

一套框架，多种平台
同时适用手机与桌面

开始吧！



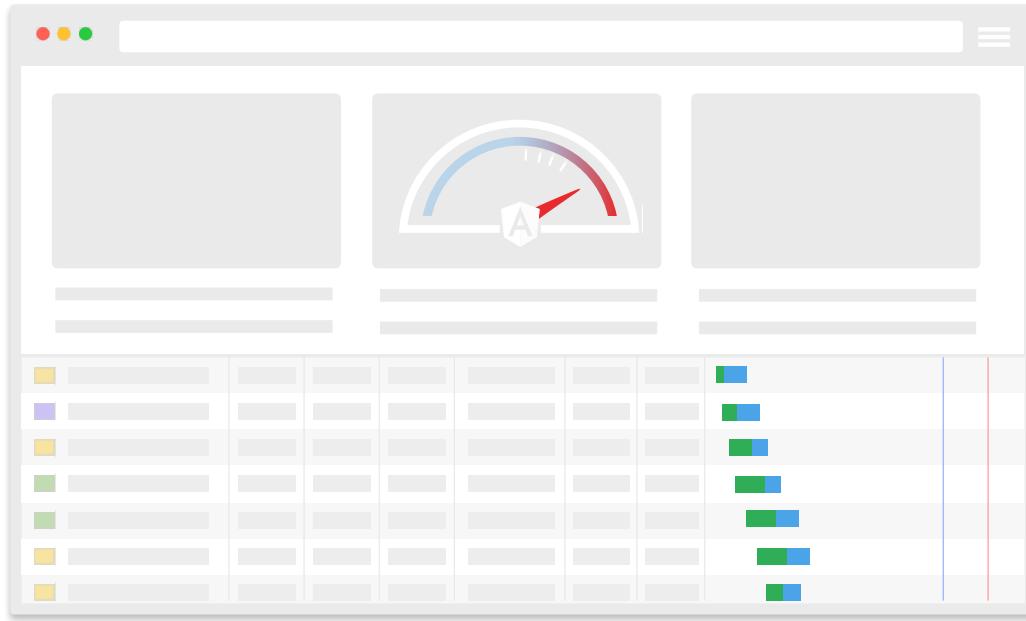
参加11月14-16号在阿姆斯特丹召开的DEVintersection！

立即报名



跨平台开发

学习如何基于 Angular 构建应用程序，并复用代码和技能来构建适用于所有平台的应用。比如：Web 应用、移动 Web 应用、原生移动应用和原生桌面应用等。



速度与性能

通过 Web Worker 和服务端渲染，达到在如今（以及未来）的 Web 平台上所能达到的最高速度。

Angular 让你有效掌控可伸缩性。基于 RxJS、Immutable.js 和其它推送模型，能适应海量数据需求。



```
• app.ts
1 import {Component} from '@angular/core'
2
3 @Component({
4   template: `<h1 [style.font]="'font'></h1>
5   <div>{{person.address.street}} font
6   <span (click)='updatePerson()' fontFamily
7   </>
8
9
10`
```

美妙的工具

使用简单的声明式模板，快速实现各种特性。使用自定义组件和大量现有组件，扩展模板语言。在几乎所有的 IDE 中获得针对 Angular 的即时帮助和反馈。所有这一切，都是为了帮助你编写漂亮的应用，而不是绞尽脑汁的让代码“能用”。



百万粉丝热捧

从原型到全球部署，Angular 都能带给你支撑 Google 大型应用的那些高延展性基础设施与技术。

特性与优点

跨平台

渐进式 Web 应用

借助现代化 Web 平台的力量，交付 app 式体验。
高性能、离线化、零安装。

原生

借助来自 Ionic 、 NativeScript 和 React Native 中的技术与思想，构建原生移动应用。

桌面

借助你已经在 Web 开发中学过的能力，结合访问原生操作系统 API 的能力，创造能在桌面环境下安装的应用，横跨 Mac 、 Windows 和 Linux 平台。

速度与性能

代码生成

Angular 会把你的模板转换成代码，针对现代 JavaScript 虚拟机进行高度优化，轻松获得框架

提供的高生产率，同时又能保留所有手写代码的优点。

统一

在服务端渲染应用的首屏，像只有 HTML 和 CSS 的页面那样几乎瞬间展现，支持 node.js、.NET、PHP，以及其它服务器，为通过 SEO 来优化站点铺平了道路。

代码拆分

Angular 应用通过新的组件路由（Component Router）模块实现快速加载，提供了自动拆分代码的功能，为用户单独加载它们请求的视图中需要的那部分代码。

生产率

模板

通过简单而强大的模板语法，快速创建 UI 视图。

Angular 命令行工具

命令行工具：快速进入构建环节、添加组件和测试，然后立即部署。

各种 IDE

在常用 IDE 和编辑器中获得智能代码补全、实时错误反馈及其它反馈等特性。

完整开发故事

测试

使用 Karma 进行单元测试，让你在每次存盘时都能立即知道是否弄坏了什么。 Protractor 则让你的场景测试运行得又快又稳定。

动画

通过 Angular 中直观简便的 API 创建高性能复杂编排和动画时间线——只要非常少的代码。

可访问性

通过支持 ARIA 的组件、开发者指南和内置的一体化测试基础设施，创建具有完备可访问性的应用。

ANGULAR中文文档

TYPESCRIPT

当前的 Angular 版本是 **2.1**。请查看 [文档更新记录](#)。参见 [Angular 更新记录](#) 了解 Angular 产品的最新的增强、修复和破坏性变更。

快速起步

一个简短的初学者指南，用以解释 Angular 的基本概念

[查看“快速起步”](#)

开发指南

一个面向中级开发者的指南，覆盖了 Angular 的所有主要特性

[查看开发指南](#)

API 参考手册

一个高级的参考手册，包括所有的 Angular 类、方法等。

[查看 API 参考手册](#)

高级文档

[Angular 模块](#)

[动画](#)

[属性型指令](#)

[浏览器支持](#)

[组件样式](#)

[查看全部.....](#)

烹饪宝典

[预编译](#)

[Angular 1 迁移到 Angular 2](#)

[Angular 模块 FAQ](#)

[组件交互](#)

[相对于组件的路径](#)

[查看全部 ...](#)

工具与库

[Angular Universal](#)

[Augury](#)

[Celerio Angular 快速起步](#)

[Codalyzer](#)

[查看全部.....](#)

快速起步

TYPESCRIPT

本“快速起步”指南将演示如何用 TypeScript 构建并运行简单的 Angular 应用。

不喜欢 TYPESCRIPT ?

虽然我们正在用 TypeScript , 但你也可以用 JavaScript 和 Dart 来写 Angular 2 应用。只要从本指南左侧导航区的语言选择器选择想用哪种语言就可以了。

概览

这个“快速起步”应用具有真实 Angular 应用的典型结构，并显示一条消息：

在线例子的链接会在 [Plunker](#) 中打开最终的应用，以便你亲自体验代码的效果。在大多数章节的开头部分都能找到在线例子。

My First Angular App

点 [在线例子](#) 的链接来 **试用一下**。

你还可以去 [GitHub](#) 克隆这个完整的应用。

构建此应用！

- **环境准备**：安装 Node.js and npm 。
- **步骤 1**：创建并配置此项目。

- 步骤 2 : 创建应用
- 步骤 3 : 创建组件并添加到应用程序中。
- 步骤 4 : 启动应用程序。
- 步骤 5 : 定义作为该应用的宿主的页面。
- 步骤 6 : 构建并运行此应用。
- 步骤 7 : 做一些修改，并立即查看效果。
- 收工，下一步

环境准备：安装 Node.js and npm

如果你的机器上还没有 Node.js 和 npm，[安装它们](#)。我们的例子需要 node **v5.x.x** 或更高版本以及 npm **3.x.x** 或更高版本。要检查你正在使用的版本，请在终端窗口中运行 `node -v` 和 `npm -v` 命令。

步骤 1 : 创建并配置本项目

这一步我们将：

- [创建项目目录](#)
- [创建配置文件](#)
- [安装包](#)

创建项目目录

使用终端窗口，为项目创建目录，并进入此目录。

```
mkdir angular-quickstart
cd      angular-quickstart
```

创建配置文件

典型的 Angular 项目需要一系列配置文件

- **package.json** 用来标记出本项目所需的 npm 依赖包。
- **tsconfig.json** 定义了 TypeScript 编译器如何从项目源文件生成 JavaScript 代码。
- **systemjs.config.js** 为模块加载器提供了该到哪里查找应用模块的信息，并注册了所有必备的依赖包。它还包括文档中后面的例子需要用到的包。

在你的项目目录中创建这些文件，并从下面的例子框中拷贝粘贴文本来填充它们。

```
1.  {
2.    "name": "angular-quickstart",
3.    "version": "1.0.0",
4.    "scripts": {
5.      "start": "tsc && concurrently \"tsc -w\" \"lite-server\" ",
6.      "lite": "lite-server",
7.      "tsc": "tsc",
8.      "tsc:w": "tsc -w"
9.    },
10.   "licenses": [
11.     {
12.       "type": "MIT",
13.       "url":
14.         "https://github.com/angular/angular.io/blob/master/LICENSE"
15.     }
16.   ],
17.   "dependencies": {
18.     "@angular/common": "~2.1.1",
19.     "@angular/compiler": "~2.1.1",
20.     "@angular/core": "~2.1.1",
21.     "@angular/forms": "~2.1.1",
22.     "@angular/http": "~2.1.1",
23.     "@angular/platform-browser": "~2.1.1",
24.     "@angular/platform-browser-dynamic": "~2.1.1",
25.     "@angular/router": "~3.1.1",
26.     "@angular/upgrade": "~2.1.1",
27.     "angular-in-memory-web-api": "~0.1.13",
28.     "core-js": "^2.4.1",
29.     "reflect-metadata": "^0.1.8",
30.     "rxjs": "5.0.0-beta.12",
31.     "systemjs": "0.19.39",
```

```
32.     "zone.js": "^0.6.25"
33.   },
34.   "devDependencies": {
35.     "@types/core-js": "^0.9.34",
36.     "@types/node": "^6.0.45",
37.     "concurrently": "^3.0.0",
38.     "lite-server": "^2.2.2",
39.     "typescript": "^2.0.3"
40.   }
41. }
```

要了解这些配置文件的更多知识，参见 [npm 包配置指南](#) 和 [TypeScript 配置指南](#)。对模块加载器的详细探讨则超出了本指南的范畴。

SYSTEMJS 还是 WEBPACK ?

