- 能把所做的修改保存到它的报税单中，或者放弃它们。

Hero Tax Returns

- RubberMan
- Tornado

实现方式之一就是让 `HeroTaxReturnComponent` 有逻辑来管理和还原那些更改。 这对于简单的报税单来说是很容易的。 不过，在现实世界中，报税单的数据模型非常复杂，对这些修改的管理可能不得不投机取巧。 于是你可以把这种管理任务委托给一个辅助服务，就像这个例子中所做的。

这是一个报税单服务 `HeroTaxReturnService`。 它缓存了单条 `HeroTaxReturn`，用于跟踪那个申报单的变更，并且可以保存或还原它。 它还委托给了全应用级的单例服务 `HeroService`，它是通过依赖注入机制取得的。

src/app/hero-tax-return.service.ts

```
import { Injectable }    from '@angular/core';
import { HeroTaxReturn } from './hero';
import { HeroesService } from './heroes.service';

@Injectable()
export class HeroTaxReturnService {
  private currentTaxReturn: HeroTaxReturn;
  private originalTaxReturn: HeroTaxReturn;

  constructor(private heroService: HeroesService) { }

  set taxReturn (htr: HeroTaxReturn) {
    this.originalTaxReturn = htr;
    this.currentTaxReturn  = htr.clone();
  }

  get taxReturn (): HeroTaxReturn {
    return this.currentTaxReturn;
  }

  restoreTaxReturn() {
    this.taxReturn = this.originalTaxReturn;
  }

  saveTaxReturn() {
    this.taxReturn = this.currentTaxReturn;
    this.heroService.saveTaxReturn(this.currentTaxReturn).subscribe();
  }
}
```

下面是正在使用它的 `HeroTaxReturnComponent` 组件。

src/app/hero-tax-return.component.ts

```
1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2. import { HeroTaxReturn }        from './hero';
3. import { HeroTaxReturnService } from './hero-tax-return.service';
4.
5. @Component({
6.   selector: 'app-hero-tax-return',
7.   templateUrl: './hero-tax-return.component.html',
8.   styleUrls: [ './hero-tax-return.component.css' ],
9.   providers: [ HeroTaxReturnService ]
10. })
11. export class HeroTaxReturnComponent {
12.   message = '';
13.   @Output() close = new EventEmitter<void>();
14.
15.   get taxReturn(): HeroTaxReturn {
16.     return this.heroTaxReturnService.taxReturn;
17.   }
18.   @Input()
19.   set taxReturn (htr: HeroTaxReturn) {
20.     this.heroTaxReturnService.taxReturn = htr;
21.   }
22.
23.   constructor(private heroTaxReturnService: HeroTaxReturnService ) { }
24.
25.   onCanceled()  {
26.     this.flashMessage('Canceled');
27.     this.heroTaxReturnService.restoreTaxReturn();
28.   };
29.
30.   onClose()  { this.close.emit(); };
31.
32.   onSaved() {
33.     this.flashMessage('Saved');
34.     this.heroTaxReturnService.saveTaxReturn();
35.   }
36.
37.   flashMessage(msg: string) {
38.     this.message = msg;
39.     setTimeout(() => this.message = '', 500);
40.   }
41. }
```

通过输入属性可以得到**要编辑的报税单**，这个属性被实现成了读取器（getter）和设置器（setter）。 设置器根据传进来的报税单初始化了组件自己的 `HeroTaxReturnService` 实例。 读取器总是返回该服务所存英雄的当前状态。 组件也会请求该服务来保存或还原这个报税单。

这里有个大问题，那就是如果**这个**服务是一个全应用范围的单例，每个组件就都会共享同一个服务实例，每个组件也都会覆盖属于其他英雄的报税单，真是一团糟！

但仔细看 `HeroTaxReturnComponent` 的元数据，注意 `providers` 属性。

src/app/hero-tax-return.component.ts (providers)

```
providers: [ HeroTaxReturnService ]
```

`HeroTaxReturnComponent` 有它自己的 `HeroTaxReturnService` 提供商。 回忆一下，每个组件的**实例**都有它自己的注入器。 在组件级提供服务可以确保组件的**每个**实例都得到一个自己的、私有的服务实例。 报税单不会再被意外覆盖，这下清楚了。

> 该场景代码中的其它部分依赖另一些 Angular 的特性和技术，你将会在本文档的其它章节学到。 你可以到在线例子 / 下载范例查看代码和下载它。

场景：专门的提供商

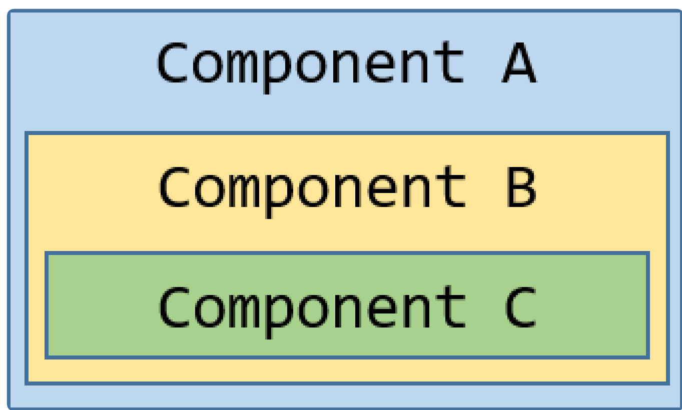
重新提供服务的另一个原因，是在组件树的深层中把该服务替换为一个**更特殊的**实现。

再次考虑依赖注入一章中车辆（Car）的例子。 假设你在根注入器（代号 A）中配置了**通用的**提供商：`CarService`、`EngineService` 和 `TiresService`。

你创建了一个车辆组件（A），它显示一个从另外三个通用服务构造出的车辆。

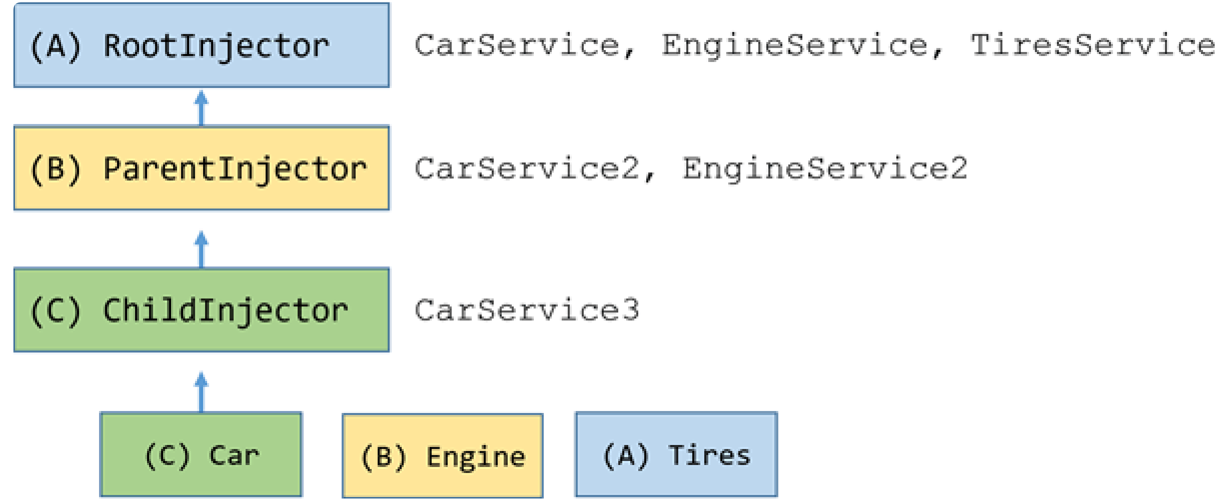
然后，你创建一个子组件（B），它为 `CarService` 和 `EngineService` 定义了自己的**特殊的**提供商，它们具有更特殊的能力，适用于组件 B 的。

组件 B 是另一个组件 C 的父组件，而组件 C 又定义了自己的，**更特殊的**`CarService` 提供商。



在幕后，每个组件都有自己的注入器，这个注入器带有为组件本身准备的 0 个、1 个或多个提供商。

当你在最深层的组件 C 解析 `Car` 的实例时，它使用注入器 C 解析生成了一个 `Car` 的实例，使用注入器 B 解析了 `Engine`，而 `Tires` 则是由根注入器 A 解析的。



> **车辆**场景下的代码位于 `car.components.ts` 和 `car.services.ts` 文件中，这个例子你可以在在线例子 / 下载范例查看和下载。

依赖注入

依赖注入是一个用来管理代码依赖的强大模式。本文会讨论 Angular 依赖注入的许多特性。

要获取本“烹饪宝典”的代码，参见在线例子 / 下载范例。

应用程序全局依赖

在服务本身的 `@Injectable()` 装饰器中注册那些将被应用程序全局使用的依赖提供商。

src/app/heroes/hero.service.3.ts

```
import { Injectable } from '@angular/core';
import { HEROES }     from './mock-heroes';

@Injectable({
  // we declare that this service should be created
  // by the root application injector.

  providedIn: 'root',
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

这里的 `providedIn` 告诉 Angular，要由根注入器负责创建 `HeroService` 的实例。 所有用这种方式提供的服务，都会自动在整个应用中可用，而不必把它们显式列在任何模块中。

这些服务类可以充当自己的提供商，因此你只要把它们定义在 `@Injectable` 装饰器中就算注册成功了。

> **提供商**是用来新建或者交付服务的。 Angular 拿到“类提供商”之后，会通过 `new` 操作来新建服务实例。从依赖注入一章可以学到关于提供商的更多知识。

现在你已经注册了这些服务，这样 Angular 就能在应用程序的**任何地方**，把它们注入到**任何**组件和服务的构造函数里。

外部模块配置

如果某个提供商不是在服务的 `@Injectable` 装饰器中配置的，那么就要在根模块 `AppModule` 中把它注册为全应用级的提供商，而不是在 `AppComponent` 中。 一般来说，要在 `NgModule` 中注册提供商，而不是在应用程序根组件中。

下列情况下会用到这种方法：1. 当用户应该明确选择所用的服务时。2. 当你要在惰性加载的上下文中提供该服务时。3. 当你要在应用启动之前配置应用中的另一个全局服务时。

下面的例子就属于这些情况，它为组件路由器配置了一个非默认的地址策略（location strategy），并把它加入到 `AppModule` 的 `providers` 数组中。

src/app/app.module.ts (providers)

```
providers: [
  { provide: LocationStrategy, useClass: HashLocationStrategy }
]
```

@Injectable和嵌套服务依赖

这些被注入服务的消费者不需要知道如何创建这个服务，它也不应该在乎。新建和缓存这个服务是依赖注入器的工作。

有时候一个服务依赖其它服务...而其它服务可能依赖另外的更多服务。按正确的顺序解析这些嵌套依赖也是框架的工作。 在每一步，依赖的使用者只要在它的构造函数里简单声明它需要什么，框架就会完成所有剩下的事情。

下面的例子往 `AppComponent` 里注入的 `LoggerService` 和 `UserContext`。

src/app/app.component.ts

```
constructor(logger: LoggerService, public userContext: UserContextService) {
  userContext.loadUser(this.userId);
  logger.logInfo('AppComponent initialized');
}
```

`UserContext` 有两个依赖 `LoggerService`(再一次)和负责获取特定用户信息的 `UserService`。

user-context.service.ts (injection)

```
@Injectable()
export class UserContextService {
  constructor(private userService: UserService, private loggerService:
LoggerService) {
  }
}
```

当 Angular 新建 `AppComponent` 时，依赖注入框架先创建一个 `LoggerService` 的实例，然后创建 `UserContextService` 实例。 `UserContextService` 需要框架已经创建好的 `LoggerService` 实例和尚未创建的 `UserService` 实例。 `UserService` 没有其它依赖，所以依赖注入框架可以直接 `new` 一个实例。

依赖注入最帅的地方在于，`AppComponent` 的作者不需要在乎这一切。作者只是在(`LoggerService` 和 `UserContextService` 的)构造函数里面简单的声明一下，框架就完成了剩下的工作。

一旦所有依赖都准备好了，`AppComponent` 就会显示用户信息：

Logged in user

Name: Bombasto
Role: Admin

@Injectable() 注解

注意在 `UserContextService` 类里面的 `@Injectable()` 装饰器。

user-context.service.ts (@Injectable)

```
@Injectable()
export class UserContextService {
}
```

`@Injectable` 装饰器会向 Angular DI 系统指明应该为 `UserContextService` 创建一个实例还是多个实例。

把服务作用域限制到一个组件支树

所有被注入的服务依赖都是单例的，也就是说，在任意一个依赖注入器("injector")中，每个服务只有唯一的实例。

但是 Angular 应用程序有多个依赖注入器，组织成一个与组件树平行的树状结构。所以，可以在任何组件级别**提供**(和建立)特定的服务。如果在多个组件中注入，服务就会被新建出多个实例，分别提供给不同的组件。

默认情况下，一个组件中注入的服务依赖，会在该组件的所有子组件中可见，而且 Angular 会把同样的服务实例注入到需要该服务的子组件中。

所以，在根部的 `AppComponent` 提供的依赖单例就能被注入到应用程序中**任何地方**的**任何**组件。

但这不一定总是想要的。有时候你想要把服务的有效性限制到应用程序的一个特定区域。

通过**在组件树的子级根组件**中提供服务，可以把一个被注入服务的作用域局限在应用程序结构中的某个**分支**中。 这个例子中展示了为子组件和根组件 `AppComponent` 提供服务的相似之处，它们的语法是相同的。 这里通过列入 `providers` 数组，在 `HeroesBaseComponent` 中提供了 `HeroService`：

src/app/sorted-heroes.component.ts (HeroesBaseComponent excerpt)

```
@Component({
  selector: 'app-unsorted-heroes',
  template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
  providers: [HeroService]
})
export class HeroesBaseComponent implements OnInit {
  constructor(private heroService: HeroService) { }
}
```

当 Angular 新建 `HeroBaseComponent` 的时候，它会同时新建一个 `HeroService` 实例，该实例只在该组件及其子组件(如果有)中可见。

也可以在应用程序别处的**不同的**组件里提供 `HeroService`。这样就会导致在**不同**注入器中存在该服务的**不同**实例。

> 这个例子中，局部化的 `HeroService` 单例，遍布整份范例代码，包括 `HeroBiosComponent`、`HeroOfTheMonthComponent` 和 `HeroBaseComponent`。 这些组件每个都有自己的 `HeroService` 实例，用来管理独立的英雄库。

> ## 休息一下!
>
> 对一些 Angular 开发者来说，这么多依赖注入知识可能已经是它们需要知道的全部了。不是每个人都需要更复杂的用法。

多个服务实例(sandboxing)

在**同一个级别的组件树**里，有时需要一个服务的多个实例。

一个用来保存其伴生组件的实例状态的服务就是个好例子。 每个组件都需要该服务的单独实例。 每个服务有自己的工作状态，与其它组件的服务和状态隔离。这叫做**沙箱化**，因为每个服务和组件实例都在自己的沙箱里运行。

想象一下，一个 `HeroBiosComponent` 组件显示三个 `HeroBioComponent` 的实例。

ap/hero-bios.component.ts

```
@Component({
  selector: 'app-hero-bios',
  template: `
    <app-hero-bio [heroId]="1"></app-hero-bio>
    <app-hero-bio [heroId]="2"></app-hero-bio>
    <app-hero-bio [heroId]="3"></app-hero-bio>`,
  providers: [HeroService]
})
export class HeroBiosComponent {
}
```

每个 `HeroBioComponent` 都能编辑一个英雄的生平。`HeroBioComponent` 依赖 `HeroCacheService` 服务来对该英雄进行读取、缓存和执行其它持久化操作。

src/app/hero-cache.service.ts

```
1. @Injectable()
2. export class HeroCacheService {
3.   hero: Hero;
4.   constructor(private heroService: HeroService) {}
5.
6.   fetchCachedHero(id: number) {
7.     if (!this.hero) {
8.       this.hero = this.heroService.getHeroById(id);
9.     }
10.     return this.hero;
11.   }
12. }
```

很明显，这三个 `HeroBioComponent` 实例不能共享一样的 `HeroCacheService`。要不然它们会相互冲突，争相把自己的英雄放在缓存里面。

通过在自己的元数据(metadata)`providers` 数组里面列出 `HeroCacheService`, 每个 `HeroBioComponent` 就能**拥有**自己独立的 `HeroCacheService` 实例。

src/app/hero-bio.component.ts

```
1. @Component({
2.   selector: 'app-hero-bio',
3.   template: `
4.     <h4>{{hero.name}}</h4>
5.     <ng-content></ng-content>
6.     <textarea cols="25" [(ngModel)]="hero.description"></textarea>`,
7.   providers: [HeroCacheService]
8. })
9.
10. export class HeroBioComponent implements OnInit  {
11.   @Input() heroId: number;
12.
13.   constructor(private heroCache: HeroCacheService) { }
14.
15.   ngOnInit() { this.heroCache.fetchCachedHero(this.heroId); }
16.
17.   get hero() { return this.heroCache.hero; }
18. }
```

父组件 `HeroBiosComponent` 把一个值绑定到 `heroId`。`ngOnInit` 把该 `id` 传递到服务，然后服务获取和缓存英雄。`hero` 属性的 getter 从服务里面获取缓存的英雄，并在模板里显示它绑定到属性值。

到在线例子 / 下载范例中找到这个例子，确认三个 `HeroBioComponent` 实例拥有自己独立的英雄数据缓存。

Hero Bios

RubberMan

Hero of many talents

Magma

Hero of all trades

Mr. Nice

The name says it all

使用@Optional()和 `@Host()` 装饰器来限定依赖查找方式

你知道，依赖可以被注入到任何组件级别。

当组件申请一个依赖时，Angular 从该组件本身的注入器开始，沿着依赖注入器的树往上找，直到找到第一个符合要求的提供商。如果 Angular 不能在这个过程中找到合适的依赖，它就会抛出一个错误。

大部分时候，你确实**想要**这个行为。 但是有时候，需要限制这个(依赖)查找逻辑，且/或提供一个缺失的依赖。 单独或联合使用 `@Host` 和 `@Optional` 限定型装饰器，就可以修改 Angular 的查找行为。

当 Angular 找不到依赖时， `@Optional` 装饰器会告诉 Angular 继续执行。Angular 把此注入参数设置为 `null`(而不用默认的抛出错误的行为)。

`@Host` 装饰器将把往上搜索的行为截止在**宿主组件**

宿主组件通常是申请这个依赖的组件。但当这个组件被投影(projected)进一个**父组件**后，这个父组件就变成了宿主。 下一个例子会演示第二种情况。

示范

`HeroBiosAndContactsComponent` 是前面见过的 `HeroBiosComponent` 的修改版。

src/app/hero-bios.component.ts (HeroBiosAndContactsComponent)

```
1. @Component({
2.   selector: 'app-hero-bios-and-contacts',
3.   template: `
4.     <app-hero-bio [heroId]="1"> <app-hero-contact></app-hero-contact>
   </app-hero-bio>
5.     <app-hero-bio [heroId]="2"> <app-hero-contact></app-hero-contact>
   </app-hero-bio>
6.     <app-hero-bio [heroId]="3"> <app-hero-contact></app-hero-contact>
   </app-hero-bio>`,
7.   providers: [HeroService]
8. })
9. export class HeroBiosAndContactsComponent {
10.   constructor(logger: LoggerService) {
11.     logger.logInfo('Creating HeroBiosAndContactsComponent');
12.   }
13. }
```

注意看模板：

dependency-injection-in-action/src/app/hero-bios.component.ts

```
template: `
  <app-hero-bio [heroId]="1"> <app-hero-contact></app-hero-contact> </app-hero-bio>
  <app-hero-bio [heroId]="2"> <app-hero-contact></app-hero-contact> </app-hero-bio>
  <app-hero-bio [heroId]="3"> <app-hero-contact></app-hero-contact> </app-hero-
bio>`,
```

在 `<hero-bio>` 标签中是一个新的 `<hero-contact>` 元素。Angular 就会把相应的 `HeroContactComponent`**投影(transclude)**进 `HeroBioComponent` 的视图里， 将它放在 `HeroBioComponent` 模板的 `<ng-content>` 标签槽里。

src/app/hero-bio.component.ts (template)

```
template: `
  <h4>{{hero.name}}</h4>
  <ng-content></ng-content>
  <textarea cols="25" [(ngModel)]="hero.description"></textarea>`,
```

从 `HeroContactComponent` 获得的英雄电话号码，被投影到上面的英雄描述里，就像这样：

RubberMan

Phone #: 123-456-7899

Hero of many talents

下面的 `HeroContactComponent`，示范了限定型装饰器(@Optional 和@Host)：

src/app/hero-contact.component.ts

```
@Component({
  selector: 'app-hero-contact',
  template: `
  <div>Phone #: {{phoneNumber}}
  <span *ngIf="hasLogger">!!!</span></div>`
})
export class HeroContactComponent {

  hasLogger = false;

  constructor(
      @Host() // limit to the host component's instance of the HeroCacheService
      private heroCache: HeroCacheService,

      @Host()     // limit search for logger; hides the application-wide logger
      @Optional() // ok if the logger doesn't exist
      private loggerService: LoggerService
  ) {
    if (loggerService) {
      this.hasLogger = true;
      loggerService.logInfo('HeroContactComponent can log!');
    }
  }

  get phoneNumber() { return this.heroCache.hero.phone; }

}
```

注意看构造函数的参数：

src/app/hero-contact.component.ts

```
@Host() // limit to the host component's instance of the HeroCacheService
private heroCache: HeroCacheService,

@Host()     // limit search for logger; hides the application-wide logger
@Optional() // ok if the logger doesn't exist
private loggerService: LoggerService
```

`@Host()` 函数是 `heroCache` 属性的装饰器，确保从其父组件 `HeroBioComponent` 得到一个缓存服务。如果该父组件不存在这个服务，Angular 就会抛出错误，即使组件树里的再上级有某个组件拥有这个服务，Angular 也会抛出错误。

另一个 `@Host()` 函数是属性 `loggerService` 的装饰器。 在本应用程序中只有一个在 `AppComponent` 级提供的 `LoggerService` 实例。 该宿主 `HeroBioComponent` 没有自己的 `LoggerService` 提供商。

如果没有同时使用 `@Optional()` 装饰器的话，Angular 就会抛出错误。多亏了 `@Optional()`，Angular 把 `loggerService` 设置为 null，并继续执行组件而不会抛出错误。

下面是 `HeroBiosAndContactsComponent` 的执行结果：

Hero Bios and Contacts

RubberMan

Phone #: 123-456-7899

Hero of many talents

Magma

Phone #: 555-555-5555

Hero of all trades

Mr. Nice

Phone #: 111-222-3333

The name says it all

如果注释掉 `@Host()` 装饰器，Angular 就会沿着注入器树往上走，直到在 `AppComponent` 中找到该日志服务。日志服务的逻辑加入进来，更新了英雄的显示信息，这表明确实找到了日志服务。

RubberMan

Phone #: 123-456-7899!!!

Hero of many talents

另一方面，如果恢复 `@Host()` 装饰器，注释掉 `@Optional`，应用程序就会运行失败，因为它在宿主组件级别找不到需要的日志服务。

`EXCEPTION: No provider for LoggerService! (HeroContactComponent -> LoggerService)`

注入组件的 DOM 元素

偶尔，可能需要访问一个组件对应的 DOM 元素。尽量避免这样做，但还是有很多视觉效果和第三方工具(比如 jQuery)需要访问 DOM。

要说明这一点，请在属性型指令`HighlightDirective` 的基础上，编写一个简化版。

src/app/highlight.directive.ts

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  @Input('appHighlight') highlightColor: string;

  private el: HTMLElement;

  constructor(el: ElementRef) {
    this.el = el.nativeElement;
  }

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || 'cyan');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.style.backgroundColor = color;
  }
}
```

当用户把鼠标移到 DOM 元素上时，指令将该元素的背景设置为一个高亮颜色。

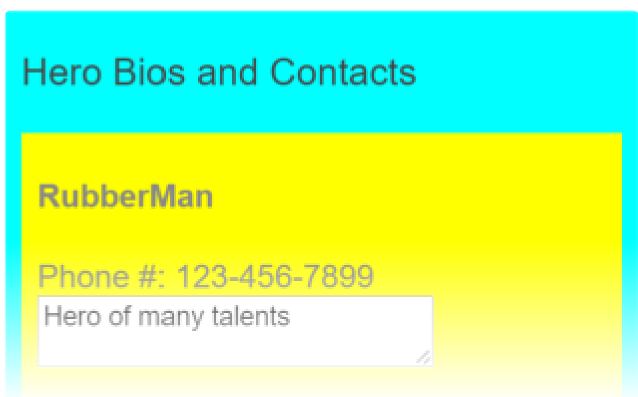
Angular 把构造函数参数 `el` 设置为注入的 `ElementRef`，该 `ElementRef` 代表了宿主的 DOM 元素， 它的 `nativeElement` 属性把该 DOM 元素暴露给了指令。

下面的代码把指令的 `myHighlight` 属性(Attribute)填加到两个 `<div>` 标签里，一个没有赋值，一个赋值了颜色。

src/app/app.component.html (highlight)

```
<div id="highlight"  class="di-component"  appHighlight>
  <h3>Hero Bios and Contacts</h3>
  <div appHighlight="yellow">
    <app-hero-bios-and-contacts></app-hero-bios-and-contacts>
  </div>
</div>
```

下图显示了鼠标移到 `<hero-bios-and-contacts>` 标签的效果：



使用提供商来定义依赖

本节将演示如何编写提供商来提供被依赖的服务。

给依赖注入器提供**令牌**来获取服务。

你通常在构造函数里面，为参数指定类型，让 Angular 来处理依赖注入。该参数类型就是依赖注入器所需的**令牌**。 Angular 把该令牌传给注入器，然后把得到的结果赋给参数。下面是一个典型的例子：

src/app/hero-bios.component.ts (component constructor injection)

```
constructor(logger: LoggerService) {
  logger.logInfo('Creating HeroBiosComponent');
}
```

Angular 向注入器请求与 `LoggerService` 对应的服务，并将返回值赋给 `logger` 参数。

注入器从哪得到的依赖？ 它可能在自己内部容器里已经有该依赖了。 如果它没有，也能在**提供商**的帮助下新建一个。 **提供商**就是一个用于交付服务的配方，它被关联到一个令牌。

> 如果注入器无法根据令牌在自己内部找到对应的提供商，它便将请求移交给它的父级注入器，这个过程不断重复，直到没有更多注入器为止。 如果没找到，注入器就抛出一个错误...除非这个请求是可选的。

新建的注入器中没有提供商。 Angular 会使用一些自带的提供商来初始化这些注入器。你必须自行注册属于**自己**的提供商，通常会在该服务的 `@Injectable` 装饰器中，或在 `NgModule` 或 `Directive` 元数据的 `providers` 数组中进行注册。

src/app/app.component.ts (providers)

```
providers: [ LoggerService, UserContextService, UserService ]
```

定义提供商

建议直接在服务类的 `@Injectable` 装饰器中定义服务提供商。

src/app/heroes/hero.service.0.ts

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

@Injectable({
  providedIn: 'root',
})
export class HeroService {
  constructor() { }
}
```

备选方案是在 `@NgModule` 的 `providers` 数组中引用下这个类就可以了。

src/app/hero-bios.component.ts (class provider)

```
providers: [HeroService]
```

注册类提供商之所以这么简单，是因为最常见的可注入服务就是一个类的实例。 但是，并不是所有的依赖都只要创建一个类的新实例就可以交付了。你还需要其它的交付方式，这意味着你也要用其它方式来指定提供商。

`HeroOfTheMonthComponent` 例子示范了一些替代方案，展示了为什么需要它们。 它看起来很简单：一些属性和一个日志输出。

Hero of the Month

Winner: **Magma**
Reason for award: **Had a great month!**
Runners-up: **RubberMan, Mr. Nice**

Logs:

INFO: starting up at Fri Apr 01 2016
23:31:10 GMT-0700 (Pacific Daylight Time)

这段代码的背后有很多值得深入思考的地方。

hero-of-the-month.component.ts

```
1. import { Component, Inject } from '@angular/core';
2.
3. import { DateLoggerService } from './date-logger.service';
4. import { Hero }              from './hero';
5. import { HeroService }       from './hero.service';
6. import { LoggerService }     from './logger.service';
7. import { MinimalLogger }     from './minimal-logger.service';
8. import { RUNNERS_UP,
9.          runnersUpFactory }  from './runners-up';
10.
11. @Component({
12.   selector: 'app-hero-of-the-month',
13.   templateUrl: './hero-of-the-month.component.html',
14.   providers: [
15.     { provide: Hero,          useValue:    someHero },
16.     { provide: TITLE,         useValue:   'Hero of the Month' },
17.     { provide: HeroService,   useClass:    HeroService },
18.     { provide: LoggerService, useClass:    DateLoggerService },
19.     { provide: MinimalLogger, useExisting: LoggerService },
20.     { provide: RUNNERS_UP,    useFactory:  runnersUpFactory(2), deps:
    [Hero, HeroService] }
21.   ]
22. })
23. export class HeroOfTheMonthComponent {
24.   logs: string[] = [];
25.
26.   constructor(
27.       logger: MinimalLogger,
28.       public heroOfTheMonth: Hero,
29.       @Inject(RUNNERS_UP) public runnersUp: string,
30.       @Inject(TITLE) public title: string)
31.   {
32.     this.logs = logger.logs;
33.     logger.logInfo('starting up');
34.   }
35. }
```

provide 对象

该 `provide` 对象需要一个**令牌**和一个**定义对象**。该**令牌**通常是一个类，但并非一定是

该**定义**对象有一个必填属性(即 `useValue`)，用来标识该提供商会如何新建和返回该服务的单例对象。

useValue - *值-提供商

把一个*固定的值，也就是该提供商可以将其作为依赖对象返回的值，赋给 `useValue` 属性。

使用该技巧来进行**运行期常量设置**，比如网站的基础地址和功能标志等。 你通常在单元测试中使用**值-提供商**，用一个假的或模仿的(服务)来取代一个生产环境的服务。

`HeroOfTheMonthComponent` 例子有两个**值-提供商**。 第一个提供了一个 `Hero` 类的实例；第二个指定了一个字符串资源：

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: Hero,          useValue:    someHero },
{ provide: TITLE,         useValue:   'Hero of the Month' },
```

`Hero` 提供商的令牌是一个类，这很合理，因为它提供的结果是一个 `Hero` 实例，并且被注入该英雄的消费者也需要知道它类型信息。

`TITLE` 提供商的令牌**不是一个类**。它是一个特别类型的提供商查询键，名叫InjectionToken. 你可以把 `InjectionToken` 用作任何类型的提供商的令牌，但是它在依赖是简单类型（比如字符串、数字、函数）时会特别有帮助。

一个**值-提供商**的值必须要**立即**定义。不能事后再定义它的值。很显然，标题字符串是立刻可用的。 该例中的 `someHero` 变量是以前在下面这个文件中定义的：

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
const someHero = new Hero(42, 'Magma', 'Had a great month!', '555-555-5555');
```

其它提供商只在需要注入它们的时候才创建并**惰性加载**它们的值。

useClass - **类-提供商**

`userClass` 提供商创建并返回一个指定类的新实例。

使用该技术来为公共或默认类**提供备选实现**。该替代品能实现一个不同的策略，比如拓展默认类或者在测试的时候假冒真实类。

请看下面 `HeroOfTheMonthComponent` 里的两个例子：

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: HeroService,    useClass:    HeroService },
{ provide: LoggerService, useClass:    DateLoggerService },
```

第一个提供商是**展开了语法糖的**，是一个典型情况的展开。一般来说，被新建的类(`HeroService`)同时也是该提供商的注入令牌。 这里用完整形态来编写它，来反衬更受欢迎的缩写形式。

第二个提供商使用 `DateLoggerService` 来满足 `LoggerService`。该 `LoggerService` 在 `AppComponent` 级别已经被注册。当**这个组件**要求 `LoggerService` 的时候，它得到的却是 `DateLoggerService` 服务。

> 这个组件及其子组件会得到 `DateLoggerService` 实例。这个组件树之外的组件得到的仍是 `LoggerService` 实例。

`DateLoggerService` 从 `LoggerService` 继承；它把当前的日期/时间附加到每条信息上。

src/app/date-logger.service.ts

```
@Injectable()
export class DateLoggerService extends LoggerService
{
  logInfo(msg: any)  { super.logInfo(stamp(msg)); }
  logDebug(msg: any) { super.logInfo(stamp(msg)); }
  logError(msg: any) { super.logError(stamp(msg)); }
}

function stamp(msg: any) { return msg + ' at ' + new Date(); }
```

useExisting - **别名-提供商**

使用 `useExisting`，提供商可以把一个令牌映射到另一个令牌上。实际上，第一个令牌是第二个令牌所对应的服务的一个**别名**，创造了**访问同一个服务对象的两种方法**。

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: MinimalLogger, useExisting: LoggerService },
```

通过使用别名接口来把一个 API 变窄，是**一个**很重要的该技巧的使用例子。下面的例子中使用别名就是为了这个目的。

想象一下如果 `LoggerService` 有个很大的 API 接口(虽然它其实只有三个方法，一个属性)，通过使用 `MinimalLogger` **类-接口**别名，就能成功的把这个 API 接口缩小到只暴露两个成员：

src/app/minimal-logger.service.ts

```
// Class used as a "narrowing" interface that exposes a minimal logger
// Other members of the actual implementation are invisible
export abstract class MinimalLogger {
  logs: string[];
  logInfo: (msg: string) => void;
}
```

现在，在一个简化版的 `HeroOfTheMonthComponent` 中使用它。

src/app/hero-of-the-month.component.ts (minimal version)

```
@Component({
  selector: 'app-hero-of-the-month',
  templateUrl: './hero-of-the-month.component.html',
  // TODO: move this aliasing, `useExisting` provider to the AppModule
  providers: [{ provide: MinimalLogger, useExisting: LoggerService }]
})
export class HeroOfTheMonthComponent {
  logs: string[] = [];
  constructor(logger: MinimalLogger) {
    logger.logInfo('starting up');
  }
}
```

`HeroOfTheMonthComponent` 构造函数的 `logger` 参数是一个 `MinimalLogger` 类型，支持 TypeScript 的编辑器里，只能看到它的两个成员 `logs` 和 `logInfo`：

```
this.logs = logger.logs;
logger.logInfo('sta   logInfo  (method) MinimalLogger.logInfo(msg: string): void
                      logs
```

实际上，Angular 确实想把 `logger` 参数设置为注入器里 `LoggerService` 的完整版本。只是在之前的提供商注册里使用了 `useClass`， 所以该完整版本被 `DateLoggerService` 取代了。

> 在下面的图片中，显示了日志日期，可以确认这一点：

> INFO: starting up at Fri Apr 01 2016 23:31:10 GMT-0700 (Pacific Daylight Time)

useFactory - **工厂-提供商**

`useFactory` 提供商通过调用工厂函数来新建一个依赖对象，如下例所示。

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: RUNNERS_UP,    useFactory:  runnersUpFactory(2), deps: [Hero, HeroService] }
```

使用这项技术，可以用包含了一些**依赖服务和本地状态**输入的工厂函数来**建立一个依赖对象。**

该**依赖对象**不一定是一个类实例。它可以是任何东西。在这个例子里，**依赖对象**是一个字符串，代表了本月英雄比赛的亚军的名字。

本地状态是数字 `2`，该组件应该显示的亚军的个数。它就会立刻用 `2` 来执行 `runnersUpFactory`。

`runnersUpFactory` 自身不是提供商工厂函数。真正的提供商工厂函数是 `runnersUpFactory` 返回的函数。

runners-up.ts (excerpt)

```
export function runnersUpFactory(take: number) {
  return (winner: Hero, heroService: HeroService): string => {
    /* ... */
  };
};
```

这个返回的函数需要一个 `Hero` 和一个 `HeroService` 参数。

Angular 通过使用 `deps` 数组中的两个**令牌**，来识别注入的值，用来提供这些参数。这两个 `deps` 值是供注入器使用的**令牌**，用来提供工厂函数的依赖。

一些内部工作后，这个函数返回名字字符串，Angular 将其注入到 `HeroOfTheMonthComponent` 组件的 `runnersUp` 参数里。

> 该函数从 `HeroService` 获取英雄参赛者，从中取 `2` 个作为亚军，并把他们的名字拼接起来。请到在线例子 / 下载范例查看全部原代码。

备选提供商令牌：**类-接口**和 InjectionToken

Angular 依赖注入当**令牌**是类的时候是最简单的，该类同时也是返回的依赖对象的类型(通常直接称之为**服务**)。

但令牌不一定都是类，就算它是一个类，它也不一定都返回类型相同的对象。这是下一节的主题。

类-接口

前面的**月度英雄**的例子使用了 `MinimalLogger` 类作为 `LoggerService` 提供商的令牌。

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: MinimalLogger, useExisting: LoggerService },
```

该 `MinimalLogger` 是一个抽象类。

dependency-injection-in-action/src/app/minimal-logger.service.ts

```
// Class used as a "narrowing" interface that exposes a minimal logger
// Other members of the actual implementation are invisible
export abstract class MinimalLogger {
  logs: string[];
  logInfo: (msg: string) => void;
}
```

你通常从一个抽象类继承。但这个应用中**并没有**类会继承 `MinimalLogger`。

`LoggerService` 和 `DateLoggerService`**本可以**从 `MinimalLogger` 中继承。 它们也可以实现 `MinimalLogger`，而不用单独定义接口。 但它们没有。 `MinimalLogger` 在这里仅仅被用作一个 "依赖注入令牌"。

这种用法的类叫做**类-接口**。它关键的好处是：提供了接口的强类型，能像正常类一样**把它当做提供商令牌使用**。

类-接口应该**只**定义允许它的消费者调用的成员。窄的接口有助于解耦该类的具体实现和它的消费者。

> ### 为什么 **MinimalLogger** 是一个类而不是一个 TypeScript 接口
>
> 不能把接口当做提供商的令牌，因为接口不是有效的 JavaScript 对象。 它们只存在在 TypeScript 的设计空间里。它们会在被编译为 JavaScript 之后消失。
>
> 一个提供商令牌必须是一个真实的 JavaScript 对象，比如：一个函数，一个对象，一个字符串，或一个类。

> 把类当做接口使用，可以为你在一个 JavaScript 对象上提供类似于接口的特性。
>
> 当然，一个真实的类会占用内存。为了节省内存占用，该类应该**没有具体的实现**。`MinimalLogger` 会被转译成下面这段没有优化过的，尚未最小化的 JavaScript：
>
> dependency-injection-in-action/src/app/minimal-logger.service.ts
>
> ```
> var MinimalLogger = (function () {
>   function MinimalLogger() {}
>   return MinimalLogger;
> }());
> exports("MinimalLogger", MinimalLogger);
> ```
>
> 注意，**只要不实现它**，不管添加多少成员，它永远不会增长大小。

InjectionToken 值

依赖对象可以是一个简单的值，比如日期，数字和字符串，或者一个无形的对象，比如数组和函数。

这样的对象没有应用程序接口，所以不能用一个类来表示。更适合表示它们的是：唯一的和符号性的令牌，一个 JavaScript 对象，拥有一个友好的名字，但不会与其它的同名令牌发生冲突。

`InjectionToken` 具有这些特征。在**Hero of the Month**例子中遇见它们两次，一个是 **title** 的值，一个是 **runnersUp** 工厂提供商。

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: TITLE,         useValue:   'Hero of the Month' },
{ provide: RUNNERS_UP,    useFactory:  runnersUpFactory(2), deps: [Hero,
HeroService] }
```

这样创建 `TITLE` 令牌：

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
import { InjectionToken } from '@angular/core';

export const TITLE = new InjectionToken<string>('title');
```

类型参数，虽然是可选的，但可以向开发者和开发工具传达类型信息。 而且这个令牌的描述信息也可以为开发者提供帮助。

注入到派生类

当编写一个继承自另一个组件的组件时，要格外小心。如果基础组件有依赖注入，必须要在派生类中重新提供和重新注入它们，并将它们通过构造函数传给基类。

在这个刻意生成的例子里，`SortedHeroesComponent` 继承自 `HeroesBaseComponent`，显示一个**被排序**的英雄列表。

Sorted Heroes

Magma
Mr. Nice
RubberMan

`HeroesBaseComponent` 能自己独立运行。它在自己的实例里要求 `HeroService`，用来得到英雄，并将他们按照数据库返回的顺序显示出来。

src/app/sorted-heroes.component.ts (HeroesBaseComponent)

```
1. @Component({
2.   selector: 'app-unsorted-heroes',
3.   template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
4.   providers: [HeroService]
5. })
6. export class HeroesBaseComponent implements OnInit {
7.   constructor(private heroService: HeroService) { }
8. 
9.   heroes: Array<Hero>;
10. 
11.   ngOnInit() {
12.     this.heroes = this.heroService.getAllHeroes();
13.     this.afterGetHeroes();
14.   }
15. 
16.   // Post-process heroes in derived class override.
17.   protected afterGetHeroes() {}
18. 
19. }
```

> **让构造函数保持简单。**它们只应该用来初始化变量。 这条规则用于在测试环境中放心的构造组件，以免在构造它们时，无意做了一些非常戏剧化的动作(比如与服务器进行会话)。 这就是为什么你要在

> `ngOnInit` 里面调用 `HeroService`，而不是在构造函数中。

用户希望看到英雄按字母顺序排序。与其修改原始的组件，不如派生它，新建 `SortedHeroesComponent`，以便展示英雄之前进行排序。 `SortedHeroesComponent` 让基类来获取英雄。

可惜，Angular 不能直接在基类里直接注入 `HeroService`。必须在**这个**组件里再次提供 `HeroService`，然后通过构造函数传给基类。

src/app/sorted-heroes.component.ts (SortedHeroesComponent)

```
@Component({
  selector: 'app-sorted-heroes',
  template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
  providers: [HeroService]
})
export class SortedHeroesComponent extends HeroesBaseComponent {
  constructor(heroService: HeroService) {
    super(heroService);
  }

  protected afterGetHeroes() {
    this.heroes = this.heroes.sort((h1, h2) => {
      return h1.name < h2.name ? -1 :
            (h1.name > h2.name ? 1 : 0);
    });
  }
}
```

现在，请注意 `afterGetHeroes()` 方法。 你的第一反应是在 `SortedHeroesComponent` 组件里面建一个 `ngOnInit` 方法来做排序。但是 Angular 会先调用**派生**类的 `ngOnInit`，后调用基类的 `ngOnInit`， 所以可能在**英雄到达之前**就开始排序。这就产生了一个讨厌的错误。

覆盖基类的 `afterGetHeroes()` 方法可以解决这个问题。

分析上面的这些复杂性是为了强调**避免使用组件继承**这一点。

通过注入来找到一个父组件

应用程序组件经常需要共享信息。使用松耦合的技术会更好一点，比如数据绑定和服务共享。 但有时候组件确实需要拥有另一个组件的引用，用来访问该组件的属性值或者调用它的方法。

在 Angular 里，获取一个组件的引用比较复杂。虽然 Angular 应用程序是一个组件树，但它没有公共 API 来在该树中巡查和穿梭。

有一个 API 可以获取子级的引用(请看API 参考手册中的 `Query`, `QueryList`, `ViewChildren`,和 `ContentChildren`)。

但没有公共 API 来获取父组件的引用。但是因为每个组件的实例都被加到了依赖注入器的容器中，可以使用 Angular 依赖注入来找到父组件。

本章节描述了这项技术。

找到已知类型的父组件

你使用标准的类注入来获取已知类型的父组件。

在下面的例子中，父组件 `AlexComponent` 有几个子组件，包括 `CathyComponent`:

parent-finder.component.ts (AlexComponent v.1)

```
@Component({
  selector: 'alex',
  template: `
    <div class="a">
      <h3>{{name}}</h3>
      <cathy></cathy>
      <craig></craig>
      <carol></carol>
    </div>`,
})
export class AlexComponent extends Base
{
  name = 'Alex';
}
```

在注入**AlexComponent` 进来后，**Cathy **报告它是否对** Alex* 有访问权:

parent-finder.component.ts (CathyComponent)

```
@Component({
  selector: 'cathy',
  template: `
  <div class="c">
    <h3>Cathy</h3>
    {{alex ? 'Found' : 'Did not find'}} Alex via the component class.<br>
  </div>`
})
export class CathyComponent {
  constructor( @Optional() public alex: AlexComponent ) { }
}
```

注意，这里为安全起见而添加了@Optional装饰器，在线例子 / 下载范例显示 `alex` 参数确实被设置了。

无法通过它的基类找到一个父级

如果**不**知道具体的父组件类名怎么办?

一个可复用的组件可能是多个组件的子级。想象一个用来渲染金融工具头条新闻的组件。由于商业原因，该新闻组件在实时变化的市场数据流过时，要频繁的直接调用其父级工具。

该应用程序可能有多于一打的金融工具组件。如果幸运，它们可能会从同一个基类派生，其 API 是 `NewsComponent` 组件所能理解的。

> 更好的方式是通过接口来寻找实现了它的组件。但这是不可能的，因为 TypeScript 的接口在编译成 JavaScript 以后就消失了，JavaScript 不支持接口。没有东西可查。

这并不是好的设计。问题是**一个组件是否能通过它父组件的基类来注入它的父组件呢**?

`CraigComponent` 例子探究了这个问题。[往回看 Alex]{guide/dependency-injection-in-action#alex}，你看到 `Alex` 组件**扩展**(**派生**)自一个叫 `Base` 的类。

parent-finder.component.ts (Alex class signature)

```
export class AlexComponent extends Base
```

`CraigComponent` 试图把 `Base` 注入到到它的 `alex` 构造函数参数，来报告是否成功。

parent-finder.component.ts (CraigComponent)

```
@Component({
  selector: 'craig',
  template: `
  <div class="c">
    <h3>Craig</h3>
    {{alex ? 'Found' : 'Did not find'}} Alex via the base class.
  </div>`
})
export class CraigComponent {
  constructor( @Optional() public alex: Base ) { }
}
```

可惜这样不行。在线例子 / 下载范例显示 `alex` 参数是 null。 **不能通过基类注入父组件。**

通过类-接口找到父组件

可以通过类-接口找到一个父组件。

该父组件必须通过提供一个与**类-接口**令牌同名的**别名**来与之合作。

请记住 Angular 总是从它自己的注入器添加一个组件实例；这就是为什么在之前可以 **Alex** 注入到 **Carol**。

编写一个**别名提供商** — 一个拥有 `useExisting` 定义的 `provide` 函数 — 它新建一个**备选的**方式来注入同一个组件实例，并把这个提供商添加到 `AlexComponent` 的 `@Component` 元数据里的 `providers` 数组。

parent-finder.component.ts (AlexComponent providers)

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```

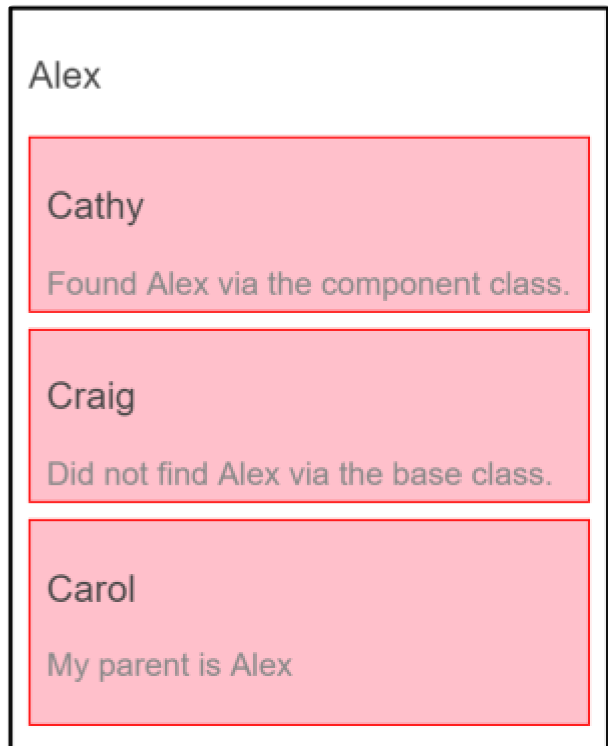
Parent是该提供商的**类-接口**令牌。`AlexComponent` 引用了自身，造成循环引用，使用**forwardRef**打破了该循环。

Carol，**Alex** 的第三个子组件，把父级注入到了自己的 `parent` 参数，和之前做的一样：

parent-finder.component.ts (CarolComponent class)

```
export class CarolComponent {
  name = 'Carol';
  constructor( @Optional() public parent: Parent ) { }
}
```

下面是 **Alex** 和其家庭的运行结果：



通过父级树找到父组件

想象组件树中的一个分支为：**Alice** -> **Barry** -> **Carol**。**Alice** 和 **Barry** 都实现了这个 `Parent`**类-接口**。

Barry 是个问题。它需要访问它的父组件 **Alice**，但同时它也是 **Carol** 的父组件。这个意味着它必须同时**注入**`Parent`**类-接口**来获取 **Alice**，和**提供**一个 `Parent` 来满足 **Carol**。

下面是 **Barry** 的代码：

parent-finder.component.ts (BarryComponent)

```
const templateB = `
  <div class="b">
    <div>
      <h3>{{name}}</h3>
      <p>My parent is {{parent?.name}}</p>
    </div>
    <carol></carol>
    <chris></chris>
  </div>`;