虽然我们这里使用了 SystemJS 来达到目标，不过它也只是加载模块的选项之一。尽管用你喜欢的模块加载器吧。关于 Webpack 和 Angular，参见 [Webpack 简介](#)。或者到 [这里](#) 学习关于 SystemJS 配置的知识。

安装依赖包

使用 `npm` 命令来安装 `package.json` 中列出的依赖包。请在终端窗口（或 Windows 的 cmd 窗口）中输入下列命令：

```
npm install
```

在安装期间可能出现红色的错误信息，你还会看到 `npm WARN` 信息。不过不用担心，只要末尾处没有 `npm ERR!` 信息就算成功了。

你现在应该得到了如下结构：

```
angular-quickstart
```

```
node_modules ...
package.json
systemjs.config.js
tsconfig.json
```

现在，你可以开始写代码了！

步骤 2：创建应用

我们用 **NgModules** 把 Angular 应用组织成了一些功能相关的代码块。Angular 本身也被拆成了一些独立的 Angular 模块。这让你可以只导入你应用中所需的 Angular 部件，以得到较小的文件体积。

每个 Angular 应用都至少有一个模块：**根模块**，在这里它叫做 `AppModule`。

** 在应用的根目录下创建 app 子目录：

```
mkdir app
```

使用下列内容创建 `app/app.module.ts` 文件：

app/app.module.ts

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

@NgModule({
  imports:      [ BrowserModule ],
})
export class AppModule { }
```

这里是应用的入口点。

由于 QuickStart 是运行在浏览器中的 Web 应用，所以根模块需要从 `@angular/platform-browser` 中导入 `BrowserModule` 并添加到 `imports` 数组中。

这是要让最小的应用在浏览器中运行时，对 Angular 的最低需求。

QuickStart 应用不做别的，也就先不需要其它模块。在真实的应用中，我们可能还得导入 `FormsModule`、`RouterModule` 和 `HttpModule`。这些会在 [英雄指南教程](#) 中引入。

步骤 3：创建组件并添加到应用中

每个 Angular 应用都至少有一个组件：**根组件**，这里名叫 `AppComponent`。

组件是 Angular 应用的基本构造块。每个组件都会通过与它相关的模板来控制屏幕上的一个小块（视图）。

使用下列内容创建组件文件 `app/app.component.ts`：

```
app/app.component.ts

1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
5.   template: '<h1>My First Angular App</h1>'
6. })
7. export class AppComponent { }
```

QuickStart 应用具有和其它 Angular 组件相同的基本结构：

- **import 语句**。它让你能访问 Angular 核心库中的 `@Component` 装饰器函数。
- **@Component 装饰器**，它会把一份 元数据 关联到 `AppComponent` 组件类上：
 - `selector` 为用来代表该组件的 HTML 元素指定简单的 CSS 选择器。
 - `template` 用来告诉 Angular 如何渲染该组件的视图。

- **组件类** 通过它的模板来控制视图的外观和行为。这里，你只有一个根组件 `AppComponent`。由于这个简单的 QuickStart 范例中并不需要应用逻辑，因此它是空的。

我们还要 **导出** 这个 `AppComponent` 类，以便让刚刚创建的这个应用导入它。

编辑 `app/app.module.ts` 文件，导入这个新的 `AppComponent`，并把它添加到 `NgModule` 装饰器中的 `declarations` 和 `bootstrap` 字段：

app/app.module.ts

```

1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3.
4. import { AppComponent }  from './app.component';
5.
6. @NgModule({
7.   imports:      [ BrowserModule ],
8.   declarations: [ AppComponent ],
9.   bootstrap:   [ AppComponent ]
10. })
11.
12. export class AppModule { }

```

步骤 4：启动应用

现在，我们还需要做点什么来让 Angular 加载这个根组件。

添加新文件

，内容如下：

app/main.ts

```

1. import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';

```

```
2.  
3. import { AppModule } from './app.module';  
4.  
5. const platform = platformBrowserDynamic();  
6. platform.bootstrapModule(AppModule);
```

这些代码初始化了应用所在的平台，然后使用此平台引导你的 `AppModule`。

为什么要分别创建 `main.ts`、应用模块 和应用组件的文件呢？

应用的引导过程与 创建模块或者 展现视图是相互独立的关注点。如果该组件不会试图运行整个应用，那么测试它就会更容易。

引导过程是与平台有关的

但我们应该用正确的方式组织 Angular 应用的文件结构。启动 App 与展现视图是两个相互分离的关注点。把这些关注点混在一起会增加不必要的难度。可以通过使用不同的引导器 (bootstraper) 来在不同的环境中启动 `AppComponent`。测试组件也变得更容易，因为不需要再运行整个程序才能跑测试。让我们多花一点精力来用“**正确的方式**”实现它。

步骤 5：定义该应用的宿主页面

在 目录下创建 `index.html` 文件，并粘贴下列内容：

`index.html`

```
1. <html>  
2.   <head>  
3.     <title>Angular QuickStart</title>  
4.     <meta charset="UTF-8">  
5.     <meta name="viewport" content="width=device-width, initial-  
       scale=1">  
6.     <link rel="stylesheet" href="styles.css">  
7.  
8.     <!-- 1. Load libraries -->  
9.     <!-- Polyfill(s) for older browsers -->
```

```

10.      <script src="node_modules/core-js/client/shim.min.js"></script>
11.
12.      <script src="node_modules/zone.js/dist/zone.js"></script>
13.      <script src="node_modules/reflect-metadata/Reflect.js"></script>
14.      <script src="node_modules/systemjs/dist/system.src.js"></script>
15.
16.      <!-- 2. Configure SystemJS -->
17.      <script src="systemjs.config.js"></script>
18.      <script>
19.          System.import('app').catch(function(err){ console.error(err);
})];
20.      </script>
21.  </head>
22.
23.  <!-- 3. Display the application -->
24.  <body>
25.      <my-app>Loading...</my-app>
26.  </body>
27. </html>

```

这里值得注意的地方有：

- JavaScript 库： `core-js` 是为老式浏览器提供的填充库， `zone.js` 和 `reflect-metadata` 库是 Angular 需要的，而 `SystemJS` 库是用来做模块加载的。
- `SystemJS` 的配置文件和一段脚本，它导入并运行了我们刚刚在 `main` 文件中写的 `app` 模块。
- `<body>` 中的 `<my-app>` 标签是 **应用程序生活的地方！**

添加一些样式

样式不是必须的，但能让应用更漂亮。`index.html` 中假定有一个叫做 `styles.css` 的样式表。

在 `project root` 目录下创建 `styles.css` 文件，并且用下面列出的最小化样式作为初始样式。

styles.css (excerpt)

```
/* Master Styles */  
h1 {  
  color: #369;  
  font-family: Arial, Helvetica, sans-serif;  
  font-size: 250%;  
}  
h2, h3 {  
  color: #444;  
  font-family: Arial, Helvetica, sans-serif;  
  font-weight: lighter;  
}  
body {  
  margin: 2em;  
}
```

要查看本文档中所用到的主样式表的完整集合，参见 [styles.css](#).

步骤 6：编译并运行应用程序

打开终端窗口，并输入如下命令：

```
npm start
```

该命令会同时运行两个并行的 node 进程：

- TypeScript 编译器运行在监听模式。
- 一个名叫 **lite-server** 的静态文件服务器，它把 `index.html` 加载到浏览器中，并且在该应用中的文件发生变化时刷新浏览器。

到 [这里](#) 了解更多关于 `package.json` 中其它脚本的知识。

稍后，一个浏览器页标签就会打开并显示出来。

My First Angular App

步骤 7 : 做一些即时修改

尝试把 `app/app.component.ts` 中的消息修改成 "My SECOND Angular App"。

TypeScript 编译器和 `lite-server` 将会检测这些修改，重新把 TypeScript 编译成 JavaScript，刷新浏览器，并显示修改过的消息。

当终止了编译器和服务器之后，可以关闭 terminal 窗口。

收工，下一步

项目的最终目录结构看起来是这样的：

```
angular-quickstart
├── app
│   ├── app.component.ts
│   ├── app.module.ts
│   └── main.ts
└── node_modules ...
    ...
index.html
package.json
styles.css
systemjs.config.js
tsconfig.json
```

要查看文件的内容，请打开 [在线例子](#)。

下一步干什么？

第一个应用没做什么，它只是 Angular 的 "Hello, World" 而已。

我们写了很小的 Angular 组件，创建了简单的 `index.html`，并且启动了静态文件服务器。

我们还设置了基本的应用环境，你可以把它用在本指南的其它部分。以后，我们对 `package.json` 和 `index.html` 的修改将仅限于添加库或一些 css 样式，不用再需要修改模块加载部分。

下一步我们会开始构建小型的应用，用于示范能通过 Angular 构建的真实特性。来吧！
[开始“英雄指南”教程！](#)

文档概览



这是一份 Angular 实战指南。面向的是正在用 HTML 和 TypeScript 构建客户端应用的、有经验的程序员。

组织结构

本文档分成几大主题区，每个区包含一组围绕自己主题的页面。

快速起步

本文档中每一个页面和范例的基础工作。

指南

Angular 开发中必不可少的要素。

API 参考手册

关于 Angular 库中每一个成员的详尽、权威的资料。

教程

按部就班、沉浸式的 Angular 学习之旅，在应用场景中介绍了 Angular 的各个主要特性。

高级

	深入分析 Angular 的特性和开发实践。
烹饪宝典	一组解决实际应用中某些特定挑战的“菜谱”，大部分是代码片段随带少量的详细阐述。

学习路径

我们不需要从头到尾依次阅读本指南。大部分页面都是独立的。

对于 Angular 新手，建议的学习路径是走完“指南”区：

1. 要了解全景图，请阅读 [架构 概览](#)。
2. 试用 “[快速起步](#)”。“快速起步”是 Angular 的“Hello, World”。它会告诉我们如何设置任何 Angular 应用程序都会用到的库和工具。
3. 学习 [英雄指南 教程](#)，它将从“快速起步”出发，最终构建出一个简单的数据驱动的应用。它虽简单，但也具有我们写一个专业应用时所需的一切基本特性：实用的项目结构、数据绑定、主从视图、服务、依赖注入、导航，以及远程数据访问。
4. [显示数据](#) 解释了如何把信息显示到屏幕上。
5. [用户输入](#) 解释了 Angular 如何响应用户行为。
6. [表单](#) 用来在 UI 中处理用户输入的数据，并进行有效性验证。
7. [依赖注入](#) 是我们把小型、单一用途的部件组装成大型、可维护的应用的方法。
8. [模板语法](#) 是对 Angular 模板 HTML 的全面讲解。

读完这些，你就可以跳到本网站的任意页面去阅读了。

代码范例

每个页面都包含一些能在我们自己的应用中复用的代码片段。这些片段来自于相应页面中附带的范例应用。

在每页靠近顶部的地方都可以看到一个链接，指向这个范例的可执行版本，比如 [架构](#) 一章中的 [live example](#)。

这个链接启动一个基于浏览器的代码编辑器，在这里，我们可以查看、修改、保存和下载这些代码。

少量早期页面是作为教程来写的，并被清晰的标注出来。其它页面的目的是展示代码中的关键点，而不是解释构建这个范例所需的每一个步骤。我们可以通过在线例子的链接找到完整的源代码。

参考资料

[速查表](#) 列出了 Angular 在常见场景下的语法。

[词汇表](#) 定义了 Angular 开发者需要知道的术语。

[API 参考手册](#) 是关于 Angular 库中每一个公有成员的权威参考资料。

提供反馈

我们期待您的反馈！

到 [angular.io Github 库](#) 提交 [文档相关](#) 的 issues 和 pull requests.

到 [Angular Github 库](#) 报告与 [Angular 本身](#) 有关的 issues。

下一步

[架构概览](#)

架构概览

Angular 应用的基本构造块

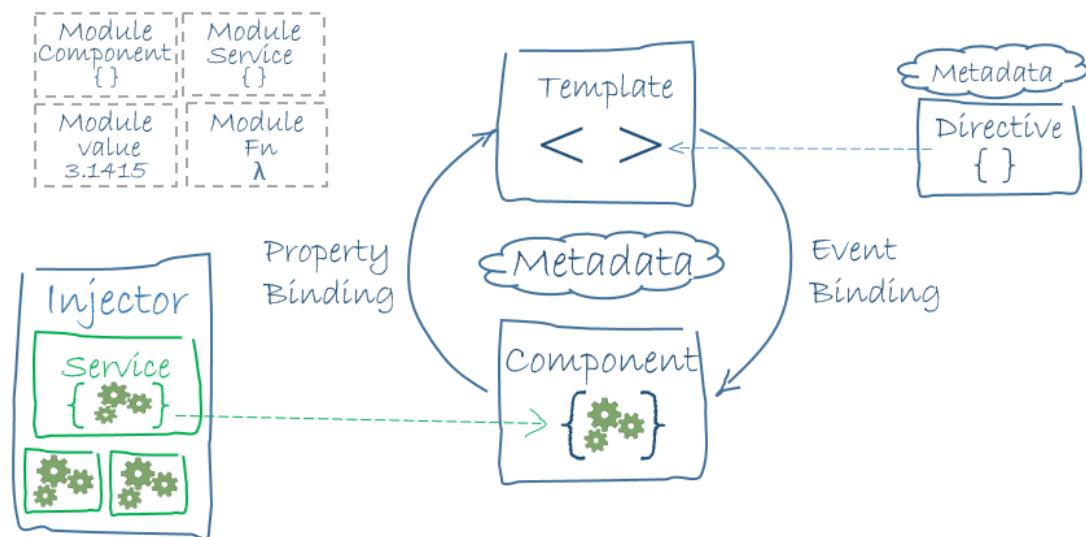
Angular 2 是一个用 HTML 和 JavaScript 或者一个可以编译成 JavaScript 的语言（比如 Dart 或者 TypeScript），来构建客户端应用的框架。

该框架包括一系列库文件，有些是核心库，有些是可选库。

我们是这样写 Angular 应用的：用带 Angular 扩展语法的 HTML 写 **模板**，用 **组件** 类管理这些模板，用 **服务** 添加应用逻辑，并在 **模块** 中打包发布组件与服务。

然后，我们通过 **引导根模块** 来启动该应用。Angular 在浏览器中接管、展现应用的内容，并根据我们提供的操作指令响应用户的交互。

当然，这只是冰山一角。在以后的页面中，我们还会学到更多细节知识。不过，目前我们还是先关注全景图吧。



这个架构图展现了 Angular 应用中的 8 个主要构造块：

- 模块 (Modules)
- 组件 (Components)
- 模板 (Templates)
- 元数据 (Metadata)
- 数据绑定 (Data Binding)
- 指令 (Directives)
- 服务 (Services)
- 依赖注入 (Dependency Injection)

掌握这些构造块，我们就可以开始使用 Angular 2 编写应用程序了。

本章所引用的代码可以从这个 [在线例子](#)。

模块

Module
Component
{ }

Angular 应用是模块化的，并且 Angular 有自己的模块系统，它被称为 **Angular 模块** 或 **NgModules**。

Angular 模块 很重要。这里我们只做一个简介，在 [Angular 模块](#) 中会做深入讲解。

每个 Angular 应用至少有一个模块（**根模块**），习惯上命名为 `AppModule`。

根模块 在一些小型应用中可能是唯一的模块，不过大多数应用可能会有很多 **特性模块**，它们由一组领域类、工作流、或紧密相关的功能聚合而成。

Angular 模块（无论是 **根** 模块还是 **特性模块**）都是一个带有 `@NgModule` 装饰器的类。

装饰器是用来修饰 JavaScript 类的函数。Angular 有很多装饰器，它们负责把元数据附加到类上，以了解那些类的设计意图以及如何使用它们。点此 [学习更多知识](#)。

`NgModule` 是一个装饰器函数，它接收一个用来描述模块属性的元数据对象。其中最重要的属性是：

- `declarations`（声明）- 本模块中拥有的视图类。Angular 有三种视图类：[组件](#)、[指令](#) 和 [管道](#)。
- `exports` - 声明（`declaration`）的子集，它可用于其它模块中的组件 [模板](#)。
- `imports` - 本模块组件模板中需要由其它模块导出的类。
- `providers` - [服务](#) 的创建者。本模块把它们加入全局的服务表中，让它们在应用中的任何部分都可被访问到。
- `bootstrap` - 标识出应用的主视图（被称为 [根组件](#)），它是所有其它视图的宿主。
只有 **根模块** 才能设置 `bootstrap` 属性。

下面是一个最简单的根模块：

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 { }
```