@Component({
  selector:   'barry',
  template:   templateB,
  providers:  [{ provide: Parent, useExisting: forwardRef(() => BarryComponent) }]
})
export class BarryComponent implements Parent {
  name = 'Barry';
  constructor( @SkipSelf() @Optional() public parent: Parent ) { }
}
```

Barry 的 `providers` 数组看起来很像**Alex** 的那个. 如果准备一直像这样编写**别名提供商**的话，你应该建立一个辅助函数。

眼下，请注意 **Barry** 的构造函数：

Barry's constructor *Carol's constructor*

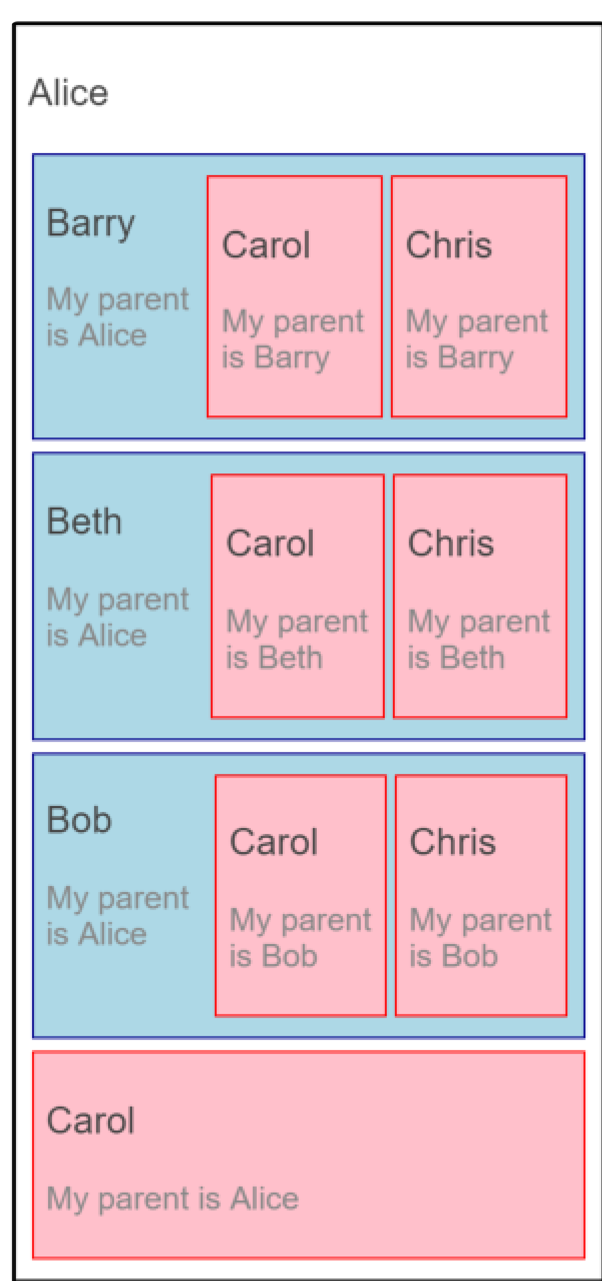
```
constructor( @SkipSelf() @Optional() public parent: Parent ) { }
```

除额外添加了一个的 `@SkipSelf` 外，它和 **Carol** 的构造函数一样。

添加 `@SkipSelf` 主要是出于两个原因：

1. 它告诉注入器从一个在自己**上一级**的组件开始搜索一个 `Parent` 依赖。
2. 如果没写 `@SkipSelf` 装饰器的话，Angular 就会抛出一个循环依赖错误。
   `不能创建循环依赖实例！(BethComponent -> Parent -> BethComponent)`

这里是 **Alice**，**Barry** 和该家庭的操作演示：



Parent 类-接口

你以前学过：**类-接口**是一个抽象类，被当成一个接口使用，而非基类。

这个例子定义了一个 `Parent` **类-接口**。

parent-finder.component.ts (Parent class-interface)

```
export abstract class Parent { name: string; }
```

该 `Parent` **类-接口**定义了 `Name` 属性，它有类型声明，但是**没有实现**，该 `name` 是该父级的所有子组件们唯一能调用的属性。 这种“窄接口”有助于解耦子组件类和它的父组件。

一个能用做父级的组件**应该**实现**类-接口**，和下面的 `AliceComponent` 的做法一样：

parent-finder.component.ts (AliceComponent class signature)

```
export class AliceComponent implements Parent
```

这样做可以提升代码的清晰度，但严格来说并不是必须的。虽然 `AlexComponent` 有一个 `name` 属性(来自 `Base` 类的要求)，但它的类签名并不需要提及 `Parent`。

parent-finder.component.ts (AlexComponent class signature)

```
export class AlexComponent extends Base
```

> 为了正确的代码风格，该 `AlexComponent`**应该**实现 `Parent`。在这个例子里它没有这样，只是为了演示在没有该接口的情况下，该代码仍会被正确编译并运行。

provideParent()助手函数

编写父组件相同的各种**别名提供商**很快就会变得啰嗦，在用**forwardRef**的时候尤其绕口：

dependency-injection-in-action/src/app/parent-finder.component.ts

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```

可以像这样把该逻辑抽取到一个助手函数里：

dependency-injection-in-action/src/app/parent-finder.component.ts

```
// Helper method to provide the current component instance in the name of a
`parentType`.
const provideParent =
  (component: any) => {
    return { provide: Parent, useExisting: forwardRef(() => component) };
  };
```

现在就可以为组件添加一个更简单、直观的父级提供商了：

dependency-injection-in-action/src/app/parent-finder.component.ts

```
providers:  [ provideParent(AliceComponent) ]
```

你可以做得更好。当前版本的助手函数只能为 `Parent`**类-接口**提供别名。应用程序可能有很多类型的父组件，每个父组件有自己的**类-接口**令牌。

下面是一个修改版本，默认接受一个 `Parent`，但同时接受一个可选的第二参数，可以用来指定一个不同的父级**类-接口**。

dependency-injection-in-action/src/app/parent-finder.component.ts

```
// Helper method to provide the current component instance in the name of a
`parentType`.
// The `parentType` defaults to `Parent` when omitting the second parameter.
const provideParent =
  (component: any, parentType?: any) => {
    return { provide: parentType || Parent, useExisting: forwardRef(() =>
component) };
  };
```

下面的代码演示了如何使它添加一个不同类型的父级：

dependency-injection-in-action/src/app/parent-finder.component.ts

```
providers:  [ provideParent(BethComponent, DifferentParent) ]
```

使用一个前向引用(forwardRef)来打破循环

在 TypeScript 里面，类声明的顺序是很重要的。如果一个类尚未定义，就不能引用它。

这通常不是一个问题，特别是当你遵循**一个文件一个类**规则的时候。 但是有时候循环引用可能不能避免。当一个类**A 引用类 B**，同时'B'引用'A'的时候，你就陷入困境了：它们中间的某一个必须要先定义。

Angular 的 `forwardRef()` 函数建立一个**间接地**引用，Angular 可以随后解析。

Parent Finder是一个充满了无法解决的循环引用的例子

当一个类**需要引用自身**的时候，你面临同样的困境，就像在 `AlexComponent` 的 `provdiers` 数组中遇到的困境一样。 该 `providers` 数组是一个 `@Component` 装饰器函数的一个属性，它必须在类定义**之前**出现。

使用 `forwardRef` 来打破这种循环：

parent-finder.component.ts (AlexComponent providers)

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```

HttpClient

大多数前端应用都需要通过 HTTP 协议与后端服务器通讯。现代浏览器支持使用两种不同的 API 发起 HTTP 请求：`XMLHttpRequest` 接口和 `fetch()` API。

`@angular/common/http` 中的 `HttpClient` 类为 Angular 应用程序提供了一个简化的 API 来实现 HTTP 客户端功能。它基于浏览器提供的 `XMLHttpRequest` 接口。 `HttpClient` 带来的其它优点包括：可测试性、强类型的请求和响应对象、发起请求与接收响应时的拦截器支持，以及更好的、基于可观察（Observable）对象的 API 以及流式错误处理机制。

你可以到 在线例子 / 下载范例 中运行本章的代码。

> 该应用代码并不需要数据服务器。 它基于 Angular **in-memory-web-api** 库，该库会替换 `HttpClient` 模块中的 `HttpBackend`。用于替换的这个服务会模拟 REST 风格的后端的行为。
>
> 到 `AppModule` 的 `imports` 中查看这个库是如何配置的。

准备工作

要想使用 `HttpClient`，就要先导入 Angular 的 `HttpClientModule`。大多数应用都会在根模块 `AppModule` 中导入它。

app/app.module.ts (excerpt)

```
import { NgModule }         from '@angular/core';
import { BrowserModule }    from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

在 `AppModule` 中导入 `HttpClientModule` 之后，你可以把 `HttpClient` 注入到应用类中，就像下面的 `ConfigService` 例子中这样。

app/config/config.service.ts (excerpt)

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```

获取 JSON 数据

应用通常会从服务器上获取 JSON 数据。 比如，该应用可能要从服务器上获取配置文件 `config.json`，其中指定了一些特定资源的 URL。

assets/config.json

```
{
  "heroesUrl": "api/heroes",
  "textfile": "assets/textfile.txt"
}
```

`ConfigService` 会通过 `HttpClient` 的 `get()` 方法取得这个文件。

app/config/config.service.ts (getConfig v.1)

```
configUrl = 'assets/config.json';

getConfig() {
  return this.http.get(this.configUrl);
}
```

像 `ConfigComponent` 这样的组件会注入 `ConfigService`，并调用其 `getConfig` 方法。

app/config/config.component.ts (showConfig v.1)

```
showConfig() {
  this.configService.getConfig()
    .subscribe((data: Config) => this.config = {
        heroesUrl: data['heroesUrl'],
        textfile:  data['textfile']
    });
}
```

这个服务方法返回配置数据的 `Observable` 对象，所以组件要订阅（subscribe） 该方法的返回值。 订阅时的回调函数会把这些数据字段复制到组件的 `config` 对象中，它会在组件的模板中绑定，以供显示。

为什么要写服务

这个例子太简单，所以它也可以在组件本身的代码中调用 `Http.get()`，而不用借助服务。

不过，数据访问很少能一直这么简单。 你通常要对数据做后处理、添加错误处理器，还可能加一些重试逻辑，以便应对网络抽风的情况。

该组件很快就会因为这些数据方式的细节而变得杂乱不堪。 组件变得难以理解、难以测试，并且这些数据访问逻辑无法被复用，也无法标准化。

这就是为什么最佳实践中要求把数据展现逻辑从数据访问逻辑中拆分出去，也就是说把数据访问逻辑包装进一个单独的服务中， 并且在组件中把数据访问逻辑委托给这个服务。就算是这么简单的应用也要如此。

带类型检查的响应

该订阅的回调需要用通过括号中的语句来提取数据的值。

```
.subscribe((data: Config) => this.config = {
    heroesUrl: data['heroesUrl'],
    textfile:  data['textfile']
});
```

你没法写成 `data.heroesUrl` ，因为 TypeScript 会报告说来自服务器的 `data` 对象中没有名叫 `heroesUrl` 的属性。

这是因为 `HttpClient.get()` 方法把 JSON 响应体解析成了匿名的 `Object` 类型。它不知道该对象的具体形态如何。

你可以告诉 `HttpClient` 该响应体的类型，以便让对这种输出的消费更容易、更明确。

首先，定义一个具有正确形态的接口：

```
export interface Config {
  heroesUrl: string;
  textfile: string;
}
```

然后，在服务器中把该接口指定为 `HttpClient.get()` 调用的类型参数。

app/config/config.service.ts (getConfig v.2)

```
getConfig() {
  // now returns an Observable of Config
  return this.http.get<Config>(this.configUrl);
}
```

修改后的组件方法，其回调函数中获取一个带类型的对象，它易于使用，且消费起来更安全：

app/config/config.component.ts (showConfig v.2)

```
config: Config;

showConfig() {
  this.configService.getConfig()
    // clone the data object, using its known Config shape
    .subscribe((data: Config) => this.config = { ...data });
}
```

读取完整的响应体

响应体可能并不包含你需要的全部信息。有时候服务器会返回一个特殊的响应头或状态码，以标记出特定的条件，因此读取它们可能是必要的。

要这样做，你就要通过 `observe` 选项来告诉 `HttpClient`，你想要完整的响应信息，而不是只有响应体：

```
getConfigResponse(): Observable<HttpResponse<Config>> {
  return this.http.get<Config>(
    this.configUrl, { observe: 'response' });
}
```

现在 `HttpClient.get()` 会返回一个 `HttpResponse` 类型的 `Observable`，而不只是 JSON 数据。

该组件的 `showConfigResponse()` 方法会像显示配置数据一样显示响应头：

app/config/config.component.ts (showConfigResponse)

```
showConfigResponse() {
  this.configService.getConfigResponse()
    // resp is of type `HttpResponse<Config>`
    .subscribe(resp => {
      // display its headers
      const keys = resp.headers.keys();
      this.headers = keys.map(key =>
        `${key}: ${resp.headers.get(key)}`);

      // access the body directly, which is typed as `Config`.
      this.config = { ... resp.body };
    });
}
```

如你所见，该响应对象具有一个带有正确类型的 `body` 属性。

错误处理

如果这个请求导致了服务器错误怎么办？甚至，在烂网络下请求都没到服务器该怎么办？`HttpClient` 就会返回一个错误（error）而不再是成功的响应。

通过在 `.subscribe()` 中添加第二个回调函数，你**可以**在组件中处理它：

app/config/config.component.ts (showConfig v.3 with error handling)

```
showConfig() {
  this.configService.getConfig()
    .subscribe(
      (data: Config) => this.config = { ...data }, // success path
      error => this.error = error // error path
    );
}
```

在数据访问失败时给用户一些反馈，确实是个好主意。 不过，直接显示由 `HttpClient` 返回的原始错误数据还远远不够。

获取错误详情

检测错误的发生是第一步，不过如果知道具体发生了什么错误才会更有用。上面例子中传给回调函数的 `err` 参数的类型是 `HttpErrorResponse`，它包含了这个错误中一些很有用的信息。

可能发生的错误分为两种。如果后端返回了一个失败的返回码（如 404、500 等），它会返回一个错误响应体。

或者，如果在客户端这边出了错误（比如在 RxJS 操作符 (operator) 中抛出的异常或某些阻碍完成这个请求的网络错误），就会抛出一个 `Error` 类型的异常。

`HttpClient` 会在 `HttpErrorResponse` 中捕获所有类型的错误信息，你可以查看这个响应体以了解到底发生了什么。

错误的探查、解释和解决是你应该在**服务**中做的事情，而不是在**组件**中。

你可能首先要设计一个错误处理器，就像这样：

app/config/config.service.ts (handleError)

```
private handleError(error: HttpErrorResponse) {
  if (error.error instanceof ErrorEvent) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error.message);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong,
    console.error(
      `Backend returned code ${error.status}, ` +
      `body was: ${error.error}`);
  }
  // return an observable with a user-facing error message
  return throwError(
    'Something bad happened; please try again later.');
};
```

注意，该处理器返回一个带有用户友好的错误信息的 RxJS `ErrorObservable` 对象。 该服务的消费者期望服务的方法返回某种形式的 `Observable`，就算是“错误的”也可以。

现在，你获取了由 `HttpClient` 方法返回的 `Observable`，并**把它们通过管道**传给错误处理器。

app/config/config.service.ts (getConfig v.3 with error handler)

```
getConfig() {
  return this.http.get<Config>(this.configUrl)
    .pipe(
      catchError(this.handleError)
    );
}
```

`retry()`

有时候，错误只是临时性的，只要重试就可能会自动消失。 比如，在移动端场景中可能会遇到网络中断的情况，只要重试一下就能拿到正确的结果。

RxJS 库提供了几个 `retry` 操作符，它们值得仔细看看。 其中最简单的是 `retry()`，它可以对失败的 `Observable` 自动重新订阅几次。对 `HttpClient` 方法调用的结果进行**重新订阅**会导致重新发起 HTTP 请求。

把它插入到 `HttpClient` 方法结果的**管道**中，就放在错误处理器的紧前面。

app/config/config.service.ts (getConfig with retry)

```
getConfig() {
  return this.http.get<Config>(this.configUrl)
    .pipe(
      retry(3), // retry a failed request up to 3 times
      catchError(this.handleError) // then handle the error
    );
}
```

可观察对象 (Observable) 与操作符 (operator)

本章的前一节中引用了 RxJS 的 `Observable` 和 `operator`，比如 `catchError` 和 `retry`。 接下来你还会遇到更多 RxJS 中的概念。

RxJS 是一个库，用于把异步调用和基于回调的代码组合成**函数式（functional）的、响应式（reactive）的**风格。 很多 Angular API，包括 `HttpClient` 都会生成和消费 RxJS 的 `Observable`。

RxJS 本身超出了本章的范围。你可以在网络上找到更多的学习资源。 虽然只用少量的 RxJS 知识就可以获得解决方案，不过以后你会逐步提高 RxJS 技能，以便更高效的使用 `HttpClient`。

如果你在跟着教程敲下面这些代码片段，要注意你要自己导入这里出现的 RxJS 的可观察对象和操作符。就像 `ConfigService` 中的这些导入。

app/config/config.service.ts (RxJS imports)

```
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
```

请求非 JSON 格式的数据

不是所有的 API 都会返回 JSON 数据。在下面这个例子中，`DownloaderService` 中的方法会从服务器读取文本文件， 并把文件的内容记录下来，然后把这些内容使用 `Observable<string>` 的形式返回给调用者。

app/downloader/downloader.service.ts (getTextFile)

```
getTextFile(filename: string) {
  // The Observable returned by get() is of type Observable<string>
  // because a text response was specified.
  // There's no need to pass a <string> type parameter to get().
  return this.http.get(filename, {responseType: 'text'})
    .pipe(
      tap( // Log the result or error
        data => this.log(filename, data),
        error => this.logError(filename, error)
      )
    );
}
```

这里的 `HttpClient.get()` 返回字符串而不是默认的 JSON 对象，因为它的 `responseType` 选项是 `'text'`。

RxJS 的 `tap` 操作符（可看做 wiretap - 窃听），让这段代码探查由可观察对象传过来的正确值和错误值，而不用打扰它们。

在 `DownloaderComponent` 中的 `download()` 方法通过订阅这个服务中的方法来发起一次请求。

app/downloader/downloader.component.ts (download)

```
download() {
  this.downloaderService.getTextFile('assets/textfile.txt')
    .subscribe(results => this.contents = results);
}
```

把数据发送到服务器

除了从服务器获取数据之外，`HttpClient` 还支持修改型的请求，也就是说，通过 `PUT`、`POST`、`DELETE` 这样的 HTTP 方法把数据发送到服务器。

本指南中的这个范例应用包括一个简化版本的《英雄指南》，它会获取英雄数据，并允许用户添加、删除和修改它们。

下面的这些章节中包括该范例的 `HeroesService` 中的一些方法片段。

添加请求头

很多服务器在进行保存型操作时需要额外的请求头。 比如，它们可能需要一个 `Content-Type` 头来显式定义请求体的 MIME 类型。 也可能服务器会需要一个认证用的令牌（token）。

`HeroesService` 在 `httpOptions` 对象中就定义了一些这样的请求头，并把它传给每个 `HttpClient` 的保存型方法。

app/heroes/heroes.service.ts (httpOptions)

```
import { HttpHeaders } from '@angular/common/http';

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type':  'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

发起一个 POST 请求

应用经常把数据 `POST` 到服务器。它们会在提交表单时进行 `POST`。 下面这个例子中，`HeroesService` 在把英雄添加到数据库中时，就会使用 `POST`。

app/heroes/heroes.service.ts (addHero)

```
/** POST: add a new hero to the database */
addHero (hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('addHero', hero))
    );
}
```

`HttpClient.post()` 方法像 `get()` 一样也有类型参数（你会希望服务器返回一个新的英雄对象），它包含一个资源 URL。

它还接受另外两个参数：

1. `hero` - 要 `POST` 的请求体数据。
2. `httpOptions` - 这个例子中，该方法的选项指定了所需的请求头。

当然，它捕获错误的方式很像前面描述的操作方式。

`HeroesComponent` 通过订阅该服务方法返回的 `Observable` 发起了一次实际的 `POST` 操作。

app/heroes/heroes.component.ts (addHero)

```
this.heroesService.addHero(newHero)
  .subscribe(hero => this.heroes.push(hero));
```

当服务器成功做出响应时，会带有这个新创建的英雄，然后该组件就会把这个英雄添加到正在显示的 `heroes` 列表中。

发起 `DELETE` 请求

该应用可以把英雄的 id 传给 `HttpClient.delete` 方法的请求 URL 来删除一个英雄。

app/heroes/heroes.service.ts (deleteHero)

```
/** DELETE: delete the hero from the server */
deleteHero (id: number): Observable<{}> {
  const url = `${this.heroesUrl}/${id}`; // DELETE api/heroes/42
  return this.http.delete(url, httpOptions)
    .pipe(
      catchError(this.handleError('deleteHero'))
    );
}
```

当 `HeroesComponent` 订阅了该服务方法返回的 `Observable` 时，就会发起一次实际的 `DELETE` 操作。

app/heroes/heroes.component.ts (deleteHero)

```
this.heroesService.deleteHero(hero.id).subscribe();
```

该组件不会等待删除操作的结果，所以它的 subscribe （订阅）中没有回调函数。不过就算你不关心结果，也仍然要订阅它。调用 `subscribe()` 方法会执行这个可观察对象，这时才会真的发起 DELETE 请求。

> 你必须调用 `subscribe()`，否则什么都不会发生。仅仅调用 `HeroesService.deleteHero()` 是不会发起 DELETE 请求的。

```
// oops ... subscribe() is missing so nothing happens
this.heroesService.deleteHero(hero.id);
```

别忘了**订阅**!

在调用方法返回的可观察对象的 `subscribe()` 方法之前，`HttpClient` 方法不会发起 HTTP 请求。这适用于 `HttpClient` 的**所有方法**。

> `AsyncPipe` 会自动为你订阅（以及取消订阅）。

`HttpClient` 的所有方法返回的可观察对象都设计为**冷的**。 HTTP 请求的执行都是**延期执行的**，让你可以用 `tap` 和 `catchError` 这样的操作符来在实际执行什么之前，先对这个可观察对象进行扩展。

调用 `subscribe(...)` 会触发这个可观察对象的执行，并导致 `HttpClient` 组合并把 HTTP 请求发给服务器。

你可以把这些可观察对象看做实际 HTTP 请求的**蓝图**。

> 实际上，每个 `subscribe()` 都会初始化此可观察对象的一次单独的、独立的执行。 订阅两次就会导致发起两个 HTTP 请求。
>
> ```
> const req = http.get<Heroes>('/api/heroes');
> // 0 requests made - .subscribe() not called.
> req.subscribe();
> // 1 request made.
> req.subscribe();
> // 2 requests made.
> ```

发起 PUT 请求

应用可以发送 PUT 请求，来使用修改后的数据完全替换掉一个资源。 下面的 `HeroesService` 例子和 POST 的例子很像。

app/heroes/heroes.service.ts (updateHero)

```
/** PUT: update the hero on the server. Returns the updated hero upon success. */
updateHero (hero: Hero): Observable<Hero> {
  return this.http.put<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('updateHero', hero))
    );
}
```

因为前面解释过的原因，调用者（这里是 `HeroesComponent.update()`）必须 `subscribe()` 由 `HttpClient.put()` 返回的可观察对象，以发起这个调用。

高级用法

我们已经讨论了 `@angular/common/http` 的基本 HTTP 功能，但有时候除了单纯发起请求和取回数据之外，你还要再做点别的。

配置请求

待发送请求的其它方面可以通过传给 `HttpClient` 方法最后一个参数中的配置对象进行配置。

以前你曾在 `HeroesService` 中通过在其保存方法中传入配置对象 `httpOptions` 设置过默认头。 你还可以做更多。

修改这些头

你没法直接修改前述配置对象中的现有头，因为这个 `HttpHeaders` 类的实例是不可变的。

改用 `set()` 方法代替。 它会返回当前实例的一份克隆，其中应用了这些新修改。

比如在发起下一个请求之前，如果旧的令牌已经过期了，你可能还要修改认证头。

```
httpOptions.headers =
  httpOptions.headers.set('Authorization', 'my-new-auth-token');
```

URL 参数

添加 URL 搜索参数也与此类似。 这里的 `searchHeroes` 方法会查询名字中包含搜索词的英雄列表。

```
/* GET heroes whose name contains search term */
searchHeroes(term: string): Observable<Hero[]> {
  term = term.trim();

  // Add safe, URL encoded search parameter if there is a search term
  const options = term ?
   { params: new HttpParams().set('name', term) } : {};

  return this.http.get<Hero[]>(this.heroesUrl, options)
    .pipe(
      catchError(this.handleError<Hero[]>('searchHeroes', []))
    );
}
```

如果有搜索词，这段代码就会构造一个包含进行过 URL 编码的搜索词的选项对象。如果这个搜索词是“foo”，这个 GET 请求的 URL 就会是 `api/heroes/?name=foo`。

如果有搜索词，这段代码就会构造一个包含进行过 URL 编码的搜索词的选项对象。如果这个搜索词是“foo”，这个 GET 请求的 URL 就会是 `api/heroes/?name=foo`。

`HttpParams` 是不可变的，所以你也要使用 `set()` 方法来修改这些选项。

请求的防抖（debounce）

这个例子还包含了**搜索 npm 包**的特性。

当用户在搜索框中输入名字时，`PackageSearchComponent` 就会把一个根据名字搜索包的请求发送给 NPM 的 web api。

下面是模板中的相关代码片段：

app/package-search/package-search.component.html (search)

```
<input (keyup)="search($event.target.value)" id="name" placeholder="Search"/>

<ul>
  <li *ngFor="let package of packages$ | async">
    <b>{{package.name}} v.{{package.version}}</b> -
    <i>{{package.description}}</i>
  </li>
</ul>
```

`(keyup)` 事件绑定把每次击键都发送给了组件的 `search()` 方法。

如果每次击键都发送一次请求就太昂贵了。 最好能等到用户停止输入时才发送请求。 使用 RxJS 的操作符就能轻易实现它，参见下面的代码片段：

app/package-search/package-search.component.ts (excerpt))

```
withRefresh = false;
packages$: Observable<NpmPackageInfo[]>;
private searchText$ = new Subject<string>();

search(packageName: string) {
  this.searchText$.next(packageName);
}

ngOnInit() {
  this.packages$ = this.searchText$.pipe(
    debounceTime(500),
    distinctUntilChanged(),
    switchMap(packageName =>
      this.searchService.search(packageName, this.withRefresh))
  );
}

constructor(private searchService: PackageSearchService) { }
```

`searchText$` 是一个序列，包含用户输入到搜索框中的所有值。 它定义成了 RxJS 的 `Subject` 对象，这表示它是一个多播 `Observable`，同时还可以自行调用 `next(value)` 来产生值。 `search()` 方法中就是这么做的。

除了把每个 `searchText` 的值都直接转发给 `PackageSearchService` 之外， `ngOnInit()` 中的代码还通过下列三个操作符对这些搜索值进行**管道**处理：

1. `debounceTime(500)` - 等待，直到用户停止输入（这个例子中是停止 1/2 秒）。
2. `distinctUntilChanged()` - 等待，直到搜索内容发生了变化。
3. `switchMap()` - 把搜索请求发送给服务。

这些代码把 `packages$` 设置成了使用搜索结果组合出的 `Observable` 对象。 模板中使用 AsyncPipe 订阅了 `packages$`，一旦搜索结果的值发回来了，就显示这些搜索结果。

这样，只有当用户停止了输入且搜索值和以前不一样的时候，搜索值才会传给服务。

> 稍后 解释了这个 `withRefresh` 选项。

switchMap()

这个 `switchMap()` 操作符有三个重要的特征：

1. 它的参数是一个返回 `Observable` 的函数。`PackageSearchService.search` 会返回 `Observable`，其它数据服务也一样。
2. 如果以前的搜索结果仍然是**在途**状态（这会出现在慢速网络中），它会取消那个请求，并发起这个新的搜索。
3. 它会按照原始的请求顺序返回这些服务的响应，而不用关心服务器实际上是以乱序返回的它们。

> 如果你觉得将来会复用这些防抖逻辑， 可以把它移到单独的工具函数中，或者移到 `PackageSearchService` 中。

拦截请求和响应

HTTP 拦截机制是 `@angular/common/http` 中的主要特性之一。 使用这种拦截机制，你可以声明**一些拦截器**，用它们监视和转换从应用发送到服务器的 HTTP 请求。 拦截器还可以用监视和转换从服务器返回到本应用的那些响应。 多个选择器会构成一个“请求/响应处理器”的双向链表。

拦截器可以用一种常规的、标准的方式对每一次 HTTP 的请求/响应任务执行从认证到记日志等很多种**隐式**任务。

如果没有拦截机制，那么开发人员将不得不对每次 `HttpClient` 调用**显式**实现这些任务。

编写拦截器

要实现拦截器，就要实现一个实现了 `HttpInterceptor` 接口中的 `intercept()` 方法的类。

这里是一个什么也不做的**空白**拦截器，它只会不做任何修改的传递这个请求。

app/http-interceptors/noop-interceptor.ts

```
import { Injectable } from '@angular/core';
import {
  HttpEvent, HttpInterceptor, HttpHandler, HttpRequest
} from '@angular/common/http';

import { Observable } from 'rxjs';

/** Pass untouched request through to the next request handler. */
@Injectable()
export class NoopInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

`intercept` 方法会把请求转换成一个最终返回 HTTP 响应体的 `Observable`。 在这个场景中，每个拦截器都完全能自己处理这个请求。

大多数拦截器拦截都会在传入时检查请求，然后把（可能被修改过的）请求转发给 `next` 对象的 `handle()` 方法，而 `next` 对象实现了 `HttpHandler` 接口。

```
export abstract class HttpHandler {
  abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>;
}
```

像 `intercept()` 一样， `handle()` 方法也会把 HTTP 请求转换成 `HttpEvents` 组成的 `Observable`，它最终包含的是来自服务器的响应。 `intercept()` 函数可以检查这个可观察对象，并在把它返回给调用者之前修改它。

这个**无操作的**拦截器，会直接使用原始的请求调用 `next.handle()`，并返回它返回的可观察对象，而不做任何后续处理。

`next` 对象

`next` 对象表示拦截器链表中的下一个拦截器。 这个链表中的最后一个 `next` 对象就是 `HttpClient` 的后端处理器（backend handler），它会把请求发给服务器，并接收服务器的响应。

大多数的拦截器都会调用 `next.handle()`，以便这个请求流能走到下一个拦截器，并最终传给后端处理器。 拦截器也**可以**不调用 `next.handle()`，使这个链路短路，并返回一个带有人工构造出来的服务器响应的 自己的 `Observable`。

这是一种常见的中间件模式，在像 Express.js 这样的框架中也会找到它。

提供这个拦截器

这个 `NoopInterceptor` 就是一个由 Angular 依赖注入 (DI)系统管理的服务。 像其它服务一样，你也必须先提供这个拦截器类，应用才能使用它。

由于拦截器是 `HttpClient` 服务的（可选）依赖，所以你必须在提供 `HttpClient` 的同一个（或其各级父注入器）注入器中提供这些拦截器。 那些在 DI 创建完 `HttpClient` **之后**再提供的拦截器将会被忽略。

由于在 `AppModule` 中导入了 `HttpClientModule`，导致本应用在其根注入器中提供了 `HttpClient`。所以你也同样要在 `AppModule` 中提供这些拦截器。

在从 `@angular/common/http` 中导入了 `HTTP_INTERCEPTORS` 注入令牌之后，编写如下的 `NoopInterceptor` 提供商注册语句：

```
{ provide: HTTP_INTERCEPTORS, useClass: NoopInterceptor, multi: true },
```

注意 `multi: true` 选项。 这个必须的选项会告诉 Angular `HTTP_INTERCEPTORS` 是一个**多重提供商**的令牌，表示它会注入一个多值的数组，而不是单一的值。

你**也可以**直接把这个提供商添加到 `AppModule` 中的提供商数组中，不过那样会非常啰嗦。况且，你将来还会用这种方式创建更多的拦截器并提供它们。 你还要特别注意提供这些拦截器的顺序。

认真考虑创建一个封装桶（barrel）文件，用于把所有拦截器都收集起来，一起提供给 `httpInterceptorProviders` 数组，可以先从这个 `NoopInterceptor` 开始。

app/http-interceptors/index.ts

```
/* "Barrel" of Http Interceptors */
import { HTTP_INTERCEPTORS } from '@angular/common/http';

import { NoopInterceptor } from './noop-interceptor';

/** Http interceptor providers in outside-in order */
export const httpInterceptorProviders = [
  { provide: HTTP_INTERCEPTORS, useClass: NoopInterceptor, multi: true },
];
```

然后导入它，并把它加到 `AppModule` 的 `providers` **数组**中，就像这样：

app/app.module.ts (interceptor providers)

```
providers: [
  httpInterceptorProviders
],
```

当你再创建新的拦截器时，就同样把它们添加到 `httpInterceptorProviders` 数组中，而不用再修改 `AppModule`。

> 在完整版的范例代码中还有更多的拦截器。

拦截器的顺序

Angular 会按照你提供它们的顺序应用这些拦截器。 如果你提供拦截器的顺序是先 **A**，再 **B**，再 **C**，那么请求阶段的执行顺序就是 **A->B->C**，而响应阶段的执行顺序则是 **C->B->A**。

以后你就再也不能修改这些顺序或移除某些拦截器了。 如果你需要动态启用或禁用某个拦截器，那就要在那个拦截器中自行实现这个功能。

HttpEvents

你可能会期望 `intercept()` 和 `handle()` 方法会像大多数 `HttpClient` 中的方法那样返回 `HttpResponse<any>` 的可观察对象。

然而并没有，它们返回的是 `HttpEvent<any>` 的可观察对象。

这是因为拦截器工作的层级比那些 `HttpClient` 方法更低一些。每个 HTTP 请求都可能会生成很多个**事件**，包括上传和下载的进度事件。 实际上，`HttpResponse` 类本身就是一个事件，它的类型（`type`）是 `HttpEventType.HttpResponseEvent`。

很多拦截器只关心发出的请求，而对 `next.handle()` 返回的事件流不会做任何修改。

但那些要检查和修改来自 `next.handle()` 的响应体的拦截器希望看到所有这些事件。 所以，你的拦截器应该返回**你没碰过的所有事件**，除非你**有充分的理由不这么做**。

不可变性

虽然拦截器有能力改变请求和响应，但 `HttpRequest` 和 `HttpResponse` 实例的属性却是只读（`readonly`）的， 因此，它们在很大意义上说是不可变对象。

有充足的理由把它们做成不可变对象：应用可能会重试发送很多次请求之后才能成功，这就意味着这个拦截器链表可能会多次重复处理同一个请求。 如果拦截器可以修改原始的请求对象，那么重试阶段的操作就会从修改过的请求开始，而不是原始请求。 而这种不可变性，可以确保这些拦截器在每次重试时看到的都是同样的原始请求。

通过把 `HttpRequest` 的属性设置为只读的，TypeScript 可以防止你犯这种错误。

```
// Typescript disallows the following assignment because req.url is readonly
req.url = req.url.replace('http://', 'https://');
```

要想修改该请求，就要先克隆它，并修改这个克隆体，然后再把这个克隆体传给 `next.handle()`。 你可以用一步操作中完成对请求的克隆和修改，例子如下：

app/http-interceptors/ensure-https-interceptor.ts (excerpt)

```
// clone request and replace 'http://' with 'https://' at the same time
const secureReq = req.clone({
  url: req.url.replace('http://', 'https://')
});
// send the cloned, "secure" request to the next handler.
return next.handle(secureReq);
```

这个 `clone()` 方法的哈希型参数允许你在复制出克隆体的同时改变该请求的某些特定属性。

请求体

`readonly` 这种赋值保护，无法防范深修改（修改子对象的属性），也不能防范你修改请求体对象中的属性。

```
req.body.name = req.body.name.trim(); // bad idea!
```

如果你必须修改请求体，那么就要先复制它，然后修改这个复本，`clone()` 这个请求，然后把这个请求体的复本作为新的请求体，例子如下：

app/http-interceptors/trim-name-interceptor.ts (excerpt)

```
// copy the body and trim whitespace from the name property
const newBody = { ...body, name: body.name.trim() };
// clone request and set its body
const newReq = req.clone({ body: newBody });
// send the cloned request to the next handler.
return next.handle(newReq);
```

清空请求体

有时你需要清空请求体，而不是替换它。 如果你把克隆后的请求体设置成 `undefined`，Angular 会认为你是想让这个请求体保持原样。 这显然不是你想要的。 但如果把克隆后的请求体设置成 `null`，那 Angular 就知

道你是想清空这个请求体了。

```
newReq = req.clone({ ... }); // body not mentioned => preserve original body
newReq = req.clone({ body: undefined }); // preserve original body
newReq = req.clone({ body: null }); // clear the body
```

设置默认请求头

应用通常会使用拦截器来设置外发请求的默认请求头。

该范例应用具有一个 `AuthService`，它会生成一个认证令牌。 在这里，`AuthInterceptor` 会注入该服务以获取令牌，并对每一个外发的请求添加一个带有该令牌的认证头：

app/http-interceptors/auth-interceptor.ts

```
1. import { AuthService } from '../auth.service';
2.
3. @Injectable()
4. export class AuthInterceptor implements HttpInterceptor {
5.
6.   constructor(private auth: AuthService) {}
7.
8.   intercept(req: HttpRequest<any>, next: HttpHandler) {
9.     // Get the auth token from the service.
10.    const authToken = this.auth.getAuthorizationToken();
11.
12.    // Clone the request and replace the original headers with
13.    // cloned headers, updated with the authorization.
14.    const authReq = req.clone({
15.      headers: req.headers.set('Authorization', authToken)
16.    });
17.
18.    // send cloned request with header to the next handler.
19.    return next.handle(authReq);
20.  }
21. }
```

这种在克隆请求的同时设置新请求头的操作太常见了，因此它还有一个快捷方式 `setHeaders`：

```
// Clone the request and set the new header in one step.
const authReq = req.clone({ setHeaders: { Authorization: authToken } });
```

这种可以修改头的拦截器可以用于很多不同的操作，比如：

- 认证 / 授权
- 控制缓存行为。比如 `If-Modified-Since`
- XSRF 防护

记日志

因为拦截器可以**同时**处理请求和响应，所以它们也可以对整个 HTTP 操作进行计时和记录日志。

考虑下面这个 `LoggingInterceptor`，它捕获请求的发起时间、响应的接收时间，并使用注入的 `MessageService` 来发送总共花费的时间。

app/http-interceptors/logging-interceptor.ts)

```
import { finalize, tap } from 'rxjs/operators';
import { MessageService } from '../message.service';

@Injectable()
export class LoggingInterceptor implements HttpInterceptor {
  constructor(private messenger: MessageService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const started = Date.now();
    let ok: string;

    // extend server response observable with logging
    return next.handle(req)
      .pipe(
        tap(
          // Succeeds when there is a response; ignore other events
          event => ok = event instanceof HttpResponse ? 'succeeded' : '',
          // Operation failed; error is an HttpErrorResponse
          error => ok = 'failed'
        ),
        // Log when response observable either completes or errors
        finalize(() => {
          const elapsed = Date.now() - started;
          const msg = `${req.method} "${req.urlWithParams}"
             ${ok} in ${elapsed} ms.`;
          this.messenger.add(msg);
        })
      );
  }
}
```

RxJS 的 `tap` 操作符会捕获请求成功了还是失败了。 RxJS 的 `finalize` 操作符无论在响应成功还是失败时都会调用（这是必须的），然后把结果汇报给 `MessageService`。

在这个可观察对象的流中，无论是 `tap` 还是 `finalize` 接触过的值，都会照常发送给调用者。

缓存

拦截器还可以自行处理这些请求，而不用转发给 `next.handle()`。

比如，你可能会想缓存某些请求和响应，以便提升性能。 你可以把这种缓存操作委托给某个拦截器，而不破坏你现有的各个数据服务。

`CachingInterceptor` 演示了这种方式。

app/http-interceptors/caching-interceptor.ts)

```
@Injectable()
export class CachingInterceptor implements HttpInterceptor {
  constructor(private cache: RequestCache) {}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    // continue if not cachable.
    if (!isCachable(req)) { return next.handle(req); }

    const cachedResponse = this.cache.get(req);
    return cachedResponse ?
      of(cachedResponse) : sendRequest(req, next, this.cache);
  }
}
```

`isCachable()` 函数用于决定该请求是否允许缓存。 在这个例子中，只有发到 npm 包搜索 API 的 GET 请求才是可以缓存的。

如果该请求是不可缓存的，该拦截器只会把该请求转发给链表中的下一个处理器。

如果可缓存的请求在缓存中找到了，该拦截器就会通过 `of()` 函数返回一个已缓存的响应体的**可观察对象**，然后把它传给 `next` 处理器（以及所有其它下游拦截器）。

如果可缓存的请求在缓存中没找到，代码就会调用 `sendRequest`。

```
/**
 * Get server response observable by sending request to `next()`.
 * Will add the response to the cache on the way out.
 */
function sendRequest(
  req: HttpRequest<any>,
  next: HttpHandler,
  cache: RequestCache): Observable<HttpEvent<any>> {

  // No headers allowed in npm search request
  const noHeaderReq = req.clone({ headers: new HttpHeaders() });

  return next.handle(noHeaderReq).pipe(
    tap(event => {
      // There may be other events besides the response.
      if (event instanceof HttpResponse) {
        cache.put(req, event); // Update the cache.
      }
    })
  );
}
```

`sendRequest` 函数创建了一个不带请求头的请求克隆体，因为 npm API 不会接受它们。

它会把这个请求转发给 `next.handle()`，它最终会调用服务器，并且返回服务器的响应。

注意 `sendRequest` 是如何在发回给应用之前**拦截这个响应的**。 它会通过 `tap()` 操作符对响应进行管道处理，并在其回调中把响应加到缓存中。

然后，原始的响应会通过这些拦截器链，原封不动的回到服务器的调用者那里。

数据服务，比如 `PackageSearchService`，并不知道它们收到的某些 `HttpClient` 请求实际上是从缓存的请求中返回来的。

返回多值可观察对象

`HttpClient.get()` 方法正常情况下只会返回一个**可观察对象**，它或者发出数据，或者发出错误。 有些人说它是“一次性完成”的可观察对象。

但是拦截器也可以把这个修改成发出多个值的**可观察对象**。

修改后的 `CachingInterceptor` 版本可以返回一个立即发出缓存的响应，然后仍然把请求发送到 NPM 的 Web API，然后再把修改过的搜索结果重新发出一次。

```
// cache-then-refresh
if (req.headers.get('x-refresh')) {
  const results$ = sendRequest(req, next, this.cache);
  return cachedResponse ?
    results$.pipe( startWith(cachedResponse) ) :
    results$;
}
// cache-or-fetch
return cachedResponse ?
  of(cachedResponse) : sendRequest(req, next, this.cache);
```

这种**缓存并刷新**的选项是由自定义的 `x-refresh` 头触发的。

> `PackageSearchComponent` 中的一个检查框会切换 `withRefresh` 标识， 它是 `PackageSearchService.search()` 的参数之一。 `search()` 方法创建了自定义的 `x-refresh` 头，并在调用 `HttpClient.get()` 前把它添加到请求里。

修改后的 `CachingInterceptor` 会发起一个服务器请求，而不管有没有缓存的值。 就像 前面 的 `sendRequest()` 方法一样进行订阅。 在订阅 `results$` 可观察对象时，就会发起这个请求。

如果没有缓存的值，拦截器直接返回 `result$`。

如果有缓存的值，这些代码就会把缓存的响应加入到 `result$` 的管道中，使用重组后的可观察对象进行处理，并发出两次。 先立即发出一次缓存的响应体，然后发出来自服务器的响应。 订阅者将会看到一个包含这**两个**响应的序列。

监听进度事件

有时，应用会传输大量数据，并且这些传输可能会花费很长时间。 典型的例子是文件上传。 可以通过在传输过程中提供进度反馈，来提升用户体验。

要想开启进度事件的响应，你可以创建一个把 `reportProgress` 选项设置为 `true` 的 `HttpRequest` 实例，以开启进度跟踪事件。

app/uploader/uploader.service.ts (upload request)

```
const req = new HttpRequest('POST', '/upload/file', file, {
  reportProgress: true
});
```

> 每个进度事件都会触发变更检测，所以，你应该只有当确实希望在 UI 中报告进度时才打开这个选项。

接下来，把这个请求对象传给 `HttpClient.request()` 方法，它返回一个 `HttpEvents` 的 `Observable`，同样也可以在拦截器中处理这些事件。

app/uploader/uploader.service.ts (upload body)

```
// The `HttpClient.request` API produces a raw event stream
// which includes start (sent), progress, and response events.
return this.http.request(req).pipe(
  map(event => this.getEventMessage(event, file)),
  tap(message => this.showProgress(message)),
  last(), // return last (completed) message to caller
  catchError(this.handleError(file))
);
```

`getEventMessage` 方法会解释事件流中的每一个 `HttpEvent` 类型。

app/uploader/uploader.service.ts (getEventMessage)

```
/** Return distinct message for sent, upload progress, & response events */
private getEventMessage(event: HttpEvent<any>, file: File) {
  switch (event.type) {
    case HttpEventType.Sent:
      return `Uploading file "${file.name}" of size ${file.size}.`;

    case HttpEventType.UploadProgress:
      // Compute and show the % done:
      const percentDone = Math.round(100 * event.loaded / event.total);
      return `File "${file.name}" is ${percentDone}% uploaded.`;

    case HttpEventType.Response:
      return `File "${file.name}" was completely uploaded!`;

    default:
      return `File "${file.name}" surprising upload event: ${event.type}.`;
  }
}
```

> 这个范例应用中并没有一个用来接收上传的文件的真实的服务器。 `app/http-interceptors/upload-interceptor.ts` 中的 `UploadInterceptor` 会拦截并短路掉上传请求，改为返回一个带有各个模拟事件的可观察对象。

安全：XSRF 防护

跨站请求伪造 (XSRF)是一个攻击技术，它能让攻击者假冒一个已认证的用户在你的网站上执行未知的操作。 `HttpClient` 支持一种通用的机制来防范 XSRF 攻击。当执行 HTTP 请求时，一个拦截器会从 cookie 中读取 XSRF 令牌（默认名字为 `XSRF-TOKEN`），并且把它设置为一个 HTTP 头 `X-XSRF-TOKEN`，由于只有运行在你自己的域名下的代码才能读取这个 cookie，因此后端可以确认这个 HTTP 请求真的来自你的客户端应用，而不是攻击者。

默认情况下，拦截器会在所有的修改型请求中（比如 POST 等）把这个 cookie 发送给使用相对 URL 的请求。但不会在 GET/HEAD 请求中发送，也不会发送给使用绝对 URL 的请求。

要获得这种优点，你的服务器需要在页面加载或首个 GET 请求中把一个名叫 `XSRF-TOKEN` 的令牌写入可被 JavaScript 读到的会话 cookie 中。 而在后续的请求中，服务器可以验证这个 cookie 是否与 HTTP 头 `X-XSRF-TOKEN` 的值一致，以确保只有运行在你自己域名下的代码才能发起这个请求。这个令牌必须对每个用户都是唯一的，并且必须能被服务器验证，因此不能由客户端自己生成令牌。把这个令牌设置为你的站点认证信息并且加了盐（salt）的摘要，以提升安全性。

为了防止多个 Angular 应用共享同一个域名或子域时出现冲突，要给每个应用分配一个唯一的 cookie 名称。

> **注意，`HttpClient` 支持的只是 XSRF 防护方案的客户端这一半。** 你的后端服务必须配置为给页面设置 cookie ，并且要验证请求头，以确保全都是合法的请求。否则，Angular 默认的这种防护措施就会失效。

配置自定义 cookie/header 名称

如果你的后端服务中对 XSRF 令牌的 cookie 或 头使用了不一样的名字，就要使用 `HttpClientXsrfModule.withConfig()` 来覆盖掉默认值。

```
imports: [
  HttpClientModule,
  HttpClientXsrfModule.withOptions({
    cookieName: 'My-Xsrf-Cookie',
    headerName: 'My-Xsrf-Header',
  }),
],
```

测试 HTTP 请求

如同所有的外部依赖一样，HTTP 后端也需要在良好的测试实践中被 Mock 掉。`@angular/common/http` 提供了一个测试库 `@angular/common/http/testing`，它让你可以直截了当的进行这种 Mock 。

Mock 方法论

Angular 的 HTTP 测试库是专为其中的测试模式而设计的。在这种模式下，会首先在应用中执行代码并发起请求。

然后，每个测试会期待发起或未发起过某个请求，对这些请求进行断言， 最终对每个所预期的请求进行刷新（flush）来对这些请求提供响应。

最终，测试可能会验证这个应用不曾发起过非预期的请求。

> 你可以到在线编程环境中运行在线例子 / 下载范例。
>
> 本章所讲的这些测试位于 `src/testing/http-client.spec.ts` 中。 在 `src/app/heroes/heroes.service.spec.ts` 中还有一些测试，用于测试那些调用了 `HttpClient` 的数据服务。