`AppComponent` 的 `export` 语句只是用于演示“如何导出”的，它在这个例子中并无必要。根模块不需要 **导出** 任何东西，因为其它组件不需要导入根模块。

我们通过 **引导** 根模块来启动应用。在开发期间，我们通常在一个 `main.ts` 文件中来引导 `AppModule`，就像这样：

app/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-  
dynamic';  
import { AppModule } from './app.module';  
  
platformBrowserDynamic().bootstrapModule(AppModule);
```

Angular 模块 vs. JavaScript 模块

Angular 模块（一个用 `@NgModule` 装饰的类）是 Angular 的基础特性。

JavaScript 还有自己的模块系统，它用来管理一组 JavaScript 对象。它和 Angular 的模块体系是完全不同而且完全无关的。

在 JavaScript 中，每个 **文件** 就是一个模块，并且该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开的。别的 JavaScript 模块中可以使用 `import` 语句 来访问这些公开对象。

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

```
export class AppModule { }
```

可到其它网站深入学习关于 JavaScript 模块的知识。

这两个模块化系统是互补的，我们在写程序时都会用到。

模块库

Library Module

Component Directive
{ } { }

Service Value Fn
{ } 3.1415 λ

Angular 发布了一组 JavaScript 模块。我们可以把它们看做库模块。

每个 Angular 库的名字都带有 @angular 前缀。

我们可以用 npm 包管理工具安装它们，并且用 JavaScript 的 import 语句导入它们的某些部件。

比如，我们从 @angular/core 库中导入 Component 装饰器，就像这样：

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

我们还使用 JavaScript 的导入语句从 Angular 库 中导入 Angular 的 某些模块。

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

在上面这个关于根模块的小例子中，应用模块需要来自 BrowserModule 的某些素材。要访问这些素材，我们就得把它加入 @NgModule 元数据的 imports 中去，就像这样：

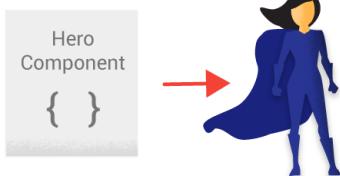
```
imports:      [ BrowserModule ],
```

这种情况下，我们正在同时使用 Angular 和 JavaScript 的模块化系统。

这两个系统比较容易混淆，因为它们共享相同的词汇（“ imports ”和“ exports ”）。先放一放，随着时间经验和增长，自然就会澄清了。

要了解更多，参见 [Angular 模块](#) 页。

组件



组件 负责控制屏幕上的一小块地方，我们称之为 视图 。

例如，下列视图都是由组件控制的：

- 带有导航链接的应用根组件。
- 英雄列表。
- 英雄编辑器。

我们在类中定义组件的应用逻辑（它被用来为视图提供支持）。组件通过一些由属性和方法组成的 API 与视图交互。

例如，`HeroListComponent` 有一个 `heroes` 属性，它返回一个“英雄”数组，这个数组是由一个服务提供的。`HeroListComponent` 还有一个 `selectHero()` 方法，当用户从列表中点选一个英雄时，就把它 / 她设置到 `selectedHero` 属性。

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; }
}

```

当用户在这个应用中“移动”时，Angular 会创建、更新和销毁组件。本应用可以通过 [生命周期钩子](#) 在组件生命周期的各个时间点上插入自己的操作，比如上面声明的 `ngOnInit()`。

模板



我们通过组件的自带的 **模板** 来定义视图。模板以 HTML 形式存在，用来告诉 Angular 如何渲染组件（视图）。

多数情况下，模板看起来很像标准 HTML当然也有一点奇怪的地方。下面是 `HeroListComponent` 组件的一个模板。

app/hero-list.component.html

```

1.  <h2>Hero List</h2>
2.
3.  <p><i>Pick a hero from the list</i></p>
4.  <ul>
5.    <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
6.      {{hero.name}}
7.    </li>

```

```

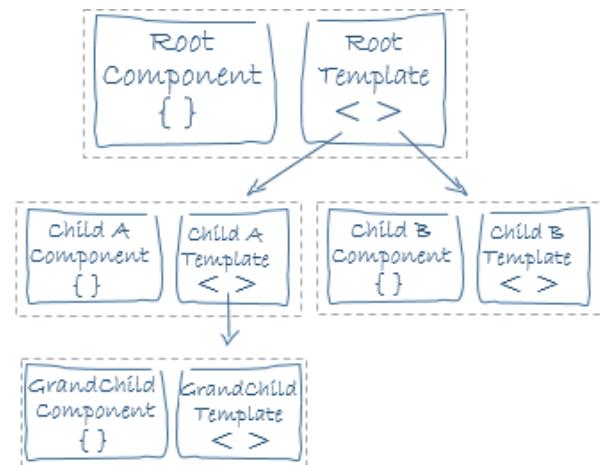
8.   </u>>
9.
10.  <hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>

```

虽然这个模板使用的是典型的 HTML 元素，比如 `<h2>` 和 `<p>`，但它也能使用别的。比如 `*ngFor`、`{hero.name}`、`(click)`、`[hero]` 和 `<hero-detail>` 使用的就是 Angular 的 模板语法。

在模板的最后一行，`<hero-detail>` 标签就是一个用来表示新组件 `HeroDetailComponent` 的自定义元素。

`HeroDetailComponent` 跟以前见到过的 `HeroListComponent` 是 **不同** 的组件。
`HeroDetailComponent` (未展示代码) 用于展现一个特定英雄的情况，这个英雄是用户从 `HeroListComponent` 列表中选择的。`HeroDetailComponent` 是 `HeroListComponent` 的子组件。



注意：`<hero-detail>` 竟如此和谐的出现在那些原生 HTML 元素中。在同一个格局中，自定义组件和原生 HTML 融合得天衣无缝。

元数据



元数据告诉 Angular 如何处理一个类。

回头看看 `HeroListComponent` 就会明白：它只是一个类。一点框架的痕迹也没有，里面完全没有出现 "Angular" 的字样。

实际上，`HeroListComponent` 真的 **只是一个类**。直到我们 **告诉 Angular 这是一个组件**为止。

只要把 **元数据** 附加到这个类，就相当于告诉 Angular：`HeroListComponent` 是个组件。

在 TypeScript 中，我们用 **装饰器 (decorator)** 来附加元数据。下面就是 `HeroListComponent` 的一些元数据。

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

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

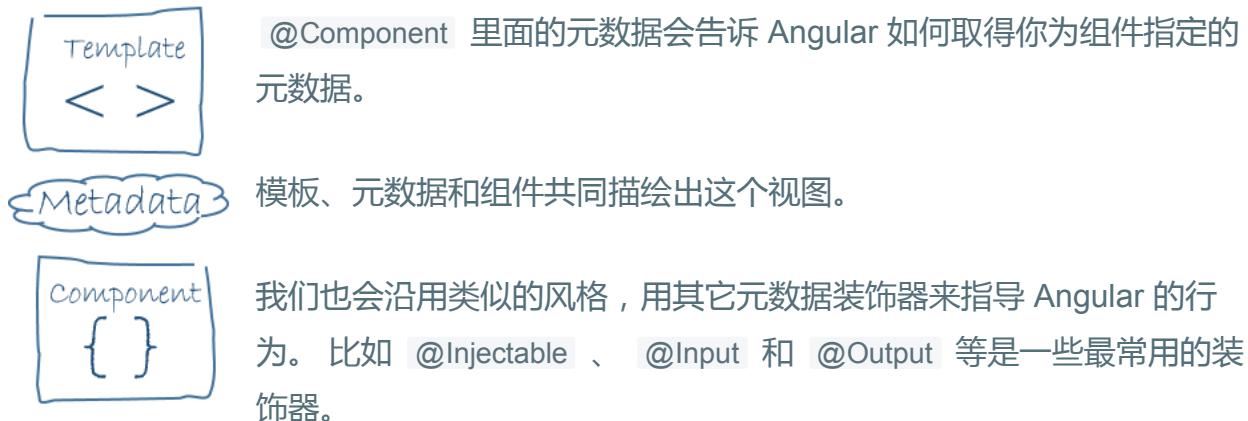
这里看到 `@Component` 装饰器（毫无悬念的）把紧随其后的类标记成了组件类。

`@Component` 装饰器能接受一个配置对象，Angular 会基于这些信息创建和展示组件及其视图。

来看下 `@Component` 中的一些配置项：

- `moduleId`：为与模块相关的 URL（如 `templateUrl`）提供基地址。
- `selector`：一个 CSS 选择器，它告诉 Angular 在 **父级** HTML 中寻找一个 `<hero-list>` 标签，然后创建该组件，并插入此标签中。比如，如果应用的 HTML 包含 `<hero-list></hero-list>`，Angular 就会把 `HeroListComponent` 的一个实例插入到这个标签中。
- `templateUrl`：组件 HTML 模板的模块相对地址，我们在 [前面](#) 展示过它。

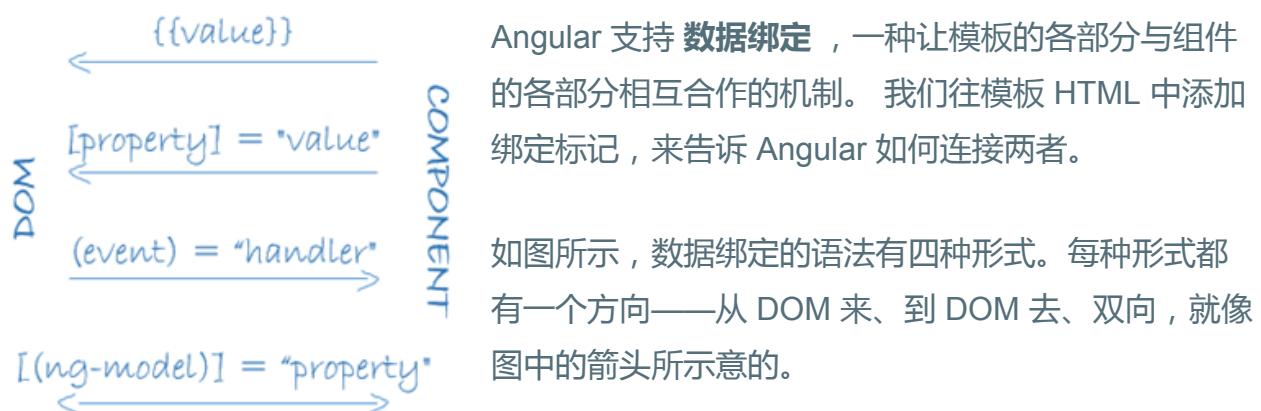
- `providers` - 一个数组，包含组件所依赖的服务所需要的 **依赖注入提供商**。这是在告诉 Angular：该组件的构造函数需要一个 `HeroService` 服务，这样组件就可以从服务中获得用来显示英雄列表的数据。



这种架构所决定的是：必须往代码中添加元数据，以便 Angular 知道该做什么。

数据绑定

如果没有框架，我们就得自己把数据值推送到 HTML 控件中，并把用户的反馈转换成动作和值更新。如果手工写代码来实现这些推 / 拉逻辑，肯定会枯燥乏味、容易出错，读起来简直是噩梦——写过 jQuery 的程序员大概都对此深有体会。



`HeroListComponent` 范例 的模板中包含了三种形式：

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

```

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

```

- `{{hero.name}}` **插值表达式**：在 `` 标签中显示了组件的 `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。
- `(click)` **事件绑定**：当用户点击英雄的名字时，调用组件的 `selectHero` 方法。

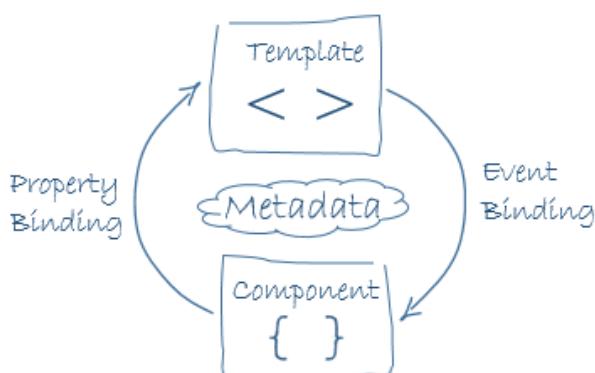
双向数据绑定：这是很重要的第四种绑定形式，它在 `ngModel` 指令这个单一标记中同时实现了属性绑定和事件绑定的功能。下面是一个来自 `HeroDetailComponent` 模板的范例：

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

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

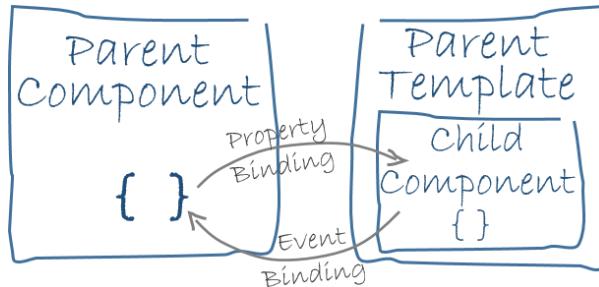
在双向绑定中，数据属性的值会从具有属性绑定的组件传到输入框。通过事件绑定，用户的修改被传回到组件，把属性值设置为最新的值。

Angular 在每个 JavaScript 事件周期中一次性处理 **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。



数据绑定在模板与对应组件的交互中扮演了一个很重要的角色。

数据绑定在父组件与子组件的通讯中 **也同样重要**。



指令



Angular 模板是 **动态的**。当 Angular 渲染它们时，它会根据 **指令** 提供的操作指南对 DOM 进行修改。



指令是一个带有“指令元数据”的类。在 TypeScript 中，要通过 `@Directive` 装饰器把元数据附加到类上。

我们已经遇到过指令的形式之一：组件。组件是一个 **带模板的指令**，而且 `@Component` 装饰器实际上就是一个 `@Directive` 装饰器，只是扩展了一些面向模板的属性。

虽然 **严格来说组件就是一个指令**，但是组件非常独特，并在 Angular 中位于中心地位，所以在架构概览中，我们把组件从指令中独立了出来。

有两种 **其它** 类型的指令，我们称之为“结构型”指令和“属性 (attribute) 型”指令。

它们往往像属性 (attribute) 一样出现在元素标签中，偶尔会以名字的形式出现但多数时候还是作为赋值目标或绑定目标出现。

结构型指令 通过在 DOM 中添加、移除和替换元素来修改布局。

我们在 [范例模板](#) 中用到了两个内置的结构型指令。

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

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

- `*ngFor` 告诉 Angular 为 `heroes` 列表中的每个英雄生成一个 `` 标签。
- `*ngIf` 表示只有在选择的英雄存在时，才会包含 `HeroDetail` 组件。

属性型指令 修改一个现有元素的外观或行为。在模板中，它们看起来就像是标准的 HTML 属性，故名。

`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。它修改了现有元素（一般是 `<input>`）的行为：它设置了显示属性值，并对 `change` 事件做出相应回应。

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

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

Angular 还有少量指令，它们或者修改结构布局（如 `ngSwitch`），或者修改 DOM 元素和组件的各个方面（如 `ngStyle` 和 `ngClass`）。

当然，我们也能编写自己的指令。像 `HeroListComponent` 这样的组件就是一种自定义指令。

服务



服务 分为很多种，包括：值、函数，以及应用所需的特性。

几乎任何东西都可以是一个服务。典型的服务是一个类，具有专注的、良好定义的用途。它应该做一件具体的事情，把它做好。

例如：

- 日志服务
- 数据服务
- 消息总线
- 税款计算器
- 应用程序配置

服务没有什么特别属于 **Angular** 的特性。 Angular 对于服务也没有什么定义。 它甚至都没有定义服务的基类（ Base Class ），也没有地方注册一个服务。

即便如此，服务仍然是任何 Angular 应用的基础。 组件就是最大的 **服务** 消费者。

这里是一个“服务”类的范例，用于把日志记录到浏览器的控制台：

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); }  
}
```

下面是 `HeroService` 类，用于获取英雄数据，并通过一个已解析的 **承诺**（ Promise ）返回它们。`HeroService` 还依赖于 `Logger` 服务和另一个用来处理服务器通讯工作的 `BackendService` 服务。

app/hero.service.ts (class)

```
export class HeroService {  
    private heroes: Hero[] = [];  
  
    constructor(  
        private backend: BackendService,  
        private logger: Logger  
    ) {  
        this.getHeroes();  
    }  
  
    getHeroes(): void {  
        this.backend.get().subscribe(  
            heroes => this.heroes = heroes  
        );  
    }  
  
    addHero(hero: Hero): void {  
        this.heroes.push(hero);  
    }  
  
    deleteHero(hero: Hero): void {  
        const index = this.heroes.indexOf(hero);  
        if (index !== -1) {  
            this.heroes.splice(index, 1);  
        }  
    }  
}
```