环境设置

要开始测试那些通过 `HttpClient` 发起的请求，就要导入 `HttpClientTestingModule` 模块，并把它加到你的 `TestBed` 设置里去，代码如下：

app/testing/http-client.spec.ts (imports)

```
// Http testing module and mocking controller
import { HttpClientTestingModule, HttpTestingController } from
'@angular/common/http/testing';

// Other imports
import { TestBed } from '@angular/core/testing';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
```

然后把 `HTTPClientTestingModule` 添加到 `TestBed` 中，并继续设置**被测服务**。

app/testing/http-client.spec.ts(setup)

```
describe('HttpClient testing', () => {
  let httpClient: HttpClient;
  let httpTestingController: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [ HttpClientTestingModule ]
    });

    // Inject the http service and test controller for each test
    httpClient = TestBed.get(HttpClient);
    httpTestingController = TestBed.get(HttpTestingController);
  });
  /// Tests begin ///
});
```

现在，在测试中发起的这些请求将会被这些测试后端（testing backend）处理，而不是标准的后端。

这种设置还会调用 `TestBed.get()`，来获取注入的 `HttpClient` 服务和模拟对象的控制器 `HttpTestingController`，以便在测试期间引用它们。

期待并回复请求

现在，你就可以编写测试，等待 GET 请求并给出模拟响应。

app/testing/http-client.spec.ts(httpClient.get)

```
it('can test HttpClient.get', () => {
  const testData: Data = {name: 'Test Data'};

  // Make an HTTP GET request
  httpClient.get<Data>(testUrl)
    .subscribe(data =>
      // When observable resolves, result should match test data
      expect(data).toEqual(testData)
    );

  // The following `expectOne()` will match the request's URL.
  // If no requests or multiple requests matched that URL
  // `expectOne()` would throw.
  const req = httpTestingController.expectOne('/data');

  // Assert that the request is a GET.
  expect(req.request.method).toEqual('GET');

  // Respond with mock data, causing Observable to resolve.
  // Subscribe callback asserts that correct data was returned.
  req.flush(testData);

  // Finally, assert that there are no outstanding requests.
  httpTestingController.verify();
});
```

最后一步，验证没有发起过预期之外的请求，足够通用，因此你可以把它移到 `afterEach()` 中：

```
afterEach(() => {
  // After every test, assert that there are no more pending requests.
  httpTestingController.verify();
});
```

自定义对请求的预期

如果仅根据 URL 匹配还不够，你还可以自行实现匹配函数。 比如，你可以验证外发的请求是否带有某个认证头：

```
// Expect one request with an authorization header
const req = httpTestingController.expectOne(
  req => req.headers.has('Authorization')
);
```

和前面根据 URL 进行测试时一样，如果零或两个以上的请求匹配上了这个期待，它就会抛出异常。

处理一个以上的请求

如果你需要在测试中对重复的请求进行响应，可以使用 `match()` API 来代替 `expectOne()`，它的参数不变，但会返回一个与这些请求相匹配的数组。一旦返回，这些请求就会从将来要匹配的列表中移除，你要自己验证和刷新（flush）它。

```
// get all pending requests that match the given URL
const requests = httpTestingController.match(testUrl);
expect(requests.length).toEqual(3);

// Respond to each request with different results
requests[0].flush([]);
requests[1].flush([testData[0]]);
requests[2].flush(testData);
```

测试对错误的预期

你还要测试应用对于 HTTP 请求失败时的防护。

调用 `request.flush()` 并传入一个错误信息，如下所示：

```
it('can test for 404 error', () => {
  const emsg = 'deliberate 404 error';

  httpClient.get<Data[]>(testUrl).subscribe(
    data => fail('should have failed with the 404 error'),
    (error: HttpErrorResponse) => {
      expect(error.status).toEqual(404, 'status');
      expect(error.error).toEqual(emsg, 'message');
    }
  );

  const req = httpTestingController.expectOne(testUrl);

  // Respond with mock error
  req.flush(emsg, { status: 404, statusText: 'Not Found' });
});
```

另外，你还可以使用 `ErrorEvent` 来调用 `request.error()`.

```
it('can test for network error', () => {
  const emsg = 'simulated network error';

  httpClient.get<Data[]>(testUrl).subscribe(
    data => fail('should have failed with the network error'),
    (error: HttpErrorResponse) => {
      expect(error.error.message).toEqual(emsg, 'message');
    }
  );

  const req = httpTestingController.expectOne(testUrl);

  // Create mock ErrorEvent, raised when something goes wrong at the network level.
  // Connection timeout, DNS error, offline, etc
  const mockError = new ErrorEvent('Network error', {
    message: emsg,
  });

  // Respond with mock error
  req.error(mockError);
});
```

路由与导航

在用户使用应用程序时，Angular 的**路由器**能让用户从一个视图导航到另一个视图。

本章涵盖了该路由器的主要特性，通过一个小型应用的成长演进来讲解它。参见在线例子 / 下载范例。

概览

浏览器具有熟悉的导航模式：

- 在地址栏输入 URL，浏览器就会导航到相应的页面。
- 在页面中点击链接，浏览器就会导航到一个新页面。
- 点击浏览器的前进和后退按钮，浏览器就会在你的浏览历史中向前或向后导航。

Angular 的 `Router`（即“路由器”）借鉴了这个模型。它把浏览器中的 URL 看做一个操作指南， 据此导航到一个由客户端生成的视图，并可以把参数传给支撑视图的相应组件，帮它决定具体该展现哪些内容。 你可以为页面中的链接绑定一个路由，这样，当用户点击链接时，就会导航到应用中相应的视图。 当用户点击按钮、从下拉框中选取，或响应来自任何地方的事件时，你也可以在代码控制下进行导航。 路由器还在浏览器的历史日志中记录下这些活动，这样浏览器的前进和后退按钮也能照常工作。

基础知识

本章包括一系列里程碑，从一个单模块、两个页面的简单程序逐步走向带有多个子路由的多视图设计。

先对路由的一些核心概念做一个介绍，它能帮你逐步过渡到细节。

<base href> 元素

大多数带路由的应用都要在`index.html`的 `<head>` 标签下先添加一个 `<base>` 元素，来告诉路由器该如何合成导航用的 URL。

如果 `app` 文件夹是该应用的根目录（就像范例应用中一样），那就把 `href` 的值设置为下面这样：

src/index.html (base-href)

```
<base href="/">
```

从路由库中导入

Angular 的路由器是一个可选的服务，它用来呈现指定的 URL 所对应的视图。 它并不是 Angular 核心库的一部分，而是在它自己的 `@angular/router` 包中。 像其它 Angular 包一样，你可以从它导入所需的一切。

src/app/app.module.ts (import)

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

> 你将会在后面学到更多选项。

配置

每个带路由的 Angular 应用都有一个`Router` **（路由器）**服务的单例对象。 当浏览器的 URL 变化时，路由器会查找对应的 `Route`（路由），并据此决定该显示哪个组件。

路由器需要先配置才会有路由信息。 下面的例子创建了四个路由定义，并用 `RouterModule.forRoot` 方法来配置路由器， 并把它的返回值添加到 `AppModule` 的 `imports` 数组中。

src/app/app.module.ts (excerpt)

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
    // other imports here
  ],
  ...
})
export class AppModule { }
```

这里的路由数组 `appRoutes` 描述如何进行导航。 把它传给 `RouterModule.forRoot` 方法并传给本模块的 `imports` 数组就可以配置路由器。

每个 `Route` 都会把一个 URL 的 `path` 映射到一个组件。 注意，`path` 不能以**斜杠（`/`）**开头。 路由器会为解析和构建最终的 URL，这样当你在应用的多个视图之间导航时，可以任意使用相对路径和绝对路径。

第二个路由中的 `:id` 是一个路由参数的令牌(Token)。比如 `/hero/42` 这个 URL 中，“42”就是 `id` 参数的值。 此 URL 对应的 `HeroDetailComponent` 组件将据此查找和展现 `id` 为 42 的英雄。 在本章中稍后的部分，你将会学习关于路由参数的更多知识。

第三个路由中的 `data` 属性用来存放于每个具体路由有关的任意信息。该数据可以被任何一个激活路由访问，并能用来保存诸如 页标题、面包屑以及其它静态只读数据。本章稍后的部分，你将使用resolve 守卫来获取动态数据。

第四个路由中的空路径（`''`）表示应用的默认路径，当 URL 为空时就会访问那里，因此它通常会作为起点。这个默认路由会重定向到 URL `/heroes`，并显示 `HeroesListComponent`。

最后一个路由中的 `**` 路径是一个通配符。当所请求的 URL 不匹配前面定义的路由表中的任何路径时，路由器就会选择此路由。 这个特性可用于显示“404 - Not Found”页，或自动重定向到其它路由。

这些路由的定义顺序是刻意如此设计的。路由器使用先匹配者优先的策略来匹配路由，所以，具体路由应该放在通用路由的前面。在上面的配置中，带静态路径的路由被放在了前面，后面是空路径路由，因此它会作为默认路由。而通配符路由被放在最后面，这是因为它能匹配上**每一个 URL**，因此应该只有在前面找不到其它能匹配的路由时才匹配它。

如果你想要看到在导航的生命周期中发生过哪些事件，可以使用路由器默认配置中的 enableTracing 选项。它会把每个导航生命周期中的事件输出到浏览器的控制台。 这应该只用于**调试**。你只需要把 `enableTracing: true` 选项作为第二个参数传给 `RouterModule.forRoot()` 方法就可以了。

路由出口

有了这份配置，当本应用在浏览器中的 URL 变为 `/heroes` 时，路由器就会匹配到 `path` 为 `heroes` 的 `Route`，并在宿主视图中的`RouterOutlet`之后显示 `HeroListComponent` 组件。

```
<router-outlet></router-outlet>
<!-- Routed views go here -->
```

路由器链接

现在，你已经有了配置好的一些路由，还找到了渲染它们的地方，但又该如何导航到它呢？固然，从浏览器的地址栏直接输入 URL 也能做到，但是大多数情况下，导航是某些用户操作的结果，比如点击一个 A 标签。

考虑下列模板：

src/app/app.component.ts (template)

```
template: `
  <h1>Angular Router</h1>
  <nav>
    <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
    <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
  </nav>
  <router-outlet></router-outlet>
`
```

`a` 标签上的 `RouterLink` 指令让路由器得以控制这个 `a` 元素。 这里的导航路径是固定的，因此可以把一个字符串赋给 `routerLink`（“一次性”绑定）。

如果需要更加动态的导航路径，那就把它绑定到一个返回**链接参数数组**的模板表达式。 路由器会把这个数组解析成完整的 URL。

每个 `a` 标签上的`RouterLinkActive`指令可以帮用户在外观上区分出当前选中的“活动”路由。 当与它关联的 **RouterLink** 被激活时，路由器会把 CSS 类 `active` 添加到这个元素上。 你可以把该指令添加到 `a` 元素或它的父元素上。

路由器状态

在导航时的每个生命周期成功完成时，路由器会构建出一个 `ActivatedRoute` 组成的树，它表示路由器的当前状态。 你可以在应用中的任何地方用 `Router` 服务及其 `routerState` 属性来访问当前的 `RouterState` 值。

`RouterState` 中的每个 `ActivatedRoute` 都提供了从任意激活路由开始向上或向下遍历路由树的一种方式，以获得关于父、子、兄弟路由的信息。

激活的路由

该路由的路径和参数可以通过注入进来的一个名叫ActivatedRoute的路由服务来获取。 它有一大堆有用的信息，包括：

属性	说明
`url`	路由路径的 `Observable` 对象，是一个由路由路径中的各个部分组成的字符串数组。
`data`	一个 `Observable`，其中包含提供给路由的 `data` 对象。也包含由解析守卫（resolve guard）解析而来的值。
`paramMap`	一个 `Observable`，其中包含一个由当前路由的必要参数和可选参数组成的map对象。用这个 map 可以获取来自同名参数的单一值或多重值。
`queryParamMap`	一个 `Observable`，其中包含一个对所有路由都有效的查询参数组成的map对象。 用这个 map 可以获取来自查询参数的单一值或多重值。
`fragment`	一个适用于所有路由的 URL 的 fragment（片段）的 `Observable`。
`outlet`	要把该路由渲染到的 `RouterOutlet` 的名字。对于无名路由，它的路由名是 `primary`，而不是空串。
`routeConfig`	用于该路由的路由配置信息，其中包含原始路径。
`parent`	当该路由是一个子路由时，表示该路由的父级 `ActivatedRoute`。
`firstChild`	包含该路由的子路由列表中的第一个 `ActivatedRoute`。
`children`	包含当前路由下所有已激活的子路由。

> 有两个旧式属性仍然是有效的，但它们不如其替代品那样强力，建议不再用它们，它们还将在未来的 Angular 版本中废弃。
>
> `params` —— 一个 `Observable` 对象，其中包含当前路由的必要参数和可选参数。请改用 `paramMap`。
>
> `queryParams` —— 一个 `Observable` 对象，其中包含对所有路由都有效的查询参数。请改用 `queryParamMap`。

路由事件

在每次导航中，`Router` 都会通过 `Router.events` 属性发布一些导航事件。这些事件的范围涵盖了从开始导航到结束导航之间的很多时间点。下表中列出了全部导航事件：

路由器事件	说明
`NavigationStart`	本事件会在导航开始时触发。
`RoutesRecognized`	本事件会在路由器解析完 URL，并识别出了相应的路由时触发
`RouteConfigLoadStart`	本事件会在 `Router` 对一个路由配置进行惰性加载之前触发。
`RouteConfigLoadEnd`	本事件会在路由被惰性加载之后触发。
`NavigationEnd`	本事件会在导航成功结束之后触发。
`NavigationCancel`	本事件会在导航被取消之后触发。 这可能是因为在导航期间某个路由守卫返回了 `false`。
`NavigationError`	这个事件会在导航由于意料之外的错误而失败时触发。

当打开了 `enableTracing` 选项时，这些事件也同时会记录到控制台中。由于这些事件是以 `Observable` 的形式提供的，所以你可以对自己感兴趣的事件进行 `filter()`，并 `subscribe()` 它们，以便根据导航过程中的事件顺序做出决策。

总结一下

该应用有一个配置过的路由器。 外壳组件中有一个 `RouterOutlet`，它能显示路由器所生成的视图。 它还有一些 `RouterLink`，用户可以点击它们，来通过路由器进行导航。

下面是一些**路由器**中的关键词汇及其含义：

路由器部件	含义
`Router`（路由器）	为激活的 URL 显示应用组件。管理从一个组件到另一个组件的导航
`RouterModule`	一个独立的 Angular 模块，用于提供所需的服务提供商，以及用来在应用视图之间进行导航的指令。
`Routes`（路由数组）	定义了一个路由数组，每一个都会把一个 URL 路径映射到一个组件。
`Route`（路由）	定义路由器该如何根据 URL 模式（pattern）来导航到组件。大多数路由都由路径和组件类构成。
`RouterOutlet`（路由出口）	该指令（`<router-outlet>`）用来标记出路由器该在哪里显示视图。
`RouterLink`（路由链接）	这个指令把可点击的 HTML 元素绑定到某个路由。点击带有 `routerLink` 指令（绑定到*字符串*或*链接参数数组*）的元素时就会触发一次导航。
`RouterLinkActive`（活动路由链接）	当 HTML 元素上或元素内的`routerLink`变为激活或非激活状态时，该指令为这个 HTML 元素添加或移除 CSS 类。
`ActivatedRoute`（激活的路由）	为每个路由组件提供提供的一个服务，它包含特定于路由的信息，比如路由参数、静态数据、解析数据、全局查询参数和全局碎片（fragment）。
`RouterState`（路由器状态）	路由器的当前状态包含了一棵由程序中激活的路由构成的树。它包含一些用于遍历路由树的快捷方法。
链接参数数组	这个数组会被路由器解释成一个路由操作指南。你可以把一个`RouterLink`绑定到该数组，或者把它作为参数传给`Router.navigate`方法。
路由组件	一个带有`RouterOutlet`的 Angular 组件，它根据路由器的导航来显示相应的视图。

范例应用

本章要讲的是如何开发一个带路由的多页面应用。 接下来会重点讲它的设计决策，并描述路由的关键特性，比如：

- 把应用的各个特性组织成模块。
- 导航到组件（**Heroes** 链接到“英雄列表”组件）。
- 包含一个路由参数（当路由到“英雄详情”时，把该英雄的 `id` 传进去）。
- 子路由（**危机中心**特性有一组自己的路由）。
- `CanActivate` 守卫（检查路由的访问权限）。
- `CanActivateChild` 守卫（检查子路由的访问权限）。
- `CanDeactivate` 守卫（询问是否丢弃未保存的更改）。
- `Resolve` 守卫（预先获取路由数据）。
- 惰性加载特性模块。
- `CanLoad` 守卫（在加载特性模块之前进行检查）。

如果打算一步步构建出本应用，本章就会经过一系列里程碑。 但是，本章并不是一个教程，它隐藏了构造 Angular 应用的细节，那些细节会在本文档的其它地方展开。

本应用的最终版源码可以在在线例子 / 下载范例中查看和下载。

范例程序的动图

假设本程序会用来帮助“英雄管理局”运行他们的业务。 英雄们需要找工作，而“英雄管理局”为他们寻找待解决的危机。

本应用具有三个主要的特性区：

1. **危机中心**用于维护要指派给英雄的危机列表。
2. **英雄**区用于维护管理局雇佣的英雄列表。
3. **管理**区会管理危机和英雄的列表。

点击英雄职介中心的在线例子 / 下载范例试用一下。

等应用热身完毕，你就会看到一排导航按钮，以及一个**英雄列表**视图。

选择其中之一，该应用就会把你带到此英雄的编辑页面。

Crisis Center　Heroes　Admin　Login

HEROES

"Magneta"

Id: 15
Name: Magneta

Back

修改完名字，再点击“后退”按钮，应用又回到了英雄列表页，其中显示的英雄名已经变了。注意，对名字的修改会立即生效。

另外你也可以点击浏览器本身的后退按钮，这样也同样会回到英雄列表页。 在 Angular 应用中导航也会和标准的 Web 导航一样更新浏览器中的历史。

现在，点击**危机中心**链接，前往**危机**列表页。

Crisis Center　Heroes　Admin　Login

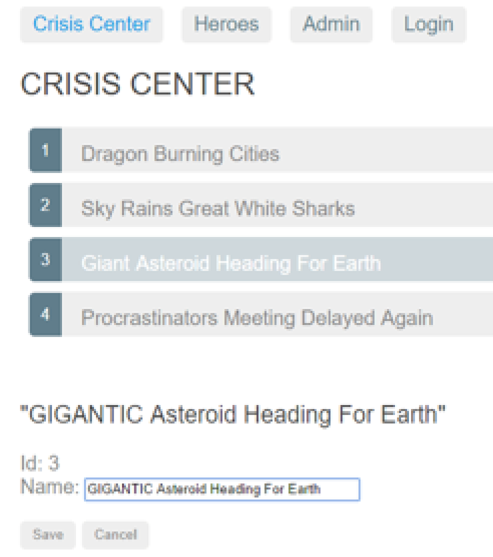
CRISIS CENTER

1 Dragon Burning Cities
2 Sky Rains Great White Sharks
3 Giant Asteroid Heading For Earth
4 Procrastinators Meeting Delayed Again

Welcome to the Crisis Center

选择其中之一，该应用就会把你带到此危机的编辑页面。 **危机详情**出现在了当前页的子视图区，也就是在列表的紧下方。

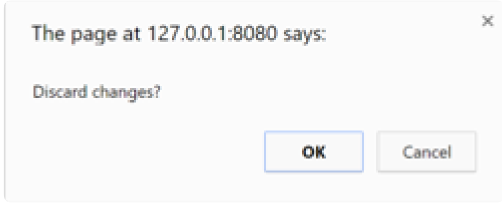
修改危机的名称。 注意，危机列表中的相应名称并没有修改。



这和**英雄详情**页略有不同。**英雄详情**会立即保存你所做的更改。 而**危机详情**页中，你的更改都是临时的 —— 除非按“保存”按钮保存它们，或者按“取消”按钮放弃它们。 这两个按钮都会导航回**危机中心**，显示危机列表。

先不要点击这些按钮。 而是点击浏览器的后退按钮，或者点击“Heroes”链接。

这时会弹出一个对话框。



你可以回答“确定”以放弃这些更改，或者回答“取消”来继续编辑。

这种行为的幕后是路由器的 `CanDeactivate` 守卫。 该守卫让你有机会进行清理工作或在离开当前视图之前请求用户的许可。

`Admin` 和 `Login` 按钮用于演示路由器的其它能力，本章稍后的部分会讲解它们。这里只是个简短的讲解。

这就开始本应用的第一个里程碑。

里程碑 1：从路由器开始

开始本应用的一个简版，它在两个空路由之间导航。

Component Router

Crisis Center Heroes

设置<base href>

路由器使用浏览器的history.pushState进行导航。 感谢 `pushState`！ 有了它，你就能按所期望的样子来显示应用内部的 URL 路径，比如：`localhost:3000/crisis-center`。虽然你使用的全部是客户端合成的视图，但应用内部的这些 URL 看起来和来自服务器的没有什么不同。

现代 HTML 5 浏览器是最早支持 `pushState` 的，这也就是很多人喜欢把这种 URL 称作“HTML 5 风格的”URL 的原因。

> HTML 5 风格的导航是路由器的默认值。请到下面的附录浏览器 URL 风格中学习为什么首选“HTML 5”风格、如何调整它的行为，以及如何在必要时切换回老式的 hash（#）风格。

你必须往本应用的 `index.html` 中添加一个<base href> 元素，这样 `pushState` 才能正常工作。 当引用 CSS 文件、脚本和图片时，浏览器会用 `<base href>` 的值作为**相对**URL 的前缀。

把 `<base>` 元素添加到 `<head>` 元素中。 如果 `app` 目录是应用的根目录，对于本应用，可以像这样设置 `index.html`中的 `href` 值：

src/index.html (base-href)

```
<base href="/">
```

在线例子说明

像 Stackblitz 这样的在线编程环境会动态设置应用的基地址（base href），因此你没办法指定固定的地址。 这就是为什么范例代码中要用一个脚本动态写入 `<base>` 标签，而不是直接写 `<base href...>`。

```
<script>document.write('<base href="' + document.location + '" />');</script>
```

你只应该在在线例子这种情况下使用这种小花招，不要把它用到产品的正式代码中。

从路由库中导入

先从路由库导入一些符号。 路由器在它自己的 `@angular/router` 包中。 它不是 Angular 内核的一部分。该路由器是可选的服务，这是因为并不是所有应用都需要路由，并且，如果需要，你还可能需要另外的路由库。

通过一些路由来配置路由器，你可以教路由器如何进行导航。

定义路由

路由器必须用"路由定义"的列表进行配置。

第一个配置中定义了由两个路由构成的数组，它们分别通过路径(path)导航到了 `CrisisListComponent` 和 `HeroListComponent` 组件。

每个定义都被翻译成了一个Route对象。该对象有一个 `path` 字段，表示该路由中的 URL 路径部分，和一个 `component` 字段，表示与该路由相关联的组件。

当浏览器的 URL 变化时或在代码中告诉路由器导航到一个路径时，路由器就会翻出它用来保存这些路由定义的注册表。

直白的说，你可以这样解释第一个路由：

- 当浏览器地址栏的 URL 变化时，如果它匹配上了路径部分 `/crisis-center`，路由器就会激活一个 `CrisisListComponent` 的实例，并显示它的视图。
- 当应用程序请求导航到路径 `/crisis-center` 时，路由器激活一个 `CrisisListComponent` 的实例，显示它的视图，并将该路径更新到浏览器地址栏和历史。

下面是第一个配置。把路由数组传递到 `RouterModule.forRoot` 方法，该方法返回一个包含已配置的 `Router` 服务提供商模块和一些其它路由包需要的服务提供商。应用启动时，`Router` 将在当前浏览器 URL 的基础上进行初始导航。

src/app/app.module.ts (first-config)

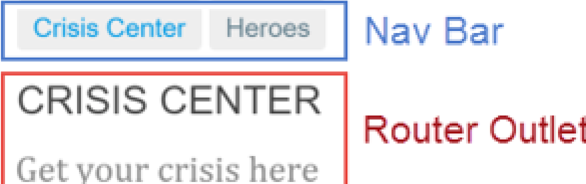
```
import { NgModule }              from '@angular/core';
import { BrowserModule }         from '@angular/platform-browser';
import { FormsModule }           from '@angular/forms';
import { RouterModule, Routes } from '@angular/router';

import { AppComponent }          from './app.component';
import { CrisisListComponent }   from './crisis-list.component';
import { HeroListComponent }     from './hero-list.component';

const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes', component: HeroListComponent },
];

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    CrisisListComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

> 作为简单的路由配置，将添加配置好的 `RouterModule` 到 `AppModule` 中就足够了。 随着应用的成长，你将需要将路由配置重构到单独的文件，并创建路由模块 - 一种特别的、专门为特性模块的路由器服务的服务模块。

在 `AppModule` 中提供 `RouterModule`，让该路由器在应用的任何地方都能被使用。

AppComponent 外壳组件

根组件 `AppComponent` 是本应用的壳。它在顶部有一个标题、一个带两个链接的导航条，在底部有一个**路由器出口**，路由器会在它所指定的位置上把视图切入或调出页面。就像下图中所标出的：



该组件所对应的模板是这样的：

src/app/app.component.ts (template)

```
template: `
  <h1>Angular Router</h1>
  <nav>
    <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
    <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
  </nav>
  <router-outlet></router-outlet>
`
```

路由出口

`RouterOutlet` 是一个来自路由库的组件。 路由器会在 `<router-outlet>` 标签中显示视图。

> 路由器会把 `<router-outlet>` 元素添加到了 DOM 中，紧接着立即在这个 `<router-outlet>` **之后**插入导航到的视图元素。

`routerLink` 绑定

在出口上方的 A 标签中，有一个绑定 `RouterLink` 指令的属性绑定，就像这样：`routerLink="..."`。

例子中的每个链接都有一个字符串型的路径，也就是你以前配置过的路由路径，但还没有指定路由参数。

你还可以通过提供查询字符串参数为 `RouterLink` 提供更多情境信息，或提供一个 URL 片段（Fragment 或 hash）来跳转到本页面中的其它区域。 查询字符串可以由 `[queryParams]` 绑定来提供，它需要一个对象型参

数（如 `{ name: 'value' }`），而 URL 片段需要一个绑定到 `[fragment]` 的单一值。

> 还可以到后面的附录中学习如何使用链接参数数组。

`routerLinkActive` 绑定

每个 A 标签还有一个到 `RouterLinkActive` 指令的属性绑定，就像 `routerLinkActive="..."`。

等号（=）右侧的模板表达式包含用空格分隔的一些 CSS 类。当路由激活时路由器就会把它们添加到此链接上（反之则移除）。你还可以把 `RouterLinkActive` 指令绑定到一个 CSS 类组成的数组，如 `[routerLinkActive]="['...']"`。

`RouterLinkActive` 指令会基于当前的 `RouterState` 对象来为激活的 `RouterLink` 切换 CSS 类。 这会一直沿着路由树往下进行级联处理，所以父路由链接和子路由链接可能会同时激活。 要改变这种行为，可以把 `[routerLinkActiveOptions]` 绑定到 `{exact: true}` 表达式。 如果使用了 `{ exact: true }`，那么只有在其 URL 与当前 URL 精确匹配时才会激活指定的 `RouterLink`。

路由器指令集

`RouterLink`、`RouterLinkActive` 和 `RouterOutlet` 是由 `RouterModule` 包提供的指令。 现在你可以把它用在模板中了。

`app.component.ts` 目前是这样的：

src/app/app.component.ts (excerpt)

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Angular Router</h1>
    <nav>
      <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
      <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
    </nav>
    <router-outlet></router-outlet>
  `
})
export class AppComponent { }
```

通配符路由

你以前在应用中创建过两个路由，一个是 `/crisis-center`，另一个是 `/heroes`。 所有其它 URL 都会导致路由器抛出错误，并让应用崩溃。

可以添加一个通配符路由来拦截所有无效的 URL，并优雅的处理它们。 **通配符**路由的 `path` 是两个星号（`**`），它会匹配**任何** URL。 当路由器匹配不上以前定义的那些路由时，它就会选择**这个**路由。 通配符路由可以导航到自定义的“404 Not Found”组件，也可以重定向到一个现有路由。

> 路由器使用先匹配者优先的策略来选择路由。 通配符路由是路由配置中最没有特定性的那个，因此务必确保它是配置中的**最后一个**路由。

要测试本特性，请往 `HeroListComponent` 的模板中添加一个带 `RouterLink` 的按钮，并且把它的链接设置为 `"/sidekicks"`。

src/app/hero-list.component.ts (excerpt)

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

@Component({
  template: `
    <h2>HEROES</h2>
    <p>Get your heroes here</p>

    <button routerLink="/sidekicks">Go to sidekicks</button>
  `
})
export class HeroListComponent { }
```

当用户点击该按钮时，应用就会失败，因为你尚未定义过 `"/sidekicks"` 路由。

不要添加 `"/sidekicks"` 路由，而是定义一个“通配符”路由，让它直接导航到 `PageNotFoundComponent` 组件。

src/app/app.module.ts (wildcard)

```
{ path: '**', component: PageNotFoundComponent }
```

创建 `PageNotFoundComponent`，以便在用户访问无效网址时显示它。

src/app/not-found.component.ts (404 component)

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

@Component({
  template: '<h2>Page not found</h2>'
})
export class PageNotFoundComponent {}
```

像其它组件一样，把 `PageNotFoundComponent` 添加到 `AppModule` 的声明中。

现在，当用户访问 `/sidekicks` 或任何无效的 URL 时，浏览器就会显示“Page not found”。 浏览器的地址栏仍指向无效的 URL。

把**默认**路由设置为英雄列表

应用启动时，浏览器地址栏中的初始 URL 是这样的：

```
localhost:3000
```

它不能匹配上任何具体的路由，于是就会走到通配符路由中去，并且显示 `PageNotFoundComponent`。

这个应用需要一个有效的默认路由，在这里应该用英雄列表作为默认页。当用户点击"Heroes"链接或把 `localhost:3000/heroes` 粘贴到地址栏时，它应该导航到列表页。

重定向路由

首选方案是添加一个 `redirect` 路由来把最初的相对路径（`''`）转换成期望的默认路径（`/heroes`）。 浏览器地址栏会显示 `.../heroes`，就像你直接导航到那里一样。

在通配符路由**上方**添加一个默认路由。 在下方的代码片段中，它出现在通配符路由的紧上方，展示了这个里程碑的完整 `appRoutes`。

src/app/app-routing.module.ts (appRoutes)

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes',        component: HeroListComponent },
  { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];
```

重定向路由需要一个 `pathMatch` 属性，来告诉路由器如何用 URL 去匹配路由的路径，否则路由器就会报错。在本应用中，路由器应该只有在**完整的 URL**等于 `''` 时才选择 `HeroListComponent` 组件，因此要把 `pathMatch` 设置为 `'full'`。

> 从技术角度说，`pathMatch = 'full'` 导致 URL 中**剩下的**、未匹配的部分必须等于 `''`。 在这个例子中，跳转路由在一个顶级路由中，因此**剩下的**URL 和**完整的**URL 是一样的。
>
> `pathMatch` 的另一个可能的值是 `'prefix'`，它会告诉路由器：当**剩下的**URL 以这个跳转路由中的 `prefix` 值开头时，就会匹配上这个跳转路由。
>
> 在这里不能这么做！如果 `pathMatch` 的值是 `'prefix'`，那么**每个**URL 都会匹配上 `''`。
>
> 尝试把它设置为 `'prefix'`，然后点击 `Go to sidekicks` 按钮。别忘了，它是一个无效 URL，本应显示“Page not found”页。 但是，你仍然在“英雄列表”页中。在地址栏中输入一个无效的 URL，你又被路由到了 `/heroes`。 **每一个** URL，无论有效与否，都会匹配上这个路由定义。
>
> 默认路由应该只有在**整个**URL 等于 `''` 时才重定向到 `HeroListComponent`，别忘了把重定向路由设置为 `pathMatch = 'full'`。
>
> 要了解更多，参见 Victor Savkin 的帖子关于重定向。

“起步阶段”总结

你得到了一个非常基本的、带导航的应用，当用户点击链接时，它能在两个视图之间切换。

你学到了如何：

- 加载路由库
- 往壳组件的模板中添加一个导航条，导航条中有一些 A 标签、`routerLink` 指令和 `routerLinkActive` 指令
- 往壳组件的模板中添加一个 `router-outlet` 指令，视图将会被显示在那里
- 用 `RouterModule.forRoot` 配置路由器模块
- 设置路由器，使其合成 HTML5 模式的浏览器 URL。
- 使用通配符路由来处理无效路由
- 当应用在空路径下启动时，导航到默认路由

这个初学者应用的其它部分有点平淡无奇，从路由器的角度来看也很平淡。 如果你还是倾向于在这个里程碑里构建它们，参见下面的构建详情。

这个初学者应用的结构是这样的：

```
router-sample
```

```
    src
        app
            app.component.ts
            app.module.ts
            crisis-list.component.ts
            hero-list.component.ts
            not-found.component.ts
        main.ts
        index.html
        styles.css
        tsconfig.json
    node_modules ...
    package.json
```

下面是当前里程碑中讨论过的文件列表：

app.component.ts | *app.module.ts* | *main.ts* | *hero-list.component.ts*

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Angular Router</h1>
    <nav>
      <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
      <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
    </nav>
    <router-outlet></router-outlet>
  `
```

```
13. })
14. export class AppComponent { }
```

里程碑 #2：路由模块

在原始的路由配置中，你提供了仅有两个路由的简单配置来设置应用的路由。对于简单的路由，这没有问题。 随着应用的成长，你用到了更多路由器特性，比如守卫、解析器和子路由等，你会很自然地想要重构路由。 我们建议将路由信息移到一个单独的特殊用途的模块，叫做路由模块。

路由模块有一系列特性：

- 把路由这个关注点从其它应用类关注点中分离出去。
- 测试特性模块时，可以替换或移除路由模块。
- 为路由服务提供商（包括守卫和解析器等）提供一个共同的地方。
- 不要声明组件。

将路由配置重构为**路由模块**

在 `/app` 目录下创建一个名叫 `app-routing.module.ts` 的文件，以包含这个路由模块。

导入 `CrisisListComponent` 和 `HeroListComponent` 组件，就像 `app.module.ts` 中一样。然后把 `Router` 的导入语句和路由配置以及 `RouterModule.forRoot` 移入这个路由模块中。

遵循规约，添加一个 `AppRoutingModule` 类并导出它，以便稍后在 `AppModule` 中导入它。

最后，可以通过把它添加到该模块的 `exports` 数组中来再次导出 `RouterModule`。 通过在 `AppModule` 中导入 `AppRoutingModule` 并再次导出 `RouterModule`，那些声明在 `AppModule` 中的组件就可以访问路由指令了，比如 `RouterLink` 和 `RouterOutlet`。

做完这些之后，该文件变成了这样：

src/app/app-routing.module.ts

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

import { CrisisListComponent }   from './crisis-list.component';
import { HeroListComponent }     from './hero-list.component';
import { PageNotFoundComponent } from './not-found.component';

const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes',        component: HeroListComponent },
  { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

接下来，修改 `app.module.ts` 文件，首先从 `app-routing.module.ts` 中导入新创建的 `AppRoutingModule`，然后把 `imports` 数组中的 `RouterModule.forRoot` 替换为 `AppRoutingModule`。

src/app/app.module.ts

```
import { NgModule }       from '@angular/core';
import { BrowserModule }  from '@angular/platform-browser';
import { FormsModule }    from '@angular/forms';

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

import { CrisisListComponent }   from './crisis-list.component';
import { HeroListComponent }     from './hero-list.component';
import { PageNotFoundComponent } from './not-found.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    CrisisListComponent,
    PageNotFoundComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

> 本章稍后的部分，你将创建一个多路由模块，并揭示你为何必须以正确的顺序导入那些路由模块。

应用继续正常运行，你可以把路由模块作为为每个特性模块维护路由配置的中心地方。

你需要路由模块吗?

路由模块在根模块或者特性模块替换了路由配置。在路由模块或者在模块内部配置路由，但不要同时在两处都配置。

路由模块是设计选择，它的价值在配置很复杂，并包含专门守卫和解析器服务时尤其明显。 在配置很简单时，它可能看起来很多余。

在配置很简单时，一些开发者跳过路由模块（例如 `AppRoutingModule`），并将路由配置直接混合在关联模块中（比如 `AppModule` ）。

从中选择一种模式，并坚持模式的一致性。

大多数开发者都应该采用路由模块，以保持一致性。 它在配置复杂时，能确保代码干净。 它让测试特性模块更加容易。 它的存在让人一眼就能看出这个模块是带路由的。 开发者可以很自然的从路由模块中查找和扩展路由配置。

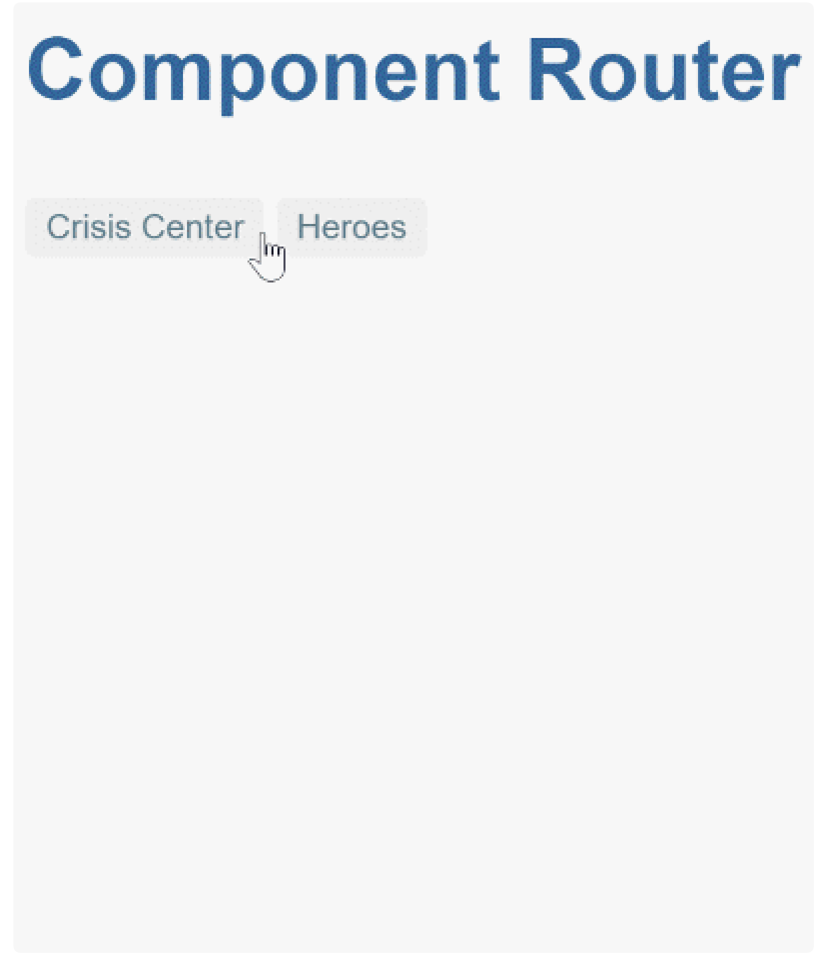
里程碑 #2 英雄特征区

你刚刚学习了如何用 `RouterLink` 指令进行导航。接下来要：

- 用模块把应用和路由组织为一些**特性区**
- 命令式的从一个组件导航到另一个
- 通过路由传递必要信息和可选信息

这个例子重写了《英雄指南》的“服务”部分的英雄列表特性，你可以从英雄指南：各个服务的范例代码 / 下载范例中复制大部分代码过来。

下面是用户将看到的版本：



典型的应用具有多个**特性区**，每个特性区都专注于特定的业务用途。

虽然你也可以把文件都放在 `src/app/` 目录下，但那样是不现实的，而且很难维护。 大部分开发人员更喜欢把每个特性区都放在它自己的目录下。

你准备把应用拆分成多个不同的**特性模块**，每个特有模块都有自己的关注点。 然后，你就会把它们导入到主模块中，并且在它们之间导航。

添加英雄管理功能

按照下列步骤：

- 创建 `src/app/heroes` 文件夹，你将会把**英雄管理**功能的实现文件放在这里。
- 在 `app` 目录下删除占位用的 `hero-list.component.ts` 文件。
- 在 `src/app/heroes` 目录下创建新的 `hero-list.component.ts` 文件。
- 把《英雄指南》：各个服务的范例代码 / 下载范例复制到 `app.component.ts` 中。
- 做一些微小但必要的修改：
    - 删除 `selector`（路由组件不需要它们）。
    - 删除 `<h1>`。
    - 给 `<h2>` 加文字，改成 `<h2>HEROES</h2>`。
    - 删除模板底部的 `<hero-detail>`。
    - 把 `AppComponent` 类改名为 `HeroListComponent`。
- 把 `hero-detail.component.ts` 和 `hero.service.ts` 复制到 `heroes` 子目录下。
- 在 `heroes` 子目录下（不带路由）的 `heroes.module.ts` 文件，内容如下：

src/app/heroes/heroes.module.ts (pre-routing)

```
import { NgModule }       from '@angular/core';
import { CommonModule }   from '@angular/common';
import { FormsModule }    from '@angular/forms';

import { HeroListComponent }    from './hero-list.component';
import { HeroDetailComponent }  from './hero-detail.component';

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

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
  ],
  declarations: [
    HeroListComponent,
    HeroDetailComponent
  ],
  providers: [ HeroService ]
})
export class HeroesModule {}
```

安排完这些，你就有了四个**英雄管理**特性区的文件：

```
src/app/heroes
  hero-detail.component.ts
  hero-list.component.ts
  hero.service.ts
  heroes.module.ts
```

英雄特性区的路由需求

“英雄”特性有两个相互协作的组件，列表和详情。 列表视图是自给自足的，你导航到它，它会自行获取英雄列表并显示他们。

详情视图就不同了。它要显示一个特定的英雄，但是它本身却无法知道显示哪一个，此信息必须来自外部。

当用户从列表中选择了一个英雄时，应用就导航到详情页以显示那个英雄。 通过把所选英雄的 id 编码进路由的 URL 中，就能告诉详情视图该显示哪个英雄。

英雄特性区的路由配置

在 `heroes` 目录下创建一个新的 `heroes-routing.module.ts` 文件，使用的技术和以前创建 `AppRoutingModule` 时的一样。

src/app/heroes/heroes-routing.module.ts

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

import { HeroListComponent }    from './hero-list.component';
import { HeroDetailComponent }  from './hero-detail.component';

const heroesRoutes: Routes = [
  { path: 'heroes',  component: HeroListComponent },
  { path: 'hero/:id', component: HeroDetailComponent }
];

@NgModule({
  imports: [
    RouterModule.forChild(heroesRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class HeroRoutingModule { }
```

> 把路由模块文件和它对应的模块文件放在同一个目录下。 比如这里的 `heroes-routing.module.ts` 和 `heroes.module.ts` 都位于 `src/app/heroes` 目录下。
>
> 将路由模块文件放到它相关的模块文件所在目录里。 这里，`heroes-routing.module.ts` 和 `heroes.module.ts` 都在 `app/heroes` 目录中。

从新位置 `src/app/heroes/` 目录中导入英雄相关的组件，定义两个“英雄管理”路由，并导出 `HeroRoutingModule` 类。

现在，你有了 `Heroes` 模块的路由，还得在 `RouterModule` 中把它们注册给**路由器**，和 `AppRoutingModule` 中的做法几乎完全一样。

这里有少量但是关键的不同点。 在 `AppRoutingModule` 中，你使用了静态的 `RouterModule.forRoot`方法来注册路由和全应用级服务提供商。 在特性模块中，你要改用`forChild`静态方法。

> 只在根模块 `AppRoutingModule` 中调用 `RouterModule.forRoot` （如果在 `AppModule` 中注册应用的顶级路由，那就在 `AppModule` 中调用）。 在其它模块中，你就必须调用`RouterModule.forChild`方法来注册附属路由。

把路由模块添加到 `HeroesModule` 中

把 `HeroRoutingModule` 添加到 `HeroModule` 中，就像为 `AppModule` 添加 `AppRoutingModule` 一样。

打开 `heroes.module.ts`，从 `heroes-routing.module.ts` 中导入 `HeroRoutingModule` 并把它添加到 `HeroesModule` 的 `imports` 数组中。 写完后的 `HeroesModule` 是这样的：

src/app/heroes/heroes.module.ts

```
import { NgModule }       from '@angular/core';
import { CommonModule }   from '@angular/common';
import { FormsModule }    from '@angular/forms';

import { HeroListComponent }    from './hero-list.component';
import { HeroDetailComponent }  from './hero-detail.component';

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

import { HeroRoutingModule } from './heroes-routing.module';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    HeroRoutingModule
  ],
  declarations: [
    HeroListComponent,
    HeroDetailComponent
  ],
  providers: [ HeroService ]
})
export class HeroesModule {}
```

移除重复的“英雄管理”路由

英雄类的路由目前定义在两个地方：`HeroesRoutingModule` 中（并最终给 `HeroesModule`）和 `AppRoutingModule` 中。

由特性模块提供的路由会被路由器再组合上它们所导入的模块的路由。 这让你可以继续定义特性路由模块中的路由，而不用修改主路由配置。

但你显然不希望把同一个路由定义两次，那就移除 `HeroListComponent` 的导入和来自 `app-routing.module.ts` 中的 `/heroes` 路由。

保留默认路由和通配符路由！ 它们是应用程序顶层该自己处理的关注点。

src/app/app-routing.module.ts (v2)

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

import { CrisisListComponent }   from './crisis-list.component';
// import { HeroListComponent }  from './hero-list.component';  // <-- delete this
line
import { PageNotFoundComponent } from './not-found.component';

const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  // { path: 'heroes',     component: HeroListComponent }, // <-- delete this line
  { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

把“英雄管理”模块导入到 AppModule

英雄这个特性模块已经就绪，但应用仍然不知道 `HeroesModule` 的存在。 打开 `app.module.ts`，并按照下述步骤修改它。

导入 `HeroesModule` 并且把它加到根模块 `AppModule` 的 `@NgModule` 元数据中的 `imports` 数组中。

从 `AppModule` 的 `declarations` 中移除 `HeroListComponent`，因为它现在已经改由 `HeroesModule` 提供了。 这一步很重要！因为一个组件只能声明在**一个**属主模块中。 这个例子中，`Heroes` 模块就是 `Heroes` 组件的属主模块，而 `AppModule` 要通过导入 `HeroesModule` 才能使用这些组件。

最终，`AppModule` 不再了解那些特定于“英雄”特性的知识，比如它的组件、路由细节等。 你可以让“英雄”特性独立演化，添加更多的组件或各种各样的路由。 这就是为每个特性区创建独立模块后获得的核心优势。

经过这些步骤，`AppModule` 变成了这样：

src/app/app.module.ts

```
import { NgModule }       from '@angular/core';
import { BrowserModule }  from '@angular/platform-browser';
import { FormsModule }    from '@angular/forms';

import { AppComponent }     from './app.component';
import { AppRoutingModule } from './app-routing.module';
import { HeroesModule }     from './heroes/heroes.module';

import { CrisisListComponent }   from './crisis-list.component';
import { PageNotFoundComponent } from './not-found.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HeroesModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    CrisisListComponent,
    PageNotFoundComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

导入模块的顺序很重要

看看该模块的 `imports` 数组。注意，`AppRoutingModule` 是**最后一个**。最重要的是，它位于 `HeroesModule` 之后。

src/app/app.module.ts (module-imports)

```
imports: [
  BrowserModule,
  FormsModule,
  HeroesModule,
  AppRoutingModule
],
```

路由配置的顺序很重要。 路由器会接受第一个匹配上导航所要求的路径的那个路由。

当所有路由都在同一个 `AppRoutingModule` 时，你要把默认路由和通配符路由放在最后（这里是在 `/heroes` 路由后面）， 这样路由器才有机会匹配到 `/heroes` 路由，否则它就会先遇到并匹配上该通配符路由，并导航到“页面未找到”路由。

这些路由不再位于单一文件中。他们分布在两个不同的模块中：`AppRoutingModule` 和 `HeroesRoutingModule`。

每个路由模块都会根据**导入的顺序**把自己的路由配置追加进去。 如果你先列出了 `AppRoutingModule`，那么通配符路由就会被注册在“英雄管理”路由**之前**。 通配符路由（它匹配**任意**URL）将会拦截住每一个到“英雄管理”路由的导航，因此事实上屏蔽了所有“英雄管理”路由。

> 反转路由模块的导入顺序，你就会看到当点击英雄相关的链接时被导向了“页面未找到”路由。 要学习如何在运行时查看路由器配置，参见稍后的内容。

带参数的路由定义

回到 `HeroesRoutingModule` 并再次检查这些路由定义。 `HeroDetailComponent` 的路由有点特殊。

src/app/heroes/heroes-routing.module.ts (excerpt)

```
{ path: 'hero/:id', component: HeroDetailComponent }
```