```

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;
}
}

```

服务无处不在。

我们更倾向于让组件保持精简。组件不需要从服务器获得数据、不需要验证输入，也不需要直接往控制台写日志。它们把这些任务委托给了服务。

组件的任务就是提供用户体验，仅此而已。它介于视图（由模板渲染）和应用逻辑（通常包括 **模型（model）** 的观念）之间。设计良好的组件为数据绑定提供属性和方法，把那些其它对它们不重要的事情都委托给服务。

Angular 不会 **强制要求** 我们遵循这些原则。即使我们花 3000 行代码写了一个“厨房洗碗槽”组件，它也不会抱怨什么。

Angular 帮助我们 **追随** 这些原则——它让我们能更容易的把应用逻辑拆分到服务，并通过 **依赖注入** 来在组件中使用这些服务。

依赖注入



“依赖注入”是提供类的新实例的一种方式，还负责处理好类所需的全部依赖。大多数依赖都是服务。Angular 也使用依赖注入提供我们需要的组件以及这些组件所需的服务。

Angular 能通过查看构造函数的参数类型，来得知组件需要哪些服务。例如，`HeroListComponent` 组件的构造函数需要一个 `HeroService`：

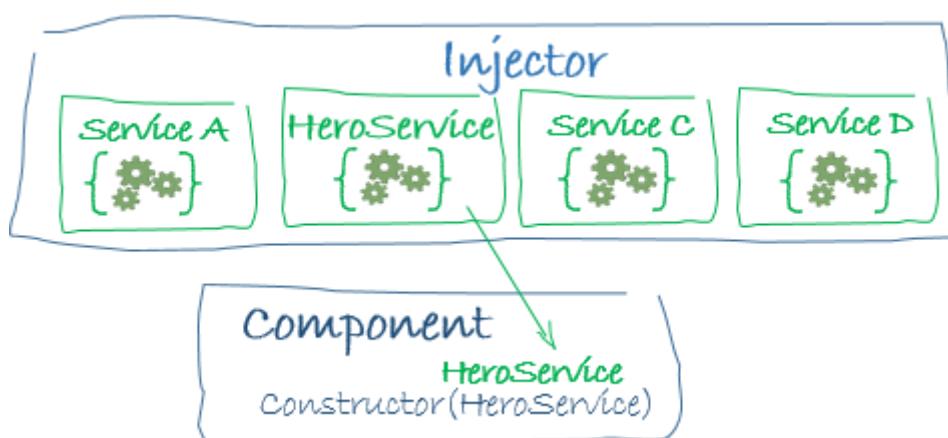
app/hero-list.component.ts (constructor)

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

当 Angular 创建组件时，会首先为组件所需的服务找一个 **注入器（Injector）**。

注入器是一个维护服务实例的容器，存放着以前创建的实例。如果容器中还没有所请求的服务实例，注入器就会创建一个服务实例，并且添加到容器中，然后把这个服务返回给 Angular。当所有的服务都被解析完并返回时，Angular 会以这些服务为参数去调用组件的构造函数。这就是 **依赖注入**。

`HeroService` 注入的过程看起来有点像这样：



如果注入器还没有 `HeroService`，它怎么知道该如何创建一个呢？

简单的说，必须在要求注入 `HeroService` 之前，把一个叫 `HeroService` 的 **提供商 Provider** 到注入器。提供商可以创建并返回服务，通常返回的就是这个“服务类”本身。

我们可以在模块上或组件上注册提供商。

我们通常会把提供商添加到 **根模块** 上，以便任何地方使用的都是服务的同一个实例。

app/app.module.ts (module providers)

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

或者，也可以在 `@Component` 元数据中的 `providers` 属性中把它注册在组件层：

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

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

把它注册在组件级表示该组件的每一个新实例都会有一个（在该组件注册的）服务的新实例。

需要记住的关于依赖注入的要点是：

- 依赖注入渗透在整个 Angular 框架中，并且被到处使用。
- **注入器（Injector）** 是本机制的核心。
 - 注入器负责维护一个 **容器**，用于存放它创建过的服务实例。
 - 注入器能使用 **提供商** 创建一个新的服务实例。
- **提供商** 是一个用于创建服务的“配方”。
- 把 **提供商** 注册到注入器。

总结

我们学到的这些只是关于 Angular 应用程序的八个主要构造块的基础知识：

- 模块
- 组件
- 模板
- 元数据
- 数据绑定
- 指令
- 服务
- 依赖注入

这是 Angular 应用程序中所有其它东西的基础，要使用 Angular 2，以这些作为开端就绰绰有余了。但它仍然没有包含我们需要知道的全部。

这里是一个简短的、按字母排序的列表，列出了其它重要的 Angular 特性和服务。它们大多数已经（或即将）包括在这份开发文档中：

动画：用 Angular 的动画库让组件动起来，而不需要对动画技术或 CSS 有深入的了解。

变更检测：“变更检测”文档会告诉你 Angular 是如何决定组件的属性值变化和什么时候该更新到屏幕的。以及它是如何用 **zones** 来拦截异步行为并执行变更检测策略。

事件：“事件”文档会告诉你如何使用组件和服务触发支持发布和订阅的事件。

表单：通过基于 HTML 的验证和脏检查机制支持复杂的数据输入场景。

HTTP：通过这个 HTTP 客户端，可以与服务器通讯，以获得数据、保存数据和触发服务端动作。

生命周期钩子：通过实现生命周期钩子接口，可以切入组件生命中的几个关键点：从创建到销毁。

管道 : 在模板中使用管道转换成可显示的值 , 以增强用户体验。比如这个

currency 管道表达式 :

```
price | currency:'USD':true
```

它把 "42.33" 显示为 \$42.33 。

路由器 : 在应用程序客户端导航 , 并且不离开浏览器。

测试 : 使用 Angular 测试工具运行单元测试 , 在你的应用与 Angular 框架交互时进行测试。

下一步

[显示数据](#)

显示数据

属性绑定机制把数据显示到 UI 上。

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，我们将创建一个英雄列表组件。我们将显示英雄名字的列表，当选中一位英雄时，就在列表下方显示一条消息。

最终的 UI 是这样的：

The screenshot shows a web page titled "Tour of Heroes". At the top, it says "My favorite hero is: Windstorm". Below that, under the heading "Heroes:", there is a list of heroes: Windstorm, Bombasto, Magneta, and Tornado. At the bottom of the list, there is a message: "There are many heroes!".

目录

- [通过插值表达式显示组件的属性](#)
- [通过 NgFor 显示数组型属性](#)
- [通过 NgIf 实现按条件显示](#)

这个 [在线例子](#) 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重花括号里放进视图模板，如 `{{myHero}}`。

我们来一起做个简明的小例子。创建一个新的项目文件夹 (`displaying-data`)，并且完成“快速起步”中的步骤。

你还可以 [下载“快速起步”的源码作为起步。](#)

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

修改完之后，它看起来应该是这样的：

app/app.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
5.   template: `
6.     <h1>{{title}}</h1>
7.     <h2>My favorite hero is: {{myHero}}</h2>
8.   `
9. })
10. export class AppComponent {
11.   title = 'Tour of Heroes';
12.   myHero = 'Windstorm';
13. }
```

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：

```
template: `

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

模板是包在反引号 (`) 中的一个多行字符串。 反引号 (`) ——注意，不是单引号 (') ——有很多好用的特性。 我们在这里用到的是它把一个字符串写在多行上的能力，这样我们的 HTML 模板就会更容易阅读。

Angular 会自动从组件中提取 `title` 和 `myHero` 属性的值，并且把这些值插入浏览器中。一旦这些属性发生变化，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，比如：按键、定时器或收到异步 `XHR` 响应。

注意，我们从没调用过 `new` 来创建 `AppComponent` 类的实例，是 Angular 替我们创建了它。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。回忆下，在“[快速起步](#)”一章中，我们曾把一个 `<my-app>` 元素添加到 `index.html` 的 `body` 里。

index.html (body)

```
<body>
<my-app>loading...</my-app>
```

```
</body>
```

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

试一下本应用。它应该显示出标题和英雄名。

Tour of Heroes

My favorite hero is: Windstorm

我们来回顾一下以前所做的决定，看看还有哪些其它选择。

内联 (inline) 模板还是模板文件？

我们有两种地方可以用来存放组件模板。我们可以使用 `template` 属性把它定义为 **内联** 的，就像这里所做的一样。或者，可以把模板定义在一个独立的 HTML 文件中，并且通过在 `@Component` 装饰器中的 `templateUrl` 属性把它链接到组件。

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。这里我们选择内联 HTML，是因为模板很小，而且这个演示很简单，没有别的 HTML 文件。

无论用哪种风格，模板中的数据绑定在访问组件属性方面都是完全一样的。

初始化：使用构造函数还是变量？

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

app/app-ctor.component.ts (class)

```
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

我们准备显示一个英雄列表。先往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。

app/app.component.ts (class)

```

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

```

现在，我们在模板中使用 Angular 的 `ngFor` “重复器”指令来显示 `heroes` 列表中的每一个条目。

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 `` 和 `` 标签组成的无序列表。 `` 元素里的 `*ngFor` 是 Angular 的“迭代”指令。 它将 `` 元素和它的子级标记为“迭代模板”：

```
<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。要了解关于此语法和 `ngFor` 的更多知识，请参见 [模板语法](#) 一章。

注意看 `ngFor` 双引号表达式中的 `hero`。它是一个 [模板输入变量](#) (译注：即 `ngFor` 模板中从外界输入的变量)。

Angular 为列表中的每一个条目复制一个 `` 元素。在每个迭代中，都会把 `hero` 变量设置为当前条目 (此英雄)。Angular 把 `hero` 变量作为双花括号插值表达式的上下文。

本例中，`ngFor` 显示“数组”，但 `ngFor` 可以为任何 [可迭代 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，我们绑定到了一个字符串数组。在真实的应用中偶尔这么做。但绝大多数时候，我们会绑定一些对象数组上。

要将此绑定转换成使用特殊对象，把英雄名字放到 `Hero` 对象的数组。但首先得有一个 `Hero` 类。

在 `app` 目录下创建一个名叫 `hero.ts` 的新文件，内容如下：

app/hero.ts (excerpt)

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

我们这样就定义了一个带有构造函数和两个属性：`id` 和 `name` 的类。

它可能看上去不像是有属性的类，但我们确实有。我们利用的是 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：

app/hero.ts (id)

```
public id: number,
```

这个简写语法做了很多：

- 定义了一个构造函数参数及其类型
- 定义了一个同名的公开属性
- 当我们 `new` 出该类的一个实例时，把该属性初始化为相应的参数值

使用 Hero 类

我们让组件中的 `heroes` 属性返回这些 `Hero` 对象的数组。

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];
```

我们还得更新一下模板。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。

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>  
`
```

从显示上看还是一样，但现在除了名字之外，我们可以有更多英雄信息。

通过 NgIf 进行条件显示

有时候，应用只需要在特定情况下显示视图或视图的一部分。

让我们来修改这个例子，让它显示在多于 3 位英雄的情况下，显示一条消息。

Angular 的 `NgIf` 指令会基于条件的真假来显示或移除一个元素。我们来亲自动手试一下，把下列语句加到模板的底部：

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。要学习关于此语法和 `NgIf` 的更多知识，请参见 [模板语法](#) 一章。

双引号中的 [模板表达式](#)，看起来很像 JavaScript，但它也 **只是“像”** JavaScript。当组件中的英雄列表有三个以上的条目时，Angular 把这些语句添加到 DOM 中，于是消息显示了出来。如果少于或等于三个条目，Angular 会移除这些语句，于是没有消息显示。

Angular 并不是在显示和隐藏这条消息，它是在从 DOM 中添加和移除这些元素。在这个范例中，它们（在性能上）几乎等价。但是如果我们想要有条件的包含或排除“一大堆”带着很多数据绑定的 HTML，性能上的区别就会更加显著。

试一下。因为数组中有四个条目，所以消息应该显示出来。回到 `app.component.ts`，并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，而消息应该会消失。

小结

现在我们知道了如何：

- 用带有双花括号的 **插值表达式 Interpolation** 来显示组件的一个属性
- 用 **ngFor** 来显示条目列表
- 用一个 TypeScript 类来为我们的组件描述 **数据模型** 并显示模型的各个属性。
- **ngIf** 用来根据一个布尔表达式有条件的显示一段 HTML

下面是我们的最终代码：

```
1. import { Component } from '@angular/core';
2.
3. import { Hero } from './hero';
4.
5. @Component({
6.   selector: 'my-app',
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. }
```

下一步

[用户输入](#)

用户输入

用户输入触发 DOM 事件。我们通过事件绑定来监听它们，把更新过的数据导入回我们的组件和 model。

当用户点击链接、按下按钮或者输入文字时，我们得知道发生了什么。这些用户动作都会产生 DOM 事件。本章中，我们将学习如何使用 Angular 事件绑定语法来绑定这些事件。

运行 [在线例子](#)

绑定到用户输入事件

我们可以使用 [Angular 事件绑定](#) 机制来响应 **任何** DOM 事件。

语法非常简单。我们只要把 DOM 事件的名字包裹在圆括号中，然后用一个放在引号中的“模板语句”对它赋值就可以了。下面的例子中，点击事件被绑定到了一个事件处理方法上：

```
<button (click)="onClickMe()">click me!</button>
```

等号左边的 `(click)` 表示把该按钮的点击事件作为 **绑定目标**。等号右边，引号中的文本是一个 **模板语句**，在这里我们通过调用组件的 `onClickMe` 方法来响应这个点击事件。**模板语句** 的语法是 JavaScript 语法的一个受限子集，但它也添加了少量“小花招”。

写绑定时，我们必须知道模板语句的 **执行上下文**。出现在模板语句中的各个标识符，全都属于一个特定的上下文对象。这个对象通常都是控制此模板的 Angular 组件……在本例中它很明确，因为这段 HTML 片段属于下面这个组件：

app/click-me.component.ts

```
@Component({
  selector: 'click-me',
  template: `
    <button (click)="onClickMe()">click me!</button>
    {{clickMessage}}
  `
})
export class ClickMeComponent {
  clickMessage = '';

  onClickMe() {
    this.clickMessage = 'You are my hero!';
  }
}
```

当用户点击此按钮时，Angular 调用组件的 `onClickMe` 方法。

通过 `$event` 对象取得用户输入

我们可以绑定到所有类型的事件。让我们试试绑定到一个输入框的 `keyup` 事件，并且把用户输入的东西回显到屏幕上。

这次，我们将 (1) 监听一个事件 (2) 捕获用户输入。

app/keyup.components.ts (模板 v.1)

```
template: `
  <input (keyup)="onKey($event)">
  <p>{{values}}</p>
`
```

Angular 把事件对象存入 `$event` 变量中，也就是我们传到组件的 `onKey()` 方法中的那个。我们希望取得的用户数据就在该变量中的某个地方。

app/keyup.components.ts (类 v.1)

```
export class KeyUpComponent_v1 {
  values = '';

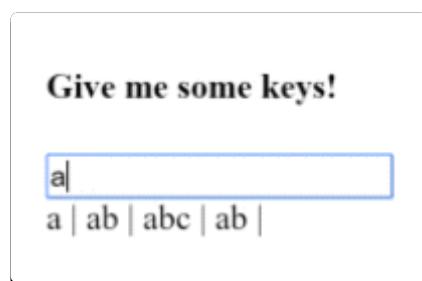
  // without strong typing
  onKey(event:any) {
    this.values += event.target.value + ' | ';
  }
}
```

`$event` 对象的形态取决于所触发的事件。这个 `keyup` 事件来自 DOM，所以 `$event` 必须是一个 **标准的 DOM 事件对象**。`$event.target` 给了我们一个 `HTMLInputElement` 对象，它有一个 `value` 属性，是用户所输入的数据。

组件的 `onKey()` 方法是用来从事件对象中提取出用户输入的，它把输入的值累加组件的 `values` 属性里，我们利用这个属性来累计用户数据。然后我们使用 **插值表达式** 来把存放累加结果的 `values` 属性回显到屏幕上。

输入字母 "abc"，然后用退格键删除它们。UI 上的显示如下：

a | ab | abc | ab | a | |



我们把 `$event` 变量声明成了 `any` 类型，这意味着我们放弃了强类型，以简化代码。我们更建议您好好使用 TypeScript 提供的强类型机制。我们可以重写此方法，把它声明为 HTML DOM 对象，就像这样：

app/keyup.components.ts (类 v.1 - 强类型版本)

```
export class KeyUpComponent_v1 {
  values = '';
```

```
// with strong typing
onKey(event: KeyboardEvent) {
    this.values += (<HTMLInputElement>event.target).value +
    ' | ';
}
}
```

使用强类型后，我们就能看出直接把 DOM 事件对象传到方法里的做法有一个严重的问题：过多模板细节，太少关注点分离。（译注：onKey 不应该理会模板的实现细节，只接收传入字符串。需要强制转换类型是代码的坏味道。）

我们将在下次再处理用户按键时处理这个问题。

从一个模板引用变量中获得用户输入

还有另一种方式，不用通过 `$event` 变量来获得用户数据。

Angular 有一个叫做 **模板引用变量** 的语法特性。这些变量给了我们直接访问元素的能力。通过在标识符前加上井号 (#)，我们就能定义一个模板引用变量。

下面的例子就通过使用局部模板变量，在一个超简单的模板中实现了一个聪明的按键反馈循环。

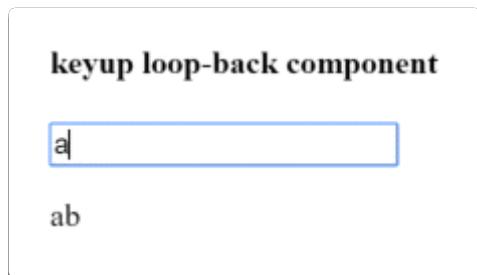
app/loop-back.component.ts

```
@Component({
  selector: 'loop-back',
  template:
    `<input #box (keyup)="0">
    <p>{{box.value}}</p>
    `
})
export class LoopbackComponent { }
```

我们在 `<input>` 元素上定义了一个名叫 `box` 的模板引用变量。 `box` 变量引用的就是 `<input>` 元素本身，这意味着我们可以获得 `input` 元素的 `value` 值，并通过插值表达式把它显示在 `<p>` 标签中。

这个模板完全是独立而完整的。它不需要绑定到组件，即使绑定了，组件也不需要额外做什么。

在输入框中输入，就会看到每次按键时，显示也随之更新了。 **完工！**



我们必须有一个事件绑定，否则这将完全无法工作。

如果我们在异步事件（如击键）的响应中做点什么，Angular 只会更新绑定（并最终影响到屏幕）。

这就是我们为什么需要把 `keyup` 事件绑定到一个语句，它做了……好吧，它啥也没做。它被绑定到了数字 0，因为这是我们所能想到的最短语句。这么做完全是为了讨好 Angular。我们说过会很聪明！

模板引用变量很有意思。它用一个变量就简洁明了的获得了文本框，而不再需要通过 `$event` 对象。也许我们可以重写前面的 `keyup` 范例，以便它能用这个变量来获得用户输入。我们这就试试看。

app/keyup.components.ts (v2)

```
@Component({
  selector: 'key-up2',
  template: `
    <input #box (keyup)="onKey(box.value)">
    <p>{{values}}</p>
  `
```

```

})
export class KeyUpComponent_v2 {
  values = '';
  onKey(value: string) {
    this.values += value + ' | ';
  }
}

```

看起来真是简单多了。该方案最漂亮的一点是：我们的组件代码从视图中获得了干干净净的数据值。再也不用了解 `$event` 变量及其结构了。

按键事件过滤 (通过 `key.enter`)

或许我们并不关心每一次按键，只在用户按下回车 (enter) 键的时候，我们才会关心输入框的值，所有其它按键都可以忽略。当绑定到 `(keyup)` 事件的时候，我们的事件处理语句会听到 **每一次按键**。我们应该先过滤一下按键，比如每一个 `$event.keyCode`，并且只有当这个按键是回车键的时候才更新 `values` 属性。

Angular 可以为我们过滤键盘事件。Angular 有一个关于键盘事件的特殊语法。通过绑定到 Angular 的 `keyup.enter` 伪事件，我们可以只监听回车键的事件。

只有在这种情况下，我们才更新组件的 `values` 属性。（在这个例子中，更新代码是写在事件绑定语句中的。但在实践中更好的方式是把更新代码放到组件中。）

app/keyup.components.ts (v3)

```

@Component({
  selector: 'key-up3',
  template: `
    <input #box (keyup.enter)="values=box.value">
    <p>{{values}}</p>
  `
})
export class KeyUpComponent_v3 {
  values = '';
}

```

下面展示了它是如何工作的。



blur(失去焦点) 事件

前一个例子中，如果用户移开了鼠标，并且点击了页面中别的地方，它不会传出输入框的值。而我们希望它在失去焦点时的行为也等同于按下回车键。只有在输入框得到焦点，并且用户按下了回车键的时候，我们才能更新组件的 `values` 属性。

我们来修正这个问题——通过同时监听输入框失去焦点的事件。

```
app/keyup.components.ts (v4)

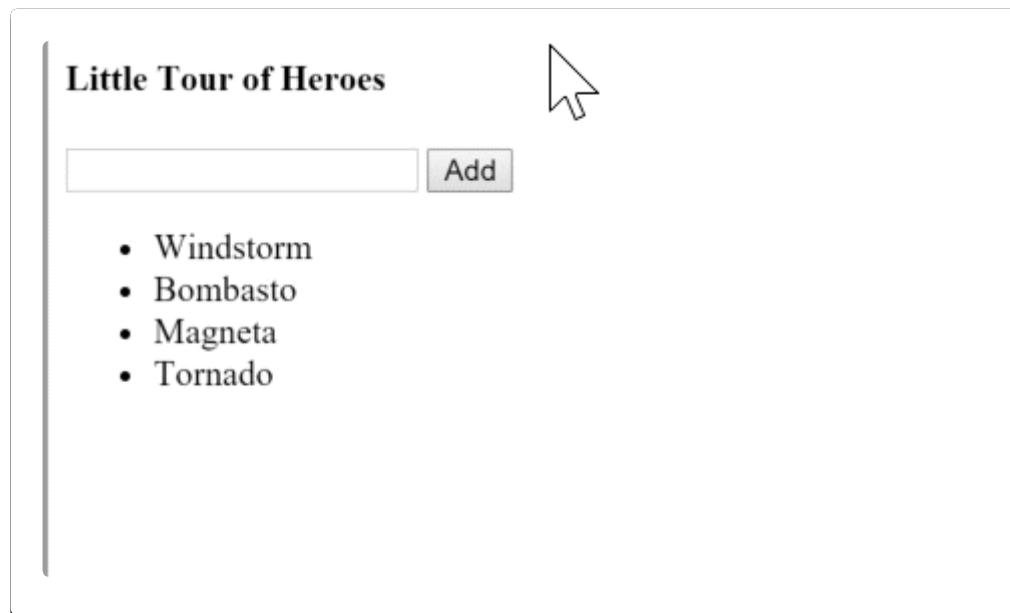
@Component({
  selector: 'key-up4',
  template: `
    <input #box
      (keyup.enter)="values=box.value"
      (blur)="values=box.value">

    <p>{{values}}</p>
  `
})
export class KeyUpComponent_v4 {
  values = '';
}
```

把它们放在一起

在前一章中，我们学到了如何 [显示数据](#)。在本章中，我们得到了一个关于事件绑定技术的小型武器库。

让我们在一个微型应用中把它们放在一起，它能显示一个英雄列表，并且把新的英雄加到列表中。用户可以通过下列步骤添加英雄：先在输入框中输入，然后按下回车键、按下“添加”按钮或点击页面中的其它地方。



下面就是“简版英雄指南”组件。短暂看一下即可，我们接下来将对它们分别讲解。

```
app/little-tour.component.ts

@Component({
  selector: '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);
    }
  }
}
```