注意路径中的 `:id` 令牌。它为**路由参数**在路径中创建一个“空位”。在这里，路由器把英雄的 `id` 插入到那个“空位”中。

如果要告诉路由器导航到详情组件，并让它显示“Magneta”，你会期望这个英雄的 `id` 像这样显示在浏览器的 URL 中：

```
localhost:3000/hero/15
```

如果用户把此 URL 输入到浏览器的地址栏中，路由器就会识别出这种模式，同样进入“Magneta”的详情视图。

> 路由参数：必须的还是可选的?
>
> 在这个场景下，把路由参数的令牌 `:id` 嵌入到路由定义的 `path` 中是一个好主意，因为对于 `HeroDetailComponent` 来说 `id` 是**必须的**， 而且路径中的值 `15` 已经足够把到“Magneta”的路由和到其它英雄的路由明确区分开。

在列表视图中设置路由参数

然后导航到 `HeroDetailComponent` 组件。在那里，你期望看到所选英雄的详情，这需要两部分信息：导航目标和该英雄的 `id`。

因此，这个**链接参数数组**中有两个条目：目标路由的`path`（路径），和一个用来指定所选英雄 `id` 的路由参数。

src/app/heroes/hero-list.component.ts (link-parameters-array)

```
['/hero', hero.id] // { 15 }
```

路由器从该数组中组合出了目标 URL： `localhost:3000/hero/15`。

> 目标组件 `HeroDetailComponent` 该怎么知道这个 `id` 参数呢？ 当然不会是自己去分析 URL 了！那是路由器的工作。
>
> 路由器从 URL 中解析出路由参数（`id:15`），并通过 ActivatedRoute 服务来把它提供给 `HeroDetailComponent` 组件。

Activated Route 实战

从路由器（`router`）包中导入 `Router`、`ActivatedRoute` 和 `Params` 类。

src/app/heroes/hero-detail.component.ts (activated route)

```
import { Router, ActivatedRoute, ParamMap } from '@angular/router';
```

这里导入 `switchMap` 操作符是因为你稍后将会处理路由参数的可观察对象 `Observable`。

src/app/heroes/hero-detail.component.ts (switchMap operator import)

```
import { switchMap } from 'rxjs/operators';
```

通常，你会直接写一个构造函数，让 Angular 把组件所需的服务注入进来，自动定义同名的私有变量，并把它们存进去。

src/app/heroes/hero-detail.component.ts (constructor)

```
constructor(
  private route: ActivatedRoute,
  private router: Router,
  private service: HeroService
) {}
```

然后，在 `ngOnInit` 方法中，你用 `ActivatedRoute` 服务来接收路由的参数，从参数中取得该英雄的 `id`，并接收此英雄用于显示。

src/app/heroes/hero-detail.component.ts (ngOnInit)

```
ngOnInit() {
  this.hero$ = this.route.paramMap.pipe(
    switchMap((params: ParamMap) =>
      this.service.getHero(params.get('id')))
  );
}
```

`paramMap` 的处理过程有点稍复杂。当这个 map 的值变化时，你可以从变化之后的参数中 `get()` 到其 `id` 参数。

然后，让 `HeroService` 去获取一个具有此 `id` 的英雄，并返回这个 `HeroService` 请求的结果。

你可能想使用 RxJS 的 `map` 操作符。 但 `HeroService` 返回的是一个 `Observable<Hero>`。 所以你要改用 `switchMap` 操作符来打平这个 `Observable`。

`switchMap` 操作符还会取消以前未完成的在途请求。如果用户使用新的 `id` 再次导航到该路由，而 `HeroService` 仍在接受老 `id` 对应的英雄，那么 `switchMap` 就会抛弃老的请求，并返回这个新 `id` 的英雄信息。

这个可观察对象的 `Subscription`（订阅）将会由 `AsyncPipe` 处理，并且组件的 `hero` 属性将会设置为刚刚接收到的这个英雄。

ParamMap API

`ParamMap` API 是参照URLSearchParams 接口来设计的。它提供了一些方法来处理对路由参数（`paramMap`）和查询参数(`queryParamMap`)中的参数访问。

成员	说明
`has(name)`	如果参数名位于参数列表中，就返回 `true` 。
`get(name)`	如果这个 map 中有参数名对应的参数值（字符串），就返回它，否则返回 `null`。如果参数值实际上是一个数组，就返回它的**第一个**元素。
`getAll(name)`	如果这个 map 中有参数名对应的值，就返回一个字符串数组，否则返回空数组。当一个参数名可能对应多个值的时候，请使用 `getAll`。
`keys`	返回这个 map 中的所有参数名组成的字符串数组。

参数的可观察对象（Observable）与组件复用

在这个例子中，你接收了路由参数的 `Observable` 对象。 这种写法暗示着这些路由参数在该组件的生存期内可能会变化。

确实如此！默认情况下，如果它没有访问过其它组件就导航到了同一个组件实例，那么路由器倾向于复用组件实例。如果复用，这些参数可以变化。

假设父组件的导航栏有“前进”和“后退”按钮，用来轮流显示英雄列表中中英雄的详情。 每次点击都会强制导航到带前一个或后一个 `id` 的 `HeroDetailComponent` 组件。

你不希望路由器仅仅从 DOM 中移除当前的 `HeroDetailComponent` 实例，并且用下一个 `id` 重新创建它。 那可能导致界面抖动。 更好的方式是复用同一个组件实例，并更新这些参数。

不幸的是，`ngOnInit` 对每个实例只调用一次。 你需要一种方式来检测**在同一个实例中**路由参数什么时候发生了变化。 而 `params` 属性这个可观察对象（Observable）干净漂亮的处理了这种情况。

> 当在组件中订阅一个可观察对象时，你通常总是要在组件销毁时取消这个订阅。
>
> 但是也有少数例外情况不需要取消订阅。 `ActivateRoute` 中的各种可观察对象就是属于这种情况。
>
> `ActivateRoute` 及其可观察对象都是由 `Router` 本身负责管理的。 `Router` 会在不再需要时销毁这个路由组件，而注入进去的 `ActivateRoute` 也随之销毁了。
>
> 不过，你仍然可以随意取消订阅，这不会造成任何损害，而且也不是一项坏的实践。

Snapshot（快照）：当不需要 Observable 时的替代品

本应用不需要复用 `HeroDetailComponent`。 用户总是会先返回英雄列表，再选择另一位英雄。 所以，不存在从一个英雄详情导航到另一个而不用经过英雄列表的情况。 这意味着路由器每次都会创建一个全新的 `HeroDetailComponent` 实例。

假如你很确定这个 `HeroDetailComponent` 组件的实例**永远、永远**不会被复用，那就可以使用**快照**来简化这段代码。

`route.snapshot` 提供了路由参数的初始值。 你可以通过它来直接访问参数，而不用订阅或者添加 Observable 的操作符。 这样在读写时就会更简单：

src/app/heroes/hero-detail.component.ts (ngOnInit snapshot)

```
ngOnInit() {
  let id = this.route.snapshot.paramMap.get('id');

  this.hero$ = this.service.getHero(id);
}
```

> 记住：，用这种技巧，你只得到了这些参数的**初始**值。 如果有可能连续多次导航到此组件，那么就该用 `paramMap` 可观察对象的方式。 这个例子中仍然使用了 `paramMap` 的可观察对象策略。

导航回列表组件

`HeroDetailComponent` 组件有一个“Back”按钮，关联到它的 `gotoHeroes` 方法，该方法会导航回 `HeroListComponent` 组件。

路由的 `navigate` 方法同样接受一个单条目的**链接参数数组**，你也可以把它绑定到 `[routerLink]` 指令上。 它保存着到 `HeroListComponent` 组件的路径：

src/app/heroes/hero-detail.component.ts (excerpt)

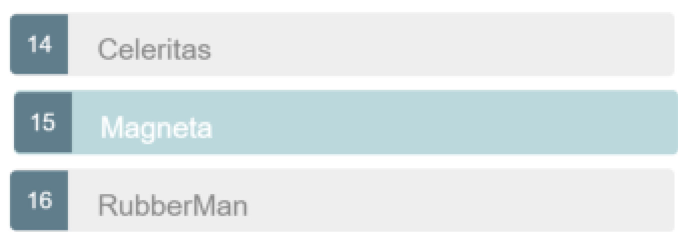
```
gotoHeroes() {
  this.router.navigate(['/heroes']);
}
```

路由参数：必须还是可选？

如果想导航到 `HeroDetailComponent` 以对 id 为 15 的英雄进行查看并编辑，就要在路由的 URL 中使用**路由参数**来指定**必要**参数值。

```
localhost:3000/hero/15
```

你也能在路由请求中添加**可选**信息。 比如，当从 `HeroDetailComponent` 返回英雄列表时，如果能自动选中刚刚查看过的英雄就好了。



当从 `HeroDetailComponent` 返回时，你很快就会通过把正在查看的英雄的 `id` 作为可选参数包含在 URL 中来实现这个特性。

可选信息有很多种形式。搜索条件通常就不是严格结构化的，比如 `name='wind*'`；有多个值也很常见，如 `after='12/31/2015'&before='1/1/2017'`； 而且顺序无关，如 `before='1/1/2017'&after='12/31/2015'`，还可能有很多种变体格式，如 `during='currentYear'`。

这么多种参数要放在 URL 的**路径**中可不容易。即使你能制定出一个合适的 URL 方案，实现起来也太复杂了，得通过模式匹配才能把 URL 翻译成命名路由。

可选参数是在导航期间传送任意复杂信息的理想载体。 可选参数不涉及到模式匹配并在表达上提供了巨大的灵活性。

和必要参数一样，路由器也支持通过可选参数导航。 在你定义完必要参数之后，再通过一个**独立的对象**来定义**可选参数**。

通常，对于强制性的值（比如用于区分两个路由路径的）使用**必备参数**；当这个值是可选的、复杂的或多值的时，使用可选参数。

英雄列表：选定一个英雄（也可不选）

当导航到 `HeroDetailComponent` 时，你可以在**路由参数**中指定一个所要编辑的英雄 `id`，只要把它作为链接参数数组中的第二个条目就可以了。

src/app/heroes/hero-list.component.ts (link-parameters-array)

```
['/hero', hero.id] // { 15 }
```

路由器在导航 URL 中内嵌了 `id` 的值，这是因为你把它用一个 `:id` 占位符当做路由参数定义在了路由的 `path` 中：

src/app/heroes/heroes-routing.module.ts (hero-detail-route)

```
{ path: 'hero/:id', component: HeroDetailComponent }
```

当用户点击后退按钮时，`HeroDetailComponent` 构造了另一个**链接参数数组**，可以用它导航回 `HeroListComponent`。

src/app/heroes/hero-detail.component.ts (gotoHeroes)

```
gotoHeroes() {
  this.router.navigate(['/heroes']);
}
```

该数组缺少一个路由参数，这是因为你那时没有理由往 `HeroListComponent` 发送信息。

但现在有了。你要在导航请求中同时发送当前英雄的 id，以便 `HeroListComponent` 可以在列表中高亮这个英雄。 这是一个**有更好，没有也无所谓**的特性，就算没有它，列表照样能显示得很完美。

传送一个包含**可选**`id` 参数的对象。 为了演示，这里还在对象中定义了一个没用的额外参数（`foo`），`HeroListComponent` 应该忽略它。 下面是修改过的导航语句：

src/app/heroes/hero-detail.component.ts (go to heroes)

```
gotoHeroes(hero: Hero) {
  let heroId = hero ? hero.id : null;
  // Pass along the hero id if available
  // so that the HeroList component can select that hero.
  // Include a junk 'foo' property for fun.
  this.router.navigate(['/heroes', { id: heroId, foo: 'foo' }]);
}
```

该应用仍然能工作。点击“back”按钮返回英雄列表视图。

注意浏览器的地址栏。

它应该是这样的，不过也取决于你在哪里运行它：

```
localhost:3000/heroes;id=15;foo=foo
```

`id` 的值像这样出现在 URL 中（`;id=15;foo=foo`），但不在 URL 的路径部分。 “Heroes”路由的路径部分并没有定义 `:id`。

可选的路由参数没有使用“？”和“&”符号分隔，因为它们将用在 URL 查询字符串中。 它们是用“;”分隔的。 这是**矩阵 URL**标记法 —— 你以前可能从未见过。

> **Matrix URL**写法首次提出是在1996 提案中，提出者是 Web 的奠基人：Tim Berners-Lee。
>
> 虽然 Matrix 写法未曾进入过 HTML 标准，但它是合法的。而且在浏览器的路由系统中，它作为从父路由和子路由中单独隔离出参数的方式而广受欢迎。Angular 的路由器正是这样一个路由系统，并支持跨浏览器的 Matrix 写法。
>
> 这种语法对你来说可能有点奇怪，不过用户不会在意这一点，因为该 URL 可以正常的通过邮件发出去或粘贴到浏览器的地址栏中。

ActivatedRoute 服务中的路由参数

英雄列表仍没有改变，没有哪个英雄列被加亮显示。

> 在线例子 / 下载范例**高亮了**选中的行，因为它演示的是应用的最终状态，因此包含了你**即将**接触到的步骤。 此刻，本文描述的仍是那些步骤**之前**的状态。

`HeroListComponent` 还完全不需要任何参数，也不知道该怎么处理它们。你可以改变这一点。

以前，当从 `HeroListComponent` 导航到 `HeroDetailComponent` 时，你通过 `ActivatedRoute` 服务订阅了路由参数这个 `Observable`，并让它能用在 `HeroDetailComponent` 中。 你把该服务注入到了 `HeroDetailComponent` 的构造函数中。

这次，你要进行反向导航，从 `HeroDetailComponent` 到 `HeroListComponent`。

首先，你扩展该路由的导入语句，以包含进 `ActivatedRoute` 服务的类；

src/app/heroes/hero-list.component.ts (import)

```
import { ActivatedRoute, ParamMap } from '@angular/router';
```

导入 `switchMap` 操作符，在路由参数的 `Observable` 对象上执行操作。

src/app/heroes/hero-list.component.ts (rxjs imports)

```
import { Observable } from 'rxjs';
import { switchMap } from 'rxjs/operators';
```

接着，你注入 `ActivatedRoute` 到 `HeroListComponent` 的构造函数中。

src/app/heroes/hero-list.component.ts (constructor and ngOnInit)

```
export class HeroListComponent implements OnInit {
  heroes$: Observable<Hero[]>;

  private selectedId: number;

  constructor(
    private service: HeroService,
    private route: ActivatedRoute
  ) {}

  ngOnInit() {
    this.heroes$ = this.route.paramMap.pipe(
      switchMap((params: ParamMap) => {
        // (+) before `params.get()` turns the string into a number
        this.selectedId = +params.get('id');
        return this.service.getHeroes();
      })
    );
  }
}
```

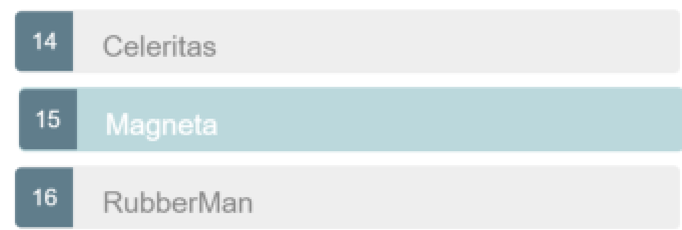
ActivatedRoute.paramMap 属性是一个路由参数的可观察对象。当用户导航到这个组件时，paramMap 会发射一个新值，其中包含 `id`。 在 ngOnInit 中，你订阅了这些值，设置到 selectedId，并获取英雄数据。

用 CSS 类绑定更新模板，把它绑定到 `isSelected` 方法上。 如果该方法返回 `true`，此绑定就会添加 CSS 类 `selected`，否则就移除它。 在 `<li>` 标记中找到它，就像这样：

src/app/heroes/hero-list.component.ts (template)

```
template: `
  <h2>HEROES</h2>
  <ul class="items">
    <li *ngFor="let hero of heroes$ | async"
      [class.selected]="hero.id === selectedId">
      <a [routerLink]="['/hero', hero.id]">
        <span class="badge">{{ hero.id }}</span>{{ hero.name }}
      </a>
    </li>
  </ul>

  <button routerLink="/sidekicks">Go to sidekicks</button>
`
```

当用户从英雄列表导航到英雄“Magneta”并返回时，“Magneta”看起来是选中的：



这儿可选的 `foo` 路由参数人畜无害，并继续被忽略。

为路由组件添加动画

这个“英雄”特性模块就要完成了，但这个特性还没有平滑的转场效果。

在这一节，你将为**英雄详情**组件添加一些动画。

首先导入 `BrowserAnimationsModule`：

src/app/app.module.ts (animations-module)

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({
  imports: [
    BrowserAnimationsModule
```

在根目录 `src/app/` 下创建一个 `animations.ts`。内容如下：

src/app/animations.ts (excerpt)

```
import { animate, state, style, transition, trigger } from '@angular/animations';

// Component transition animations
export const slideInDownAnimation =
  trigger('routeAnimation', [
    state('*',
      style({
        opacity: 1,
        transform: 'translateX(0)'
      })
    ),
    transition(':enter', [
      style({
        opacity: 0,
        transform: 'translateX(-100%)'
      }),
      animate('0.2s ease-in')
    ]),
    transition(':leave', [
      animate('0.5s ease-out', style({
        opacity: 0,
        transform: 'translateY(100%)'
      }))
    ])
  ]);
```

该文件做了如下工作：

- 导入动画符号以构建动画触发器、控制状态并管理状态之间的过渡。
- 导出了一个名叫 `slideInDownAnimation` 的常量，并把它设置为一个名叫*`routeAnimation` 的动画触发器。带动画的组件将会引用这个名字。
- 指定了一个**通配符状态** —— `*`，它匹配该路由组件存在时的任何动画状态。
- 定义两个**过渡效果**，其中一个（`:enter`）在组件进入应用视图时让它从屏幕左侧缓动进入（ease-in），另一个（`:leave`）在组件离开应用视图时让它向下飞出。

你可以为其它路由组件用不同的转场效果创建更多触发器。现在这个触发器已经足够当前的里程碑用了。

返回 `HeroDetailComponent`，从 `'./animations.ts` 中导入 `slideInDownAnimation`。 从 `@angular/core` 中导入 `HostBinding` 装饰器，你很快就会用到它。

把一个包含 `slideInDownAnimation` 的 `animations` 数组添加到 `@Component` 的元数据中。

然后把三个 `@HostBinding` 属性添加到类中以设置这个路由组件元素的动画和样式。

src/app/heroes/hero-detail.component.ts (host bindings)

```
@HostBinding('@routeAnimation') routeAnimation = true;
@HostBinding('style.display')   display = 'block';
@HostBinding('style.position')  position = 'absolute';
```

传给了第一个 `@HostBinding` 的 `'@routeAnimation'` 匹配了 `slideInDownAnimation`**触发器**的名字 `routeAnimation`。 把 `routeAnimation` 属性设置为 `true`，因为你只关心 `:enter` 和 `:leave` 这两个状态。

另外两个 `@HostBinding` 属性指定组件的外观和位置。

当进入该路由时，`HeroDetailComponent` 将会从左侧缓动进入屏幕，而离开路由时，将会向下划出。

> 在这样简单的演示程序中，可以把路由动画应用到独立的组件中，但是在真实的应用中，最好能基于**路由路径**应用路由动画。

里程碑#3 的总结

你学到了如何：

- 把应用组织成**特性区**
- 命令式的从一个组件导航到另一个
- 通过路由参数传递信息，并在组件中订阅它们
- 把这个特性分区模块导入根模块 `AppModule`
- 把动画应用到路由组件上

做完这些修改之后，目录结构是这样的：

```
router-sample
  src
    app
      heroes
        hero-detail.component.ts
        hero-list.component.ts
```

```
            hero.service.ts
            heroes.module.ts
            heroes-routing.module.ts
        app.component.ts
        app.module.ts
        app-routing.module.ts
        crisis-list.component.ts
    main.ts
    index.html
    styles.css
    tsconfig.json
  node_modules ...
  package.json
```

这里是当前版本的范例程序相关文件。

app.component.ts *app.module.ts* *app-routing.module.ts* *hero-list.c*

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-root',
5.   template: `
6.     <h1>Angular Router</h1>
7.     <nav>
8.       <a routerLink="/crisis-center" routerLinkActive="active">Crisis
   Center</a>
9.       <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
10.     </nav>
11.     <router-outlet></router-outlet>
12.   `
13. })
```

```
14. export class AppComponent { }
```

里程碑#4：危机中心

是时候往该应用的危机中心（现在是占位符）中添加一些真实的特性了。

先从模仿“英雄管理”中的特性开始：

- 删除危机中心的占位文件。
- 创建 `app/crisis-center` 文件夹。
- 把 `app/heroes` 中的文件复制到新的危机中心文件夹。
- 在这些新文件中，把每一个对“hero”替换为“crisis”，并把“heroes”替换为“crises”。

你将会把 `CrisisService` 转换成模拟的危机列表，而不再是模拟的英雄列表：

src/app/crisis-center/crisis.service.ts (mock-crises)

```
import { BehaviorSubject } from 'rxjs';
import { map } from 'rxjs/operators';

export class Crisis {
  constructor(public id: number, public name: string) { }
}

const CRISES = [
  new Crisis(1, 'Dragon Burning Cities'),
  new Crisis(2, 'Sky Rains Great White Sharks'),
  new Crisis(3, 'Giant Asteroid Heading For Earth'),
  new Crisis(4, 'Procrastinators Meeting Delayed Again'),
];
```

最终的危机中心可以作为引入子路由这个新概念的基础。 你可以把**英雄管理**保持在当前状态，以便和**危机中心**进行对比，以后再根据这些差异是否有价值来决定后续行动。

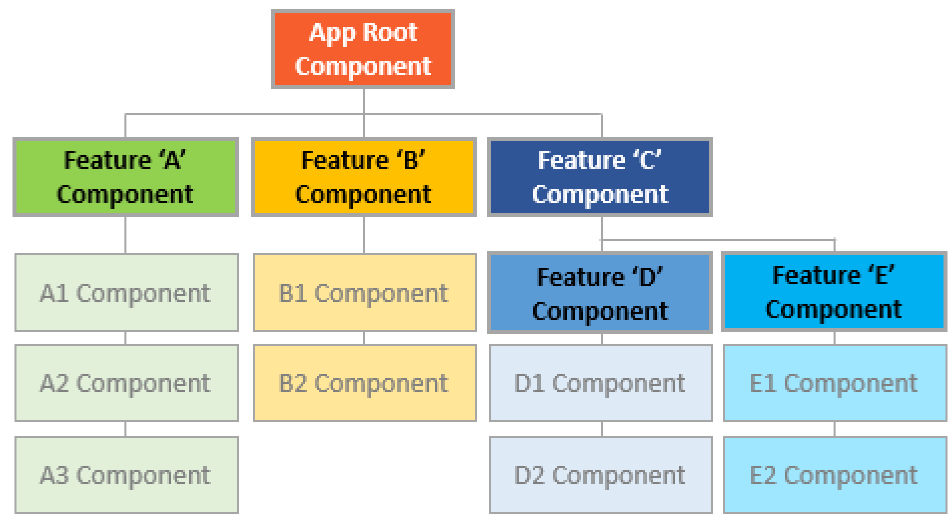
> 遵循**关注点分离（Separation of Concerns）**原则， 对**危机中心**的修改不会影响 `AppModule` 或其它特性模块中的组件。

带有子路由的危机中心

本节会展示如何组织危机中心，来满足 Angular 应用所推荐的模式：

- 把每个特性放在自己的目录中。
- 每个特性都有自己的 Angular 特性模块。
- 每个特性区都有自己的根组件。
- 每个特性区的根组件中都有自己的路由出口及其子路由。
- 特性区的路由很少（或完全不）与其它特性区的路由交叉。

如果你还有更多特性区，它们的组件树是这样的：



子路由组件

往 `crisis-center` 目录下添加下列 `crisis-center.component.ts` 文件：

src/app/crisis-center/crisis-center.component.ts

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

@Component({
  template:  `
    <h2>CRISIS CENTER</h2>
    <router-outlet></router-outlet>
  `
})
export class CrisisCenterComponent { }
```

`CrisisCenterComponent` 和 `AppComponent` 有下列共同点：

- 它是危机中心特性区的**根**，正如 `AppComponent` 是整个应用的根。
- 它是危机管理特性区的**壳**，正如 `AppComponent` 是管理高层工作流的壳。

就像大多数的壳一样，`CrisisCenterComponent` 类也非常简单，甚至比 `AppComponent` 更简单：它没有业务逻辑，它的模板中没有链接，只有一个标题和用于放置危机中心的子视图的 `<router-outlet>`。

与 `AppComponent` 和大多数其它组件不同的是，它甚至都**没有指定选择器** `selector`。它不**需要**选择器，因为你不会把这个组件嵌入到某个父模板中，而是使用路由器**导航**到它。

子路由配置

把下面这个 `crisis-center-home.component.ts` 添加到 `crisis-center` 目录下，作为 "危机中心" 特性区的宿主页面。

src/app/crisis-center/crisis-center-home.component.ts

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

@Component({
  template: `
    <p>Welcome to the Crisis Center</p>
  `
})
export class CrisisCenterHomeComponent { }
```

像 `heroes-routing.module.ts` 文件一样，你也创建一个 `crisis-center-routing.module.ts`。但这次，你要把子路由定义在父路由 `crisis-center` 中。

src/app/crisis-center/crisis-center-routing.module.ts (Routes)

```
const crisisCenterRoutes: Routes = [
  {
    path: 'crisis-center',
    component: CrisisCenterComponent,
    children: [
      {
        path: '',
        component: CrisisListComponent,
        children: [
          {
            path: ':id',
            component: CrisisDetailComponent
          },
          {
            path: '',
            component: CrisisCenterHomeComponent
          }
        ]
      }
    ]
  }
];
```

注意，父路由 `crisis-center` 有一个 `children` 属性，它有一个包含 `CrisisListComponent` 的路由。`CrisisListModule` 路由还有一个带两个路由的 `children` 数组。

这两个路由导航到了**危机中心**的两个子组件：`CrisisCenterHomeComponent` 和 `CrisisDetailComponent`。

对这些路由的处理中有一些**重要的不同**。

路由器会把这些路由对应的组件放在 `CrisisCenterComponent` 的 `RouterOutlet` 中，而不是 `AppComponent` 壳组件中的。

`CrisisListComponent` 包含危机列表和一个 `RouterOutlet`，用以显示 `Crisis Center Home` 和 `Crisis Detail` 这两个路由组件。

`Crisis Detail` 路由是 `Crisis List` 的子路由。由于路由器默认会复用组件，因此当你选择了另一个危机时，`CrisisDetailComponent` 会被复用。
作为对比，回到 `Hero Detail` 路由时，每当你选择了不同的英雄时，该组件都会被重新创建。

在顶级，以 `/` 开头的路径指向的总是应用的根。 但这里是子路由。 它们是在父路由路径的基础上做出的扩展。 在路由树中每深入一步，你就会在该路由的路径上添加一个斜线 `/`（除非该路由的路径是**空的**）。

如果把该逻辑应用到危机中心中的导航，那么父路径就是 `/crisis-center`。

- 要导航到 `CrisisCenterHomeComponent`，完整的 URL 是 `/crisis-center` (`/crisis-center` + `''` + `''`)。
- 要导航到 `CrisisDetailComponent` 以展示 `id=2` 的危机，完整的 URL 是 `/crisis-center/2` (`/crisis-center` + `''` + `'/2'`)。

本例子中包含站点部分的绝对 URL，就是：

```
localhost:3000/crisis-center/2
```

这里是完整的 `crisis-center.routing.ts` 及其导入语句。

src/app/crisis-center/crisis-center-routing.module.ts (excerpt)

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

import { CrisisCenterHomeComponent } from './crisis-center-home.component';
import { CrisisListComponent }       from './crisis-list.component';
import { CrisisCenterComponent }     from './crisis-center.component';
import { CrisisDetailComponent }     from './crisis-detail.component';

const crisisCenterRoutes: Routes = [
  {
    path: 'crisis-center',
    component: CrisisCenterComponent,
    children: [
      {
        path: '',
        component: CrisisListComponent,
        children: [
          {
            path: ':id',
            component: CrisisDetailComponent
          },
          {
            path: '',
            component: CrisisCenterHomeComponent
          }
        ]
      }
    ]
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(crisisCenterRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class CrisisCenterRoutingModule { }
```

把危机中心模块导入到 `AppModule` 的路由中

就像 `HeroesModule` 模块中一样，你必须把 `CrisisCenterModule` 添加到 `AppModule` 的 `imports` 数组中，就在 `AppRoutingModule` 前面：

src/app/app.module.ts (import CrisisCenterModule)

```
import { NgModule }       from '@angular/core';
import { CommonModule }   from '@angular/common';
import { FormsModule }    from '@angular/forms';

import { AppComponent }            from './app.component';
import { PageNotFoundComponent }   from './not-found.component';

import { AppRoutingModule }        from './app-routing.module';
import { HeroesModule }            from './heroes/heroes.module';
import { CrisisCenterModule }      from './crisis-center/crisis-center.module';

import { DialogService }           from './dialog.service';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    HeroesModule,
    CrisisCenterModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    PageNotFoundComponent
  ],
  providers: [
    DialogService
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

从 `app.routing.ts` 中移除危机中心的初始路由。 这些特性路由现在是由 `HeroesModule` 和 `CrisisCenter` 特性模块提供的。

`app-routing.module.ts` 文件中只有应用的顶级路由，比如默认路由和通配符路由。

src/app/app-routing.module.ts (v3)

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

import { ComposeMessageComponent } from './compose-message.component';
import { PageNotFoundComponent }  from './not-found.component';

const appRoutes: Routes = [
  { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

相对导航

虽然构建出了危机中心特性区，你却仍在使用以斜杠开头的绝对路径来导航到危机详情的路由。

路由器会从路由配置的顶层来匹配像这样的**绝对路径**。

你固然可以继续像**危机中心**特性区一样使用绝对路径，但是那样会把链接钉死在特定的父路由结构上。 如果你修改了父路径 `/crisis-center`，那就不得不修改每一个链接参数数组。

通过改成定义**相对于**当前 URL 的路径，你可以把链接从这种依赖中解放出来。 当你修改了该特性区的父路由路径时，该特性区**内部**的导航仍然完好无损。

例子如下：

> 路由器支持在**链接参数数组**中使用"目录式"语法来为查询路由名提供帮助：
>
> `./` 或 `无前导斜线` 形式是相对于当前级别的。

> `../` 会回到当前路由路径的上一级。
>
> 你可以把相对导航语法和一个祖先路径组合起来用。 如果不得不导航到一个兄弟路由，你可以用 `../<sibling>` 来回到上一级，然后进入兄弟路由路径中。

用 `Router.navigate` 方法导航到相对路径时，你必须提供当前的 `ActivatedRoute`，来让路由器知道你现在位于路由树中的什么位置。

在**链接参数数组**中，添加一个带有 `relativeTo` 属性的对象，并把它设置为当前的 `ActivatedRoute`。 这样路由器就会基于当前激活路由的位置来计算出目标 URL。

> 当调用路由器的 `navigateByUrl` 时，总是要指定完整的**绝对路径**。

使用相对 URL 导航到危机列表

你已经注入过了 `ActivatedRoute`，你需要把它来和相对导航路径组合在一起。

如果用 `RouterLink` 来代替 `Router` 服务进行导航，就要使用**相同**的链接参数数组，不过不再需要提供 `relativeTo` 属性。 `ActivatedRoute` 已经隐含在了 `RouterLink` 指令中。

修改 `CrisisDetailComponent` 的 `gotoCrises` 方法，来使用相对路径返回**危机中心**列表。

src/app/crisis-center/crisis-detail.component.ts (relative navigation)

```
// Relative navigation back to the crises
this.router.navigate(['../', { id: crisisId, foo: 'foo' }], { relativeTo: this.route
});
```

注意这个路径使用了 `../` 语法返回上一级。 如果当前危机的 `id` 是 `3`，那么最终返回到的路径就是 `/crisis-center/;id=3;foo=foo`。

用命名出口（outlet）显示多重路由

你决定给用户提供一种方式来联系危机中心。 当用户点击“Contact”按钮时，你要在一个弹出框中显示一条消息。

即使在应用中的不同页面之间切换，这个弹出框也应该始终保持打开状态，直到用户发送了消息或者手动取消。 显然，你不能把这个弹出框跟其它放到页面放到同一个路由出口中。

迄今为止，你只定义过单路由出口，并且在其中嵌套了子路由以便对路由分组。 在每个模板中，路由器只能支持一个**无名**主路由出口。

模板还可以有多个**命名的**路由出口。 每个命名出口都自己有一组带组件的路由。 多重出口可以在同一时间根据不同的路由来显示不同的内容。

在 `AppComponent` 中添加一个名叫“popup”的出口，就在无名出口的下方。

src/app/app.component.ts (outlets)

```
<router-outlet></router-outlet>
<router-outlet name="popup"></router-outlet>
```

一旦你学会了如何把一个弹出框组件路由到该出口，那里就是将会出现弹出框的地方。

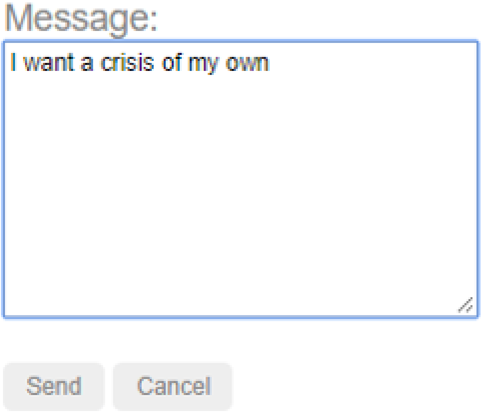
第二路由

命名出口是**第二路由**的目标。

第二路由很像主路由，配置方式也一样。它们只有一些关键的不同点：

- 它们彼此互不依赖。
- 它们与其它路由组合使用。
- 它们显示在命名出口中。

在 `src/app/compose-message.component.ts` 中创建一个名叫 `ComposeMessageComponent` 的新组件。 它显示一个简单的表单，包括一个头、一个消息输入框和两个按钮：“Send”和“Cancel”。



下面是该组件及其模板：

src/app/compose-message.component.ts *src/app/compose-message.component.html*

```
1. import { Component, HostBinding } from '@angular/core';
2. import { Router }                 from '@angular/router';
```

```
3.
4. import { slideInDownAnimation }   from './animations';
5.
6. @Component({
7.   templateUrl: './compose-message.component.html',
8.   styles: [ ':host { position: relative; bottom: 10%; }' ],
9.   animations: [ slideInDownAnimation ]
10. })
11. export class ComposeMessageComponent {
12.   @HostBinding('@routeAnimation') routeAnimation = true;
13.   @HostBinding('style.display')   display = 'block';
14.   @HostBinding('style.position')  position = 'absolute';
15.
16.   details: string;
17.   sending = false;
18.
19.   constructor(private router: Router) {}
20.
21.   send() {
22.     this.sending = true;
23.     this.details = 'Sending Message...';
24.
25.     setTimeout(() => {
26.       this.sending = false;
27.       this.closePopup();
28.     }, 1000);
29.   }
30.
31.   cancel() {
32.     this.closePopup();
33.   }
34.
35.   closePopup() {
36.     // Providing a `null` value to the named outlet
37.     // clears the contents of the named outlet
38.     this.router.navigate([{ outlets: { popup: null }}]);
39.   }
40. }
```

它看起来几乎和你以前见过其它组件一样，但有两个值得注意的区别。

主要 `send()` 方法在发送消息和关闭弹出框之前通过等待模拟了一秒钟的延迟。

`closePopup()` 方法用把 `popup` 出口导航到 `null` 的方式关闭了弹出框。 这个奇怪的用法在稍后的部分有讲解。

像其它组件一样，你还要把 `ComposeMessageComponent` 添加到 `AppModule` 的 `declarations` 中。

添加第二路由

打开 `AppRoutingModule`，并把一个新的 `compose` 路由添加到 `appRoutes` 中。

src/app/app-routing.module.ts (compose route)

```
{
  path: 'compose',
  component: ComposeMessageComponent,
  outlet: 'popup'
},
```

对 `path` 和 `component` 属性应该很熟悉了吧。 注意这个新的属性 `outlet` 被设置成了 `'popup'`。 这个路由现在指向了 `popup` 出口，而 `ComposeMessageComponent` 也将显示在那里。

用户需要某种途径来打开这个弹出框。 打开 `AppComponent`，并添加一个“Contact”链接。

src/app/app.component.ts (contact-link)

```
<a [routerLink]="[{ outlets: { popup: ['compose'] } }]">Contact</a>
```

虽然 `compose` 路由被钉死在了 `popup` 出口上，但这仍然不足以向 `RouterLink` 指令表明要加载该路由。 你还要在**链接参数数组**中指定这个命名出口，并通过属性绑定的形式把它绑定到 `RouterLink` 上。

链接参数数组包含一个只有一个 `outlets` 属性的对象，它的值是另一个对象，这个对象以一个或多个路由的出口名作为属性名。 在这里，它只有一个出口名“popup”，它的值则是另一个**链接参数数组**，用于指定 `compose` 路由。

意思是，当用户点击此链接时，在路由出口 `popup` 中显示与 `compose` 路由相关联的组件。

> 当有且只有一个**无名**出口时，外部对象中的这个 `outlets` 对象并不是必须的。
>
> 路由器假设这个路由指向了**无名**的主出口，并为你创建这些对象。
>
> 路由到一个命名出口就会揭示一个以前被隐藏的真相： 你可以在同一个 `RouterLink` 指令中为多个路由出口指定多个路由。
>
> 这里你实际上没能这样做。要想指向命名出口，你就得使用一种更强大也更啰嗦的语法。

第二路由导航：在导航期间合并路由

导航到**危机中心**并点击“Contact”，你将会在浏览器的地址栏看到如下 URL：

```
http://.../crisis-center(popup:compose)
```

这个 URL 中有意思的部分是 `...` 后面的这些：

- `crisis-center` 是主导航。
- 圆括号包裹的部分是第二路由。
- 第二路由包括一个出口名称（`popup`）、一个冒号分隔符和第二路由的路径（`compose`）。

点击 **Heroes** 链接，并再次查看 URL：

```
http://.../heroes(popup:compose)
```

主导航的部分变化了，而第二路由没有变。

路由器在导航树中对两个独立的分支保持追踪，并在 URL 中对这棵树进行表达。

你还可以添加更多出口和更多路由（无论是在顶层还是在嵌套的子层）来创建一个带有多个分支的导航树。路由器将会生成相应的 URL。

通过像前面那样填充 `outlets` 对象，你可以告诉路由器立即导航到一棵完整的树。 然后把这个对象通过一个**链接参数数组**传给 `router.navigate` 方法。

有空的时候你可以自行试验这些可能性。

清除第二路由

正如你刚刚学到的，除非导航到新的组件，否则路由出口中的组件会始终存在。 这里涉及到的第二出口也同样如此。

每个第二出口都有自己独立的导航，跟主出口的导航彼此独立。 修改主出口中的当前路由并不会影响到 `popup` 出口中的。 这就是为什么在危机中心和英雄管理之间导航时，弹出框始终都是可见的。

点击“send”或“cancel”按钮，则**会**清除弹出框视图。 为何如此？再看看 `closePopup()` 方法：

src/app/compose-message.component.ts (closePopup)

```
closePopup() {
  // Providing a `null` value to the named outlet
  // clears the contents of the named outlet
  this.router.navigate([{ outlets: { popup: null }}]);
}
```

它使用 `Router.navigate()` 方法进行强制导航，并传入了一个链接参数数组。

就像在 `AppComponent` 中绑定到的 **Contact** `RouterLink` 一样，它也包含了一个带 `outlets` 属性的对象。`outlets` 属性的值是另一个对象，该对象用一些出口名称作为属性名。 唯一的命名出口是 `'popup'`。

但这次，`'popup'` 的值是 `null`。`null` 不是一个路由，但却是一个合法的值。 把 `popup` 这个 `RouterOutlet` 设置为 `null` 会清除该出口，并且从当前 URL 中移除第二路由 `popup`。

里程碑 5：路由守卫

现在，**任何用户**都能在**任何时候**导航到**任何地方**。 但有时候这样是不对的。

- 该用户可能无权导航到目标组件。
- 可能用户得先登录（认证）。
- 在显示目标组件前，你可能得先获取某些数据。
- 在离开组件前，你可能要先保存修改。
- 你可能要询问用户：你是否要放弃本次更改，而不用保存它们?

你可以往路由配置中添加**守卫**，来处理这些场景。

守卫返回一个值，以控制路由器的行为：

- 如果它返回 `true`，导航过程会继续
- 如果它返回 `false`，导航过程会终止，且用户会留在原地。

> 守卫还可以告诉路由器导航到别处，这样也取消当前的导航。

守卫**可以**用同步的方式返回一个布尔值。但在很多情况下，守卫无法用同步的方式给出答案。 守卫可能会向用户问一个问题、把更改保存到服务器，或者获取新数据，而这些都是异步操作。

因此，路由的守卫可以返回一个 `Observable<boolean>` 或 `Promise<boolean>`，并且路由器会等待这个可观察对象被解析为 `true` 或 `false`。

路由器可以支持多种守卫接口：

- 用`CanActivate`来处理导航**到**某路由的情况。
- 用`CanActivateChild`来处理导航**到**某子路由的情况。
- 用`CanDeactivate`来处理从当前路由**离开**的情况.
- 用`Resolve`在路由激活**之前**获取路由数据。
- 用`CanLoad`来处理**异步**导航到某特性模块的情况。

在分层路由的每个级别上，你都可以设置多个守卫。 路由器会先按照从最深的子路由由下往上检查的顺序来检查 `CanDeactivate()` 和 `CanActivateChild()` 守卫。 然后它会按照从上到下的顺序检查 `CanActivate()` 守卫。 如果特性模块是异步加载的，在加载它之前还会检查 `CanLoad()` 守卫。 如果**任何**一个守卫返回 `false`，其它尚未完成的守卫会被取消，这样整个导航就被取消了。

接下来的小节中有一些例子。

CanActivate: 要求认证

应用程序通常会根据访问者来决定是否授予某个特性区的访问权。 你可以只对已认证过的用户或具有特定角色的用户授予访问权，还可以阻止或限制用户访问权，直到用户账户激活为止。

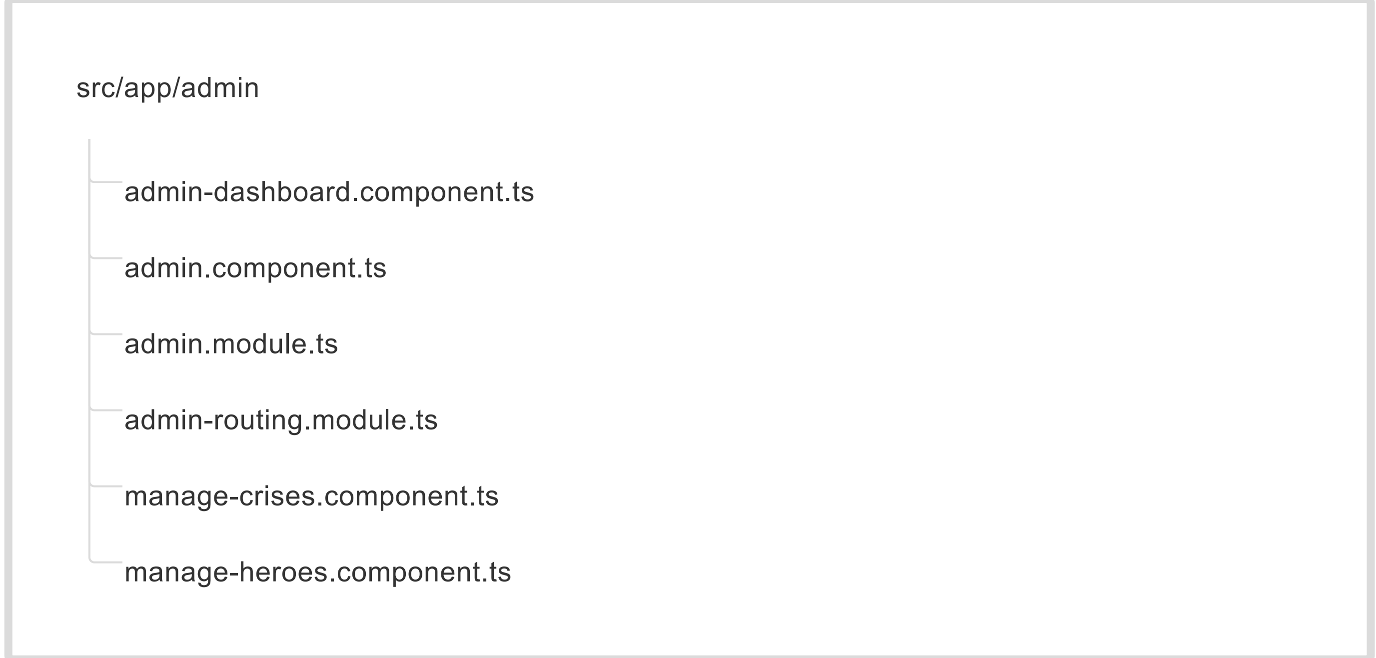
`CanActivate` 守卫是一个管理这些导航类业务规则的工具。

添加一个“管理”特性模块

在下一节，你将会使用一些新的**管理**特性来扩展危机中心。 那些特性尚未定义，但是你可以先从添加一个名叫 `AdminModule` 的特性模块开始。

创建一个 `admin` 目录，它带有一个特性模块文件、一个路由配置文件和一些支持性组件。

管理特性区的文件是这样的：



管理特性模块包含 `AdminComponent`，它用于在特性模块内的仪表盘路由以及两个尚未完成的用于管理危机和英雄的组件之间进行路由。

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

@Component({
  template:  `
    <p>Dashboard</p>
  `
})
export class AdminDashboardComponent { }
```

> 由于 `AdminModule` 中 `AdminComponent` 中的 `RouterLink` 是一个空路径的路由，所以它会匹配到管理特性区的任何路由。 但你只有在访问 `Dashboard` 路由时才希望该链接被激活。 往 `Dashboard` 这个 routerLink 上添加另一个绑定 `[routerLinkActiveOptions]="{ exact: true }"`， 这样就只有当用户导航到 `/admin` 这个 URL 时才会激活它，而不会在导航到它的某个子路由时。

最初的管理路由配置如下：

src/app/admin/admin-routing.module.ts (admin routing)

```
const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    children: [
      {
        path: '',
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ]
      }
    ]
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(adminRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AdminRoutingModule {}
```

无组件路由: 不借助组件对路由进行分组

来看 `AdminComponent` 下的子路由，这里有一个带 path 和 children 的子路由， 但它没有使用 component。这并不是配置中的失误，而是在使用无组件路由。

这里的目标是对 `admin` 路径下的 `危机中心` 管理类路由进行分组，但并不需要另一个仅用来分组路由的组件。一个**无组件**的路由能让守卫子路由变得更容易。

接下来，把 `AdminModule` 导入到 `app.module.ts` 中，并把它加入 `imports` 数组中来注册这些管理类路由。

src/app/app.module.ts (admin module)

```
import { NgModule }       from '@angular/core';
import { CommonModule }   from '@angular/common';
import { FormsModule }    from '@angular/forms';

import { AppComponent }            from './app.component';
import { PageNotFoundComponent }   from './not-found.component';

import { AppRoutingModule }        from './app-routing.module';
import { HeroesModule }            from './heroes/heroes.module';
import { CrisisCenterModule }      from './crisis-center/crisis-center.module';
import { AdminModule }             from './admin/admin.module';

import { DialogService }           from './dialog.service';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    HeroesModule,
    CrisisCenterModule,
    AdminModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    PageNotFoundComponent
  ],
  providers: [
    DialogService
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

然后往壳组件 `AppComponent` 中添加一个链接，让用户能点击它，以访问该特性。

src/app/app.component.ts (template)

```
template: `
  <h1 class="title">Angular Router</h1>
  <nav>
    <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
    <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
    <a routerLink="/admin" routerLinkActive="active">Admin</a>
    <a [routerLink]="[{ outlets: { popup: ['compose'] } }]">Contact</a>
  </nav>
  <router-outlet></router-outlet>
  <router-outlet name="popup"></router-outlet>
  `
```

守护“管理特性”区

现在“危机中心”的每个路由都是对所有人开放的。这些新的**管理特性**应该只能被已登录用户访问。

你可以在用户登录之前隐藏这些链接，但这样会有点复杂并难以维护。

你可以换种方式：写一个 `CanActivate()` 守卫，将正在尝试访问管理组件匿名用户重定向到登录页。

这是一种具有通用性的守护目标（通常会有其它特性需要登录用户才能访问），所以你要在应用的根目录下创建一个 `auth-guard.ts` 文件。

此刻，你的兴趣在于看看守卫是如何工作的，所以第一个版本没做什么有用的事情。它只是往控制台写日志，并且立即返回 `true`，让导航继续：

src/app/auth-guard.service.ts (excerpt)

```
import { Injectable }     from '@angular/core';
import { CanActivate }    from '@angular/router';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate() {
    console.log('AuthGuard#canActivate called');
    return true;
  }
}
```

接下来，打开 `crisis-center.routes.ts`，导入 `AuthGuard` 类，修改管理路由并通过 `CanActivate()` 守卫来引用 `AuthGuard`：

src/app/admin/admin-routing.module.ts (guarded admin route)

```
import { AuthGuard }                from '../auth-guard.service';

const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard],
    children: [
      {
        path: '',
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ],
      }
    ]
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(adminRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AdminRoutingModule {}
```

管理特性区现在受此守卫保护了，不过这样的保护还不够。

教 AuthGuard 进行认证

先让 `AuthGuard` 至少能“假装”进行认证。

`AuthGuard` 可以调用应用中的一项服务，该服务能让用户登录，并且保存当前用户的信息。下面是一个 `AuthService` 的示范：

src/app/auth.service.ts (excerpt)

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

import { Observable, of } from 'rxjs';
import { tap, delay } from 'rxjs/operators';

@Injectable()
export class AuthService {
  isLoggedIn = false;

  // store the URL so we can redirect after logging in
  redirectUrl: string;

  login(): Observable<boolean> {
    return of(true).pipe(
      delay(1000),
      tap(val => this.isLoggedIn = true)
    );
  }

  logout(): void {
    this.isLoggedIn = false;
  }
}
```

虽然它不会真的进行登录，但足够让你进行这个讨论了。 它有一个 `isLoggedIn` 标志，用来标识是否用户已经登录过了。 它的 `login` 方法会仿真一个对外部服务的 API 调用，返回一个可观察对象（observable）。在短暂的停顿之后，这个可观察对象就会解析成功。 `redirectUrl` 属性将会保存在 URL 中，以便认证完之后导航到它。

这就修改 `AuthGuard` 来调用它。

src/app/auth-guard.service.ts (v2)

```
import { Injectable }       from '@angular/core';
import {
  CanActivate, Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot
}                           from '@angular/router';
import { AuthService }      from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean
{
    let url: string = state.url;

    return this.checkLogin(url);
  }

  checkLogin(url: string): boolean {
    if (this.authService.isLoggedIn) { return true; }

    // Store the attempted URL for redirecting
    this.authService.redirectUrl = url;

    // Navigate to the login page with extras
    this.router.navigate(['/login']);
    return false;
  }
}
```

注意，你把 `AuthService` 和 `Router` 服务**注入到**构造函数中。 你还没有提供 `AuthService`，这里要说明的是：可以往路由守卫中注入有用的服务。

该守卫返回一个同步的布尔值。如果用户已经登录，它就返回 `true`，导航会继续。

这个 `ActivatedRouteSnapshot` 包含了**即将**被激活的路由，而 `RouterStateSnapshot` 包含了该应用**即将**到达的状态。 你应该通过守卫进行检查。

如果用户还没有登录，你就会用 `RouterStateSnapshot.url` 保存用户来自的 URL 并让路由器导航到登录页（你尚未创建该页）。 这间接导致路由器自动中止了这次导航，`checkLogin()` 返回 `false` 并不是必须的，但这样可以更清楚的表达意图。

添加 LoginComponent

你需要一个 `LoginComponent` 来让用户登录进这个应用。在登录之后，你就会跳转到前面保存的 URL，如果没有，就跳转到默认 URL。 该组件没有什么新内容，你把它放进路由配置的方式也没什么新意。

在 `login-routing.module.ts` 中注册一个 `/login` 路由，并把必要的提供商添加 `providers` 数组中。 在 `app.module.ts` 中，导入 `LoginComponent` 并把它加入根模块的 `declarations` 中。 同时在 `AppModule` 中导入并添加 `LoginRoutingModule`。

src/app/app.module.ts *src/app/login.component.ts* *src/app/login-routing.module.ts*

```
1. import { NgModule }       from '@angular/core';
2. import { BrowserModule }  from '@angular/platform-browser';
3. import { FormsModule }    from '@angular/forms';
4. import { BrowserAnimationsModule } from '@angular/platform-
   browser/animations';
5.
6. import { Router } from '@angular/router';
7.
8. import { AppComponent }            from './app.component';
9. import { AppRoutingModule }        from './app-routing.module';
10.
11. import { HeroesModule }            from './heroes/heroes.module';
12. import { ComposeMessageComponent } from './compose-message.component';
13. import { LoginRoutingModule }      from './login-routing.module';
14. import { LoginComponent }          from './login.component';
15. import { PageNotFoundComponent }   from './not-found.component';
16.
17. import { DialogService }           from './dialog.service';
18.
19. @NgModule({
20.   imports: [
21.     BrowserModule,
22.     FormsModule,
23.     HeroesModule,
24.     LoginRoutingModule,
25.     AppRoutingModule,
26.     BrowserAnimationsModule
27.   ],
28.   declarations: [
29.     AppComponent,
30.     ComposeMessageComponent,
31.     LoginComponent,
```

```
32.       PageNotFoundComponent
33.     ],
34.     providers: [
35.       DialogService
36.     ],
37.     bootstrap: [ AppComponent ]
38.   })
39.   export class AppModule {
40.     // Diagnostic only: inspect router configuration
41.     constructor(router: Router) {
42.       console.log('Routes: ', JSON.stringify(router.config, undefined, 2));
43.     }
44.   }
```

> 它们所需的守卫和服务提供商必须在模块一级提供。这让路由器在导航过程中可以通过 `Injector` 来取得这些服务。 同样的规则也适用于异步加载的特性模块。

`CanActivateChild`：保护子路由

你还可以使用 `CanActivateChild` 守卫来保护子路由。 `CanActivateChild` 守卫和 `CanActivate` 守卫很像。 它们的区别在于，`CanActivateChild` 会在**任何子路由**被激活之前运行。

你要保护管理特性模块，防止它被非授权访问，还要保护这个特性模块**内部**的那些子路由。

扩展 `AuthGuard` 以便在 `admin` 路由之间导航时提供保护。 打开 `auth-guard.service.ts` 并从路由库中导入 `CanActivateChild` 接口。

接下来，实现 `CanActivateChild` 方法，它所接收的参数与 `CanActivate` 方法一样：一个 `ActivatedRouteSnapshot` 和一个 `RouterStateSnapshot`。 `CanActivateChild` 方法可以返回 `Observable<boolean>` 或 `Promise<boolean>` 来支持异步检查，或 `boolean` 来支持同步检查。 这里返回的是 `boolean`：

src/app/auth-guard.service.ts (excerpt)

```
import { Injectable }       from '@angular/core';
import {
  CanActivate, Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  CanActivateChild
}                           from '@angular/router';
import { AuthService }      from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean
{
    let url: string = state.url;

    return this.checkLogin(url);
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
boolean {
    return this.canActivate(route, state);
  }

/* . . . */
}
```

同样把这个 `AuthGuard` 添加到“无组件的”管理路由，来同时保护它的所有子路由，而不是为每个路由单独添加这个 `AuthGuard`。

src/app/admin/admin-routing.module.ts (excerpt)

```
const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard],
    children: [
      {
        path: '',
        canActivateChild: [AuthGuard],
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ]
      }
    ]
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(adminRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AdminRoutingModule {}
```

CanDeactivate：处理未保存的更改

回到“Heroes”工作流，该应用毫不犹豫的接受对英雄的任何修改，不作任何校验。

在现实世界中，你得先把用户的改动积累起来。 你可能不得不进行跨字段的校验，可能要找服务器进行校验，可能得把这些改动保存成一种待定状态，直到用户或者把这些改动**作为一组**进行确认或撤销所有改动。

当用户要导航到外面时，该怎么处理这些既没有审核通过又没有保存过的改动呢？ 你不能马上离开，不在乎丢失这些改动的风险，那显然是一种糟糕的用户体验。

最好能暂停，并让用户决定该怎么做。 如果用户选择了取消，你就留下来，并允许更多改动。 如果用户选择了确认，那就进行保存。

在保存成功之前，你还可以继续推迟导航。如果你让用户立即移到下一个界面，而保存却失败了（可能因为数据不符合有效性规则），你就会丢失该错误的上下文环境。

在等待服务器的答复时，你没法阻塞它 —— 这在浏览器中是不可能的。 你只能用异步的方式在等待服务器答复之前先停止导航。

你需要 `CanDeactivate` 守卫。

取消与保存

这个范例应用不会与服务器通讯。 幸运的是，你有另一种方式来演示异步的路由器钩子。

用户在 `CrisisDetailComponent` 中更新危机信息。 与 `HeroDetailComponent` 不同，用户的改动不会立即更新危机的实体对象。当用户按下了 **Save** 按钮时，应用就更新这个实体对象；如果按了 **Cancel** 按钮，那就放弃这些更改。

这两个按钮都会在保存或取消之后导航回危机列表。

src/app/crisis-center/crisis-detail.component.ts (cancel and save methods)

```
cancel() {
  this.gotoCrises();
}

save() {
  this.crisis.name = this.editName;
  this.gotoCrises();
}
```

如果用户尝试不保存或撤销就导航到外面该怎么办？ 用户可以按浏览器的后退按钮，或点击英雄的链接。 这些操作都会触发导航。本应用应该自动保存或取消吗？

都不行。应用应该弹出一个确认对话框来要求用户明确做出选择，该对话框会**用异步的方式等用户做出选择**。

> 你也能用同步的方式等用户的答复，阻塞代码。但如果能用异步的方式等待用户的答复，应用就会响应性更好，也能同时做别的事。异步等待用户的答复和等待服务器的答复是类似的。

`DialogService`（为了在应用级使用，已经注入到了 `AppModule`）就可以做到这些。

它返回promise，当用户最终决定了如何去做时，它就会被**解析** —— 或者决定放弃更改直接导航离开（`true`），或者保留未完成的修改，留在危机编辑器中（`false`）。

创建了一个 `Guard`，它将检查这个（任意）组件中是否有 `canDeactivate()` 函数。 `CrisisDetailComponent` 就会有这个方法。 但是该守卫并不需要知道 `CrisisDetailComponent` 确认退出激活状态的详情。 它只需要检查该组件是否有一个 `canDeactivate()` 方法，并调用它。 这就让该守卫可以复用。

src/app/can-deactivate-guard.service.ts

```
import { Injectable }    from '@angular/core';
import { CanDeactivate } from '@angular/router';
import { Observable }    from 'rxjs';

export interface CanComponentDeactivate {
 canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}

@Injectable()
export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {
  canDeactivate(component: CanComponentDeactivate) {
    return component.canDeactivate ? component.canDeactivate() : true;
  }
}
```

另外，你也可以为 `CrisisDetailComponent` 创建一个特定的 `CanDeactivate` 守卫。 在需要访问外部信息时，`canDeactivate()` 方法为你提供了组件、`ActivatedRoute` 和 `RouterStateSnapshot` 的当前实例。 如果只想为这个组件使用该守卫，并且需要获取该组件属性或确认路由器是否允许从该组件导航出去时，这会非常有用。

src/app/can-deactivate-guard.service.ts (component-specific)

```
import { Injectable }           from '@angular/core';
import { Observable }           from 'rxjs';
import { CanDeactivate,
         ActivatedRouteSnapshot,
         RouterStateSnapshot }  from '@angular/router';

import { CrisisDetailComponent } from './crisis-center/crisis-detail.component';

@Injectable()
export class CanDeactivateGuard implements CanDeactivate<CrisisDetailComponent> {

  canDeactivate(
    component: CrisisDetailComponent,
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | boolean {
    // Get the Crisis Center ID
    console.log(route.paramMap.get('id'));

    // Get the current URL
    console.log(state.url);

    // Allow synchronous navigation (`true`) if no crisis or the crisis is unchanged
    if (!component.crisis || component.crisis.name === component.editName) {
      return true;
    }
    // Otherwise ask the user with the dialog service and return its
    // observable which resolves to true or false when the user decides
    return component.dialogService.confirm('Discard changes?');
  }
}
```

看看 `CrisisDetailComponent` 组件，它已经实现了对未保存的更改进行确认的工作流。

src/app/crisis-center/crisis-detail.component.ts (excerpt)

```
canDeactivate(): Observable<boolean> | boolean {
  // Allow synchronous navigation (`true`) if no crisis or the crisis is unchanged
  if (!this.crisis || this.crisis.name === this.editName) {
    return true;
  }
  // Otherwise ask the user with the dialog service and return its
  // observable which resolves to true or false when the user decides
  return this.dialogService.confirm('Discard changes?');
}
```

注意，`canDeactivate` 方法**可以**同步返回，如果没有危机，或者没有未定的修改，它就立即返回 `true`。但是它也可以返回一个承诺（`Promise`）或可观察对象（`Observable`），路由器将等待它们被解析为真值（继续导航）或假值（留下）。

往 `crisis-center.routing.module.ts` 的危机详情路由中用 `canDeactivate` 数组添加一个 `Guard`（守卫）。

src/app/crisis-center/crisis-center-routing.module.ts (can deactivate guard)

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

import { CrisisCenterHomeComponent } from './crisis-center-home.component';
import { CrisisListComponent }       from './crisis-list.component';
import { CrisisCenterComponent }     from './crisis-center.component';
import { CrisisDetailComponent }     from './crisis-detail.component';

import { CanDeactivateGuard }    from '../can-deactivate-guard.service';

const crisisCenterRoutes: Routes = [
  {
    path: '',
    redirectTo: '/crisis-center',
    pathMatch: 'full'
  },
  {
    path: 'crisis-center',
    component: CrisisCenterComponent,
    children: [
      {
        path: '',
        component: CrisisListComponent,
        children: [
          {
            path: ':id',
            component: CrisisDetailComponent,
            canDeactivate: [CanDeactivateGuard]
          },
          {
            path: '',
            component: CrisisCenterHomeComponent
          }
        ]
      }
    ]
  }
];

@NgModule({
  imports: [
```

```
    RouterModule.forChild(crisisCenterRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class CrisisCenterRoutingModule { }
```

还要把这个 `Guard` 添加到 `AppRoutingModule` 的 `providers` 中去，以便 `Router` 可以在导航过程中注入它。

```
1. import { NgModule }               from '@angular/core';
2. import { RouterModule, Routes }  from '@angular/router';
3.
4. import { ComposeMessageComponent } from './compose-message.component';
5. import { CanDeactivateGuard }      from './can-deactivate-guard.service';
6. import { PageNotFoundComponent }   from './not-found.component';
7.
8. const appRoutes: Routes = [
9.   {
10.     path: 'compose',
11.     component: ComposeMessageComponent,
12.     outlet: 'popup'
13.   },
14.   { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
15.   { path: '**', component: PageNotFoundComponent }
16. ];
17.
18. @NgModule({
19.   imports: [
20.     RouterModule.forRoot(
21.       appRoutes,
22.       { enableTracing: true } // <-- debugging purposes only
23.     )
24.   ],
25.   exports: [
26.     RouterModule
27.   ],
28.   providers: [
29.     CanDeactivateGuard
30.   ]
31. })
32. export class AppRoutingModule {}
```

现在，你已经给了用户一个能保护未保存更改的安全守卫。

Resolve: 预先获取组件数据

在 `Hero Detail` 和 `Crisis Detail` 中，它们等待路由读取完对应的英雄和危机。

这种方式没有问题，但是它们还有进步的空间。 如果你在使用真实 api，很有可能数据返回有延迟，导致无法即时显示。 在这种情况下，直到数据到达前，显示一个空的组件不是最好的用户体验。

最好预先从服务器上获取完数据，这样在路由激活的那一刻数据就准备好了。 还要在路由到此组件之前处理好错误。 但当某个 `id` 无法对应到一个危机详情时，就没办法处理它。 这时最好把用户带回到“危机列表”中，那里显示了所有有效的“危机”。

总之，你希望的是只有当所有必要数据都已经拿到之后，才渲染这个路由组件。

你需要 `Resolve` 守卫。

导航前预先加载路由信息

目前，`CrisisDetailComponent` 会接收选中的危机。 如果该危机没有找到，它就会导航回危机列表视图。

如果能在该路由将要激活时提前处理了这个问题，那么用户体验会更好。 `CrisisDetailResolver` 服务可以接收一个 `Crisis`，而如果这个 `Crisis` 不存在，就会在激活该路由并创建 `CrisisDetailComponent` 之前先行离开。

在“危机中心”特性区中创建 `crisis-detail-resolver.service.ts` 文件。

src/app/crisis-center/crisis-detail-resolver.service.ts

```
import { Injectable }             from '@angular/core';
import { Router, Resolve, RouterStateSnapshot,
         ActivatedRouteSnapshot } from '@angular/router';
import { Observable }             from 'rxjs';
import { map, take }              from 'rxjs/operators';

import { Crisis, CrisisService }  from './crisis.service';

@Injectable()
export class CrisisDetailResolver implements Resolve<Crisis> {
  constructor(private cs: CrisisService, private router: Router) {}

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Crisis> {
    let id = route.paramMap.get('id');

    return this.cs.getCrisis(id).pipe(
      take(1),
      map(crisis => {
        if (crisis) {
          return crisis;
        } else { // id not found
          this.router.navigate(['/crisis-center']);
          return null;
        }
      })
    );
  }
}
```

在 `CrisisDetailComponent.ngOnInit` 中拿到相关的危机检索逻辑，并且把它们移到 `CrisisDetailResolver` 中。 导入 `Crisis` 模型、`CrisisService` 和 `Router` 以便让你可以在找不到指定的危机时导航到别处。

为了更明确一点，可以实现一个带有 `Crisis` 类型的 `Resolve` 接口。

注入 `CrisisService` 和 `Router`，并实现 `resolve()` 方法。 该方法可以返回一个 `Promise`、一个 `Observable` 来支持异步方式，或者直接返回一个值来支持同步方式。

`CrisisService.getCrisis` 方法返回了一个可观察对象，这是为了防止在数据获取完毕前加载路由。 如果它没有返回一个有效的 `Crisis`，就把用户导航回 `CrisisListComponent`，并取消以前到 `CrisisDetailComponent` 尚未完成的导航。

把这个解析器（resolver）导入到 `crisis-center-routing.module.ts` 中，并往 `CrisisDetailComponent` 的路由配置中添加一个 `resolve` 对象。

别忘了把 `CrisisDetailResolver` 服务添加到 `CrisisCenterRoutingModule` 的 `providers` 数组中。

src/app/crisis-center/crisis-center-routing.module.ts (resolver)

```
import { CrisisDetailResolver }   from './crisis-detail-resolver.service';

@NgModule({
  imports: [
    RouterModule.forChild(crisisCenterRoutes)
  ],
  exports: [
    RouterModule
  ],
  providers: [
    CrisisDetailResolver
  ]
})
export class CrisisCenterRoutingModule { }
```

`CrisisDetailComponent` 不应该再去获取这个危机的详情。 把 `CrisisDetailComponent` 改成从 `ActivatedRoute.data.crisis` 属性中获取危机详情，这正是你重新配置路由的恰当时机。 当 `CrisisDetailComponent` 要求取得危机详情时，它就已经在那里了。

src/app/crisis-center/crisis-detail.component.ts (ngOnInit v2)

```
ngOnInit() {
  this.route.data
    .subscribe((data: { crisis: Crisis }) => {
      this.editName = data.crisis.name;
      this.crisis = data.crisis;
    });
}
```

两个关键点

1. 路由器的这个 `Resolve` 接口是可选的。`CrisisDetailResolver` 没有继承自某个基类。路由器只要找到了这个方法，就会调用它。
2. 要依赖路由器调用此守卫。不必关心用户用哪种方式导航离开，这是路由器的工作。你只要写出这个类，等路由器从那里取出它就可以了。
3. 由路由器提供的 Observable **必须** 完成（complete），否则导航不会继续。

本里程碑中与**危机中心**有关的代码如下：

app.component.ts　*crisis-center-home.component.ts*　*crisis-center.compon*

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

@Component({
  selector: 'app-root',
  template: `
    <h1 class="title">Angular Router</h1>
    <nav>
      <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
      <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
      <a routerLink="/admin" routerLinkActive="active">Admin</a>
      <a routerLink="/login" routerLinkActive="active">Login</a>
      <a [routerLink]="[{ outlets: { popup: ['compose'] } }]">Contact</a>
    </nav>
    <router-outlet></router-outlet>
    <router-outlet name="popup"></router-outlet>
  `
})
export class AppComponent {
}
```

auth-guard.service.ts　*can-deactivate-guard.service.ts*

```
import { Injectable }       from '@angular/core';
import {
  CanActivate, Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  CanActivateChild
}                           from '@angular/router';
import { AuthService }      from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

```

```
14.   canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
15.     let url: string = state.url;
16.
17.     return this.checkLogin(url);
18.   }
19.
20.   canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
21.     return this.canActivate(route, state);
22.   }
23.
24.   checkLogin(url: string): boolean {
25.     if (this.authService.isLoggedIn) { return true; }
26.
27.     // Store the attempted URL for redirecting
28.     this.authService.redirectUrl = url;
29.
30.     // Navigate to the login page
31.     this.router.navigate(['/login']);
32.     return false;
33.   }
34. }
```

查询参数及片段

在这个查询参数例子中，你只为路由指定了参数，但是该如何定义一些所有路由中都可用的可选参数呢？ 这就该“查询参数”登场了。

片段可以引用页面中带有特定 `id` 属性的元素.

修改 `AuthGuard` 以提供 `session_id` 查询参数，在导航到其它路由后，它还会存在。

再添加一个锚点（`A`）元素，来让你能跳转到页面中的正确位置。

为 `router.navigate` 方法添加一个 `NavigationExtras` 对象，用来导航到 `/login` 路由。

src/app/auth-guard.service.ts (v3)

```
import { Injectable }       from '@angular/core';
import {
  CanActivate, Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  CanActivateChild,
  NavigationExtras
}                           from '@angular/router';
import { AuthService }      from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean
{
    let url: string = state.url;

    return this.checkLogin(url);
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
boolean {
    return this.canActivate(route, state);
  }

  checkLogin(url: string): boolean {
    if (this.authService.isLoggedIn) { return true; }

    // Store the attempted URL for redirecting
    this.authService.redirectUrl = url;

    // Create a dummy session id
    let sessionId = 123456789;

    // Set our navigation extras object
    // that contains our global query params and fragment
    let navigationExtras: NavigationExtras = {
      queryParams: { 'session_id': sessionId },
      fragment: 'anchor'
    };
```

```
    // Navigate to the login page with extras
    this.router.navigate(['/login'], navigationExtras);
    return false;
  }
}
```

还可以再导航之间保留查询参数和片段，而无需再次再导航中提供。在 `LoginComponent` 中的 `router.navigate` 方法中，添加第二个参数，该对象提供了 `preserveQueryParams` 和 `preserveFragment`，用于传递到当前的查询参数中并为下一个路由提供片段。

src/app/login.component.ts (preserve)

```
// Set our navigation extras object
// that passes on our global query params and fragment
let navigationExtras: NavigationExtras = {
  queryParamsHandling: 'preserve',
  preserveFragment: true
};

// Redirect the user
this.router.navigate([redirect], navigationExtras);
```

> `queryParamsHandling` 特性还提供了 `merge` 选项，它将会在导航时保留当前的查询参数，并与其它查询参数合并。

由于要在登录后导航到**危机管理**特征区的路由，所以你还得修改它，来处理这些全局查询参数和片段。

src/app/admin/admin-dashboard.component.ts (v2)

```
import { Component, OnInit }  from '@angular/core';
import { ActivatedRoute }     from '@angular/router';
import { Observable }        from 'rxjs';
import { map }               from 'rxjs/operators';

@Component({
  template:  `
    <p>Dashboard</p>

    <p>Session ID: {{ sessionId | async }}</p>
    <a id="anchor"></a>
    <p>Token: {{ token | async }}</p>
  `
})
export class AdminDashboardComponent implements OnInit {
  sessionId: Observable<string>;
  token: Observable<string>;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    // Capture the session ID if available
    this.sessionId = this.route
      .queryParamMap
      .pipe(map(params => params.get('session_id') || 'None'));

    // Capture the fragment if available
    this.token = this.route
      .fragment
      .pipe(map(fragment => fragment || 'None'));
  }
}
```

查询参数和**片段**可通过 `Router` 服务的 `routerState` 属性使用。和**路由参数**类似，全局查询参数和片段也是 `Observable` 对象。 在修改过的**英雄管理**组件中，你将借助 `AsyncPipe` 直接把 `Observable` 传给模板。

按照下列步骤试验下：点击**Crisis Admin**按钮，它会带着你提供的 `queryParamMap` 和 `fragment` 跳转到登录页。 点击登录按钮，你就会被重定向到 `Admin Dashboard` 页。 注意，它仍然带着上一步提供的 `queryParamMap` 和 `fragment`。

你可以用这些持久化信息来携带需要为每个页面都提供的信息，如认证令牌或会话的 ID 等。

> “查询参数”和“片段”也可以分别用 `RouterLink` 中的 preserveQueryParams 和 preserveFragment 保存。

里程碑 6：异步路由

完成上面的里程碑后，应用程序很自然的长大了。在继续构建特征区的过程中，应用的尺寸将会变得更大。在某一个时间点，你将达到一个顶点，应用将会需要过多的时间来加载。

如何才能解决这个问题呢？通过引进异步路由，可以获得在请求时才惰性加载特性模块的能力。 惰性加载有多个优点：

- 你可以只在用户请求时才加载某些特性区。
- 对于那些只访问应用程序某些区域的用户，这样能加快加载速度。
- 你可以持续扩充惰性加载特性区的功能，而不用增加初始加载的包体积。

你已经完成了一部分。通过把应用组织成一些模块：`AppModule`、`HeroesModule`、`AdminModule` 和 `CrisisCenterModule`， 你已经有了可用于实现惰性加载的候选者。

有些模块（比如 `AppModule`）必须在启动时加载，但其它的都可以而且应该惰性加载。 比如 `AdminModule` 就只有少数已认证的用户才需要它，所以你应该只有在正确的人请求它时才加载。

惰性加载路由配置

把 `admin-routing.module.ts` 中的 `admin` 路径从 `'admin'` 改为空路径 `''`。

`Router` 支持**空路径**路由，可以使用它们来分组路由，而不用往 URL 中添加额外的路径片段。 用户仍旧访问 `/admin`，并且 `AdminComponent` 仍然作为用来包含子路由的**路由组件**。

打开 `AppRoutingModule`，并把一个新的 `admin` 路由添加到它的 `appRoutes` 数组中。

给它一个 `loadChildren` 属性（注意不是 `children` 属性），把它设置为 `AdminModule` 的地址。 该地址是 `AdminModule` 的文件路径（相对于 `app` 目录的），加上一个 `#` 分隔符，再加上导出模块的类名 `AdminModule`。

app-routing.module.ts (load children)

```
{
  path: 'admin',
  loadChildren: 'app/admin/admin.module#AdminModule',
},
```

当路由器导航到这个路由时，它会用 `loadChildren` 字符串来动态加载 `AdminModule`，然后把 `AdminModule` 添加到当前的路由配置中， 最后，它把所请求的路由加载到目标 `admin` 组件中。

惰性加载和重新配置工作只会发生一次，也就是在该路由**首次**被请求时。在后续的请求中，该模块和路由都是立即可用的。

> Angular 提供一个内置模块加载器，支持`SystemJS`来异步加载模块。如果你使用其它捆绑工具比如 Webpack，则使用 Webpack 的机制来异步加载模块。

最后一步是把管理特性区从主应用中完全分离开。 根模块 `AppModule` 既不能加载也不能引用 `AdminModule` 及其文件。

在 `app.module.ts` 中，从顶部移除 `AdminModule` 的导入语句，并且从 Angular 模块的 `imports` 数组中移除 `AdminModule`。

`CanLoad` 守卫：保护对特性模块的未授权加载

你已经使用 `CanActivate` 保护 `AdminModule` 了，它会阻止未授权用户访问管理特性区。如果用户未登录，它就会跳转到登录页。

但是路由器仍然会加载 `AdminModule` —— 即使用户无法访问它的任何一个组件。 理想的方式是，只有在用户已登录的情况下你才加载 `AdminModule`。

添加一个`CanLoad`守卫，它只在用户已登录**并且**尝试访问管理特性区的时候，才加载 `AdminModule` 一次。

现有的 `AuthGuard` 的 `checkLogin()` 方法中已经有了支持 `CanLoad` 守卫的基础逻辑。

打开 `auth-guard.service.ts`，从 `@angular/router` 中导入 `CanLoad` 接口。 把它添加到 `AuthGuard` 类的 `implements` 列表中。 然后实现 `canLoad`，代码如下：

src/app/auth-guard.service.ts (CanLoad guard)

```
canLoad(route: Route): boolean {
  let url = `/${route.path}`;

  return this.checkLogin(url);
}
```

路由器会把 `canLoad()` 方法的 `route` 参数设置为准备访问的目标 URL。 如果用户已经登录了，`checkLogin()` 方法就会重定向到那个 URL。

现在，把 `AuthGuard` 导入到 `AppRoutingModule` 中，并把 `AuthGuard` 添加到 `admin` 路由的 `canLoad` 数组中。完整的 `admin` 路由是这样的：

app-routing.module.ts (lazy admin route)

```
{
  path: 'admin',
  loadChildren: 'app/admin/admin.module#AdminModule',
  canLoad: [AuthGuard]
},
```

预加载：特性区的后台加载

你已经学会了如何按需加载模块，接下来再看看如何使用**预加载**技术异步加载模块。

看起来好像应用一直都是这么做的，但其实并非如此。 `AppModule` 在应用启动时就被加载了，它是**立即**加载的。 而 `AdminModule` 只有当用户点击某个链接时才会加载，它是**惰性**加载的。

预加载是介于两者之间的一种方式。 来看看**危机中心**。 用户第一眼不会看到它。 默认情况下，**英雄管理**才是第一视图。 为了获得尽可能小的初始加载体积和最快的加载速度，你应该对 `AppModule` 和 `HeroesModule` 进行立即加载。

你可以惰性加载**危机中心**。 但是，你几乎可以肯定用户会在启动应用之后的几分钟内访问**危机中心**。 理想情况下，应用启动时应该只加载 `AppModule` 和 `HeroesModule`，然后几乎立即开始后台加载 `CrisisCenterModule`。 在用户浏览到**危机中心**之前，该模块应该已经加载完毕，可供访问了。

这就是**预加载**。

预加载的工作原理

在每次**成功的**导航后，路由器会在自己的配置中查找尚未加载并且可以预加载的模块。 是否加载某个模块，以及要加载哪些模块，取决于**预加载策略**。

`Router` 内置了两种预加载策略：

- 完全不预加载，这是默认值。惰性加载的特性区仍然会按需加载。
- 预加载所有惰性加载的特性区。

默认情况下，路由器或者完全不预加载或者预加载每个惰性加载模块。 路由器还支持自定义预加载策略，以便完全控制要预加载哪些模块以及何时加载。

在下一节，你将会把 `CrisisCenterModule` 改为默认惰性加载的，并使用 `PreloadAllModules` 策略来尽快加载它（以及**所有其它**惰性加载模块）。

惰性加载**危机中心**

修改路由配置，来惰性加载 `CrisisCenterModule`。修改的步骤和配置惰性加载 `AdminModule` 时一样。

1. 把 `CrisisCenterRoutingModule` 中的路径从 `crisis-center` 改为空字符串。
2. 往 `AppRoutingModule` 中添加一个 `crisis-center` 路由。
3. 设置 `loadChildren` 字符串来加载 `CrisisCenterModule`。
4. 从 `app.module.ts` 中移除所有对 `CrisisCenterModule` 的引用。

下面是打开预加载之前的模块修改版：

app.module.ts *app-routing.module.ts* *crisis-center-routing.module.ts*

```
1. import { NgModule }       from '@angular/core';
2. import { BrowserModule }  from '@angular/platform-browser';
3. import { FormsModule }    from '@angular/forms';
4. import { BrowserAnimationsModule } from '@angular/platform-
   browser/animations';
5.
6. import { Router } from '@angular/router';
7.
8. import { AppComponent }            from './app.component';
9. import { AppRoutingModule }        from './app-routing.module';
10.
11. import { HeroesModule }            from './heroes/heroes.module';
12. import { ComposeMessageComponent } from './compose-message.component';
13. import { LoginRoutingModule }      from './login-routing.module';
14. import { LoginComponent }          from './login.component';
15. import { PageNotFoundComponent }   from './not-found.component';
16.
17. import { DialogService }           from './dialog.service';
18.
19. @NgModule({
20.   imports: [
21.     BrowserModule,
22.     FormsModule,
23.     HeroesModule,
24.     LoginRoutingModule,
25.     AppRoutingModule,
26.     BrowserAnimationsModule
27.   ],
28.   declarations: [
29.     AppComponent,
30.     ComposeMessageComponent,
31.     LoginComponent,
32.     PageNotFoundComponent
```

```
33.   ],
34.   providers: [
35.     DialogService
36.   ],
37.   bootstrap: [ AppComponent ]
38. })
39. export class AppModule {
40.   // Diagnostic only: inspect router configuration
41.   constructor(router: Router) {
42.     console.log('Routes: ', JSON.stringify(router.config, undefined, 2));
43.   }
44. }
```

你可以现在尝试它，并确认在点击了“Crisis Center”按钮之后加载了 `CrisisCenterModule`。

要为所有惰性加载模块启用预加载功能，请从 Angular 的路由模块中导入 `PreloadAllModules`。

`RouterModule.forRoot` 方法的第二个参数接受一个附加配置选项对象。 `preloadingStrategy` 就是其中之一。 把 `PreloadAllModules` 添加到 `forRoot` 调用中：

src/app/app-routing.module.ts (preload all)

```
RouterModule.forRoot(
  appRoutes,
  {
    enableTracing: true, // <-- debugging purposes only
    preloadingStrategy: PreloadAllModules
  }
)
```

这会让 `Router` 预加载器立即加载**所有**惰性加载路由（带 `loadChildren` 属性的路由）。

当访问 `http://localhost:3000` 时， `/heroes` 路由立即随之启动，并且路由器在加载了 `HeroesModule` 之后立即开始加载 `CrisisCenterModule`。

意外的是， `AdminModule`**没有**预加载，有什么东西阻塞了它。

CanLoad 会阻塞预加载

`PreloadAllModules` 策略不会加载被CanLoad守卫所保护的特性区。这是刻意设计的。

你几步之前刚刚给 `AdminModule` 中的路由添加了 `CanLoad` 守卫，以阻塞加载那个模块，直到用户认证结束。 `CanLoad` 守卫的优先级高于预加载策略。

如果你要加载一个模块**并且**保护它防止未授权访问，请移除 `canLoad` 守卫，只单独依赖CanActivate守卫。

自定义预加载策略

在大多数场景下，预加载每个惰性加载模块就很好了，但是有时候它却并不是正确的选择，特别是在移动设备和低带宽连接下。 你可能出于用户的测量和其它商业和技术因素而选择只对某些特性模块进行预加载。

使用自定义预加载策略，你可以控制路由器预加载哪些路由以及如何加载。

在这一节，你将添加一个自定义策略，它**只**预加载那些 `data.preload` 标志为 `true` 的路由。 回忆一下，你可以往路由的 `data` 属性中添加任何东西。

在 `AppRoutingModule` 的 `crisis-center` 路由中设置 `data.preload` 标志。

src/app/app-routing.module.ts (route data preload)

```
{
  path: 'crisis-center',
  loadChildren: 'app/crisis-center/crisis-center.module#CrisisCenterModule',
  data: { preload: true }
},
```

往项目中添加一个新的名叫 `selective-preloading-strategy.ts` 的文件，并在其中定义一个服务类 `SelectivePreloadingStrategy`，代码如下：

src/app/selective-preloading-strategy.ts (excerpt)

```
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable, of } from 'rxjs';

@Injectable()
export class SelectivePreloadingStrategy implements PreloadingStrategy {
  preloadedModules: string[] = [];

  preload(route: Route, load: () => Observable<any>): Observable<any> {
    if (route.data && route.data['preload']) {
      // add the route path to the preloaded module array
      this.preloadedModules.push(route.path);

      // log the route path to the console
      console.log('Preloaded: ' + route.path);

      return load();
    } else {
      return of(null);
    }
  }
}
```

`SelectivePreloadingStrategy` 实现了 `PreloadingStrategy`，它只有一个方法 `preload`。

路由器会用两个参数调用调用 `preload` 方法：

1. 要加载的路由。
2. 一个加载器（loader）函数，它能异步加载带路由的模块。

`preload` 的实现必须返回一个 `Observable`。 如果该路由应该预加载，它就会返回调用加载器函数所返回的 `Observable`。 如果该路由由**不**应该预加载，它就返回一个 `null` 值的 `Observable` 对象。

在这个例子中，`preload` 方法只有在路由的 `data.preload` 标识为真时才会加载该路由。

它还有一个副作用。 `SelectivePreloadingStrategy` 会把所选路由的 `path` 记录在它的公共数组 `preloadedModules` 中。

很快，你就会扩展 `AdminDashboardComponent` 来注入该服务，并且显示它的 `preloadedModules` 数组。

但是首先，要对 `AppRoutingModule` 做少量修改。

1. 把 `SelectivePreloadingStrategy` 导入到 `AppRoutingModule` 中。
2. 把 `PreloadAllModules` 策略替换成对 `forRoot` 的调用，并且传入这个 `SelectivePreloadingStrategy`。
3. 把 `SelectivePreloadingStrategy` 策略添加到 `AppRoutingModule` 的 `providers` 数组中，以便它可以注入到应用中的任何地方。

现在，编辑 `AdminDashboardComponent` 以显示这些预加载路由的日志。

1. 导入 `SelectivePreloadingStrategy`（它是一个服务）。
2. 把它注入到仪表盘的构造函数中。
3. 修改模板来显示这个策略服务的 `preloadedModules` 数组。

当完成时，代码如下：

src/app/admin/admin-dashboard.component.ts (preloaded modules)

```
import { Component, OnInit }     from '@angular/core';
import { ActivatedRoute }        from '@angular/router';
import { Observable }           from 'rxjs';
import { map }                  from 'rxjs/operators';

import { SelectivePreloadingStrategy } from '../selective-preloading-strategy';


@Component({
  template:  `
    <p>Dashboard</p>

    <p>Session ID: {{ sessionId | async }}</p>
    <a id="anchor"></a>
    <p>Token: {{ token | async }}</p>

    Preloaded Modules
    <ul>
      <li *ngFor="let module of modules">{{ module }}</li>
    </ul>
  `
})
export class AdminDashboardComponent implements OnInit {
  sessionId: Observable<string>;
  token: Observable<string>;
  modules: string[];

  constructor(
    private route: ActivatedRoute,
    private preloadStrategy: SelectivePreloadingStrategy
  ) {
    this.modules = preloadStrategy.preloadedModules;
  }

  ngOnInit() {
    // Capture the session ID if available
    this.sessionId = this.route
      .queryParamMap
      .pipe(map(params => params.get('session_id') || 'None'));

    // Capture the fragment if available
```

```
      this.token = this.route
        .fragment
        .pipe(map(fragment => fragment || 'None'));
    }
  }
```

一旦应用加载完了初始路由，`CrisisCenterModule` 也被预加载了。 通过 `Admin` 特性区中的记录就可以验证它，“Preloaded Modules”中没有列出 `crisis-center`。 它也被记录到了浏览器的控制台。

使用重定向迁移 URL

你已经设置好了路由，并且用命令式和声明式的方式导航到了很多不同的路由。但是，任何应用的需求都会随着时间而改变。 你把链接 `/heroes` 和 `hero/:id` 指向了 `HeroListComponent` 和 `HeroDetailComponent` 组件。 如果有这样一个需求，要把链接 `heroes` 变成 `superheroes`，你仍然希望以前的 URL 能正常导航。 但你也不想在应用中找到并修改每一个链接，这时候，重定向就可以省去这些琐碎的重构工作。

把 `/heroes` 修改为 `/superheros`

先取得 `Hero` 路由，并把它们迁移到新的 URL。`Router`（路由器）会在开始导航之前先在配置中检查所有重定向语句，以便将来按需触发重定向。要支持这种修改，你就要在 `heroes-routing.module` 文件中把老的路由重定向到新的路由。

src/app/heroes/heroes-routing.module.ts (heroes redirects)

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

import { HeroListComponent }    from './hero-list.component';
import { HeroDetailComponent }  from './hero-detail.component';

const heroesRoutes: Routes = [
  { path: 'heroes', redirectTo: '/superheroes' },
  { path: 'hero/:id', redirectTo: '/superhero/:id' },
  { path: 'superheroes',  component: HeroListComponent },
  { path: 'superhero/:id', component: HeroDetailComponent }
];

@NgModule({
  imports: [
    RouterModule.forChild(heroesRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class HeroRoutingModule { }
```

注意，这里有两种类型的重定向。第一种是不带参数的从 `/heroes` 重定向到 `/superheroes`。这是一种非常直观的重定向。第二种是从 `/hero/:id` 重定向到 `/superhero/:id`，它还要包含一个 `:id` 路由参数。 路由器重定向时使用强大的模式匹配功能，这样，路由器就会检查 URL，并且把 `path` 中带的路由参数替换成相应的目标形式。以前，你导航到形如 `/hero/15` 的 URL 时，带了一个路由参数 `id`，它的值是 `15`。

> 在重定向的时候，路由器还支持查询参数和片段(fragment)。
>
> - 当使用绝对地址重定向时，路由器将会使用路由配置的 `redirectTo` 属性中规定的查询参数和片段。
> - 当使用相对地址重定向时，路由器将会使用源地址（跳转前的地址）中的查询参数和片段。

在修改 `app-routing.module.ts` 之前，你要先考虑一条重要的规则。 目前，你把空路径路由重定向到了 `/heroes`，它又被重定向到了 `/superheroes`。这样**不行**，从设计上就不行。 因为路由器在每一层的路由配置中只会处理一次重定向。这样可以防止出现无限循环的重定向。

所以，你要在 `app-routing.module.ts` 中修改空路径路由，让它重定向到 `/superheroes`。

src/app/app-routing.module.ts (superheroes redirect)

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

import { ComposeMessageComponent }  from './compose-message.component';
import { PageNotFoundComponent }    from './not-found.component';

import { CanDeactivateGuard }       from './can-deactivate-guard.service';
import { AuthGuard }                from './auth-guard.service';
import { SelectivePreloadingStrategy } from './selective-preloading-strategy';

const appRoutes: Routes = [
  {
    path: 'compose',
    component: ComposeMessageComponent,
    outlet: 'popup'
  },
  {
    path: 'admin',
    loadChildren: 'app/admin/admin.module#AdminModule',
    canLoad: [AuthGuard]
  },
  {
    path: 'crisis-center',
    loadChildren: 'app/crisis-center/crisis-center.module#CrisisCenterModule',
    data: { preload: true }
  },
  { path: '',   redirectTo: '/superheroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      {
        enableTracing: true, // <-- debugging purposes only
        preloadingStrategy: SelectivePreloadingStrategy,

      }
    )
  ],
```

```
    exports: [
      RouterModule
    ],
    providers: [
      CanDeactivateGuard,
      SelectivePreloadingStrategy
    ]
  })
  export class AppRoutingModule { }
```

由于 `RouterLink` 指令没有关联到路由配置，所以你需要修改相关的路由链接，以便在新的路由激活时，它们也能保持激活状态。你要修改 `app.component.ts` 模板中的 `/heroes` 路由链接。

src/app/app.component.ts (superheroes active routerLink)

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

@Component({
  selector: 'app-root',
  template: `
    <h1 class="title">Angular Router</h1>
    <nav>
      <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
      <a routerLink="/superheroes" routerLinkActive="active">Heroes</a>
      <a routerLink="/admin" routerLinkActive="active">Admin</a>
      <a routerLink="/login" routerLinkActive="active">Login</a>
      <a [routerLink]="[{ outlets: { popup: ['compose'] } }]">Contact</a>
    </nav>
    <router-outlet></router-outlet>
    <router-outlet name="popup"></router-outlet>
  `
})
export class AppComponent {
}
```

当这些重定向设置好之后，所有以前的路由都指向了它们的新目标，并且每个 URL 也仍然能正常工作。

审查路由器配置

你把大量的精力投入到在一系列路由模块文件里配置路由器上，并且小心的以合适的顺序列出它们。 这些路由是否真的如同你预想的那样执行了？ 路由器的真实配置是怎样的？

通过注入它（Router）并检查它的 `config` 属性，你可以随时审查路由器的当前配置。 例如，把 `AppModule` 修改为这样，并在浏览器的控制台窗口中查看最终的路由配置。

src/app/app.module.ts (inspect the router config)

```
import { Router } from '@angular/router';

export class AppModule {
  // Diagnostic only: inspect router configuration
  constructor(router: Router) {
    console.log('Routes: ', JSON.stringify(router.config, undefined, 2));
  }
}
```

总结与最终的应用

本章中涉及到了很多背景知识，而且本应用程序也太大了，所以没法在这里显示。请访问Router Sample in Stackblitz / 下载范例，在那里你可以下载最终的源码。

附录

本章剩下的部分是一组附录，它详尽阐述了那些曾匆匆带过的知识点。

该附件中的内容不是必须的，感兴趣的人才需要阅读它。

附录：链接参数数组

链接参数数组保存路由导航时所需的成分：

- 指向目标组件的那个路由的**路径（path）**
- 必备路由参数和可选路由参数，它们将进入该路由的 URL

你可以把 `RouterLink` 指令绑定到一个数组，就像这样：

src/app/app.component.ts (h-anchor)

```
<a [routerLink]="['/heroes']">Heroes</a>
```

在指定路由参数时，你写过一个双元素的数组，就像这样：

src/app/heroes/hero-list.component.ts (nav-to-detail)

```
<a [routerLink]="['/hero', hero.id]">
  <span class="badge">{{ hero.id }}</span>{{ hero.name }}
</a>
```

你可以在对象中提供可选的路由参数，就像这样：

src/app/app.component.ts (cc-query-params)

```
<a [routerLink]="['/crisis-center', { foo: 'foo' }]">Crisis Center</a>
```

这三个例子涵盖了你在单级路由的应用中所需的一切。在你添加一个像**危机中心**一样的子路由时，你可以创建新链接数组。

回忆一下，你曾为**危机中心**指定过一个默认的子路由，以便能使用这种简单的 `RouterLink`。

src/app/app.component.ts (cc-anchor-w-default)

```
<a [routerLink]="['/crisis-center']">Crisis Center</a>
```

分解一下。

- 数组中的第一个条目标记出了父路由(`/crisis-center`)。
- 这个父路由没有参数，因此这步已经完成了。
- 没有默认的子路由，因此你得选取一个。
- 你决定跳转到 `CrisisListComponent`，它的路由路径是'/'，但你不用显式的添加它。
- 哇！`['/crisis-center']`。

更进一步。这次要构建一个从根组件往下导航到“巨龙危机”时的链接参数数组：

src/app/app.component.ts (Dragon-anchor)

```
<a [routerLink]="['/crisis-center', 1]">Dragon Crisis</a>
```

- 数组中的第一个条目标记出了父路由(`/crisis-center`)。
- 这个父路由没有参数，因此这步已经完成了。
- 数组中的第二个条目（'/:id'）用来标记出到指定危机的详情页的子路由。
- 详细的子路由需要一个 `id` 路由参数。
- 你把**巨龙危机**的 `id` 添加为该数组中的第二个条目（`1`）。
- 最终生成的路径是 `/crisis-center/1`。

只要想，你也可以用**危机中心**路由单独重定义 `AppComponent` 的模板：

src/app/app.component.ts (template)

```
template: `
  <h1 class="title">Angular Router</h1>
  <nav>
    <a [routerLink]="['/crisis-center']">Crisis Center</a>
    <a [routerLink]="['/crisis-center/1', { foo: 'foo' }]">Dragon Crisis</a>
    <a [routerLink]="['/crisis-center/2']">Shark Crisis</a>
  </nav>
  <router-outlet></router-outlet>
`
```

总结：你可以用一级、两级或多级路由来写应用程序。 链接参数数组提供了用来表示任意深度路由的链接参数数组以及任意合法的路由参数序列、必须的路由器参数以及可选的路由参数对象。

附录：LocationStrategy 以及浏览器 URL 样式

当路由器导航到一个新的组件视图时，它会用该视图的 URL 来更新浏览器的当前地址以及历史。 严格来说，这个 URL 其实是本地的，浏览器不会把该 URL 发给服务器，并且不会重新加载此页面。

现代 HTML 5 浏览器支持history.pushState API， 这是一项可以改变浏览器的当前地址和历史，却又不会触发服务端页面请求的技术。 路由器可以合成出一个“自然的”URL，它看起来和那些需要进行页面加载的 URL 没什么区别。

下面是**危机中心**的 URL 在“HTML 5 pushState”风格下的样子：

```
localhost:3002/crisis-center/
```

老旧的浏览器在当前地址的 URL 变化时总会往服务器发送页面请求……唯一的例外规则是：当这些变化位于“#”（被称为“hash”）后面时不会发送。通过把应用内的路由 URL 拼接在 `#` 之后，路由器可以获得这条“例外规则”带来的优点。下面是到**危机中心**路由的“hash URL”：

```
localhost:3002/src/#/crisis-center/
```

路由器通过两种 `LocationStrategy` 提供商来支持所有这些风格：

1. `PathLocationStrategy` - 默认的策略，支持“HTML 5 pushState”风格。
2. `HashLocationStrategy` - 支持“hash URL”风格。

`RouterModule.forRoot` 函数把 `LocationStrategy` 设置成了 `PathLocationStrategy`，使其成为了默认策略。 你可以在启动过程中改写（override）它，来切换到 `HashLocationStrategy` 风格 —— 如果你更喜欢这种。

> 要学习关于“提供商”和启动过程的更多知识，参见依赖注入一章。

哪种策略更好?