```
    }  
}  
}
```

我们在这里几乎看到了以前接触过的每一个概念。有少量新东西，其它是复习。

使用模板变量引用元素

`newHero` 模板变量引用了 `<input>` 元素。 我们可以在 `<input>` 元素的任何兄弟节点或子节点中使用 `newHero` 。

从模板变量中获得元素，可以让按钮的点击 `click` 事件处理变得更简单。如果没有变量，我们就不得不使用“奇怪的” CSS 选择器来查找这个 `input` 元素。

传入值，不要传元素

我们可以把 `newHero` 传入组件的 `addHero` 方法。

但那需要 `addHero` 通过访问 `<input>` DOM 元素的方式先取得它——也就是我们以前在 `keyup` 组件 中学过的那种讨厌的方式。

该怎么做呢？我们该取得输入框的 **值 value**，并把它传给 `addHero`。该组件不需要知道关于 HTML 或 DOM 的任何知识，我们更喜欢这种方式。

保持模板中的语句简洁

我们把 `(blur)` 事件绑定到了 **两条** JavaScript 语句。

我们喜欢前一条，它调用了 `addHero`。我们不喜欢第二条，它把一个空值赋值给了输入框的 `value`。

第二条语句的存在理由很充分：在把新的英雄加入列表中之后，我们得清除输入框的值。组件自己做不到这一点，它不能访问输入框（我们的设计选择）。

虽然范例 **能工作**，但我们得对 HTML 中的 JavaScript 保持警惕。模板语句很强大，所以我们更得认真负责的使用它们。显然，在 HTML 中使用复杂的 JavaScript 是不负责任的表现。

我们是否要重新考虑我们的顾虑，把输入框直接传给组件？

应该有一种更好的第三条路。恩，确实有！当我们在 [表单](#) 一章学到 `ngModel` 时就明白了。

源码

下面是我们在本章中讨论过的所有源码。

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: '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

我们已经掌握了响应用户输入和操作的基础技术。虽然这些基础技术确实强大，但在处理大量用户输入时难免显得笨拙。我们在事件底层操作，但是真正应该做的是：在数据输入字段和模型属性之间建立双向数据绑定。

Angular 有一种叫做 `NgModel` 的双向数据绑定机制，我们将在 [表单](#) 一章中学到它。

下一步

表单

表单

表单创建一个有机、有效、引人注目的数据输入体验。Angular 表单协调一组数据绑定控件，跟踪变更，验证输入的有效性，并且显示错误信息。

我们全都用过表单来执行登录、求助、下单、预订机票、发起会议，以及不计其数的其它数据录入任务。表单是商业应用的主体。

不管什么样的 Web 开发者，都能使用适当的标签“捏”出一个 HTML。但是，要想做出一个优秀的表单，让它具有贴心的数据输入体验，以指导用户明晰、高效的通过表单完成背后的工作流程，这个挑战就大多了。

这当中 所需要的设计技能，坦白讲，确实超出了本章的范围。

但是，它也需要框架支持，来实现 **双向数据绑定、变更跟踪、有效性验证和错误处理**……这些 Angular 表单相关的内容，属于本章的范围。

我们将从零构建一个简单的表单，把它简化到一次一步。通过这种方式，我们将学到如何：

- 使用组件和模板构建一个 Angular 表单
- 使用 `[(ngModel)]` 语法实现双向数据绑定，以便于读取和写入输入控件的值
- 结合表单来使用 `ngModel`，能让我们跟踪状态的变化并对表单控件做验证
- 使用特殊的 CSS 类来跟踪控件状态，并提供强烈的视觉反馈
- 向用户显示有效性验证的错误提示，以及禁用 / 启用表单控件
- 通过模板引用变量，在控件之间共享信息

[运行 在线例子](#)

模板驱动的表单

我们大多数都可以使用表单特有的指令和本章所描述的技术，在模板中按照 Angular 模板语法 来构建表单。

这不是创建表单的唯一方式，但它是我们将在这章中使用的方式。（译注：Angular 支持的另一种方式叫做模型驱动表单 Model-Driven Forms）

利用 Angular 模板，我们可以构建几乎所有表单——登录表单、联系人表单……大量的各种商务表单。我们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 / 启用特定的控件、触发内置的视觉反馈等等，不胜枚举。

它的确很简单，这是因为 Angular 帮我们处理了大多数重复、单调的任务，这让我们可以不必亲自操刀、身陷其中。

我们将讨论和学习构建如下的“模板驱动”表单：

The screenshot shows a form titled "Hero Form". It contains three input fields: "Name" (with value "Dr IQ"), "Alter Ego" (with value "Chuck Overstreet"), and "Hero Power" (with value "Really Smart" and a dropdown arrow). Below the form is a "Submit" button.

这里是 **英雄管理局**，我们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。我们公司的任务就是让适当的英雄去解决它 / 她所擅长应对的危机！

这个表单中的三个字段都是必填的。这些必填的字段在左侧会有一个绿色的竖条，让它们更容易看出来。

如果我们删除了英雄的名字，表单就会用一种引人注目的样式把验证错误显示出来。

Hero Form

Name

Name is required

Alter Ego

Hero Power

Submit

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。

我们将使用标准 CSS 来定制“必填”条的颜色和位置。

我们将按照一系列很小的步骤来构建此表单：

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

搭建

创建一个新的项目文件夹 (`angular-forms`)，并且完成“[快速起步](#)”中的步骤。

你还可以 [下载“快速起步”的源码作为起步。](#)

创建一个 Hero 模型类

当用户输入表单数据时，我们要捕获它们的变化，并更新到模型的一个实例中。除非我们知道模型里有什么，否则无法设计表单。

最简单的模型就是一个“属性包”，用来存放应用中一件事物的事实。这里我们使用三个必备字段 (`id`、`name`、`power`)，和一个可选字段 (`alterEgo`，译注：中文含义：第二人格，比如 X 战警中的 Jean/ 黑凤凰)。

在应用文件夹中创建一个 `hero.ts` 文件，并且写入下列类定义内容：

app/hero.ts

```
1.  export class Hero {  
2.    
3.    constructor(  
4.      public id: number,  
5.      public name: string,  
6.      public power: string,  
7.      public alterEgo?: string  
8.    ) { }  
9.  }
```

```
10. }
```

这是一个少量需求和零行为的贫血模型。对我们的演示来说很完美。

TypeScript 编译器为构造函数中每个标为 `public` 的参数创建一个公有字段，并在创建新的英雄实例时，把参数值自动赋给这些公有字段。

`alterEgo` 是可选的，构造函数允许我们省略它，注意 `alterEgo?` 中的问号 (?)。

我们可以像这样创建一个新英雄：

```
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"
```

创建一个表单组件

每个 Angular 表单分为两部分：一个基于 HTML 的模板，和一个基于代码的组件，它用来处理数据和用户交互。

我们从组件开始，是因为它能够简要说明英雄编辑器能做什么。

创建一个名叫 `hero-form.component.ts` 的文件，并且放进下列定义：

app/hero-form.component.ts

```
1. import { Component } from '@angular/core';
2.
3. import { Hero } from './hero';
4.
5. @Component({
6.   moduleId: module.id,
7.   selector: 'hero-form',
8.   templateUrl: 'hero-form.component.html'
```

```
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.       // TODO: Remove this when we're done
22.       get diagnostic() { return JSON.stringify(this.model); }
23.   }
```

本组件没有什么特别的地方：没有表单相关的东西，也没有任何地方能把它和我们以前写过的那些组件区分开。

只需要用到前面章节中已经学过的那些概念，就可以完全理解这个组件：

1. 像往常一样，我们从 Angular 库中导入 `Component` 装饰器。
2. 导入刚刚创建的 `Hero` 模型
3. `@Component` 选择器 "hero-form" 表示我们可以通过一个 `<hero-form>` 标签，把此表单扔进父模板中。
4. `moduleId: module.id` 属性设置了以相对于模块的路径加载 `templateUrl` 时使用的基地址。
5. `templateUrl` 属性指向一个独立的 HTML 模板文件，名叫 `hero-form.component.html`。
。
6. 我们为 `model` 和 `powers` 定义了供演示用的假数据。接下来，我们可以注入一个用于获取和保存真实数据的服务，或者把这些属性暴露为 [输入与输出属性](#)，以绑定到父组件上。我们目前不关心这些，因为将来这些变化不会影响到我们的表单。

7. 我们在最后增加一个 `diagnostic` 属性，它返回这个模型的 JSON 形式。它会帮我们看清开发过程中发生的事，等最后做清理时我们会丢弃它。

这次我们为什么不像在开发指南中的其它地方那样，以内联的方式把模板放到组件文件呢？

没有什么答案在所有场合都总是“正确”的。当内联模板足够短的时候，我们更喜欢用它。但大多数的表单模板都不短。普遍来讲，TypeScript 和 JavaScript 文件不是写大型 HTML 的好地方（也不好读）。而且没有几个编辑器能对混写的 HTML 和代码提供足够的帮助。我们还是喜欢写成像这个一样清晰明确的短文件。

把 HTML 模板放在别处是一个好的选择。我们一会儿就去写那个模板。在这之前，我们先回来修改 `app.module.ts` 和 `app.component.ts` 文件，来让它用上我们新的 `HeroFormComponent` 组件。

修改 `app.module.ts`*

`app.module.ts` 定义了应用的根模块。在那里，我们指出了即将用到的外部模块，并且声明了属于本模块中的组件，比如 `HeroFormComponent`。

因为模板驱动的表单有它们自己的模块，所以我们得把 `FormsModule` 添加到本应用的 `imports` 数组中，这样我们才能使用表单。

把“快速起步”版的文件替换为如下内容：

```
app/app.module.ts

1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { AppComponent }  from './app.component';
6. import { HeroFormComponent } from './hero-form.component';
7.
8. @NgModule({
9.   imports: [
10.     BrowserModule,
11.     FormsModule
12.   ],
13.   declarations: [
14.     AppComponent,
15.     HeroFormComponent
16.   ],
17.   bootstrap: [ AppComponent ]
18. })
19. export class AppModule { }
```

```
12.     ],
13.     declarations: [
14.       AppComponent,
15.       HeroFormComponent
16.     ],
17.     bootstrap: [ AppComponent ]
18.   })
19. export class AppModule { }
```

有三处更改：

1. 我们导入了 `FormsModule` 以及新组件 `HeroFormComponent`。
2. 我们把 `FormsModule` 添加到 `ngModule` 装饰器的 `imports` 列表中。这会让我们的应用能使用模板驱动表单的所有特性，包括 `ngModel`。
3. 我们把 `HeroFormComponent` 添加到 `ngModule` 装饰器的 `declarations` 列表中。这让 `HeroFormComponent` 组件在本模块中随处都可访问。

如果一个组件、指令或管道出现在模块的 `imports` 数组中，就说明它是外来模块，**不要**再到 `declarations` 数组中声明它们。如果你自己写的它，并且它属于当前模块，**就要**把它声明在 `declarations` 数组中。

修改 app.component.ts 文件

`app.component.ts` 是本应用的根组件，我们的 `HeroFormComponent` 将被放在其中。

把“快速起步”的版本内容替换成下列代码：

app/app.component.ts

```
1. import { Component } from '@angular/core';
2.
```

```

3.  @Component({
4.    selector: 'my-app',
5.    template: '<hero-form></hero-form>'
6.  })
7.  export class AppComponent { }

```

仅有的一处修改是：

1. 直接把 `template` 的内容改成 `HeroFormComponent` 的 `selector` 属性中指定的新元素标签。于是当应用组件被加载时，将显示这个英雄表单。

创建一个初始 HTML 表单模板

创建一个新的模板文件，命名为 `hero-form.component.html`，并且填写如下内容：

app/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-default">Submit</button>
15.
16.    </form>
17.  </div>