你必须选择一种策略，并且在项目的早期就这么干。一旦该应用进入了生产阶段，要改起来可就不容易了，因为外面已经有了大量对应用 URL 的引用。

几乎所有的 Angular 项目都会使用默认的 HTML 5 风格。它生成的 URL 更易于被用户理解，它也为将来做服务端渲染预留了空间。

在服务器端渲染指定的页面，是一项可以在该应用首次加载时大幅提升响应速度的技术。那些原本需要十秒甚至更长时间加载的应用，可以预先在服务端渲染好，并在少于一秒的时间内完整呈现在用户的设备上。

只有当应用的 URL 看起来像是标准的 Web URL，中间没有 hash（#）时，这个选项才能生效。

除非你有强烈的理由不得不使用 hash 路由，否则就应该坚决使用默认的 HTML 5 路由风格。

HTML 5 URL 与**<base href>**

由于路由器默认使用“HTML 5 pushState”风格，所以你**必须**用一个base href来配置该策略（Strategy）。

配置该策略的首选方式是往 `index.html` 的 `<head>` 中添加一个<base href> element标签。

src/index.html (base-href)

```
<base href="/">
```

如果没有此标签，当通过“深链接”进入该应用时，浏览器就不能加载资源（图片、CSS、脚本）。如果有人把应用的链接粘贴进浏览器的地址栏或从邮件中点击应用的链接时，这种问题就发生。

有些开发人员可能无法添加 `<base>` 元素，这可能是因为它们没有访问 `<head>` 或 `index.html` 的权限。

它们仍然可以使用 HTML 5 格式的 URL，但要采取两个步骤进行补救：

1. 用适当的APP_BASE_HREF值提供（provide）路由器。
2. 对所有 Web 资源使用绝对地址：CSS、图片、脚本、模板 HTML。

HashLocationStrategy 策略

你可以在根模块的 `RouterModule.forRoot` 的第二个参数中传入一个带有 `useHash: true` 的对象，以回到基于 `HashLocationStrategy` 的传统方式。

src/app/app.module.ts (hash URL strategy)

```
import { NgModule }             from '@angular/core';
import { BrowserModule }        from '@angular/platform-browser';
import { FormsModule }          from '@angular/forms';
import { Routes, RouterModule } from '@angular/router';

import { AppComponent }          from './app.component';
import { PageNotFoundComponent } from './not-found.component';

const routes: Routes = [

];

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(routes, { useHash: true })  // .../#/crisis-center/
  ],
  declarations: [
    AppComponent,
    PageNotFoundComponent
  ],
  providers: [

  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

测试

该指南提供了对 Angular 应用进行单元测试和集成测试的技巧和提示。

该指南中的测试面向的是一个很像《英雄指南》教程的 CLI 范例应用。 这个范例应用及其所有测试都可以在下面的链接中进行查看和试用：

- 在线例子 / 下载范例
- 在线例子 / 下载范例

准备工作

Angular CLI 会下载并安装试用 Jasmine 测试框架 测试 Angular 应用时所需的一切。

你使用 CLI 创建的项目是可以立即用于测试的。 运行下列 CLI 命令即可：

```
ng test
```

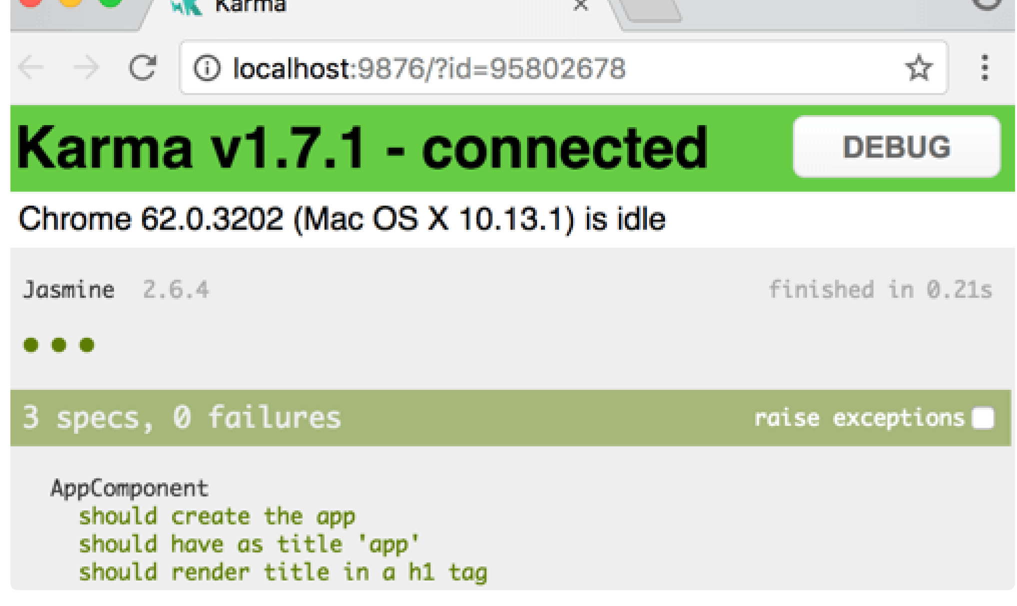
`ng test` 命令在**监视模式**下构建应用，并启动 karma 测试运行器。

它的控制台输出一般是这样的：

```
10% building modules 1/1 modules 0 active
...INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876/
...INFO [launcher]: Launching browser Chrome ...
...INFO [launcher]: Starting browser Chrome
...INFO [Chrome ...]: Connected on socket ...
Chrome ...: Executed 3 of 3 SUCCESS (0.135 secs / 0.205 secs)
```

最后一行很重要。它表示 Karma 运行了三个测试，而且这些测试都通过了。

它还会打开 Chrome 浏览器并在“ Jasmine HTML 报告器”中显示测试输出，就像这样：



大多数人都会觉得浏览器中的报告比控制台中的日志更容易阅读。 你可以点击一行测试，来单独重跑这个测试，或者点击一行描述信息来重跑所选测试组（“测试套件”）中的那些测试。

同时，`ng test` 命令还会监听这些变化。

要查看它的实际效果，就对 `app.component.ts` 做一个小修改，并保存它。 这些测试就会重新运行，浏览器也会刷新，然后新的测试结果就出现了。

配置

CLI 会为你生成 Jasmine 和 Karma 的配置文件。

不过你也可以通过编辑 `src/` 目录下的 `karma.conf.js` 和 `test.ts` 文件来微调很多选项。

`karma.conf.js` 文件是 karma 配置文件的一部分。 CLI 会基于 `angular.json` 文件中指定的项目结构和 `karma.conf.js` 文件，来在内存中构建出完整的运行时配置。

要进一步了解 Jasmine 和 Karma 的配置项，请搜索网络。

其它测试框架

你还可以使用其它的测试库和测试运行器来对 Angular 应用进行单元测试。 每个库和运行器都有自己特有的安装过程、配置项和语法。

要了解更多，请搜索网络。

测试文件名及其位置

查看 `src/app` 文件夹。

CLI 为 `AppComponent` 生成了一个名叫 `app.component.spec.ts` 的测试文件。

> 测试文件的扩展名必须是 `.spec.ts`，这样工具才能识别出它是一个测试文件，也叫规约（spec）文件。

`app.component.ts` 和 `app.component.spec.ts` 文件位于同一个文件夹中，而且相邻。 其根文件名部分（`app.component`）都是一样的。

请在你的项目中对**任意类型**的测试文件都坚持这两条约定。

对服务的测试

服务通常是单元测试中最简单的文件类型。 下面是一些针对 `ValueService` 的同步和异步单元测试， 编写它们时没有借助来自 Angular 测试工具集的任何协助。

app/demo/demo.spec.ts

```
1. // Straight Jasmine testing without Angular's testing support
2. describe('ValueService', () => {
3.   let service: ValueService;
4.   beforeEach(() => { service = new ValueService(); });
5.
6.   it('#getValue should return real value', () => {
7.     expect(service.getValue()).toBe('real value');
8.   });
9.
10.   it('#getObservableValue should return value from observable',
11.     (done: DoneFn) => {
12.     service.getObservableValue().subscribe(value => {
13.       expect(value).toBe('observable value');
14.       done();
15.     });
16.   });
17.
18.   it('#getPromiseValue should return value from a promise',
19.     (done: DoneFn) => {
20.     service.getPromiseValue().then(value => {
21.       expect(value).toBe('promise value');
22.       done();
23.     });
24.   });
25. });
```

带有依赖的服务

服务通常会依赖于一些 Angular 注入到其构造函数中的其它服务。 多数情况下，创建并在调用该服务的构造函数时，手工创建并注入这些依赖也是很容易的。

`MasterService` 就是一个简单的例子：

app/demo/demo.ts

```
@Injectable()
export class MasterService {
  constructor(private valueService: ValueService) { }
  getValue() { return this.valueService.getValue(); }
}
```

`MasterService` 把它唯一的方法 `getValue` 委托给了注入进来的 `ValueService`。

这里是几种测试它的方法。

app/demo/demo.spec.ts

```
describe('MasterService without Angular testing support', () => {
  let masterService: MasterService;

  it('#getValue should return real value from the real service', () => {
    masterService = new MasterService(new ValueService());
    expect(masterService.getValue()).toBe('real value');
  });

  it('#getValue should return faked value from a fakeService', () => {
    masterService = new MasterService(new FakeValueService());
    expect(masterService.getValue()).toBe('faked service value');
  });

  it('#getValue should return faked value from a fake object', () => {
    const fake =  { getValue: () => 'fake value' };
    masterService = new MasterService(fake as ValueService);
    expect(masterService.getValue()).toBe('fake value');
  });

  it('#getValue should return stubbed value from a spy', () => {
    // create `getValue` spy on an object representing the ValueService
    const valueServiceSpy =
      jasmine.createSpyObj('ValueService', ['getValue']);

    // set the value to return when the `getValue` spy is called.
    const stubValue = 'stub value';
    valueServiceSpy.getValue.and.returnValue(stubValue);

    masterService = new MasterService(valueServiceSpy);

    expect(masterService.getValue())
      .toBe(stubValue, 'service returned stub value');
    expect(valueServiceSpy.getValue.calls.count())
      .toBe(1, 'spy method was called once');
    expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)
      .toBe(stubValue);
  });
});
```

第一个测试使用 `new` 创建了 `ValueService`，然后把它传给了 `MasterService` 的构造函数。

不过，对于大多数没这么容易创建和控制的依赖项来说，注入真实的服务很容易出问题。

你可以改用模拟依赖的方式，你可以使用虚值或在相关的服务方法上创建一个间谍（spy）。

> 优先使用间谍，因为它们通常是 Mock 服务时最简单的方式。

这些标准的测试技巧对于在隔离的环境下对服务进行单元测试非常重要。

不过，你几乎迟早要用 Angular 的依赖注入机制来把服务注入到应用类中去，而且你应该已经有了这类测试。 Angular 的测试工具集可以让你轻松探查这种注入服务的工作方式。

使用 `TestBed`（测试机床）测试服务

你的应用中会依赖 Angular 的依赖注入 (DI) 来创建服务。 当某个服务依赖另一个服务时，DI 就会找到或创建那个被依赖的服务。 如果那个被依赖的服务还有它自己的依赖，DI 也同样会找到或创建它们。

作为服务的**消费方**，你不需要关心这些细节。 你不用关心构造函数中的参数顺序或如何创建它们。

但对于服务的**测试方**来说，你就至少要考虑服务的第一级依赖了。 不过你**可以**让 Angular DI 来负责服务的创建工作，但当你使用 `TestBed` 测试工具来提供和创建服务时，你仍然需要关心构造函数中的参数顺序。

Angular **TestBed**

`TestBed` 是 Angular 测试工具中最重要的部分。 `TestBed` 会动态创建一个用来模拟 @NgModule 的 Angular **测试**模块。

`TestBed.configureTestingModule()` 方法接收一个元数据对象，其中具有 @NgModule 中的绝大多数属性。

要测试某个服务，就要在元数据的 `providers` 属性中指定一个将要进行测试或模拟的相关服务的数组。

app/demo/demo.testbed.spec.ts (provide ValueService in beforeEach

```
let service: ValueService;

beforeEach(() => {
  TestBed.configureTestingModule({ providers: [ValueService] });
});
```

然后通过调用 `TestBed.get()`（参数为该服务类）把它注入到一个测试中。

```
it('should use ValueService', () => {
  service = TestBed.get(ValueService);
  expect(service.getValue()).toBe('real value');
});
```

或者，如果你更倾向于把该服务作为环境准备过程的一部分，就把它放在 `beforeEach()` 中。

```
beforeEach(() => {
  TestBed.configureTestingModule({ providers: [ValueService] });
  service = TestBed.get(ValueService);
});
```

如果要测试一个带有依赖项的服务，那就把模拟对象放在 `providers` 数组中。

在下面的例子中，模拟对象是一个间谍（spy）对象。

```
let masterService: MasterService;
let valueServiceSpy: jasmine.SpyObj<ValueService>;

beforeEach(() => {
  const spy = jasmine.createSpyObj('ValueService', ['getValue']);

  TestBed.configureTestingModule({
    // Provide both the service-to-test and its (spy) dependency
    providers: [
      MasterService,
      { provide: ValueService, useValue: spy }
    ]
  });
  // Inject both the service-to-test and its (spy) dependency
  masterService = TestBed.get(MasterService);
  valueServiceSpy = TestBed.get(ValueService);
});
```

该测试会像以前一样消费这个间谍对象。

```
it('#getValue should return stubbed value from a spy', () => {
  const stubValue = 'stub value';
  valueServiceSpy.getValue.and.returnValue(stubValue);

  expect(masterService.getValue())
    .toBe(stubValue, 'service returned stub value');
  expect(valueServiceSpy.getValue.calls.count())
    .toBe(1, 'spy method was called once');
  expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)
    .toBe(stubValue);
});
```

不使用 `beforeEach` 进行测试

本指南中的大多数的测试套件都会调用 `beforeEach()` 来为每个 `it()` 测试准备前置条件，并依赖 `TestBed` 来创建类和注入服务。

另一些测试教程中也可能让你不要调用 `beforeEach()`，并且更倾向于显式创建类，而不要借助 `TestBed`。

下面的例子教你如何把 `MasterService` 的测试改写成那种风格。

通过把可复用的准备代码放进一个单独的 `setup` 函数来代替 `beforeEach()`。

app/demo/demo.spec.ts (setup)

```
function setup() {
  const valueServiceSpy =
    jasmine.createSpyObj('ValueService', ['getValue']);
  const stubValue = 'stub value';
  const masterService = new MasterService(valueServiceSpy);

  valueServiceSpy.getValue.and.returnValue(stubValue);
  return { masterService, stubValue, valueServiceSpy };
}
```

`setup()` 函数返回一个带有一些变量的对象字面量，比如 `masterService`，测试中可以引用它。 这样你就不用在 `describe()` 中定义一些**半全局性的**变量了（比如 `let masterService: MasterService` ）。

然后，每个测试都会在第一行调用 `setup()`，然后再操纵被测主体以及对期望值进行断言。

```
it('#getValue should return stubbed value from a spy', () => {
  const { masterService, stubValue, valueServiceSpy } = setup();
  expect(masterService.getValue())
    .toBe(stubValue, 'service returned stub value');
  expect(valueServiceSpy.getValue.calls.count())
    .toBe(1, 'spy method was called once');
  expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)
    .toBe(stubValue);
});
```

注意这些测试是如何使用 解构赋值 来提取出所需变量的。

```
const { masterService, stubValue, valueServiceSpy } = setup();
```

很多开发者觉得这种方式相比传统的 `beforeEach()` 风格更加干净、更加明确。

虽然本章会遵循传统的风格，并且 CLI 生成的默认测试文件也用的是 `beforeEach()` 和 `TestBed`，不过你可以在自己的项目中自由选择**这种可选方式。**

测试 HTTP 服务

那些会向远端服务器发起 HTTP 调用的数据服务，通常会注入 Angular 的 `HttpClient` 服务并委托它进行 XHR 调用。

你可以像测试其它带依赖的服务那样，通过注入一个 `HttpClient` 间谍来测试这种数据服务。

app/model/hero.service.spec.ts (tests with spies)

```
let httpClientSpy: { get: jasmine.Spy };
let heroService: HeroService;

beforeEach(() => {
  // TODO: spy on other methods too
  httpClientSpy = jasmine.createSpyObj('HttpClient', ['get']);
  heroService = new HeroService(<any> httpClientSpy);
});

it('should return expected heroes (HttpClient called once)', () => {
  const expectedHeroes: Hero[] =
    [{ id: 1, name: 'A' }, { id: 2, name: 'B' }];

  httpClientSpy.get.and.returnValue(asyncData(expectedHeroes));

  heroService.getHeroes().subscribe(
    heroes => expect(heroes).toEqual(expectedHeroes, 'expected heroes'),
    fail
  );
  expect(httpClientSpy.get.calls.count()).toBe(1, 'one call');
});

it('should return an error when the server returns a 404', () => {
  const errorResponse = new HttpErrorResponse({
    error: 'test 404 error',
    status: 404, statusText: 'Not Found'
  });

  httpClientSpy.get.and.returnValue(asyncError(errorResponse));

  heroService.getHeroes().subscribe(
    heroes => fail('expected an error, not heroes'),
    error  => expect(error.message).toContain('test 404 error')
  );
});
```

> `HttpService` 中的方法会返回 `Observables`。**订阅**这些方法返回的可观察对象会让它开始执行，并且断言这些方法是成功了还是失败了。
>
> `subscribe()` 方法接受一个成功回调 (`next`) 和一个失败 (`error`) 回调。 你要确保同时提供了这两个回调，以便捕获错误。 如果忽略这些异步调用中未捕获的错误，测试运行器可能会得出截然不同的测试结论。

HttpClientTestingModule

如果将来 `HttpClient` 和数据服务之间有更多的交互，则可能会变得复杂，而且难以使用间谍进行模拟。

`HttpClientTestingModule` 可以让这些测试场景变得更加可控。

本章的**代码范例**要示范的是 `HttpClientTestingModule`，所以把部分内容移到了 HttpClient 中，那里会详细讲解如何用 `HttpClientTestingModule` 进行测试。

> 本章范例代码中的 `app/model/http-hero.service.spec.ts` 还示范了如何使用**传统的** `HttpModule` 进行验证。

组件测试基础

组件与 Angular 应用中的其它部分不同，它是由 HTML 模板和 TypeScript 类组成的。 组件其实是指模板加上与其合作的类。 要想对组件进行充分的测试，就要测试它们能否如预期的那样协作。

这些测试需要在浏览器的 DOM 中创建组件的宿主元素（就像 Angular 所做的那样），然后检查组件类和 DOM 的交互是否如同它在模板中所描述的那样。

Angular 的 `TestBed` 为所有这些类型的测试提供了基础设施。 但是很多情况下，可以**单独测试组件类本身**而不必涉及 DOM，就已经可以用一种更加简单、清晰的方式来验证该组件的大多数行为了。

单独测试组件类

你可以像测试服务类一样测试组件类。

考虑下面这个 `LightswitchComponent`，当用户点击按钮时，它会切换灯的开关状态（用屏幕上的消息展现出来）。

app/demo/demo.ts (LightswitchComp)

```
@Component({
  selector: 'lightswitch-comp',
  template: `
    <button (click)="clicked()">Click me!</button>
    <span>{{message}}</span>`
})
export class LightswitchComponent {
  isOn = false;
  clicked() { this.isOn = !this.isOn; }
  get message() { return `The light is ${this.isOn ? 'On' : 'Off'}`; }
}
```

你可能要测试 `clicked()` 方法能否正确切换灯的开关状态。

该组件类没有依赖。 要测试一个没有依赖的服务，你会用 `new` 来创建它，调用它的 API，然后对它的公开状态进行断言。 组件类也可以这么做。

app/demo/demo.spec.ts (Lightswitch tests)

```
describe('LightswitchComp', () => {
  it('#clicked() should toggle #isOn', () => {
    const comp = new LightswitchComponent();
    expect(comp.isOn).toBe(false, 'off at first');
    comp.clicked();
    expect(comp.isOn).toBe(true, 'on after click');
    comp.clicked();
    expect(comp.isOn).toBe(false, 'off after second click');
  });

  it('#clicked() should set #message to "is on"', () => {
    const comp = new LightswitchComponent();
    expect(comp.message).toMatch(/is off/i, 'off at first');
    comp.clicked();
    expect(comp.message).toMatch(/is on/i, 'on after clicked');
  });
});
```

下面这段代码是来自《英雄指南》教程的 `DashboardHeroComponent`。

app/dashboard/dashboard-hero.component.ts (component)

```
export class DashboardHeroComponent {
  @Input() hero: Hero;
  @Output() selected = new EventEmitter<Hero>();
  click() { this.selected.emit(this.hero); }
}
```

它呈现在父组件的模板中，那里把一个英雄绑定到了 `@Input` 属性上，并且通过 `@Output` 属性监听**选中英雄**时的事件。

你可以测试 `DashboardHeroComponent` 类，而不用完整创建它或其父组件。

app/dashboard/dashboard-hero.component.spec.ts (class tests)

```
it('raises the selected event when clicked', () => {
  const comp = new DashboardHeroComponent();
  const hero: Hero = { id: 42, name: 'Test' };
  comp.hero = hero;

  comp.selected.subscribe(selectedHero => expect(selectedHero).toBe(hero));
  comp.click();
});
```

当组件有依赖时，你可能要使用 `TestBed` 来同时创建该组件及其依赖。

下面的 `WelcomeComponent` 依赖于 `UserService`，并通过它知道要打招呼的那位用户的名字。

app/welcome/welcome.component.ts

```
export class WelcomeComponent  implements OnInit {
  welcome: string;
  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.welcome = this.userService.isLoggedIn ?
      'Welcome, ' + this.userService.user.name : 'Please log in.';
  }
}
```

你可能要先创建一个满足本组件最小需求的模拟板 `UserService`。

app/welcome/welcome.component.spec.ts (MockUserService)

```
class MockUserService {
  isLoggedIn = true;
  user = { name: 'Test User'};
};
```

然后在 `TestBed` 的配置中提供并注入该组件和该服务。

app/welcome/welcome.component.spec.ts (class-only setup)

```
beforeEach(() => {
  TestBed.configureTestingModule({
    // provide the component-under-test and dependent service
    providers: [
      WelcomeComponent,
      { provide: UserService, useClass: MockUserService }
    ]
  });
  // inject both the component and the dependent service.
  comp = TestBed.get(WelcomeComponent);
  userService = TestBed.get(UserService);
});
```

然后使用这个组件类，别忘了像 Angular 运行本应用时那样调用它的生命周期钩子方法。

app/welcome/welcome.component.spec.ts (class-only tests)

```
it('should not have welcome message after construction', () => {
  expect(comp.welcome).toBeUndefined();
});

it('should welcome logged in user after Angular calls ngOnInit', () => {
  comp.ngOnInit();
  expect(comp.welcome).toContain(userService.user.name);
});

it('should ask user to log in if not logged in after ngOnInit', () => {
  userService.isLoggedIn = false;
  comp.ngOnInit();
  expect(comp.welcome).not.toContain(userService.user.name);
  expect(comp.welcome).toContain('log in');
});
```

组件 DOM 的测试

测试组件**类**就像测试服务那样简单。

但组件不仅是这个类。 组件还要和 DOM 以及其它组件进行交互。 **只涉及类**的测试可以告诉你组件类的行为是否正常， 但是不能告诉你组件是否能正常渲染出来、响应用户的输入和查询或与它的父组件和子组件相集成。

上述**只涉及类**的测试没办法回答这些组件在屏幕上的行为之类的关键性问题：

- `Lightswitch.clicked()` 是否真的绑定到了某些用户可以接触到的东西？
- `Lightswitch.message` 是否真的显示出来了？
- 用户真的可以选择 `DashboardHeroComponent` 中显示的某个英雄吗？
- 英雄的名字是否如预期般显示出来了？ （比如是否大写）
- `WelcomeComponent` 的模板是否显示了欢迎信息？

这些问题对于上面这种简单的组件来说当然没有问题， 不过很多组件和它们模板中所描述的 DOM 元素之间会有复杂的交互，当组件的状态发生变化时，会导致一些 HTML 出现和消失。

要回答这类问题，你就不得不创建那些与组件相关的 DOM 元素了，你必须检查 DOM 来确认组件的状态能在恰当的时机正常显示出来，并且必须通过屏幕来仿真用户的交互，以判断这些交互是否如预期的那般工作。

要想写这类测试，你就要用到 `TestBed` 的附加功能以及其它测试助手了。

CLI 生成的测试

当你用 CLI 生成新的组件时，它也会默认创建最初的测试文件。

比如，下列 CLI 命令会在 `app/banner` 文件夹中创建带有内联模板和内联样式的 `BannerComponent`：

```
ng generate component banner --inline-template --inline-style --module app
```

它也会为组件生成最初的测试文件 `banner-external.component.spec.ts`，代码如下：

app/banner/banner-external.component.spec.ts (initial)

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { BannerComponent } from './banner.component';

describe('BannerComponent', () => {
  let component: BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(BannerComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeDefined();
  });
});
```

缩减环境准备代码

这个文件中只有最后三行是真正测试组件的，它们用来断言 Angular 可以创建该组件。

文件的其它部分都是为更高级的测试二准备的样板代码，当组件逐渐演变成更加实质性的东西时，它们才**可能**变成必备的。

稍后你将学到这些高级的测试特性。 不过目前，你可以先把这些测试文件缩减成更加可控的大小，以便理解：

app/banner/banner-initial.component.spec.ts (minimal)

```
describe('BannerComponent (minimal)', () => {
  it('should create', () => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ]
    });
    const fixture = TestBed.createComponent(BannerComponent);
    const component = fixture.componentInstance;
    expect(component).toBeDefined();
  });
});
```

在这个例子中，传给 `TestBed.configureTestingModule` 的元数据对象中只声明了 `BannerComponent` —— 待测试的组件。

```
TestBed.configureTestingModule({
  declarations: [ BannerComponent ]
});
```

> 不用声明或导入任何其它的东西。 默认的测试模块中已经预先配置好了一些东西，比如来自 `@angular/platform-browser` 的 `BrowserModule`。
>
> 稍后你将会调用带有导入模块、服务提供商和更多可声明对象的 `TestBed.configureTestingModule()` 来满足测试所需。 将来还可以用可选的 `override` 方法对这些配置进行微调。

createComponent()

在配置好 `TestBed` 之后，你还可以调用它的 `createComponent()` 方法。

```
const fixture = TestBed.createComponent(BannerComponent);
```

`TestBed.createComponent()` 会创建一个 `BannerComponent` 的实例，把相应的元素添加到测试运行器的 DOM 中，然后返回一个 `ComponentFixture` 对象。

> 在调用了 `createComponent` 之后就不能再重新配置 `TestBed` 了。
>
> `createComponent` 方法冻结了当前的 `TestBed` 定义，关闭它才能再进行后续配置。
>
> 你不能再调用任何 `TestBed` 的后续配置方法了，不能调 `configureTestingModule()`、不能调 `get()`， 也不能调用任何 `override...` 方法。 如果试图这么做， `TestBed` 就会抛出错误。

ComponentFixture

ComponentFixture 是一个测试挽具（就像马车缰绳）， 用来与所创建的组件及其 DOM 元素进行交互。

可以通过测试夹具（fixture）来访问该组件的实例，并用 Jasmine 的 `expect` 语句来确保其存在。

```
const component = fixture.componentInstance;
expect(component).toBeDefined();
```

beforeEach()

随着该组件的成长，你将会添加更多测试。 除了为每个测试都复制一份 `TestBed` 测试之外，你还可以把它们重构成 Jasmine 的 `beforeEach()` 中的准备语句以及一些支持性变量：

```
describe('BannerComponent (with beforeEach)', () => {
  let component: BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ]
    });
    fixture = TestBed.createComponent(BannerComponent);
    component = fixture.componentInstance;
  });

  it('should create', () => {
    expect(component).toBeDefined();
  });
});
```

现在，添加一个测试，用它从 `fixture.nativeElement` 中获取组件的元素，并查找是否存在所预期的文本内容。

```
it('should contain "banner works!"', () => {
  const bannerElement: HTMLElement = fixture.nativeElement;
  expect(bannerElement.textContent).toContain('banner works!');
});
```

nativeElement

`ComponentFixture.nativeElement` 的值是 `any` 类型的。 稍后你将遇到的 `DebugElement.nativeElement` 也同样是 `any` 类型的。

Angular 在编译期间没办法知道 `nativeElement` 是哪种 HTML 元素，甚至是否 HTML 元素（译注：比如可能是 SVG 元素）。 本应用还可能运行在**非浏览器平台上**，比如服务端渲染或 Web Worker 那里的元素可能只有一些缩水过的 API，甚至根本不存在。

本指南中的例子都是为运行在浏览器中而设计的，因此 `nativeElement` 的值一定会是 `HTMLElement` 及其派生类。

如果知道了它是某种 `HTMLElement`，你就可以用标准的 `querySelector` 在元素树中进行深挖了。

下面这个测试就会调用 `HTMLElement.querySelector` 来获取 `<p>` 元素，并在其中查找 Banner 文本：

```
it('should have <p> with "banner works!"', () => {
  const bannerElement: HTMLElement = fixture.nativeElement;
  const p = bannerElement.querySelector('p');
  expect(p.textContent).toEqual('banner works!');
});
```

DebugElement

Angular 的**夹具**可以通过 `fixture.nativeElement` 直接提供组件的元素。

```
const bannerElement: HTMLElement = fixture.nativeElement;
```

它实际上是 `fixture.debugElement.nativeElement` 的一个便利方法。

```
const bannerDe: DebugElement = fixture.debugElement;
const bannerEl: HTMLElement = bannerDe.nativeElement;
```

这种访问元素的迂回方式有很好的理由。

`nativeElement` 的属性取决于运行环境。 你可以在没有 DOM，或者其 DOM 模拟器无法支持全部 `HTMLElement` API 的平台上运行这些测试。

Angular 依赖于 `DebugElement` 这个抽象层，就可以安全的横跨**其支持的所有平台**。 Angular 不再创建 HTML 元素树，而是创建 `DebugElement` 树，其中包裹着相应运行平台上的**原生元素**。 `nativeElement` 属性会解开 `DebugElement`，并返回平台相关的元素对象。

因为本章的这些测试都设计为只运行在浏览器中，因此这些测试中的 `nativeElement` 总是 `HTMLElement`， 你可以在测试中使用那些熟悉的方法和属性进行浏览。

下面是对上一个测试改用 `fixture.debugElement.nativeElement` 进行的重新实现：

```
it('should find the <p> with fixture.debugElement.nativeElement)', () => {
  const bannerDe: DebugElement = fixture.debugElement;
  const bannerEl: HTMLElement = bannerDe.nativeElement;
  const p = bannerEl.querySelector('p');
  expect(p.textContent).toEqual('banner works!');
});
```

`DebugElement` 还有其它的方法和属性，它们在测试中也很有用，你将在本章的其它测试中看到它们。

你要从 Angular 核心库中导入 `DebugElement` 符号。

```
import { DebugElement } from '@angular/core';
```

By.css()

虽然本章中的测试都是运行在浏览器中的，不过有些应用可能会运行在其它平台上（至少一部分时间是这样）。

比如，作为加快慢速网络设备上应用启动速度的一种策略，组件可能要先在服务器上渲染。服务端渲染可能无法支持完全的 HTML API。 如果它不支持 `querySelector`，那么前一个测试就会失败。

`DebugElement` 提供了可以工作在所有受支持的平台上的查询方法。 这些查询方法接受一个谓词（predicate）函数，如果 `DebugElement` 树中的节点满足某个筛选条件，它就返回 `true`。

你可以在从库中导入的 `By` 类的帮助下为该运行平台创建谓词函数。下面这个 `By` 是从浏览器平台导入的：

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

下面这个例子使用 `DebugElement.query()` 和浏览器的 `By.css` 方法重新实现了前一个测试。

```
it('should find the <p> with fixture.debugElement.query(By.css)', () => {
  const bannerDe: DebugElement = fixture.debugElement;
  const paragraphDe = bannerDe.query(By.css('p'));
  const p: HTMLElement = paragraphDe.nativeElement;
  expect(p.textContent).toEqual('banner works!');
});
```

值得注意的地方有：

- `By.css()` 静态方法使用标准 CSS 选择器选择了一些 `DebugElement` 节点。
- 这次查询返回了 `<p>` 元素的一个 `DebugElement`。
- 你必须解包此结果，以获取这个 `<p>` 元素。

当你要通过 CSS 选择器过滤，并且只打算测试浏览器的**原生元素**的属性时，`By.css` 这种方法就有点多余了。

使用标准的 `HTMLElement` 方法，比如 `querySelector()` 或 `querySelectorAll()` 通常会更简单、更清晰。 你在下一组测试中就会体会到这一点。

组件测试场景

下面这些部分构成了本指南的大部分内容，它们将探讨一些常见的组件测试场景。

组件绑定

当前的 `BannerComponent` 在 HTML 模板中展示了静态标题内容。

稍作修改之后，`BannerComponent` 也可以通过绑定到组件的 `title` 属性来展示动态标题。就像这样：

app/banner/banner.component.ts

```
@Component({
  selector: 'app-banner',
  template: '<h1>{{title}}</h1>',
  styles: ['h1 { color: green; font-size: 350%}']
})
export class BannerComponent {
  title = 'Test Tour of Heroes';
}
```

很简单，你决定添加一个测试来确定这个组件真的像你预期的那样显示出了正确的内容。

查询 `<h1>`

你将会写一系列测试来探查 `<h1>` 元素的值，这个值包含在了带有 `title` 属性的插值表达式绑定中。

你要修改 `beforeEach` 来使用标准的 HTML `querySelector` 来找到该元素，并把它赋值给 `h1` 变量。

app/banner/banner.component.spec.ts (setup)

```
let component: BannerComponent;
let fixture:   ComponentFixture<BannerComponent>;
let h1:        HTMLElement;

beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  });
  fixture = TestBed.createComponent(BannerComponent);
  component = fixture.componentInstance; // BannerComponent test instance
  h1 = fixture.nativeElement.querySelector('h1');
});
```

`createComponent()` 函数不会绑定数据

你的第一个测试希望看到屏幕显示出了默认的 `title`。 你本能的写出如下测试来立即审查这个 `<h1>` 元素：

```
it('should display original title', () => {
  expect(h1.textContent).toContain(component.title);
});
```

但**测试失败了**，给出如下信息：

```
expected '' to contain 'Test Tour of Heroes'.
```

因为绑定是在 Angular 执行变更检测时才发生的。

在产品阶段，当 Angular 创建组件、用户输入或者异步动作（比如 AJAX）完成时，会自动触发变更检测。

但 `TestBed.createComponent` **不能**触发变更检测。 可以在这个修改后的测试中确定这一点：

```
it('no title in the DOM after createComponent()', () => {
  expect(h1.textContent).toEqual('');
});
```

detectChanges()

你必须通过调用 `fixture.detectChanges()` 来要求 `TestBed` 执行数据绑定。 然后 `<h1>` 中才会具有所期望的标题。

```
it('should display original title after detectChanges()', () => {
  fixture.detectChanges();
  expect(h1.textContent).toContain(component.title);
});
```

这种迟到的变更检测是故意设计的，而且很有用。 它给了测试者一个机会，**在 Angular 初始化数据绑定以及调用生命周期钩子之前**探查并改变组件的状态。

下面这个测试中，会在调用 `fixture.detectChanges()` **之前**修改组件的 `title` 属性。

```
it('should display a different test title', () => {
  component.title = 'Test Title';
  fixture.detectChanges();
  expect(h1.textContent).toContain('Test Title');
});
```

自动变更检测

`BannerComponent` 的这些测试需要频繁调用 `detectChanges`。 有些测试者更喜欢让 Angular 测试环境自动运行变更检测。

使用 `ComponentFixtureAutoDetect` 服务提供商来配置 `TestBed` 就可以做到这一点。 首先从测试工具库中导入它：

app/banner/banner.component.detect-changes.spec.ts (import)

```
import { ComponentFixtureAutoDetect } from '@angular/core/testing';
```

然后把它添加到测试模块配置的 `providers` 数组中：

app/banner/banner.component.detect-changes.spec.ts (AutoDetect)

```
TestBed.configureTestingModule({
  declarations: [ BannerComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
});
```

下列测试阐明了自动变更检测的工作原理。

app/banner/banner.component.detect-changes.spec.ts (AutoDetect Tests)

```
it('should display original title', () => {
  // Hooray! No `fixture.detectChanges()` needed
  expect(h1.textContent).toContain(comp.title);
});

it('should still see original title after comp.title change', () => {
  const oldTitle = comp.title;
  comp.title = 'Test Title';
  // Displayed title is old because Angular didn't hear the change :(
  expect(h1.textContent).toContain(oldTitle);
});

it('should display updated title after detectChanges', () => {
  comp.title = 'Test Title';
  fixture.detectChanges(); // detect changes explicitly
  expect(h1.textContent).toContain(comp.title);
});
```

第一个测试程序展示了自动检测的好处。

第二和第三个测试程序显示了它重要的局限性。 Angular 测试环境不会知道测试程序改变了组件的 `title` 属性。 自动检测只对异步行为比如承诺的解析、计时器和 DOM 事件作出反应。 但是直接修改组件属性值的这种同步更新是不会触发自动检测的。 测试程序必须手动调用 `fixture.detectChange()`，来触发新一轮的变更检测周期。

> 相比于受测试工具有没有执行变更检测的困扰，本章中的例子更愿意总是显式调用 `detectChanges()`。 即使是在不需要的时候，频繁调用 `detectChanges()` 也没有任何坏处。

使用 `dispatchEvent()` 修改输入值

要想模拟用户输入，你就要找到 `<input>` 元素并设置它的 `value` 属性。

你要调用 `fixture.detectChanges()` 来触发 Angular 的变更检测。 但那只是一个基本的中间步骤。

Angular 不知道你设置了这个 `<input>` 元素的 `value` 属性。 在你通过调用 `dispatchEvent()` 触发了该输入框的 `input` 事件之前，它不能读到那个值。 **调用完之后**你再调用 `detectChanges()`。

下面的例子演示了这个调用顺序。

app/hero/hero-detail.component.spec.ts (pipe test)

```
1. it('should convert hero name to Title Case', () => {
2.   // get the name's input and display elements from the DOM
3.   const hostElement = fixture.nativeElement;
4.   const nameInput: HTMLInputElement = hostElement.querySelector('input');
5.   const nameDisplay: HTMLElement = hostElement.querySelector('span');
6.
7.   // simulate user entering a new name into the input box
8.   nameInput.value = 'quick BROWN  fOx';
9.
10.   // dispatch a DOM event so that Angular learns of input value change.
11.   nameInput.dispatchEvent(newEvent('input'));
12.
13.   // Tell Angular to update the display binding through the title pipe
14.   fixture.detectChanges();
15.
16.   expect(nameDisplay.textContent).toBe('Quick Brown  Fox');
17. });
```

带有外部文件的组件

上面的 `BannerComponent` 定义了一个**内联模板**和**内联 CSS**，分别是在 `@Component.template` 和 `@Component.styles` 属性中指定的。

很多组件会分别用 `@Component.templateUrl` 和 `@Component.styleUrls` 属性来指定**外部模板**和**外部 CSS**，就像下面这个 `BannerComponent` 的变体中所做的一样：

app/banner/banner-external.component.ts (metadata)

```
@Component({
  selector: 'app-banner',
  templateUrl: './banner-external.component.html',
  styleUrls:  ['./banner-external.component.css']
})
```

这个语法告诉 Angular 编译器在编译期间读取外部文件。

当你运行 CLI 的 `ng test` 命令的时候这毫无问题，因为它会**在运行测试之前先编译该应用**。

不过，如果你在非 CLI 环境下运行这些测试，那么对该组件的测试就可能失败。 比如，如果你在像 plunker 这样的 Web 编程环境下运行 `BannerComponent` 的测试，就会看到如下信息：

```
Error: This test module uses the component BannerComponent
which is using a "templateUrl" or "styleUrls", but they were never compiled.
Please call "TestBed.compileComponents" before your test.
```

如果在**测试自身期间**，运行环境试图编译源码，就会出现这个测试错误信息。

要解决这个问题，可以像稍后解释的那样调用一次 `compileComponents()`。

带依赖的组件

组件经常依赖其他服务。

`WelcomeComponent` 为登陆的用户显示一条欢迎信息。它从注入的 `UserService` 的属性得知用户的身份：

app/welcome/welcome.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UserService }       from '../model/user.service';

@Component({
  selector: 'app-welcome',
  template: '<h3 class="welcome"><i>{{welcome}}</i></h3>'
})
export class WelcomeComponent  implements OnInit {
  welcome: string;
  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.welcome = this.userService.isLoggedIn ?
      'Welcome, ' + this.userService.user.name : 'Please log in.';
  }
}
```

`WelcomeComponent` 带有与服务交互的决策逻辑，这些逻辑让该组件值得测试。 下面是 `app/welcome/welcome.component.spec.ts` 中的测试模块配置：

app/welcome/welcome.component.spec.ts

```
TestBed.configureTestingModule({
   declarations: [ WelcomeComponent ],
// providers:    [ UserService ]  // NO! Don't provide the real service!
                                  // Provide a test-double instead
   providers:    [ {provide: UserService, useValue: userServiceStub } ]
});
```

这次，在测试配置里不但声明了被测试的组件，而且在 `providers` 数组中添加了 `UserService` 依赖。但不是真实的 `UserService`。

提供服务的测试替身

被测试的组件不一定要注入真正的服务。实际上，服务的替身（Stub - 桩, Fake - 假冒品, Spy - 间谍或者 Mock - 模拟对象）通常会更加合适。 spec 的主要目的是测试组件，而不是服务。真实的服务可能连自身都有问题，不应该让它干扰对组件的测试。

注入真实的 `UserService` 有可能很麻烦。真实的服务可能询问用户登录凭据，也可能试图连接认证服务器。可能很难处理这些行为。所以在真实的 `UserService` 的位置创建和注册 `UserService` 替身，会让测试更加容易和安全。

这个测试套件提供了 `UserService` 的一个最小化模拟对象，它能满足 `WelcomeComponent` 及其测试的需求：

app/welcome/welcome.component.spec.ts

```
let userServiceStub: Partial<UserService>;

userServiceStub = {
  isLoggedIn: true,
  user: { name: 'Test User'}
};
```

获取注入的服务

测试程序需要访问被注入到 `WelcomeComponent` 中的 `UserService`（stub 类）。

Angular 的注入系统是层次化的。 可以有很多层注入器，从根 `TestBed` 创建的注入器下来贯穿整个组件树。

最安全并总是有效的获取注入服务的方法，是从被测组件的注入器获取。 组件注入器是 fixture 的 `DebugElement` 的属性之一。

WelcomeComponent's injector

```
// UserService actually injected into the component
userService = fixture.debugElement.injector.get(UserService);
```

TestBed.get()

你也可能通过 `TestBed.get()` 来使用根注入器获取该服务。 这样更容易记住而且不那么啰嗦。 不过这只有当 Angular 组件需要的恰好是该测试的根注入器时才能正常工作。

在这个测试套件中，`UserService` **唯一**的提供商就是根测试模块中的，因此调用 `TestBed.get()` 就是安全的，代码如下：

TestBed injector

```
// UserService from the root injector
userService = TestBed.get(UserService);
```

> 对于那些不能用 `TestBed.get()` 的测试用例，请参见改写组件的提供商一节，那里解释了何时以及为何必须改从组件自身的注入器中获取服务。

总是从注入其中获取服务

请不要引用测试代码里提供给测试模块的 `userServiceStub` 对象。这样不行！ 被注入组件的 `userService` 实例是完全不一样的对象，它提供的是 `userServiceStub` 的克隆。

app/welcome/welcome.component.spec.ts

```
it('stub object and injected UserService should not be the same', () => {
  expect(userServiceStub === userService).toBe(false);

  // Changing the stub object has no effect on the injected service
  userServiceStub.isLoggedIn = false;
  expect(userService.isLoggedIn).toBe(true);
});
```

最终的准备及测试代码

下面是使用 `TestBed.get()` 的完整的 `beforeEach()`：

app/welcome/welcome.component.spec.ts

```
let userServiceStub: Partial<UserService>;

beforeEach(() => {
  // stub UserService for test purposes
  userServiceStub = {
    isLoggedIn: true,
    user: { name: 'Test User'}
  };

  TestBed.configureTestingModule({
     declarations: [ WelcomeComponent ],
     providers:    [ {provide: UserService, useValue: userServiceStub } ]
  });

  fixture = TestBed.createComponent(WelcomeComponent);
  comp    = fixture.componentInstance;

  // UserService from the root injector
  userService = TestBed.get(UserService);

  //  get the "welcome" element by CSS selector (e.g., by class name)
  el = fixture.nativeElement.querySelector('.welcome');
});
```

下面是一些测试程序:

app/welcome/welcome.component.spec.ts

```
it('should welcome the user', () => {
  fixture.detectChanges();
  const content = el.textContent;
  expect(content).toContain('Welcome', '"Welcome ..."');
  expect(content).toContain('Test User', 'expected name');
});

it('should welcome "Bubba"', () => {
  userService.user.name = 'Bubba'; // welcome message hasn't been shown yet
  fixture.detectChanges();
  expect(el.textContent).toContain('Bubba');
});

it('should request login if not logged in', () => {
  userService.isLoggedIn = false; // welcome message hasn't been shown yet
  fixture.detectChanges();
  const content = el.textContent;
  expect(content).not.toContain('Welcome', 'not welcomed');
  expect(content).toMatch(/log in/i, '"log in"');
});
```

第一个测试程序是合法测试程序，它确认这个被模拟的 `UserService` 是否被调用和工作正常。

> Jasmine 匹配器的第二个参数（比如 `'expected name'`）是一个可选的失败标签。 如果这个期待语句失败了，Jasmine 就会把这个标签追加到这条个期待语句的失败信息后面。 对于具有多个期待语句的规约，它可以帮助澄清到底什么出错了，以及哪个期待语句失败了。

接下来的测试程序确认当服务返回不同的值时组件的逻辑是否工作正常。 第二个测试程序验证变换用户名字的效果。 第三个测试程序检查如果用户没有登录，组件是否显示正确消息。

带有异步服务的组件

在这个例子中，`AboutComponent` 的模板中还有一个 `TwainComponent`。 `TwainComponent` 用于显示引自马克·吐温的话。

app/twain/twain.component.ts (template)

```
template: `
  <p class="twain"><i>{{quote | async}}</i></p>
  <button (click)="getQuote()">Next quote</button>
  <p class="error" *ngIf="errorMessage">{{ errorMessage }}</p>`,
```

注意该组件的 `quote` 属性的值是通过 `AsyncPipe` 传进来的。 这意味着该属性或者返回 `Promise` 或者返回 `Observable`。

在这个例子中，`TwainComponent.getQuote()` 方法告诉你 `quote` 方法返回的是 `Observable`。

app/twain/twain.component.ts (getQuote)

```
getQuote() {
  this.errorMessage = '';
  this.quote = this.twainService.getQuote().pipe(
    startWith('...'),
    catchError( (err: any) => {
      // Wait a turn because errorMessage already set once this turn
      setTimeout(() => this.errorMessage = err.message || err.toString());
      return of('...'); // reset message to placeholder
    })
  );
```

`TwainComponent` 会从一个注入进来的 `TwainService` 来获取这些引文。 在服务返回第一条引文之前，该组件会先返回一个占位值（`'...'`）的 `Observable`。

`catchError` 会拦截服务中的错误，准备错误信息，并在成功分支中返回占位值。 它必须等一拍（tick）才能设置 `errorMessage`，以免在同一个变更检测周期中两次修改这个消息而导致报错。

这就是你要测试的全部特性。

使用间谍（Spy）进行测试

当测试组件时，只应该关心服务的公共 API。 通常来说，测试不应该自己向远端服务器发起调用。 它们应该对这些调用进行仿真。`app/twain/twain.component.spec.ts` 中的准备代码展示了实现方式之一：

app/twain/twain.component.spec.ts (setup)

```
beforeEach(() => {
  testQuote = 'Test Quote';

  // Create a fake TwainService object with a `getQuote()` spy
  const twainService = jasmine.createSpyObj('TwainService', ['getQuote']);
  // Make the spy return a synchronous Observable with the test data
  getQuoteSpy = twainService.getQuote.and.returnValue( of(testQuote) );

  TestBed.configureTestingModule({
    declarations: [ TwainComponent ],
    providers:    [
      { provide: TwainService, useValue: twainService }
    ]
  });

  fixture = TestBed.createComponent(TwainComponent);
  component = fixture.componentInstance;
  quoteEl = fixture.nativeElement.querySelector('.twain');
});
```

重点看这个间谍对象（spy）。

```
// Create a fake TwainService object with a `getQuote()` spy
const twainService = jasmine.createSpyObj('TwainService', ['getQuote']);
// Make the spy return a synchronous Observable with the test data
getQuoteSpy = twainService.getQuote.and.returnValue( of(testQuote) );
```

这个间谍的设计是：任何对 `getQuote` 的调用都会收到一个包含测试引文的可观察对象。 和真正的 `getQuote()` 方法不同，这个间谍跳过了服务器，直接返回了一个能立即解析出值的同步型可观察对象。

虽然它的 `Observable` 是同步的，不过你仍然可以使用这个间谍对象写出很多有用的测试。

同步测试

同步 `Observable` 的一大优点就是你可以把那些异步的流程转换成同步测试。

```
it('should show quote after component initialized', () => {
  fixture.detectChanges(); // onInit()

  // sync spy result shows testQuote immediately after init
  expect(quoteEl.textContent).toBe(testQuote);
  expect(getQuoteSpy.calls.any()).toBe(true, 'getQuote called');
});
```

因为间谍对象的结果是同步返回的，所以 `getQuote()` 方法会在 Angular 调用 `ngOnInit` 时触发的首次变更检测周期后立即修改屏幕上的消息。

但测试出错路径的时候就没这么幸运了。 虽然该服务的间谍也会返回一个同步的错误对象，但是组件的那个方法中调用了 `setTimeout()`。 这个测试必须至少等待 JavaScript 引擎的一个周期，那个值才会变成可用状态。因此这个测试变成了**异步的**。

使用 `fakeAsync()` 进行异步测试

下列测试用于确保当服务返回 `ErrorObservable` 的时候也能有符合预期的行为。

```
1. it('should display error when TwainService fails', fakeAsync(() => {
2.   // tell spy to return an error observable
3.   getQuoteSpy.and.returnValue(
4.     throwError('TwainService test failure'));
5.
6.   fixture.detectChanges(); // onInit()
7.   // sync spy errors immediately after init
8.
9.   tick(); // flush the component's setTimeout()
10.
11.   fixture.detectChanges(); // update errorMessage within setTimeout()
12.
13.   expect(errorMessage()).toMatch(/test failure/, 'should display error');
14.   expect(quoteEl.textContent).toBe('...', 'should show placeholder');
15. }));
```

注意这个 `it()` 函数接收了一个如下形式的参数。

```
fakeAsync(() => { /* test body */ })`
```

`fakeAsync` 函数通过在一个特殊的`fakeAsync` **测试区域（zone）**中运行测试体来启用线性代码风格。 测试体看上去是同步的。 这里没有嵌套式语法（如 `Promise.then()`）来打断控制流。

`tick()` 函数

你必须调用 `tick()` 函数来向前推动（虚拟）时钟。

调用 `tick()` 会模拟时光的流逝，直到所有未决的异步活动都结束为止。 在这个例子中，它会等待错误处理器中的 `setTimeout()`。

`tick` 函数是你通过 `TestBed` 中引入的 Angular 测试工具集之一。 它总是和 `fakeAsync` 一起使用，你也只能在 `fakeAsync` 的函数体中调用它。

异步的可观察对象

你可能对这些测试的覆盖率已经很满足了。

不过你可能会因为真实的服务没有按这种方式工作而困扰。 真实的服务器会把请求发送给远端服务器。 服务需要花一些时间来作出响应，它的响应当然也不会真的像前面两个测试中那样立即可用。

如果你在 `getQuote()` 间谍中返回一个**异步**可观察对象，那它就能更忠诚的反映出真实的世界了。

```
// Simulate delayed observable values with the `asyncData()` helper
getQuoteSpy.and.returnValue(asyncData(testQuote));
```

可观察对象的异步助手

这个异步的可观察对象是用 `asyncData` 辅助函数生成的。 `asyncData` 助手是一个工具函数，你可以自己写一个，也可以从下面的范例代码中复制一份。

testing/async-observable-helpers.ts

```
/** Create async observable that emits-once and completes
 *  after a JS engine turn */
export function asyncData<T>(data: T) {
  return defer(() => Promise.resolve(data));
}
```

这个辅助函数的可观察对象会在 JavaScript 引擎的下一个工作周期中发出 `data` 的值。

RxJS 的 `defer()` （延期）操作符 会返回一个可观察对象。 它获取一个工厂函数，这个工厂函数或者返回 Promise 或者返回 Observable。 当有人订阅了这个 `defer` 的可观察对象时，它就会把这个订阅者添加到由那个工厂函数创建的新的可观察对象中。

`defer()` 操作符会把 `Promise.resolve()` 转换成一个新的可观察对象，然后像 `HttpClient` 那样的发出一个值，然后结束。 订阅者将会在接收到这个数据值之后自动被取消订阅。

下面是一个类似的用于产生异步错误的辅助函数。

```
/** Create async observable error that errors
 *  after a JS engine turn */
export function asyncError<T>(errorObject: any) {
  return defer(() => Promise.reject(errorObject));
}
```

更多异步测试

现在，`getQuote()` 间谍会返回一个异步的可观察对象，你的大多数测试也同样要变成异步的。

下面这个 `fakeAsync()` 测试演示了你所期待的和真实世界中一样的数据流。

```
it('should show quote after getQuote (fakeAsync)', fakeAsync(() => {
  fixture.detectChanges(); // ngOnInit()
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');

  tick(); // flush the observable to get the quote
  fixture.detectChanges(); // update view

  expect(quoteEl.textContent).toBe(testQuote, 'should show quote');
  expect(errorMessage()).toBeNull('should not show error');
}));
```

注意，这个 `<quote>` 元素应该在 `ngOnInit()` 之后显示占位值（`'...'`）， 但第一个引文却没有出现。

要刷出可观察对象中的第一个引文，你就要先调用 `tick()`，然后调用 `detectChanges()` 来要求 Angular 刷新屏幕。

然后你就可以断言这个 `<quote>` 元素应该显示所预期的文字了。

使用 `async()` 进行异步测试

`fakeAsync()` 工具函数有一些限制。 特别是，如果测试中发起了 `XHR` 调用，它就没用了。

测试中的 `XHR` 调用比较罕见，所以你通常会使用 `fakeAsync()`。 不过你可能迟早会需要调用 `XHR`，那就来了解一些 `async()` 的知识吧。

> `TestBed.compileComponents()` 方法（参见稍后）就会在 JIT 编译期间调用 `XHR` 来读取外部模板和 CSS 文件。 如果写调用了 `compileComponents()` 的测试，就要用到 `async()` 工具函数了。

下面是用 `async()` 工具函数重写的以前的 `fakeAsync()` 测试。

```
it('should show quote after getQuote (async)', async(() => {
  fixture.detectChanges(); // ngOnInit()
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');

  fixture.whenStable().then(() => { // wait for async getQuote
    fixture.detectChanges();        // update view with quote
    expect(quoteEl.textContent).toBe(testQuote);
    expect(errorMessage()).toBeNull('should not show error');
  });
}));
```

`async()` 工具函数通过把测试人员的代码放进在一个特殊的**async 测试区域**中，节省了一些用于异步调用的样板代码。 你不必把 Jasmine 的 `done()` 传给这个测试，并在承诺（Promise）或可观察对象的回调中调用 `done()`。

但是对 `fixture.whenStable()` 的调用揭示了该测试的异步本性，它将会打破线性的控制流。

whenStable

该测试必须等待 `getQuote()` 的可观察对象发出下一条引言。 它不再调用 `tick()`，而是调用 `fixture.whenStable()`。

`fixture.whenStable()` 返回一个承诺，这个承诺会在 JavaScript 引擎的任务队列变为空白时被解析。 在这个例子中，一旦这个可观察对象发出了第一条引言，这个任务队列就会变为空。

该测试在这个承诺的回调中继续执行，它会调用 `detectChanges()` 来用预期的文本内容修改 `<quote>` 元素。

Jasmine **done()**

虽然 `async` 和 `fakeAsync` 函数极大地简化了 Angular 的异步测试，不过你仍然可以回退到传统的技术中。 也就是说给 `it` 额外传入一个函数型参数，这个函数接受一个 `done` 回调。

现在，你就要负责对 Promise 进行链接、处理错误，并在适当的时机调用 `done()` 了。

写带有 `done()` 的测试函数会比 `async` 和 `fakeAsync` 方式更加冗长。 不过有些时候它是必要的。 比如，你不能在那些涉及到 `intervalTimer()` 或 RxJS 的 `delay()` 操作符时调用 `async` 或 `fakeAsync` 函数。

下面是对前面的测试用 `done()` 重写后的两个版本。 第一个会订阅由组件的 `quote` 属性暴露给模板的那个 `Observable`。

```
it('should show last quote (quote done)', (done: DoneFn) => {
  fixture.detectChanges();

  component.quote.pipe( last() ).subscribe(() => {
    fixture.detectChanges(); // update view with quote
    expect(quoteEl.textContent).toBe(testQuote);
    expect(errorMessage()).toBeNull('should not show error');
    done();
  });
});
```

RxJS 的 `last()` 操作符会在结束之前发出这个可观察对象的最后一个值，也就是那条测试引文。 `subscribe` 回调中会像以前一样调用 `detectChanges()` 用这条测试引文更新 `<quote>` 元素。

有些测试中，相对于在屏幕上展示了什么，你可能会更关心所注入服务的某个方法是如何被调用的，以及它的返回值是什么。

服务的间谍，比如假冒服务 `TwainService` 的 `getQuote()` 间谍，可以给你那些信息，并且对视图的状态做出断言。

```
it('should show quote after getQuote (spy done)', (done: DoneFn) => {
  fixture.detectChanges();

  // the spy's most recent call returns the observable with the test quote
  getQuoteSpy.calls.mostRecent().returnValue.subscribe(() => {
    fixture.detectChanges(); // update view with quote
    expect(quoteEl.textContent).toBe(testQuote);
    expect(errorMessage()).toBeNull('should not show error');
    done();
  });
});
```

组件的宝石测试

前面的 `TwainComponent` 测试中使用 `TwainService` 中的 `asyncData` 和 `asyncError` 工具函数仿真了可观察对象的异步响应。

那些都是你自己写的简短函数。 很不幸，它们对于很多常见场景来说都太过简单了。 可观察对象通常会发出很多次值，还可能会在显著的延迟之后。 组件可能要协调多个由正常值和错误值组成的重叠序列的可观察对象。

RxJS 的宝石测试是测试各种可观察对象场景的最佳方式 —— 无论简单还是复杂。 你可以看看宝石图，它揭示了可观察对象的工作原理。 宝石测试使用类似的宝石语言来在你的测试中指定可观察对象流和对它们的期待。

下面的例子使用宝石测试重写了 `TwainComponent` 的两个测试。

首先安装 `jasmine-marbles` 这个 npm 包，然后倒入所需的符号。

app/twain/twain.component.marbles.spec.ts (import marbles)

```
import { cold, getTestScheduler } from 'jasmine-marbles';
```

下面是对获取引文功能的完整测试：

```
it('should show quote after getQuote (marbles)', () => {
  // observable test quote value and complete(), after delay
  const q$ = cold('---x|', { x: testQuote });
  getQuoteSpy.and.returnValue( q$ );

  fixture.detectChanges(); // ngOnInit()
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');

  getTestScheduler().flush(); // flush the observables

  fixture.detectChanges(); // update view

  expect(quoteEl.textContent).toBe(testQuote, 'should show quote');
  expect(errorMessage()).toBeNull('should not show error');
});
```

注意，这个 Jasmine 测试是同步的。没有调用 `fakeAsync()`。 宝石测试使用了一个测试调度程序来用同步的方式模拟时间的流逝。

宝石测试的美妙之处在于它给出了可观察对象流的可视化定义。 这个测试定义了一个**冷的**可观察对象，它等待三帧 (`---`)，然后发出一个值（`x`），然后结束（`|`）。 在第二个参数中，你把值标记（`x`）换成了实际发出的值（`testQuote`）。

```
const q$ = cold('---x|', { x: testQuote });
```

这个宝石库会构造出相应的可观察对象，测试代码会把它当做 `getQuote` 间谍的返回值。

当你已经准备好激活这个宝石库构造出的可观察对象时，只要让 `TestScheduler` 去**刷新**准备好的任务队列就可以了。代码如下：

```
getTestScheduler().flush(); // flush the observables
```

这个步骤的目的类似于前面的 `fakeAsync()` 和 `async()` 范例中的 `tick()` 和 `whenStable()`。 这种测试的权衡方式也和那些例子中是一样的。

宝石错误测试

下面是 `getQuote()` 错误测试的宝石测试版本。

```
it('should display error when TwainService fails', fakeAsync(() => {
  // observable error after delay
  const q$ = cold('---#|', null, new Error('TwainService test failure'));
  getQuoteSpy.and.returnValue( q$ );

  fixture.detectChanges(); // ngOnInit()
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');

  getTestScheduler().flush(); // flush the observables
  tick();                     // component shows error after a setTimeout()
  fixture.detectChanges();    // update error message

  expect(errorMessage()).toMatch(/test failure/, 'should display error');
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');
}));
```

它仍然是异步测试，要调用 `fakeAsync()` 和 `tick()`，这是因为组件自身在处理错误的时候调用 `setTimeout()`。

看看宝石库生成的可观察对象的定义。

```
const q$ = cold('---#|', null, new Error('TwainService test failure'));
```

它是一个**冷的**可观察对象，它等待三帧，然后发出一个错误。 井号（`#`）标记出了发出错误的时间点，这个错误是在第三个参数中指定的。 第二个参数是空的，因为这个可观察对象永远不会发出正常值。

深入学习宝石测试

宝石帧是测试时序中的虚拟单元。 每个符号（`-`，`x`，`|`，`#`）都表示一帧过去了。

冷的可观察对象不会生成值，除非你订阅它。 应用中的大多数可观察对象都是冷的。 所有 `HttpClient`的方法返回的都是冷的可观察对象。

热的可观察对象在你订阅它之前就会生成值。 `Router.events` 可观察对象会主动汇报路由器的活动，它就是个**热的**可观察对象。

RxJS 的宝石测试是一个内容丰富的主题，超出了本章的范围。 要想在网络上进一步学习它，可以从 official documentation 开始。

带有输入输出参数的组件

带有导入和导出的组件通常出现在宿主组件的视图模板中。 宿主使用属性绑定来设置输入属性，使用事件绑定来监听输出属性触发的事件。

测试的目的是验证这样的绑定和期待的那样正常工作。 测试程序应该设置导入值并监听导出事件。

`DashboardHeroComponent` 是非常小的这种类型的例子组件。 它显示由 `DashboardCompoent` 提供的英雄个体。 点击英雄告诉 `DashbaordComponent` 用户已经选择了这个英雄。

`DashboardHeroComponent` 是这样内嵌在 `DashboardCompoent` 的模板中的：

app/dashboard/dashboard.component.html (excerpt)

```
<dashboard-hero *ngFor="let hero of heroes"  class="col-1-4"
  [hero]=hero  (selected)="gotoDetail($event)" >
</dashboard-hero>
```

`DashboardHeroComponent` 在 `*ngFor` 循环中出现，把每个组件的 `hero` input 属性设置为迭代的值，并监听组件的 `selected` 事件。

下面是该组件的完整定义：

app/dashboard/dashboard-hero.component.ts (component)

```
@Component({
  selector: 'dashboard-hero',
  template: `
    <div (click)="click()" class="hero">
      {{hero.name | uppercase}}
    </div>`,
  styleUrls: [ './dashboard-hero.component.css' ]
})
export class DashboardHeroComponent {
  @Input() hero: Hero;
  @Output() selected = new EventEmitter<Hero>();
  click() { this.selected.emit(this.hero); }
}
```

虽然测试这么简单的组件没有什么内在价值，但是它的测试程序是值得学习的。 有下列候选测试方案：

- 把它当作被 `DashbaordComponent` 使用的组件来测试
- 把它当作独立的组件来测试
- 把它当作被 `DashbaordComponent` 的替代组件使用的组件来测试

简单看看 `DashbaordComponent` 的构造函数就否决了第一种方案：

app/dashboard/dashboard.component.ts (constructor)

```
constructor(
  private router: Router,
  private heroService: HeroService) {
}
```

`DashbaordComponent` 依赖 Angular 路由器和 `HeroService` 服务。 你必须使用测试替身替换它们两个，似乎过于复杂了。 路由器尤其具有挑战性。

> 稍后的讨论涵盖了那些需要路由器的测试组件。

当前的任务是测试 `DashboardHeroComponent` 组件，而非 `DashbaordComponent`，所以无需做不必要的努力。 那就试试第二和第三种方案。

单独测试 `DashboardHeroComponent`

下面是 spec 文件的准备语句中的重点部分。

app/dashboard/dashboard-hero.component.spec.ts (setup)

```
TestBed.configureTestingModule({
  declarations: [ DashboardHeroComponent ]
})
fixture = TestBed.createComponent(DashboardHeroComponent);
comp    = fixture.componentInstance;

// find the hero's DebugElement and element
heroDe  = fixture.debugElement.query(By.css('.hero'));
heroEl = heroDe.nativeElement;

// mock the hero supplied by the parent component
expectedHero = { id: 42, name: 'Test Name' };

// simulate the parent setting the input property with that hero
comp.hero = expectedHero;

// trigger initial data binding
fixture.detectChanges();
```

注意代码是如何将模拟英雄（`expectedHero`）赋值给组件的 `hero` 属性的，模拟了 `DashbaordComponent` 在它的迭代器中通过属性绑定的赋值方式。

下面的测试会验证英雄的名字已经通过绑定的方式传播到模板中了。

```
it('should display hero name in uppercase', () => {
  const expectedPipedName = expectedHero.name.toUpperCase();
  expect(heroEl.textContent).toContain(expectedPipedName);
});
```

因为模板通过 Angular 的 `UpperCasePipe` 传入了英雄的名字，所以这个测试必须匹配该元素的值中包含了大写形式的名字。

> 这个小测试示范了 Angular 的测试如何以较低的成本验证组件的视觉表现（它们不能通过组件类测试进行验证）。 而不用借助那些更慢、更复杂的端到端测试。

点击

点击这个英雄将会发出一个 `selected` 事件，而宿主元素（可能是 `DashboardComponent`）可能会听到它：

```
it('should raise selected event when clicked (triggerEventHandler)', () => {
  let selectedHero: Hero;
  comp.selected.subscribe((hero: Hero) => selectedHero = hero);

  heroDe.triggerEventHandler('click', null);
  expect(selectedHero).toBe(expectedHero);
});
```

该组件的 `selected` 属性返回一个 `EventEmitter`，对消费者来说它和 RxJS 的同步 `Observable` 很像。 该测试会**显式**订阅它，而宿主组件会**隐式**订阅它。

如果该组件的行为符合预期，点击英雄所在的元素就会告诉组件的 `selected` 属性发出这个 `hero` 对象。

这个测试会通过订阅 `selected` 来检测是否确实如此。

triggerEventHandler

前面测试中的 `heroDe` 是一个指向英雄条目 `<div>` 的 `DebugElement`。

它有一些用于抽象与原生元素交互的 Angular 属性和方法。 这个测试会使用事件名称 `click` 来调用 `DebugElement.triggerEventHandler`。 `click` 的事件绑定到了 `DashboardHeroComponent.click()`。

Angular 的 `DebugElement.triggerEventHandler` 可以用事件的名字触发任何数据绑定事件。 第二个参数是传递给事件处理器的事件对象。

该测试使用事件对象 `null` 触发了一次 `click` 事件。

```
heroDe.triggerEventHandler('click', null);
```

测试程序假设（在这里应该这样)运行时间的事件处理器（组件的 `click()` 方法）不关心事件对象。

> 其它处理器的要求比较严格。比如，`RouterLink` 指令期望一个带有 `button` 属性的对象，该属性用于指出点击时按下的是哪个鼠标按钮。 如果不给出这个事件对象，`RouterLink` 指令就会抛出一个错误。

点击该元素

下面这个测试改为调用原生元素自己的 `click()` 方法，它对于**这个组件**来说相当完美。

```
it('should raise selected event when clicked (element.click)', () => {
  let selectedHero: Hero;
  comp.selected.subscribe((hero: Hero) => selectedHero = hero);

  heroEl.click();
  expect(selectedHero).toBe(expectedHero);
});
```

click() 辅助函数

点击按钮、链接或者任意 HTML 元素是很常见的测试任务。

把**点击事件**的处理过程包装到如下的 `click()` 辅助函数中，可以让这项任务更一致、更简单：

testing/index.ts (click helper)

```
/** Button events to pass to `DebugElement.triggerEventHandler` for RouterLink
event handler */
export const ButtonClickEvents = {
   left:  { button: 0 },
   right: { button: 2 }
};

/** Simulate element click. Defaults to mouse left-button click event. */
export function click(el: DebugElement | HTMLElement, eventObj: any =
ButtonClickEvents.left): void {
  if (el instanceof HTMLElement) {
    el.click();
  } else {
    el.triggerEventHandler('click', eventObj);
  }
}
```

第一个参数是用来点击的元素。如果你愿意，可以将自定义的事件对象传递给第二个参数。 默认的是（局部的）鼠标左键事件对象， 它被许多事件处理器接受，包括 `RouterLink` 指令。

> `click()` 辅助函数不是Angular 测试工具之一。 它是在本章的例子代码中定义的函数方法，被所有测试例子所用。 如果你喜欢它，将它添加到你自己的辅助函数集。

下面是把前面的测试用 `click` 辅助函数重写后的版本。

app/dashboard/dashboard-hero.component.spec.ts (test with click helper)

```
it('should raise selected event when clicked (click helper)', () => {
  let selectedHero: Hero;
  comp.selected.subscribe(hero => selectedHero = hero);

  click(heroDe); // click helper with DebugElement
  click(heroEl); // click helper with native element

  expect(selectedHero).toBe(expectedHero);
});
```

位于测试宿主中的组件

前面的这些测试都是自己扮演宿主元素 `DashboardComponent` 的角色。 但是当 `DashboardHeroComponent` 真的绑定到某个宿主元素时还能正常工作吗？

固然，你也可以测试真实的 `DashboardComponent`。 但要想这么做需要做很多准备工作，特别是它的模板中使用了某些特性，如 `*ngFor`、 其它组件、布局 HTML、附加绑定、注入了多个服务的构造函数、如何用正确的方式与那些服务交互等。

想出这么多需要努力排除的干扰，只是为了证明一点 —— 可以像这样造出一个令人满意的**测试宿主**：

app/dashboard/dashboard-hero.component.spec.ts (test host)

```
@Component({
  template: `
    <dashboard-hero
      [hero]="hero" (selected)="onSelected($event)">
    </dashboard-hero>`
})
class TestHostComponent {
  hero: Hero = {id: 42, name: 'Test Name' };
  selectedHero: Hero;
  onSelected(hero: Hero) { this.selectedHero = hero; }
}
```

这个测试宿主像 `DashboardComponent` 那样绑定了 `DashboardHeroComponent`，但是没有 `Router`、 没有 `HeroService`，也没有 `*ngFor`。

这个测试宿主使用其测试用的英雄设置了组件的输入属性 `hero`。 它使用 `onSelected` 事件处理器绑定了组件的 `selected` 事件，其中把事件中发出的英雄记录到了 `selectedHero` 属性中。

稍后，这个测试就可以轻松检查 `selectedHero` 以验证 `DashboardHeroComponent.selected` 事件确实发出了所期望的英雄。

这个**测试宿主**中的准备代码和独立测试中的准备过程类似：

app/dashboard/dashboard-hero.component.spec.ts (test host setup)

```
TestBed.configureTestingModule({
  declarations: [ DashboardHeroComponent, TestHostComponent ]
})
// create TestHostComponent instead of DashboardHeroComponent
fixture  = TestBed.createComponent(TestHostComponent);
testHost = fixture.componentInstance;
heroEl   = fixture.nativeElement.querySelector('.hero');
fixture.detectChanges(); // trigger initial data binding
```

这个测试模块的配置信息有三个重要的不同点：

1. 它同时声明了 `DashboardHeroComponent` 和 `TestHostComponent`。
2. 它创建了 `TestHostComponent`，而非 `DashboardHeroComponent`。
3. `TestHostComponent` 通过绑定机制设置了 `DashboardHeroComponent.hero`。

`createComponent` 返回的 `fixture` 里有 `TestHostComponent` 实例，而非 `DashboardHeroComponent` 组件实例。

当然，创建 `TestHostComponent` 有创建 `DashboardHeroComponent` 的副作用，因为后者出现在前者的模板中。 英雄元素（`heroEl`)的查询语句仍然可以在测试 DOM 中找到它，尽管元素树比以前更深。

这些测试本身和它们的孤立版本几乎相同：

app/dashboard/dashboard-hero.component.spec.ts (test-host)

```
it('should display hero name', () => {
  const expectedPipedName = testHost.hero.name.toUpperCase();
  expect(heroEl.textContent).toContain(expectedPipedName);
});

it('should raise selected event when clicked', () => {
  click(heroEl);
  // selected hero should be the same data bound hero
  expect(testHost.selectedHero).toBe(testHost.hero);
});
```

只有 selected 事件的测试不一样。它确保被选择的 `DashboardHeroComponent` 英雄确实通过事件绑定被传递到宿主组件。

路由组件

所谓**路由组件**就是指会要求 `Router` 导航到其它组件的组件。 `DashboardComponent` 就是一个**路由组件**，因为用户可以通过点击仪表盘中的某个**英雄按钮**来导航到 `HeroDetailComponent`。

路由确实很复杂。 测试 `DashboardComponent` 看上去有点令人生畏，因为它牵扯到和 `HeroService` 一起注入进来的 `Router`。

app/dashboard/dashboard.component.ts (constructor)

```
constructor(
  private router: Router,
  private heroService: HeroService) {
}
```

使用间谍来 Mock `HeroService` 是一个熟悉的故事。 但是 `Router` 的 API 很复杂，并且与其它服务和应用的前置条件纠缠在一起。它应该很难进行 Mock 吧?

庆幸的是，在这个例子中不会，因为 `DashboardComponent` 并没有深度使用 `Router`。

app/dashboard/dashboard.component.ts (goToDetail)

```
gotoDetail(hero: Hero) {
  let url = `/heroes/${hero.id}`;
  this.router.navigateByUrl(url);
}
```

这是**路由组件**中的通例。 一般来说，你应该测试组件而不是路由器，应该只关心组件有没有根据给定的条件导航到正确的地址。

为**这个组件**的测试套件提供路由器的间谍就像它提供 `HeroService` 的间谍一样简单。

app/dashboard/dashboard.component.spec.ts (spies)

```
const routerSpy = jasmine.createSpyObj('Router', ['navigateByUrl']);
const heroServiceSpy = jasmine.createSpyObj('HeroService', ['getHeroes']);

TestBed.configureTestingModule({
  providers: [
    { provide: HeroService, useValue: heroServiceSpy },
    { provide: Router,      useValue: routerSpy }
  ]
})
```

下面这个测试会点击正在显示的英雄，并确认 `Router.navigateByUrl` 曾用所期待的 URL 调用过。

app/dashboard/dashboard.component.spec.ts (navigate test)

```
it('should tell ROUTER to navigate when hero clicked', () => {

  heroClick(); // trigger click on first inner <div class="hero">

  // args passed to router.navigateByUrl() spy
  const spy = router.navigateByUrl as jasmine.Spy;
  const navArgs = spy.calls.first().args[0];

  // expecting to navigate to id of the component's first hero
  const id = comp.heroes[0].id;
  expect(navArgs).toBe('/heroes/' + id,
    'should nav to HeroDetail for first hero');
});
```

路由目标组件

路由目标组件是指 `Router` 导航到的目标。 它测试起来可能很复杂，特别是当路由到的这个组件**包含参数**的时候。 `HeroDetailComponent` 就是一个**路由目标组件**，它是某个路由定义指向的目标。

当用户点击**仪表盘**中的英雄时，`DashboardComponent` 会要求 `Router` 导航到 `heroes/:id`。 `:id` 是一个路由参数，它的值就是所要编辑的英雄的 `id`。

该 `Router` 会根据那个 URL 匹配到一个指向 `HeroDetailComponent` 的路由。 它会创建一个带有路由信息的 `ActivatedRoute` 对象，并把它注入到一个 `HeroDetailComponent` 的新实例中。

下面是 `HeroDetailComponent` 的构造函数：

app/hero/hero-detail.component.ts (constructor)

```
constructor(
  private heroDetailService: HeroDetailService,
  private route:  ActivatedRoute,
  private router: Router) {
}
```

`HeroDetailComponent` 组件需要一个 `id` 参数，以便通过 `HeroDetailService` 获取相应的英雄。 该组件只能从 `ActivatedRoute.paramMap` 属性中获取这个 `id`，这个属性是一个 `Observable`。

它不能仅仅引用 `ActivatedRoute.paramMap` 的 `id` 属性。 该组件不得不**订阅** `ActivatedRoute.paramMap` 这个可观察对象，要做好它在生命周期中随时会发生变化的准备。

app/hero/hero-detail.component.ts (ngOnInit)

```
ngOnInit(): void {
  // get hero when `id` param changes
  this.route.paramMap.subscribe(pmap => this.getHero(pmap.get('id')));
}
```

> 路由与导航一章中详细讲解了 `ActivatedRoute.paramMap`。

通过操纵注入到组件构造函数中的这个 `ActivatedRoute`，测试可以探查 `HeroDetailComponent` 是如何对不同的 `id` 参数值做出响应的。

你已经知道了如何给 `Router` 和数据服务安插间谍。

不过对于 `ActivatedRoute`，你要采用另一种方式，因为：

- 在测试期间，`paramMap` 会返回一个能发出多个值的 `Observable`。
- 你需要路由器的辅助函数 `convertToParamMap()` 来创建 `ParamMap`。
- 针对**路由目标组件**的其它测试需要一个 `ActivatedRoute` 的测试替身。

这些差异表明你需要一个可复用的桩类（stub）。

ActivatedRouteStub

下面的 `ActivatedRouteStub` 类就是作为 `ActivatedRoute` 类的测试替身使用的。

testing/activated-route-stub.ts (ActivatedRouteStub)

```
import { convertToParamMap, ParamMap, Params } from '@angular/router';
import { ReplaySubject } from 'rxjs';

/**
 * An ActivateRoute test double with a `paramMap` observable.
 * Use the `setParamMap()` method to add the next `paramMap` value.
 */
export class ActivatedRouteStub {
  // Use a ReplaySubject to share previous values with subscribers
  // and pump new values into the `paramMap` observable
  private subject = new ReplaySubject<ParamMap>();

  constructor(initialParams?: Params) {
    this.setParamMap(initialParams);
  }

  /** The mock paramMap observable */
  readonly paramMap = this.subject.asObservable();

  /** Set the paramMap observables's next value */
  setParamMap(params?: Params) {
    this.subject.next(convertToParamMap(params));
  };
}
```

考虑把这类辅助函数放进一个紧邻 `app` 文件夹的 `testing` 文件夹。 这个例子把 `ActivatedRouteStub` 放在了 `testing/activated-route-stub.ts` 中。

> 可以考虑使用宝石测试库来为此测试桩编写一个更强力的版本。

使用 `ActivatedRouteStub` 进行测试

下面的测试程序是演示组件在被观察的 `id` 指向现有英雄时的行为：

app/hero/hero-detail.component.spec.ts (existing id)

```
describe('when navigate to existing hero', () => {
  let expectedHero: Hero;

  beforeEach(async(() => {
    expectedHero = firstHero;
    activatedRoute.setParamMap({ id: expectedHero.id });
    createComponent();
  }));

  it('should display that hero\'s name', () => {
    expect(page.nameDisplay.textContent).toBe(expectedHero.name);
  });
});
```

> `createComponent()` 方法和 `page` 对象会在稍后进行讨论。 不过目前，你只要凭直觉来理解就行了。

当找不到 `id` 的时候，组件应该重新路由到 `HeroListComponent`。

测试套件的准备代码提供了一个和前面一样的路由器间谍，它会充当路由器的角色，而不用发起实际的导航。

这个测试中会期待该组件尝试导航到 `HeroListComponent`。

app/hero/hero-detail.component.spec.ts (bad id)

```
describe('when navigate to non-existent hero id', () => {
  beforeEach(async(() => {
    activatedRoute.setParamMap({ id: 99999 });
    createComponent();
  }));

  it('should try to navigate back to hero list', () => {
    expect(page.gotoListSpy.calls.any()).toBe(true, 'comp.gotoList called');
    expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');
  });
});
```

虽然本应用没有在缺少 `id` 参数的时候，继续导航到 `HeroDetailComponent` 的路由，但是，将来它可能会添加这样的路由。 当没有 `id` 时，该组件应该作出合理的反应。

在本例中，组件应该创建和显示新英雄。 新英雄的 `id` 为零，`name` 为空。本测试程序确认组件是按照预期的这样做的：

app/hero/hero-detail.component.spec.ts (no id)

```
describe('when navigate with no hero id', () => {
  beforeEach(async( createComponent ));

  it('should have hero.id === 0', () => {
    expect(component.hero.id).toBe(0);
  });

  it('should display empty hero name', () => {
    expect(page.nameDisplay.textContent).toBe('');
  });
});
```

对嵌套组件的测试

组件的模板中通常还会有嵌套组件，嵌套组件的模板还可能包含更多组件。

这棵组件树可能非常深，并且大多数时候在测试这棵树顶部的组件时，这些嵌套的组件都无关紧要。

比如，`AppComponent` 会显示一个带有链接及其 `RouterLink` 指令的导航条。

app/app.component.html

```
<app-banner></app-banner>
<app-welcome></app-welcome>
<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
  <a routerLink="/about">About</a>
</nav>
<router-outlet></router-outlet>
```

虽然 `AppComponent` **类**是空的，不过，由于稍后解释的原因，你可能会希望写个单元测试来确认这些链接是否正确使用了 `RouterLink` 指令。

要想验证这些链接，你不必用 `Router` 进行导航，也不必使用 `<router-outlet>` 来指出 `Router` 应该把**路由目标组件**插入到什么地方。

而 `BannerComponent` 和 `WelcomeComponent` （写作 `<app-banner>` 和 `<app-welcome>`）也同样风马牛不相及。

然而，任何测试，只要能在 DOM 中创建 `AppComponent`，也就同样能创建这三个组件的实例。如果要创建它们，你就要配置 `TestBed`。

如果你忘了声明它们，Angular 编译器就无法在 `AppComponent` 模板中识别出 `<app-banner>`、`<app-welcome>` 和 `<router-outlet>` 标记，并抛出一个错误。

如果你声明的这些都是真实的组件，那么也同样要声明**它们**的嵌套组件，并要为这棵组件树中的**任何**组件提供要注入的**所有**服务。

如果只是想回答有关链接的一些简单问题，做这些显然就太多了。

本节会讲减少此类准备工作的两项技术。 单独使用或组合使用它们，可以让这些测试聚焦于要测试的主要组件上。

对不需要的组件提供桩（stub）

这项技术中，你要为那些在测试中无关紧要的组件或指令创建和声明一些测试桩。

app/app.component.spec.ts (stub declaration)

```
@Component({selector: 'app-banner', template: ''})
class BannerStubComponent {}

@Component({selector: 'router-outlet', template: ''})
class RouterOutletStubComponent { }

@Component({selector: 'app-welcome', template: ''})
class WelcomeStubComponent {}
```

这些测试桩的选择器要和其对应的真实组件一致，但其模板和类是空的。

然后在 `TestBed` 的配置中那些真正有用的组件、指令、管道之后声明它们。

app/app.component.spec.ts (TestBed stubs)

```
TestBed.configureTestingModule({
  declarations: [
    AppComponent,
    RouterLinkDirectiveStub,
    BannerStubComponent,
    RouterOutletStubComponent,
    WelcomeStubComponent
  ]
})
```

`AppComponent` 是该测试的主角，因此当然要用它的真实版本。

而 `RouterLinkDirectiveStub`（稍后讲解）是一个真实的 `RouterLink` 的测试版，它能帮你对链接进行测试。

其它都是测试桩。

NO_ERRORS_SCHEMA

第二种办法就是把 `NO_ERRORS_SCHEMA` 添加到 `TestBed.schemas` 的元数据中。

app/app.component.spec.ts (NO_ERRORS_SCHEMA)

```
TestBed.configureTestingModule({
  declarations: [
    AppComponent,
    RouterLinkDirectiveStub
  ],
  schemas: [ NO_ERRORS_SCHEMA ]
})
```

`NO_ERRORS_SCHEMA` 会要求 Angular 编译器忽略不认识的那些元素和属性。

编译器将会识别出 `<app-root>` 元素和 `RouterLink` 属性，因为你在 `TestBed` 的配置中声明了相应的 `AppComponent` 和 `RouterLinkDirectiveStub`。

但编译器在遇到 `<app-banner>`、`<app-welcome>` 或 `<router-outlet>` 时不会报错。 它只会把它们渲染成空白标签，而浏览器会忽略这些标签。

你不用再提供桩组件了。

同时使用这两项技术

这些是进行**浅层**测试要用到的技术，之所以叫浅层测试是因为只包含本测试所关心的这个组件模板中的元素。

`NO_ERRORS_SCHEMA` 方法在这两者中比较简单，但也不要过度使用它。

`NO_ERRORS_SCHEMA` 还会阻止编译器告诉你因为的疏忽或拼写错误而缺失的组件和属性。 你如果人工找出这些 bug 可能要浪费几个小时，但编译器可以立即捕获它们。

桩组件方式还有其它优点。 虽然**这个**例子中的桩是空的，但你如果想要和它们用某种形式互动，也可以给它们一些裁剪过的模板和类。

在实践中，你可以在准备代码中组合使用这两种技术，例子如下：

app/app.component.spec.ts (mixed setup)

```
TestBed.configureTestingModule({
  declarations: [
    AppComponent,
    BannerStubComponent,
    RouterLinkDirectiveStub
  ],
  schemas: [ NO_ERRORS_SCHEMA ]
})
```

Angular 编译器会为 `<app-banner>` 元素创建 `BannerComponentStub`，并把 `RouterLinkStubDirective` 应用到带有 `routerLink` 属性的链接上，不过它会忽略 `<app-welcome>` 和 `<router-outlet>` 标签。

带有 `RouterLink` 的组件

真实的 `RouterLinkDirective` 太复杂了，而且与 `RouterModule` 中的其它组件和指令有着千丝万缕的联系。要在准备阶段 Mock 它以及在测试中使用它具有一定的挑战性。

这段范例代码中的 `RouterLinkDirectiveStub` 用一个代用品替换了真实的指令，这个代用品用来验证 `AppComponent` 中所用链接的类型。

testing/router-link-directive-stub.ts (RouterLinkDirectiveStub)

```
@Directive({
  selector: '[routerLink]',
  host: { '(click)': 'onClick()' }
})
export class RouterLinkDirectiveStub {
  @Input('routerLink') linkParams: any;
  navigatedTo: any = null;

  onClick() {
    this.navigatedTo = this.linkParams;
  }
}
```

这个 URL 被绑定到了 `[routerLink]` 属性，它的值流入了该指令的 `linkParams` 属性。

它的元数据中的 `host` 属性把宿主元素（即 `AppComponent` 中的 `<a>` 元素）的 `click` 事件关联到了这个桩指令的 `onClick` 方法。

点击这个链接应该触发 `onClick()` 方法，其中会设置该桩指令中的警示器属性 `navigatedTo`。 测试中检查 `navigatedTo` 以确认点击该链接确实如预期的那样根据路由定义设置了该属性。

> 路由器的配置是否正确和是否能按照那些路由定义进行导航，是测试中一组独立的问题。

`By.directive` 与注入的指令

再一步配置触发了数据绑定的初始化，获取导航链接的引用：

app/app.component.spec.ts (test setup)

```
beforeEach(() => {
  fixture.detectChanges(); // trigger initial data binding

  // find DebugElements with an attached RouterLinkStubDirective
  linkDes = fixture.debugElement
    .queryAll(By.directive(RouterLinkDirectiveStub));

  // get attached link directive instances
  // using each DebugElement's injector
  routerLinks = linkDes.map(de => de.injector.get(RouterLinkDirectiveStub));
});
```

有三点特别重要：

1. 你可以使用 `By.directive` 来定位一个带附属指令的链接元素。
2. 该查询返回包含了匹配元素的 `DebugElement` 包装器。
3. 每个 `DebugElement` 都会导出该元素中的一个依赖注入器，其中带有指定的指令实例。

`AppComponent` 中要验证的链接如下：

app/app.component.html (navigation links)

```
<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
  <a routerLink="/about">About</a>
</nav>
```

下面这些测试用来确认那些链接是否如预期般连接到了 `RouterLink` 指令中：

app/app.component.spec.ts (selected tests)

```
it('can get RouterLinks from template', () => {
  expect(routerLinks.length).toBe(3, 'should have 3 routerLinks');
  expect(routerLinks[0].linkParams).toBe('/dashboard');
  expect(routerLinks[1].linkParams).toBe('/heroes');
  expect(routerLinks[2].linkParams).toBe('/about');
});

it('can click Heroes link in template', () => {
  const heroesLinkDe = linkDes[1];   // heroes link DebugElement
  const heroesLink = routerLinks[1]; // heroes link directive

  expect(heroesLink.navigatedTo).toBeNull('should not have navigated yet');

  heroesLinkDe.triggerEventHandler('click', null);
  fixture.detectChanges();

  expect(heroesLink.navigatedTo).toBe('/heroes');
});
```

> 其实**这个例子中**的“click”测试误入歧途了。 它测试的重点其实是 `RouterLinkDirectiveStub` ，而不是该组件。 这是写桩指令时常见的错误。
>
> 在本章中，它有存在的必要。 它演示了如何在不涉及完整路由器机制的情况下，如何找到 `RouterLink` 元素、点击它并检查结果。 要测试更复杂的组件，你可能需要具备这样的能力，能改变视图和重新计算参数，或者当用户点击链接时，有能力重新安排导航选项。

这些测试有什么优点?

用 `RouterLink` 的桩指令进行测试可以确认带有链接和 outlet 的组件的设置的正确性，确认组件有应该有的链接，确认它们都指向了正确的方向。 这些测试程序不关心用户点击链接时，也不关心应用是否会成功的导航到目标组件。

对于这些有限的测试目标，使用 RouterLink 桩指令和 RouterOutlet 桩组件 是最佳选择。 依靠真正的路由器会让它们很脆弱。 它们可能因为与组件无关的原因而失败。 例如，一个导航守卫可能防止没有授权的用户访问 `HeroListComponent`。 这并不是 `AppComponent` 的过错，并且无论该组件怎么改变都无法修复这个失败的测试程序。

不同的测试程序可以探索在不同条件下（比如像检查用户是否认证），该应用是否和期望的那样导航。

> 未来对本章的更新将介绍如何使用 `RouterTestingModule` 来编写这样的测试程序。

使用页面（page）对象

`HeroDetailComponent` 是带有标题、两个英雄字段和两个按钮的简单视图。

Narco Details

id: 12

name: Narco

Save Cancel

但即使是这么简单的表单，其模板中也涉及到不少复杂性。

app/hero/hero-detail.component.html

```
<div *ngIf="hero">
  <h2><span>{{hero.name | titlecase}}</span> Details</h2>
  <div>
    <label>id: </label>{{hero.id}}</div>
  <div>
    <label for="name">name: </label>
    <input id="name" [(ngModel)]="hero.name" placeholder="name" />
  </div>
  <button (click)="save()">Save</button>
  <button (click)="cancel()">Cancel</button>
</div>
```

这些供练习用的组件需要 ……

- 等获取到英雄之后才能让元素出现在 DOM 中。
- 一个对标题文本的引用。
- 一个对 name 输入框的引用，以便对它进行探查和修改。
- 引用两个按钮，以便点击它们。
- 为组件和路由器的方法安插间谍。

即使是像这样一个很小的表单，也能产生令人疯狂的错综复杂的条件设置和 CSS 元素选择。

可以使用 `Page` 类来征服这种复杂性。`Page` 类可以处理对组件属性的访问，并对设置这些属性的逻辑进行封装。

下面是一个供 `hero-detail.component.spec.ts` 使用的 `Page` 类

app/hero/hero-detail.component.spec.ts (Page)

```
class Page {
  // getter properties wait to query the DOM until called.
  get buttons()     { return this.queryAll<HTMLButtonElement>('button'); }
  get saveBtn()     { return this.buttons[0]; }
  get cancelBtn()   { return this.buttons[1]; }
  get nameDisplay() { return this.query<HTMLElement>('span'); }
  get nameInput()   { return this.query<HTMLInputElement>('input'); }

  gotoListSpy: jasmine.Spy;
  navigateSpy:  jasmine.Spy;

  constructor(fixture: ComponentFixture<HeroDetailComponent>) {
    // get the navigate spy from the injected router spy object
    const routerSpy = <any> fixture.debugElement.injector.get(Router);
    this.navigateSpy = routerSpy.navigate;

    // spy on component's `gotoList()` method
    const component = fixture.componentInstance;
    this.gotoListSpy = spyOn(component, 'gotoList').and.callThrough();
  }

  //// query helpers ////
  private query<T>(selector: string): T {
    return fixture.nativeElement.querySelector(selector);
  }

  private queryAll<T>(selector: string): T[] {
    return fixture.nativeElement.querySelectorAll(selector);
  }
}
```

现在，用来操作和检查组件的重要钩子都被井然有序的组织起来了，可以通过 `page` 实例来使用它们。

`createComponent` 方法创建 `page`，在 `hero` 到来时，自动填补空白。

app/hero/hero-detail.component.spec.ts (createComponent)

```
/** Create the HeroDetailComponent, initialize it, set test variables  */
function createComponent() {
  fixture = TestBed.createComponent(HeroDetailComponent);
  component = fixture.componentInstance;
  page = new Page(fixture);

  // 1st change detection triggers ngOnInit which gets a hero
  fixture.detectChanges();
  return fixture.whenStable().then(() => {
    // 2nd change detection displays the async-fetched hero
    fixture.detectChanges();
  });
}
```

前面小节中的 `HeroDetailComponent` 测试示范了如何 `createComponent`，而 `page` 让这些测试保持简短而富有表达力。 而且还不用分心：不用等待承诺被解析，不必在 DOM 中找出元素的值才能进行比较。

还有更多的 `HeroDetailComponent` 测试可以证明这一点。

app/hero/hero-detail.component.spec.ts (selected tests)

```
it('should display that hero\'s name', () => {
  expect(page.nameDisplay.textContent).toBe(expectedHero.name);
});

it('should navigate when click cancel', () => {
  click(page.cancelBtn);
  expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');
});

it('should save when click save but not navigate immediately', () => {
  // Get service injected into component and spy on its`saveHero` method.
  // It delegates to fake `HeroService.updateHero` which delivers a safe test
result.
  const hds = fixture.debugElement.injector.get(HeroDetailService);
  const saveSpy = spyOn(hds, 'saveHero').and.callThrough();

  click(page.saveBtn);
  expect(saveSpy.calls.any()).toBe(true, 'HeroDetailService.save called');
  expect(page.navigateSpy.calls.any()).toBe(false, 'router.navigate not called');
});

it('should navigate when click save and save resolves', fakeAsync(() => {
  click(page.saveBtn);
  tick(); // wait for async save to complete
  expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');
}));

it('should convert hero name to Title Case', () => {
  // get the name's input and display elements from the DOM
  const hostElement = fixture.nativeElement;
  const nameInput: HTMLInputElement = hostElement.querySelector('input');
  const nameDisplay: HTMLElement = hostElement.querySelector('span');

  // simulate user entering a new name into the input box
  nameInput.value = 'quick BROWN  fOx';

  // dispatch a DOM event so that Angular learns of input value change.
  nameInput.dispatchEvent(newEvent('input'));

  // Tell Angular to update the display binding through the title pipe
  fixture.detectChanges();
```

```
  expect(nameDisplay.textContent).toBe('Quick Brown  Fox');
});
```

调用 `compileComponents()`

> 如果你**只想**使用 CLI 的 `ng test` 命令来运行测试，那么可以忽略这一节。

如果你在非 CLI 环境中运行测试，这些测试可能会报错，错误信息如下：

```
Error: This test module uses the component BannerComponent
which is using a "templateUrl" or "styleUrls", but they were never compiled.
Please call "TestBed.compileComponents" before your test.
```

问题的根源在于这个测试中至少有一个组件引用了外部模板或外部 CSS 文件，就像下面这个 `BannerComponent` 所示：

app/banner/banner-external.component.ts (external template & css)

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

@Component({
  selector: 'app-banner',
  templateUrl: './banner-external.component.html',
  styleUrls:  ['./banner-external.component.css']
})
export class BannerComponent {
  title = 'Test Tour of Heroes';
}
```

当 `TestBed` 视图创建组件时，这个测试失败了：

app/banner/banner.component.spec.ts (setup that fails)

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  });
  fixture = TestBed.createComponent(BannerComponent);
});
```

回想一下，这个应用从未编译过。 所以当你调用 `createComponent()` 的时候， `TestBed` 就会进行隐式编译。

当它的源码都在内存中的时候，这样做没问题。 不过 `BannerComponent` 需要一些外部文件，编译时必须从文件系统中读取它，而这是一个天生的**异步**操作。

如果 `TestBed` 继续执行，这些测试就会继续运行，并在编译器完成这些异步工作之前导致莫名其妙的失败。

这些错误信息告诉你要使用 `compileComponents()` 进行显式的编译。

`compileComponents()` 是异步的

你必须在异步测试函数中调用 `compileComponents()`。

> 如果你忘了把测试函数标为异步的（比如忘了像稍后的代码中那样使用 `async()`），就会看到下列错误。
>
> ```
> Error: ViewDestroyedError: Attempt to use a destroyed view
> ```

典型的做法是把准备逻辑拆成两个独立的 `beforeEach()` 函数：

1. 异步的 `beforeEach()` 负责编译组件
2. 同步的 `beforeEach()` 负责执行其余的准备代码。

要想使用这种模式，就要和其它符号一起从测试库中导入 `async()` 辅助函数。

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
```

异步的 `beforeEach`

像下面这样编写第一个异步的 `beforeEach`。

app/banner/banner-external.component.spec.ts (async beforeEach)

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  })
  .compileComponents();  // compile template and css
}));
```

`async()` 辅助函数接受一个无参函数，其内容是准备代码。

`TestBed.configureTestingModule()` 方法返回 `TestBed` 类，所以你可以链式调用其它 `TestBed` 中的静态方法，比如 `compileComponents()`。

在这个例子中，`BannerComponent` 是仅有的待编译组件。 其它例子中可能会使用多个组件来配置测试模块，并且可能引入某些具有其它组件的应用模块。 它们中的任何一个都可能需要外部文件。

`TestBed.compileComponents` 方法会异步编译测试模块中配置过的所有组件。

> 在调用了 `compileComponents()` 之后就不能再重新配置 `TestBed` 了。

调用 `compileComponents()` 会关闭当前的 `TestBed` 实例，不再允许进行配置。 你不能再调用任何 `TestBed` 中的配置方法，既不能调 `configureTestingModule()`，也不能调用任何 `override...` 方法。如果你试图这么做，`TestBed` 就会抛出错误。

确保 `compileComponents()` 是调用 `TestBed.createComponent()` 之前的最后一步。

同步的 `beforeEach`

第二个同步 `beforeEach()` 的例子包含剩下的准备步骤， 包括创建组件和查询那些要检查的元素。

app/banner/banner-external.component.spec.ts (synchronous beforeEach)

```
beforeEach(() => {
  fixture = TestBed.createComponent(BannerComponent);
  component = fixture.componentInstance; // BannerComponent test instance
  h1 = fixture.nativeElement.querySelector('h1');
});
```

测试运行器（runner）会先等待第一个异步 `beforeEach` 函数执行完再调用第二个。

整理过的准备代码

你可以把这两个 `beforeEach()` 函数重整成一个异步的 `beforeEach()`。

`compileComponents()` 方法返回一个承诺，所以你可以通过把同步代码移到 `then(...)` 回调中， 以便在编译完成**之后** 执行那些同步准备任务。

app/banner/banner-external.component.spec.ts (one beforeEach)

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  })
  .compileComponents()
  .then(() => {
    fixture = TestBed.createComponent(BannerComponent);
    component = fixture.componentInstance;
    h1 = fixture.nativeElement.querySelector('h1');
  });
}));
```

`compileComponents()` 是无害的

在不需要 `compileComponents()` 的时候调用它也不会有害处。

虽然在运行 `ng test` 时永远都不需要调用 `compileComponents()`，但 CLI 生成的组件测试文件还是会调用它。

但这篇指南中的这些测试只会在必要时才调用 `compileComponents`。

准备模块的 `imports`

此前的组件测试程序使用了一些 `declarations` 来配置模块，就像这样：

app/dashboard/dashboard-hero.component.spec.ts (configure TestBed)

```
TestBed.configureTestingModule({
  declarations: [ DashboardHeroComponent ]
})
```

`DashbaordComponent` 非常简单。它不需要帮助。 但是更加复杂的组件通常依赖其它组件、指令、管道和提供商， 所以这些必须也被添加到测试模块中。

幸运的是，`TestBed.configureTestingModule` 参数与传入 `@NgModule` 装饰器的元数据一样，也就是所你也可以指定 `providers` 和 `imports`.

虽然 `HeroDetailComponent` 很小，结构也很简单，但是它需要很多帮助。 除了从默认测试模块 `CommonModule` 中获得的支持，它还需要：

- `FormsModule` 里的 `NgModel` 和其它，来进行双向数据绑定
- `shared` 目录里的 `TitleCasePipe`
- 一些路由器服务（测试程序将 stub 伪造它们）
- 英雄数据访问服务（同样被 stub 伪造了）

一种方法是在测试模块中一一配置，就像这样：

app/hero/hero-detail.component.spec.ts (FormsModule setup)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports:      [ FormsModule ],
    declarations: [ HeroDetailComponent, TitleCasePipe ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService,    useClass: TestHeroService },
      { provide: Router,         useValue: routerSpy},
    ]
  })
  .compileComponents();
}));
```

> 注意，`beforeEach()` 是异步的，它调用 `TestBed.compileComponents` 是因为 `HeroDetailComponent` 有外部模板和 CSS 文件。
>
> 如前面的调用 `compileComponents()` 中所解释的那样，这些测试可以运行在非 CLI 环境下，那里 Angular 并不会在浏览器中编译它们。

导入共享模块

因为很多应用组件都需要 `FormsModule` 和 `TitleCasePipe`，所以开发者创建了 `SharedModule` 来把它们及其它常用的部分组合在一起。

这些测试配置也可以使用 `SharedModule`，如下所示：

app/hero/hero-detail.component.spec.ts (SharedModule setup)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports:      [ SharedModule ],
    declarations: [ HeroDetailComponent ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService,    useClass: TestHeroService },
      { provide: Router,         useValue: routerSpy},
    ]
  })
  .compileComponents();
}));
```

它的导入声明少一些（未显示），稍微干净一些，小一些。

导入特性模块

`HeroDetailComponent` 是 `HeroModule` 这个特性模块的一部分，它聚合了更多相互依赖的片段，包括 `SharedModule`。 试试下面这个导入了 `HeroModule` 的测试配置：

app/hero/hero-detail.component.spec.ts (HeroModule setup)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports:   [ HeroModule ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService,    useClass: TestHeroService },
      { provide: Router,         useValue: routerSpy},
    ]
  })
  .compileComponents();
}));
```

这样特别清爽。只有 `providers` 里面的测试替身被保留。连 `HeroDetailComponent` 声明都消失了。

事实上，如果你试图声明它，Angular 就会抛出一个错误，因为 `HeroDetailComponent` 同时声明在了 `HeroModule` 和 `TestBed` 创建的 `DynamicTestModule` 中。

> 如果模块中有很多共同依赖，并且该模块很小（这也是特性模块的应有形态），那么直接导入组件的特性模块可以成为配置这些测试的简易方式。

改写组件的服务提供商

`HeroDetailComponent` 提供自己的 `HeroDetailService` 服务。

app/hero/hero-detail.component.ts (prototype)

```
@Component({
  selector:    'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls:  ['./hero-detail.component.css' ],
  providers:  [ HeroDetailService ]
})
export class HeroDetailComponent implements OnInit {
  constructor(
    private heroDetailService: HeroDetailService,
    private route:  ActivatedRoute,
    private router: Router) {
  }
}
```

在 `TestBed.configureTestingModule` 的 `providers` 中 stub 伪造组件的 `HeroDetailService` 是不可行的。这些是测试模块的提供商，而非组件的。组件级别的供应商应该在fixture 级别准备的依赖注入器。

Angular 会使用自己的注入器来创建这些组件，这个注入器是夹具的注入器的子注入器。 它使用这个子注入器注册了该组件服务提供商（这里是 `HeroDetailService` ）。

测试没办法从测试夹具的注入器中获取子注入器中的服务，而 `TestBed.configureTestingModule` 也没法配置它们。

Angular 始终都在创建真实 `HeroDetailService` 的实例。

> 如果 `HeroDetailService` 向远程服务器发出自己的 XHR 请求，这些测试可能会失败或者超时。 这个远程服务器可能根本不存在。

> 幸运的是，`HeroDetailService` 将远程数据访问的责任交给了注入进来的 `HeroService`。
>
> app/hero/hero-detail.service.ts (prototype)
>
> ```
> @Injectable()
> export class HeroDetailService {
>   constructor(private heroService: HeroService) {  }
> /* . . . */
> }
> ```
>
> 前面的测试配置使用 `TestHeroService` 替换了真实的 `HeroService`，它拦截了发往服务器的请求，并伪造了服务器的响应。

如果你没有这么幸运怎么办？如果伪造 `HeroService` 很难怎么办？如果 `HeroDetailService` 自己发出服务器请求怎么办？

`TestBed.overrideComponent` 方法可以将组件的 `providers` 替换为容易管理的测试替身，参见下面的变体准备代码：

app/hero/hero-detail.component.spec.ts (Override setup)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports:   [ HeroModule ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: Router,         useValue: routerSpy},
    ]
  })

  // Override component's own provider
  .overrideComponent(HeroDetailComponent, {
    set: {
      providers: [
        { provide: HeroDetailService, useClass: HeroDetailServiceSpy }
      ]
    }
  })

  .compileComponents();
}));
```

注意，`TestBed.configureTestingModule` 不再提供（伪造的）`HeroService`，因为并不需要。

`overrideComponent` 方法

注意这个 `overrideComponent` 方法。

app/hero/hero-detail.component.spec.ts (overrideComponent)

```
.overrideComponent(HeroDetailComponent, {
  set: {
    providers: [
      { provide: HeroDetailService, useClass: HeroDetailServiceSpy }
    ]
  }
})
```

它接受两个参数：要改写的组件类型（`HeroDetailComponent`），以及用于改写的元数据对象。 用于改写的元数据对象是一个泛型，其定义如下：

```
type MetadataOverride = {
    add?: T;
    remove?: T;
    set?: T;
  };
```

元数据重载对象可以添加和删除元数据属性的项目，也可以彻底重设这些属性。 这个例子重新设置了组件的 `providers` 元数据。

这个类型参数，`T`，是你会传递给 `@Component` 装饰器的元数据的类型。

```
selector?: string;
template?: string;
templateUrl?: string;
providers?: any[];
...
```

提供 间谍桩 (`HeroDetailServiceSpy`)

这个例子把组件的 `providers` 数组完全替换成了一个包含 `HeroDetailServiceSpy` 的新数组。

`HeroDetailServiceSpy` 是实际 `HeroDetailService` 服务的桩版本，它伪造了该服务的所有必要特性。 但它既不需要注入也不会委托给低层的 `HeroService` 服务，因此不用为 `HeroService` 提供测试替身。

通过对该服务的方法进行刺探，`HeroDetailComponent` 的关联测试将会对 `HeroDetailService` 是否被调用过进行断言。 因此，这个桩类会把它的方法实现为刺探方法：

app/hero/hero-detail.component.spec.ts (HeroDetailServiceSpy)

```
class HeroDetailServiceSpy {
  testHero: Hero = {id: 42, name: 'Test Hero' };

  /* emit cloned test hero */
  getHero = jasmine.createSpy('getHero').and.callFake(
    () => asyncData(Object.assign({}, this.testHero))
  );

  /* emit clone of test hero, with changes merged in */
  saveHero = jasmine.createSpy('saveHero').and.callFake(
    (hero: Hero) => asyncData(Object.assign(this.testHero, hero))
  );
}
```

改写测试

现在，测试程序可以通过操控 stub 的 `testHero`，直接控制组件的英雄，并确保服务的方法被调用过。

app/hero/hero-detail.component.spec.ts (override tests)

```
let hdsSpy: HeroDetailServiceSpy;

beforeEach(async(() => {
  createComponent();
  // get the component's injected HeroDetailServiceSpy
  hdsSpy = fixture.debugElement.injector.get(HeroDetailService) as any;
}));

it('should have called `getHero`', () => {
  expect(hdsSpy.getHero.calls.count()).toBe(1, 'getHero called once');
});

it('should display stub hero\'s name', () => {
  expect(page.nameDisplay.textContent).toBe(hdsSpy.testHero.name);
});

it('should save stub hero change', fakeAsync(() => {
  const origName = hdsSpy.testHero.name;
  const newName = 'New Name';

  page.nameInput.value = newName;
  page.nameInput.dispatchEvent(newEvent('input')); // tell Angular

  expect(component.hero.name).toBe(newName, 'component hero has new name');
  expect(hdsSpy.testHero.name).toBe(origName, 'service hero unchanged before save');

  click(page.saveBtn);
  expect(hdsSpy.saveHero.calls.count()).toBe(1, 'saveHero called once');

  tick(); // wait for async save to complete
  expect(hdsSpy.testHero.name).toBe(newName, 'service hero has new name after
save');
  expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');
}));
```

更多的改写

`TestBed.overrideComponent` 方法可以在相同或不同的组件中被反复调用。 `TestBed` 还提供了类似的 `overrideDirective`、`overrideModule` 和 `overridePipe` 方法，用来深入并重载这些其它类的部件。

自己探索这些选项和组合。

属性型指令的测试

属性指令修改元素、组件和其它指令的行为。正如它们的名字所示，它们是作为宿主元素的属性来被使用的。

本例子应用的 `HighlightDirective` 使用数据绑定的颜色或者默认颜色来设置元素的背景色。 它同时设置元素的 `customProperty` 属性为 `true`，这里仅仅是为了显示它能这么做而已，并无其它原因。

app/shared/highlight.directive.ts

```
import { Directive, ElementRef, Input, OnChanges } from '@angular/core';

@Directive({ selector: '[highlight]' })
/** Set backgroundColor for the attached element to highlight color
 *  and set the element's customProperty to true */
export class HighlightDirective implements OnChanges {

  defaultColor =  'rgb(211, 211, 211)'; // lightgray

  @Input('highlight') bgColor: string;

  constructor(private el: ElementRef) {
    el.nativeElement.style.customProperty = true;
  }

  ngOnChanges() {
    this.el.nativeElement.style.backgroundColor = this.bgColor || this.defaultColor;
  }
}
```

它的使用贯穿整个应用，也许最简单的使用在 `AboutComponent` 里：

app/about/about.component.ts

```
import { Component } from '@angular/core';
@Component({
  template: `
  <h2 highlight="skyblue">About</h2>
  <h3>Quote of the day:</h3>
  <twain-quote></twain-quote>
  `
})
export class AboutComponent { }
```

要想在 `AboutComponent` 中测试 `HighlightDirective` 的具体用法，只要使用在“浅层测试”部分用过的技术即可。

app/about/about.component.spec.ts

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ AboutComponent, HighlightDirective],
    schemas:      [ NO_ERRORS_SCHEMA ]
  })
  .createComponent(AboutComponent);
  fixture.detectChanges(); // initial binding
});

it('should have skyblue <h2>', () => {
  const h2: HTMLElement = fixture.nativeElement.querySelector('h2');
  const bgColor = h2.style.backgroundColor;
  expect(bgColor).toBe('skyblue');
});
```

但是，测试单一的用例一般无法探索该指令的全部能力。 查找和测试所有使用该指令的组件非常繁琐和脆弱，并且通常无法覆盖所有组件。

只针对类的测试可能很有用， 但是像这个一样的属性型指令肯定要操纵 DOM。 隔离出的单元测试不能接触 DOM，因此也就没办法证明该指令的有效性。

更好的方法是创建一个能展示该指令所有用法的人造测试组件。

app/shared/highlight.directive.spec.ts (TestComponent)

```
@Component({
  template: `
  <h2 highlight="yellow">Something Yellow</h2>
  <h2 highlight>The Default (Gray)</h2>
  <h2>No Highlight</h2>
  <input #box [highlight]="box.value" value="cyan"/>`
})
class TestComponent { }
```

Something Yellow

The Default (Gray)

No Highlight

cyan

> `<input>` 用例将 `HighlightDirective` 绑定到输入框里输入的颜色名字。 初始只是单词“cyan”，所以输入框的背景色应该是 cyan。

下面是一些该组件的测试程序：

app/shared/highlight.directive.spec.ts (selected tests)

```
1.  beforeEach(() => {
2.    fixture = TestBed.configureTestingModule({
3.      declarations: [ HighlightDirective, TestComponent ]
4.    })
5.    .createComponent(TestComponent);
6.
7.    fixture.detectChanges(); // initial binding
8.
9.    // all elements with an attached HighlightDirective
10.   des = fixture.debugElement.queryAll(By.directive(HighlightDirective));
11.
12.   // the h2 without the HighlightDirective
13.   bareH2 = fixture.debugElement.query(By.css('h2:not([highlight])'));
14. });
15.
16. // color tests
17. it('should have three highlighted elements', () => {
18.   expect(des.length).toBe(3);
19. });
20.
21. it('should color 1st <h2> background "yellow"', () => {
22.   const bgColor = des[0].nativeElement.style.backgroundColor;
23.   expect(bgColor).toBe('yellow');
24. });
25.
26. it('should color 2nd <h2> background w/ default color', () => {
27.   const dir = des[1].injector.get(HighlightDirective) as
    HighlightDirective;
28.   const bgColor = des[1].nativeElement.style.backgroundColor;
29.   expect(bgColor).toBe(dir.defaultColor);
30. });
31.
32. it('should bind <input> background to value color', () => {
33.   // easier to work with nativeElement
34.   const input = des[2].nativeElement as HTMLInputElement;
35.   expect(input.style.backgroundColor).toBe('cyan', 'initial
    backgroundColor');
36.
37.   // dispatch a DOM event so that Angular responds to the input value
    change.
38.   input.value = 'green';
```

```
39.   input.dispatchEvent(newEvent('input'));
40.   fixture.detectChanges();
41.
42.   expect(input.style.backgroundColor).toBe('green', 'changed
    backgroundColor');
43. });
44.
45.
46. it('bare <h2> should not have a customProperty', () => {
47.   expect(bareH2.properties['customProperty']).toBeUndefined();
48. });
```

一些技巧值得注意：

- 当已知元素类型时，`By.directive` 是一种获取拥有这个指令的元素的好方法。
- `By.css('h2:not([highlight])')` 里的`:not` 伪类（pseudo-class）帮助查找不带该指令的 `<h2>` 元素。`By.css('*:not([highlight])')` 查找所有不带该指令的元素。
- `DebugElement.styles` 甚至不用借助真实的浏览器也可以访问元素的样式，感谢 `DebugElement` 提供的这层抽象！ 但是如果直接使用 `nativeElement` 会比这层抽象更简单、更清晰，也可以放心大胆的使用它。
- Angular 将指令添加到它的元素的注入器中。默认颜色的测试程序使用第二个 `<h2>` 的注入器来获取它的 `HighlightDirective` 实例以及它的 `defaultColor`。
- `DebugElement.properties` 让你可以访问由指令设置的自定义属性。

管道测试

管道很容易测试，无需 Angular 测试工具。

管道类有一个方法，`transform`，用来转换输入值到输出值。 `transform` 的实现很少与 DOM 交互。 除了 `@Pipe` 元数据和一个接口外，大部分管道不依赖 Angular。

假设 `TitleCasePipe` 将每个单词的第一个字母变成大写。 下面是使用正则表达式实现的简单代码：

app/shared/title-case.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'titlecase', pure: true})
/** Transform to Title Case: uppercase the first letter of the words in a string.*/
export class TitleCasePipe implements PipeTransform {
  transform(input: string): string {
    return input.length === 0 ? '' :
      input.replace(/\w\S*/g, (txt => txt[0].toUpperCase() +
txt.substr(1).toLowerCase() ));
  }
}
```

任何使用正则表达式的类都值得彻底的进行测试。 使用 Jasmine 来探索预期的用例和极端的用例。

app/shared/title-case.pipe.spec.ts

```
1. describe('TitleCasePipe', () => {
2.   // This pipe is a pure, stateless function so no need for BeforeEach
3.   let pipe = new TitleCasePipe();
4.
5.   it('transforms "abc" to "Abc"', () => {
6.     expect(pipe.transform('abc')).toBe('Abc');
7.   });
8.
9.   it('transforms "abc def" to "Abc Def"', () => {
10.     expect(pipe.transform('abc def')).toBe('Abc Def');
11.   });
12.
13.   // ... more tests ...
14. });
```

也能编写 DOM 测试

有些管道的测试程序是孤立的。 它们不能验证 `TitleCasePipe` 是否在应用到组件上时是否工作正常。

考虑像这样添加组件测试程序：

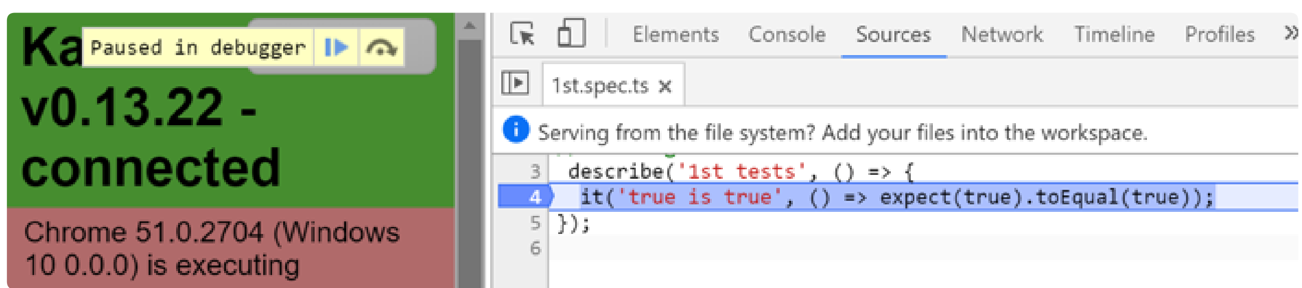
app/hero/hero-detail.component.spec.ts (pipe test)

```
1. it('should convert hero name to Title Case', () => {
2.   // get the name's input and display elements from the DOM
3.   const hostElement = fixture.nativeElement;
4.   const nameInput: HTMLInputElement = hostElement.querySelector('input');
5.   const nameDisplay: HTMLElement = hostElement.querySelector('span');
6. 
7.   // simulate user entering a new name into the input box
8.   nameInput.value = 'quick BROWN  fOx';
9. 
10.   // dispatch a DOM event so that Angular learns of input value change.
11.   nameInput.dispatchEvent(newEvent('input'));
12. 
13.   // Tell Angular to update the display binding through the title pipe
14.   fixture.detectChanges();
15. 
16.   expect(nameDisplay.textContent).toBe('Quick Brown  Fox');
17. });
```

测试程序的调试

在浏览器中，像调试应用一样调试测试程序 spec。

1. 显示 `Karma` 的浏览器窗口（之前被隐藏了）。
2. 点击“DEBUG”按钮；它打开一页新浏览器标签并重新开始运行测试程序
3. 打开浏览器的“Developer Tools”(Windows 上的 Ctrl-Shift-I 或者 OSX 上的 `Command-Option-I)。
4. 选择“sources”页
5. 打开 `1st.spec.ts` 测试文件（Control/Command-P, 然后输入文件名字）。
6. 在测试程序中设置断点。
7. 刷新浏览器...然后它就会停在断点上。



测试工具 API

本节将最有用的 Angular 测试功能提取出来，并总结了它们的作用。

Angular 的测试工具集包括 `TestBed`、`ComponentFixture` 和一些用来控制测试环境的便捷函数。 `TestBed` 和 `ComponentFixture` 部分单独讲过它们。

下面是一些独立函数的总结，以使用频率排序：

函数	说明
`async`	在一个特殊的 **async 测试区域**中运行测试（`it`）的函数体或准备函数（`beforeEach`）。 参见前面的讨论。
`fakeAsync`	在一个特殊的 **fakeAsync 测试区域**中运行测试（`it`）的函数体，以便启用线性风格的控制流。 参见前面的讨论。
`tick`	通过在 **fakeAsync 测试区域**中刷新定时器和微任务（micro-task）队列来仿真时间的流逝以及异步活动的完成。 好奇和执着的读者可能会喜欢这篇长博客： "**Tasks, microtasks, queues and schedules**". 接受一个可选参数，它可以把虚拟时钟往前推进特定的微秒数。 清除调度到那个时间帧中的异步活动。 参见前面的讨论。
`inject`	从当前的 `TestBed` 注入器中把一个或多个服务注入到一个测试函数中。 它不能用于注入组件自身提供的服务。 参见 `debugElement.injector` 部分的讨论。
`discardPeriodicTasks`	当 `fakeAsync` 测试程序以正在运行的计时器事件任务（排队中的 `setTimeOut` 和 `setInterval` 的回调）结束时， 测试会失败，并显示一条明确的错误信息。 一般来讲，测试程序应该以无排队任务结束。 当待执行计时器任务存在时，调用 `discardPeriodicTasks` 来触发任务队列，防止该错误发生。
`flushMicrotasks`	当 `fakeAsync` 测试程序以待执行微任务（比如未解析的承诺）结束时，测试会失败并显示明确的错误信息。 一般来说，测试应该等待微任务结束。 当待执行微任务存在时，调用 `flushMicrotasks` 来触发微任务队列，防止该错误发生。
`ComponentFixtureAutoDetect`	一个服务提供商令牌，用于开启自动变更检测。
`getTestBed`	获取当前 `TestBed` 实例。 通常用不上，因为 `TestBed` 的静态类方法已经够用。 `TestBed` 实例有一些很少需要用到的方法，它们没有对应的静态方法。

`TestBed` 类小结

`TestBed` 类是 Angular 测试工具的主要类之一。它的 API 很庞大，可能有点过于复杂，直到你一点一点的探索它们。 阅读本章前面的部分，了解了基本的知识以后，再试着了解完整 API。

传递给 `configureTestingModule` 的模块定义是 `@NgModule` 元数据属性的子集。

```
type TestModuleMetadata = {
  providers?: any[];
  declarations?: any[];
  imports?: any[];
  schemas?: Array<SchemaMetadata | any[]>;
};
```

每一个重载方法接受一个 `MetadataOverride<T>`，这里 `T` 是适合这个方法的元数据类型，也就是 `@NgModule`、`@Component`、`@Directive` 或者 `@Pipe` 的参数。

```
type MetadataOverride = {
    add?: T;
    remove?: T;
    set?: T;
  };
```

`TestBed` 的 API 包含了一系列静态类方法，它们更新或者引用全局的 `TestBed` 实例。

在内部，所有静态方法在 `getTestBed()` 函数返回的当前运行时间的 `TestBed` 实例上都有对应的方法。

在 `BeforeEach()` 内调用 `TestBed` 方法，这样确保在运行每个单独测试时，都有崭新的开始。

这里列出了最重要的静态方法，以使用频率排序：

方法	说明
`configureTestingModule`	测试垫片（`karma-test-shim`, `browser-test-shim`）创建了初始测试环境和默认测试模块。 默认测试模块是使用基本声明和一些 Angular 服务替代品，它们是所有测试程序都需要的。 调用 `configureTestingModule` 来为一套特定的测试定义测试模块配置，添加和删除导入、（组件、指令和管道的）声明和服务提供商。
`compileComponents`	在配置好测试模块之后，异步编译它。 如果测试模块中的**任何一个**组件具有 `templateUrl` 或 `styleUrls`，那么你必须调用这个方法，因为获取组件的模板或样式文件必须是异步的。 参见前面的讨论。 调用完 `compileComponents` 之后，`TestBed` 的配置就会在当前测试期间被冻结。
`createComponent`	基于当前 `TestBed` 的配置创建一个类型为 T 的组件实例。 一旦调用，`TestBed` 的配置就会在当前测试期间被冻结。
`overrideModule`	替换指定的 `NgModule` 的元数据。回想一下，模块可以导入其他模块。 `overrideModule` 方法可以深入到当前测试模块深处，修改其中一个内部模块。
`overrideComponent`	替换指定组件类的元数据，该组件类可能嵌套在一个很深的内部模块中。
`overrideDirective`	替换指定指令类的元数据，该指令可能嵌套在一个很深的内部模块中。
`overridePipe`	替换指定管道类的元数据，该管道可能嵌套在一个很深的内部模块中。

`get`	从当前 `TestBed` 注入器获取一个服务。
	`inject` 函数通常都能胜任这项工作，但是如果它没法提供该服务时就会抛出一个异常。
	如果该服务是可选的呢?
	`TestBed.get()` 方法可以接受可选的第二参数，当 Angular 找不到指定的服务提供商时，就会返回该对象（下面这个例子中是 `null`）：
	app/demo/demo.testbed.spec.ts
	`service = TestBed.get(NotProvided, null); // service is null`
	一旦调用，`TestBed` 的配置就会在当前测试期间被冻结。

`initTestEnvironment`	为整套测试的运行初始化测试环境。 测试垫片(`karma-test-shim`, `browser-test-shim`)会为你调用它，所以你很少需要自己调用它。 这个方法只能被调用一次。如果确实需要在测试程序运行期间改变这个默认设置，那么先调用 `resetTestEnvironment`。 指定 Angular 编译器工厂，`PlatformRef`，和默认 Angular 测试模块。以 `@angular/platform-<platform_name>/testing/<platform_name>` 的形式提供非浏览器平台的替代品。
`resetTestEnvironment`	重设初始测试环境，包括默认测试模块在内。

少数 `TestBed` 实例方法没有对应的静态方法。它们很少被使用。

`ComponentFixture` 类

`TestBed.createComponent<T>` 会创建一个组件 `T` 的实例，并为该组件返回一个强类型的 `ComponentFixture`。

`ComponentFixture` 的属性和方法提供了对组件、它的 DOM 和它的 Angular 环境方面的访问。

`ComponentFixture` 的属性

下面是对测试最重要的属性，以使用频率排序：

属性	说明
`componentInstance`	被 `TestBed.createComponent` 创建的组件类实例。
`debugElement`	与组件根元素关联的 `DebugElement`。 `debugElement` 提供了在测试和调试期间深入探查组件及其 DOM 元素的功能。它对于测试者是一个极其重要的属性。它的大多数主要成员在后面都有讲解。
`nativeElement`	组件的原生根 DOM 元素。
`changeDetectorRef`	组件的 `ChangeDetectorRef`。 在测试一个拥有 `ChangeDetectionStrategy.OnPush` 的组件，或者在组件的变化测试在你的程序控制下时，`ChangeDetectorRef` 是最重要的。

`ComponentFixture` 的方法

fixture 方法使 Angular 对组件树执行某些任务。 在触发 Angular 行为来模拟的用户行为时，调用这些方法。

下面是对测试最有用的方法。

方法	说明
`detectChanges`	为组件触发一轮变化检查。 调用它来初始化组件（它调用 `ngOnInit`）。或者在你的测试代码改变了组件的数据绑定属性值后调用它。 Angular 不能检测到你已经改变了 `personComponent.name` 属性，也不会更新 `name` 的绑定，直到你调用了 `detectChanges`。 之后，运行 `checkNoChanges`，来确认没有循环更新，除非它被这样调用：`detectChanges(false)`。
`autoDetectChanges`	设置 fixture 是否应该自动试图检测变化。 当自动检测打开时，测试 fixture 监听 zone 事件，并调用 `detectChanges`。 当你的测试代码直接修改了组件属性值时，你还是要调用 `fixture.detectChanges` 来触发数据绑定更新。 默认值是 `false`，喜欢对测试行为进行精细控制的测试者一般保持它为 `false`。
`checkNoChanges`	运行一次变更检测来确认没有待处理的变化。如果有未处理的变化，它将抛出一个错误。
`isStable`	如果 fixture 当前是稳定的，则返回 `true`。 如果有异步任务没有完成，则返回 `false`。
`whenStable`	返回一个承诺，在 fixture 稳定时解析。 要想在完成了异步活动或异步变更检测之后再继续测试，可以对那个承诺对象进行挂钩。 参见 前面。
`destroy`	触发组件的销毁。

DebugElement

`DebugElement` 提供了对组件的 DOM 的访问。

`fixture.debugElement` 返回测试根组件的 `DebugElement`，通过它你可以访问（查询）fixture 的整个元素和组件子树。

下面是 `DebugElement` 最有用的成员，以使用频率排序。

成员	说明
nativeElement	与浏览器中 DOM 元素对应（WebWorkers 时，值为 null）。
query	调用 query(predicate: Predicate<DebugElement>) 会在子树的任意深度中查找能和谓词函数匹配的第一个 DebugElement。
queryAll	调用 queryAll(predicate: Predicate<DebugElement>) 会在子树的任意深度中查找能和谓词函数匹配的所有 DebugElement。
injector	宿主依赖注入器。 比如，根元素的组件实例注入器。
componentInstance	元素自己的组件实例（如果有）。
context	为元素提供父级上下文的对象。 通常是控制该元素的祖级组件实例。 当一个元素被 *ngFor 重复，它的上下文为 NgForRow，它的 $implicit 属性值是该行的实例值。 比如，*ngFor="let hero of heroes" 里的 hero。
children	DebugElement 的直接子元素。可以通过继续深入 children 来遍历这棵树。 DebugElement 还有 childNodes，即 DebugNode 对象列表。 DebugElement 从 DebugNode 对象衍生，而且通常节点（node）比元素多。测试者通常忽略赤裸节点。
parent	DebugElement 的父级。如果 DebugElement 是根元素，parent 为 null。
name	元素的标签名字，如果它是一个元素的话。
triggerEventHandler	如果在该元素的 listeners 集合中有相应的监听器，就根据名字触发这个事件。 如果事件缺乏监听器，或者有其它问题，考虑调用 nativeElement.dispatchEvent(eventObject)。
listeners	元素的 @Output 属性以及/或者元素的事件属性所附带的回调函数。
providerTokens	组件注入器的查询令牌。 包括组件自己的令牌和组件的 providers 元数据中列出来的令牌。

`source`	source 是在源组件模板中查询这个元素的处所。
`references`	与模板本地变量（比如 `#foo`）关联的词典对象，关键字与本地变量名字配对。

`DebugElement.query(predicate)` 和 `DebugElement.queryAll(predicate)` 方法接受一个条件方法， 它过滤源元素的子树，返回匹配的 `DebugElement`。

这个条件方法是任何接受一个 `DebugElement` 并返回真值的方法。 下面的例子查询所有拥有名为 `content` 的模块本地变量的所有 `DebugElement`：

app/demo/demo.testbed.spec.ts

```
// Filter for DebugElements with a #content reference
const contentRefs = el.queryAll( de => de.references['content']);
```

Angular 的 `By` 类为常用条件方法提供了三个静态方法：

- `By.all` - 返回所有元素
- `By.css(selector)` - 返回符合 CSS 选择器的元素。
- `By.directive(directive)` - 返回 Angular 能匹配一个指令类实例的所有元素。

app/hero/hero-list.component.spec.ts

```
// Can find DebugElement either by css selector or by directive
const h2        = fixture.debugElement.query(By.css('h2'));
const directive = fixture.debugElement.query(By.directive(HighlightDirective));
```

常见问题

为什么要把测试文件和被测文件放在一起?

将单元测试的 spec 配置文件放到与应用程序源代码文件所在的同一个文件夹中是个好主意，因为：

- 这样的测试程序很容易被找到
- 你可以一眼看出应用程序的那些部分缺乏测试程序。
- 临近的测试程序可以展示代码是如何在上下文中工作的
- 当你移动代码（无可避免）时，你记得一起移动测试程序
- 当你重命名源代码文件（无可避免），你记得重命名测试程序文件。

什么时候我该把测试文件放进单独的 `test` 文件夹中?

应用程序的整合测试 spec 文件可以测试横跨多个目录和模块的多个部分之间的互动。 它们不属于任何部分，很自然，没有特别的地方存放它们。

通常，在 `test` 目录中为它们创建一个合适的目录比较好。

当然，测试助手对象的测试 spec 文件也属于 `test` 目录，与它们对应的助手文件相邻。

为什么不依赖 E2E 测试来保障 DOM 集成后的正确性?

本指南中讲的组件 DOM 测试通常需要大量的准备工作以及高级技巧，不像只针对类的测试那样简单。

为什么不等到端到端（E2E）测试阶段再对 DOM 进行集成测试呢?

E2E 测试对于整个系统的高层验证非常好用。 但是它们没法给你像单元测试这样全面的测试覆盖率。

E2E 测试很难写，并且执行性能也赶不上单元测试。 它们很容易被破坏，而且经常是因为某些远离故障点的修改或不当行为而导致的。

当出错时，E2E 测试不能轻松揭露你的组件出了什么问题， 比如丢失或错误的数据、网络失去连接或远端服务器挂了。

如果 E2E 的测试对象要更新数据库、发送发票或收取信用卡，就需要一些特殊的技巧和后门来防止远程资源被意外破坏。 它甚至可能都难以导航到你要测试的组件。

由于存在这么多障碍，你应该尽可能使用单元测试技术来测试 DOM 交互。

速查表

引导/启动	`import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';`
`platformBrowserDynamic().bootstrapModule(AppI`	用 `NgModule` 中指定的根组件进行启动。

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

`@NgModule({ declarations: ..., imports: ...,`
`exports: ..., providers: ..., bootstrap: ...})`
`class MyModule {}`
	定义一个模块，其中可以包含组件、指令、管道和服务提供商。

`declarations: [MyRedComponent, MyBlueComponent, MyDatePipe]`	属于当前模块的组件、指令和管道的列表。
`imports: [BrowserModule, SomeOtherModule]`	本模块所导入的模块列表
`exports: [MyRedComponent, MyDatePipe]`	那些导入了本模块的模块所能看到的组件、指令和管道的列表
`providers: [MyService, { provide: ... }]`	依赖注入提供商的列表，本模块以及本模块导入的所有模块中的内容都可以看见它们。
`bootstrap: [MyAppComponent]`	当本模块启动时，随之启动的组件列表。

模板语法	
`<input [value]="firstName">`	把`value`属性绑定到表达式`firstName`
`<div [attr.role]="myAriaRole">`	把属性（Attribute）`role`绑定到表达式`myAriaRole`的结果。
`<div [class.extra-sparkle]="isDelightful">`	根据`isDelightful`表达式的结果是否为真，决定 CSS 类`extra-sparkle`是否出现在当前元素上。
`<div [style.width.px]="mySize">`	把 CSS 样式属性`width`的 px（像素）值绑定到表达式`mySize`的结果。单位是可选的。
`<button (click)="readRainbow($event)">`	当这个按钮元素（及其子元素）上的 click 事件触发时，调用方法`readRainbow`，并把这个事件对象作为参数传进去。
`<div title="Hello {{ponyName}}">`	把一个属性绑定到插值字符串（如"Hello Seabiscuit"）。这种写法等价于`<div [title]="'Hello ' + ponyName">`
`<p>Hello {{ponyName}}</p>`	把文本内容绑定到插值字符串（如"Hello Seabiscuit"）
`<my-cmp [(title)]="name">`	设置双向绑定。等价于`<my-cmp [title]="name" (titleChange)="name=$event">`。
`<video #movieplayer ...>` `<button (click)="movieplayer.play()">` `</video>`	创建一个局部变量`movieplayer`，支持在当前模板的数据绑定和事件绑定表达式中访问`video`元素的实例。
`<p *myUnless="myExpression">...</p>`	这个 `*` 符号会把当前元素转换成一个内嵌的模板。它等价于：`<ng-template [myUnless]="myExpression"><p>...</p></ng-template>`

<table>
<tr><td><p>Card No.: {{cardNumber | myCardNumberFormatter}}</p></td><td>使用名叫myCardNumberFormatter的管道对表达式cardNumber的当前值进行变幻</td></tr>
<tr><td><p>Employer: {{employer?.companyName}}</p></td><td>安全导航操作符（?）表示employer字段是可选的，如果它是 undefined ，那么表达式其余的部分就会被忽略，并返回 undefined。</td></tr>
<tr><td><svg:rect x="0" y="0" width="100" height="100"/></td><td>模板中的 SVG 片段需要给它的根元素加上svg:前缀，以便把 SVG 元素和 HTML 元素区分开。</td></tr>
<tr><td><svg>
<rect x="0" y="0" width="100" height="100"/>
</svg></td><td>以<svg>作为根元素时会自动识别为 SVG 元素，不需要前缀。</td></tr>
</table>

内置指令	`import { CommonModule } from '@angular/common';`
`<section *ngIf="showSection">`	根据`showSection`表达式的结果，移除或重新创建 DOM 树的一部分。
`<li *ngFor="let item of list">`	把 li 元素及其内容变成一个模板，并使用这个模板为列表中的每一个条目实例化一个视图。

`<div [ngSwitch]="conditionExpression">`
`<ng-template [ngSwitchCase]="case1Exp">...</ng-template>`
`<ng-template ngSwitchCase="case2LiteralString">...</ng-template>`
`<ng-template ngSwitchDefault>...</ng-template>`
`</div>`
	根据`conditionExpression`的当前值选择一个嵌入式模板，并用它替换这个 div 的内容。

`<div [ngClass]="{'active': isActive, 'disabled': isDisabled}">`	根据 map 中的 value 是否为真，来决定该元素上是否出现与 name 对应的 CSS 类。右侧的表达式应该返回一个形如 `{class-name: true/false}` 的 map。
`<div [ngStyle]="{'property': 'value'}">` `<div [ngStyle]="dynamicStyles()">`	允许你使用 CSS 为 HTML 元素指定样式。你可以像第一个例子那样直接使用 CSS，也可以调用组件中的方法。
表单	`import { FormsModule } from '@angular/forms';`
`<input [(ngModel)]="userName">`	为表单控件提供双向数据绑定、解析和验证功能。

类装饰器	`import { Directive, ... } from '@angular/core';`
`@Component({...})` `class MyComponent() {}`	声明一个类是组件，并提供该组件的元数据。
`@Directive({...})` `class MyDirective() {}`	声明一个类是指令，并提供该指令的元数据。
`@Pipe({...})` `class MyPipe() {}`	声明一个类是管道，并提供该管道的元数据。
`@Injectable()` `class MyService() {}`	声明某个类具有一些依赖。当依赖注入器要创建这个类的实例时，应该把这些依赖注入到它的构造函数中。

指令配置项	`@Directive({ property1: value1, ... })`
`selector: '.cool-button:not(a)'`	指定一个 CSS 选择器，用于在模板中标记出该指令。支持的选择器类型包括：`元素名`、`[属性名]`、`.类名` 和 `:not()`。 但不支持指定父子关系的选择器。
`providers: [MyService, { provide: ... }]`	该指令及其子指令的依赖注入提供商列表。

组件配置项	`@Component` 继承自 `@Directive`， 因此，`@Directive` 的这些配置项也同样适用于组件。
`moduleId: module.id`	如果设置了，那么 `templateUrl` 和 `styleUrl` 的路径就会相对于当前组件进行解析。
`viewProviders: [MyService, { provide: ... }]`	依赖注入提供商列表，但它们的范围被限定为当前组件的视图。
`template: 'Hello {{name}}'` `templateUrl: 'my-component.html'`	当前组件视图的内联模板或外部模板的 URL 。
`styles: ['.primary {color: red}']` `styleUrls: ['my-component.css']`	用于为当前组件的视图提供样式的内联 CSS 或外部样式表 URL 的列表。

给指令和组件使用的类属性配置项	`import { Input, ... } from '@angular/core';`
`@Input() myProperty;`	声明一个输入属性，你可以通过属性绑定来更新它，如 `<my-cmp [myProperty]="someExpression">`。
`@Output() myEvent = new EventEmitter();`	声明一个输出属性，它发出事件，你可以用事件绑定来订阅它们（如：`<my-cmp (myEvent)="doSomething()">`）。
`@HostBinding('class.valid') isValid;`	把宿主元素的一个属性（这里是 CSS 类 `valid`）绑定到指令或组件上的 `isValid` 属性。
`@HostListener('click', ['$event']) onClick(e) {...}`	用指令或组件上的`onClick`方法订阅宿主元素上的`click`事件，并从中获取`$event`参数（可选）
`@ContentChild(myPredicate) myChildComponent;`	把组件内容查询（`myPredicate`）的第一个结果绑定到该类的 `myChildComponent` 属性上。
`@ContentChildren(myPredicate) myChildComponents;`	把组件内容查询（`myPredicate`）的全部结果绑定到该类的 `myChildComponents` 属性上
`@ViewChild(myPredicate) myChildComponent;`	把组件视图查询（`myPredicate`）的第一个结果绑定到该类的 `myChildComponent` 属性上。对指令无效。
`@ViewChildren(myPredicate) myChildComponents;`	把组件视图查询（`myPredicate`）的全部结果绑定到该类的 `myChildComponents` 属性上。对指令无效。

指令与组件的变更检测与生命周期钩子	由类的方法实现。
`constructor(myService: MyService, ...) { ... }`	在任何其它生命周期钩子之前调用。可以用它来注入依赖项，但不要在这里做正事。
`ngOnChanges(changeRecord) { ... }`	每当输入属性发生变化时就会调用，但位于处理内容（`ng-content`）或子视图之前。
`ngOnInit() { ... }`	在调用完构造函数、初始化完所有输入属性并首次调用过`ngOnChanges`之后调用。
`ngDoCheck() { ... }`	每当对组件或指令的输入属性进行变更检测时就会调用。可以用它来扩展变更检测逻辑，执行自定义的检测逻辑。
`ngAfterContentInit() { ... }`	`ngOnInit`完成之后，当组件或指令的内容（`ng-content`）已经初始化完毕时调用。
`ngAfterContentChecked() { ... }`	每当组件或指令的内容（`ng-content`）做变更检测时调用。
`ngAfterViewInit() { ... }`	当`ngAfterContentInit`完毕，并且组件的视图及其子视图或指令所属的视图已经初始化完毕时调用。
`ngAfterViewChecked() { ... }`	当组件的视图及其子视图或指令所属的视图每次执行变更检测时调用。
`ngOnDestroy() { ... }`	只在实例被销毁前调用一次。

依赖注入配置项	
`{ provide: MyService, useClass: MyMockService }`	把 `MyService` 的服务提供商设置或改写为 `MyMockService` 类。
`{ provide: MyService, useFactory: myFactory }`	把 `MyService` 的服务提供商设置或改写为 `myFactory` 工厂函数。
`{ provide: MyValue, useValue: 41 }`	把 `MyValue` 的服务提供商改写为一个特定的值 `41` 。

路由与导航	`import { Routes, RouterModule, ... } from '@angular/router';`

`const routes: Routes = [`
`{ path: '', component: HomeComponent },`
`{ path: 'path/:routeParam', component: MyComponent },`
`{ path: 'staticPath', component: ... },`
`{ path: '**', component: ... },`
`{ path: 'oldPath', redirectTo: '/staticPath' },`
`{ path: ..., component: ..., data: { message: 'Custom' } }`
`]);`
 	
`const routing = RouterModule.forRoot(routes);`
	为该应用配置路由。支持静态、参数化、重定向和通配符路由。也支持自定义路由数据和解析（resolve）函数。

`<router-outlet></router-outlet>` `<router-outlet name="aux"></router-outlet>`	标记出一个位置，用来加载活动路由的组件。

`<a routerLink="/path">`
`<a [routerLink]="[ '/path', routeParam ]">`
`<a [routerLink]="[ '/path', { matrixParam: 'value' } ]">`
`<a [routerLink]="[ '/path' ]" [queryParams]="{ page: 1 }">`
`<a [routerLink]="[ '/path' ]" fragment="anchor">`
	使用路由体系创建一个到其它视图的链接。路由体系由路由路径、必要参数、可选参数、查询参数和文档片段组成。要导航到根路由，请使用`/`前缀；要导航到子路由，使用`./`前缀；要导航到兄弟路由或父级路由，使用`../`前缀。

`<a [routerLink]="[ '/path' ]" routerLinkActive="active">`	当 `routerLink` 指向的路由变成活动路由时，为当前元素添加一些类（比如这里的 `active`）。

`class CanActivateGuard implements`
`CanActivate {`
`canActivate(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`):`
`Observable<boolean>|Promise<boolean>|boolean`
`{ ... }`
`}`
 	
`{ path: ..., canActivate:`
`[`**`CanActivate`**`Guard] }`
	用来定义类的接口。路由器会首先调用本接口来决定是否激活该路由。应该返回一个 `boolean` 或能解析成 `boolean` 的 `Observable/Promise`。

`class CanDeactivateGuard implements`
`CanDeactivate<T> {`
`canDeactivate(`
`component: T,`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`):`
`Observable<boolean>|Promise<boolean>|boolean`
`{ ... }`
`}`
 	
`{ path: ..., canDeactivate:`
`[`**`CanDeactivate`**`Guard] }`
	用来定义类的接口。路由器会在导航离开前首先调用本接口以决定是否取消激活本路由。应该返回一个 `boolean` 或能解析成 `boolean` 的 `Observable/Promise`。

`class CanActivateChildGuard implements`
`CanActivateChild {`
`canActivateChild(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`):`
`Observable<boolean>|Promise<boolean>|boolean`
`{ ... }`
`}`
 	
`{ path: ..., canActivateChild:`
`[CanActivateGuard],`
`children: ... }`
	用来定义类的接口。路由器会首先调用本接口来决定是否激活一个子路由。应该返回一个 `boolean` 或能解析成 `boolean` 的 `Observable/Promise`。

```
class ResolveGuard implements Resolve<T> {
resolve(
route: ActivatedRouteSnapshot,
state: RouterStateSnapshot
): Observable<any>|Promise<any>|any { ... }
}
```

```
{ path: ..., resolve: [ResolveGuard] }
```

用来定义类的接口。路由器会在渲染该路由之前，首先调用它来解析路由数据。应该返回一个值或能解析成值的 `Observable/Promise`。

```
class CanLoadGuard implements CanLoad {
canLoad(
route: Route
):
Observable<boolean>|Promise<boolean>|boolean
{ ... }
}
```

```
{ path: ..., canLoad: [CanLoadGuard],
loadChildren: ... }
```

用来定义类的接口。路由器会首先调用它来决定是否应该加载一个惰性加载模块。应该返回一个 `boolean` 或能解析成 `boolean` 的 `Observable/Promise`。