```

这是一段普通的旧式 HTML 5 代码。这里出现了两个 `Hero` 字段，`name` 和 `alterEgo`，让用户可以在输入框中输入，修改它们。

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。

我们在底部有一个 **Submit** 按钮，它有一些用来添加样式的 CSS 类。

我们还没有用到 **Angular**。没有绑定。没有额外的指令。只做了个布局。

`container`、`form-group`、`form-control` 和 `btn` 类来自 **Twitter Bootstrap**。纯粹是装饰。我们使用 Bootstrap 来打扮我们的表单。嘿，一点样式都没有的表单算个啥！

ANGULAR 表单不需要任何样式库

Angular 不需要 `container`、`form-group`、`form-control` 和 `btn` 类，或者来自任何第三方库的任何样式，Angular 应用可以使用任何 CSS 库……或者啥都不用。

我们来添加样式表。

1. 在应用的根目录下打开一个终端窗口，敲如下命令：

```
npm install bootstrap --save
```

2. 打开 `index.html` 文件并且把下列链接添加到 `<head>` 中。

```
<link rel="stylesheet"  
      href="https://unpkg.com/bootstrap@3.3.7/dist/css/bootstrap.min.css">
```

用 `ngFor` 添加超能力

我们的英雄可以从由英雄管理局认证过的固定列表中选择一项超能力。我们先在 `HeroFormComponent` 中内部维护这个列表。

我们将添加一个 `select` 到表单中，并且用 `ngFor` 把 `powers` 列表绑定到 `option` 中。前面我们应该在 [显示数据](#) 一章中见过 `ngFor`。

在 `Alter Ego` 的紧下方添加如下 HTML：

app/hero-form.component.html (节选)

```
<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control" id="power" required>
    <option *ngFor="let p of powers" [value]="p">{{p}}</option>
  </select>
</div>
```

我们为列表中的每一项超能力渲染出一个 `<option>` 标签。模板输入变量 `p` 在每个迭代中都代表一个不同的超能力，我们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。

The screenshot shows a form titled "Hero Form". It contains three fields: "Name" (empty), "Alter Ego" (empty), and "Hero Power" (set to "Really Smart"). The "Hero Power" field is a dropdown menu.

我们没有看到英雄的数据，这是因为还没有绑定到 `Hero`。从以前的章节中，我们知道该怎么解决。[显示数据](#) 教会我们属性绑定。[用户输入](#) 告诉我们如何通过事件绑定来监听 DOM 事件，以及如何用所显示的值更新组件的属性。

现在，我们需要同时进行显示、监听和提取。

虽然可以在表单中再次使用这些技术。但是，这里我们将引入一个新东西——`[(ngModel)]` 语法，它使用一种超简单的方式把我们的表单绑定到模型。

找到“ Name ”对应的 `<input>` 标签，并且像这样修改它：

app/hero-form.component.html (节选)

```
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name">
    TODO: remove this: {{model.name}}
```

我们在 `input` 标签后添加一个诊断用的插值表达式，以看清正在发生什么事。我们给自己留下了一个备注，提醒我们完成后移除它。

聚焦到绑定语法 `[(ngModel)]="..."` 上。

如果我们现在运行这个应用，并且开始在 **姓名** 输入框中键入，添加和删除字符，我们将看到它们从插值结果中显示和消失。某一瞬间，它看起来可能是这样：

Dr IQ 3000

TODO: remove this: Dr IQ 3000

诊断信息是一个证据，用来表明数据从输入框流动到模型，再反向流动回来的过程。**这就是双向数据绑定！**

注意，我们还往 `<input>` 标签上添加了一个 `name` 属性 (Attribute) 并且把它设置为 `"name"`，这表示英雄的名字。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 `FormControls` 并将它们注册到一个 `NgForm` 指令，Angular 将该指令附加到 `<form>` 标签。每个 `FormControl` 被注册为我们指定的 `name` 属性名字。本章后面讲述了 `NgForm`。

让我们用类似的方式把 `[(ngModel)]` 绑定添加到 **第二人格** 和 **超能力** 属性。我们将抛弃输入框的绑定消息，并在组件顶部添加一个到 `diagnostic` 的新绑定。这样我们能确认双向数据绑定 **在整个 Hero 模型上** 都能正常工作了。

修改之后的表单，其核心是三个 `[(ngModel)]` 绑定，看起来像这样：

app/hero-form.component.html (节选)

```
1. {{diagnostic}}
2. <div class="form-group">
3.   <label for="name">Name</label>
4.   <input type="text" class="form-control" id="name"
5.         required
6.         [(ngModel)]="model.name" name="name">
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.         [(ngModel)]="model.alterEgo" name="alterEgo">
13. </div>
14.
15. <div class="form-group">
16.   <label for="power">Hero Power</label>
17.   <select class="form-control" id="power"
18.         required
19.         [(ngModel)]="model.power" name="power">
20.     <option *ngFor="let p of powers" [value]="p">{{p}}</option>
21.   </select>
22. </div>
```

- 每一个 `input` 元素都有一个 `id` 属性，它被 `label` 元素的 `for` 属性用来把标签匹配到对应的 `input`。
- 每一个 `input` 元素都有一个 `name` 属性，Angular 的表单模块需要使用它为表单注册控制器。

如果现在我们运行本应用，并且修改 Hero 模型的每一个属性，表单看起来像这样：

Hero Form

```
{"id":18,"name":"Dr IQ 3000","power":"Super Flexible","alterEgo":"Chuck OverUnderStreet"}
```

Name

Alter Ego

Hero Power

表单顶部的诊断信息反映出了我们所做的一切更改。

表单顶部的 `{{diagnostic}}` 绑定表达式已经完成了它的使命，删除它。

[(ngModel)] 内幕

本节是对 `[(ngModel)]` 的深入剖析，它是可选的。不感兴趣？跳过它！

绑定语法中的 `[]` 是一个很好的线索。

在属性绑定中，一个值从模型中传到屏幕上的目标属性。我们通过把名字括在方括号中来标记出目标属性，`[]`。这是一个 **从模型到视图** 的单向数据绑定。

在事件绑定中，值从屏幕上的目标属性传到模型中。我们通过把名字括在圆括号中来标记出目标属性，`()`。这是一个 **从视图到模型** 的反向单向数据绑定。

不出所料，Angular 选择了组合标点 [()] 来标记出双向数据绑定和双向数据流。

事实上，我们可以把 `NgModel` 绑定拆成两个独立的绑定，就像我们重写的“Name”`<input>` 绑定一样：

app/hero-form.component.html (节选)

```
<input type="text" class="form-control" id="name"
       required
       [ngModel]="model.name" name="name"
       (ngModelChange)="model.name = $event" >
    TODO: remove this: {{model.name}}
```

这个属性绑定看起来很眼熟，但事件绑定看起来有点怪。

`ngModelChange` 并不是 `<input>` 元素的事件。它实际上是一个来自 `ngModel` 指令的事件属性。当 Angular 在表单中看到一个 [(x)] 的绑定目标时，它会期待这个 `x` 指令有一个名为 `x` 的输入属性，和一个名为 `xChange` 的输出属性。

模板表达式中的另一个古怪之处是 `model.name = $event`。我们以前看到的 `$event` 变量是来自 DOM 事件的。但 `ngModelChange` 属性不会生成 DOM 事件——它是一个 Angular `EventEmitter` 类型的属性，当它触发时，它返回的是输入框的值——它恰好和我们希望赋给模型上 `name` 属性的值一样。

很高兴知道这些，但是这样现实吗？实践上我们几乎总是优先使用 `[(ngModel)]` 形式的双向绑定。只有当我们不得不在事件处理函数中做一些特别的事情（比如合并或限制按键频率）时，才需要拆分出独立的事件处理函数。

要学习关于 `ngModel` 和其它模板语法的更多知识，请参见 [模板语法](#) 一章。

通过 ngModel 跟踪修改状态与有效性验证

表单不仅是关于数据绑定的。我们还希望知道表单中各个控件的状态。

在表单中使用 `ngModel` 能让我们比仅使用双向数据绑定获得更多的控制权。它还会告诉我们很多信息：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。它还使用三个 CSS 类来更新控件，以便反映当前状态。我们可以通过定制这些 CSS 类的样式来更改控件的外观，以及让消息被显示或隐藏。

状态	为真的 CSS 类	为假时的 CSS 类
控件已经被访问过	<code>ng-touched</code>	<code>ng-unouched</code>
控件值已经变化	<code>ng-dirty</code>	<code>ng-pristine</code>
控件值是有效的	<code>ng-valid</code>	<code>ng-invalid</code>

我们往姓名 `<input>` 标签上添加一个名叫 **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。

app/hero-form.component.html (excerpt)

```

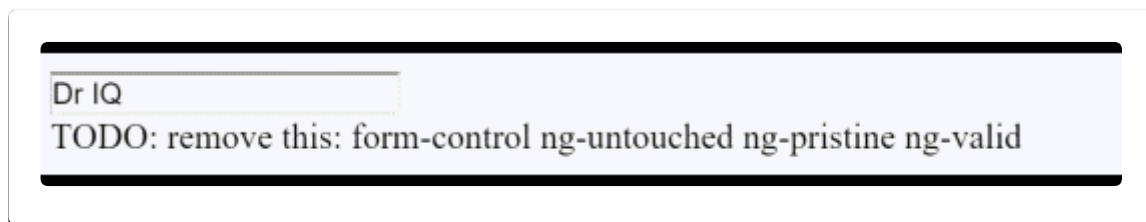
<input type="text" class="form-control" id="name"
  required
  [(ngModel)]="model.name" name="name"
  #spy >
<br>TODO: remove this: {{spy.className}}

```

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. 在名字的末尾添加一个斜杠
4. 删掉名字

动作和它对应的效果如下：



我们应该能看到下列四组类名以及它们的变迁：

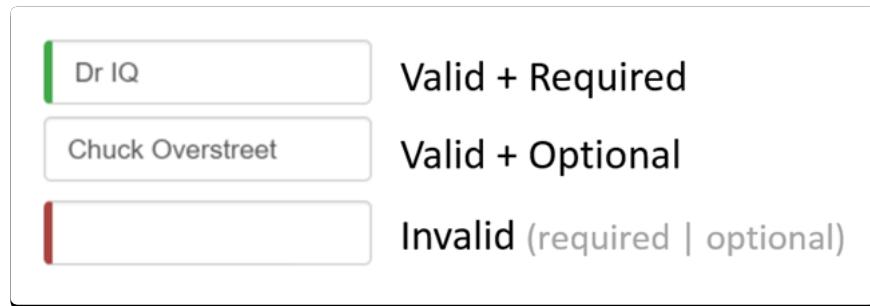


(`ng-valid` | `ng-invalid`) 这一对是我们最感兴趣的。当数据变得无效时，我们希望发出一个强力的视觉信号。我们还希望标记出必填字段。于是我们加入自定义 CSS 来提供视觉反馈。

也 **删除** 模板引用变量 `#spy` 以及 `TODO`，因为它们已经完成了使命。

添加自定义 CSS，以提供视觉反馈

我们意识到，只要在输入框的左侧添加一个带颜色的竖条，就可以同时做到标记出必填字段和无效输入：



在新建的 `forms.css` 文件中，添加两个样式的定义就达到了预期效果。我们把这个文件添加到项目中，和 `index.html` 相邻。

forms.css

```
.ng-valid[required], .ng-valid.required {
  border-left: 5px solid #42A948; /* green */
}

.ng-invalid:not(form) {
  border-left: 5px solid #a94442; /* red */
}
```

这些样式的 selector 是这两个 Angular 有效性类和 HTML5 的“`required`”属性。

我们更新 `index.html` 中的 `<head>` 标签来包含这个样式表。

index.html (节选)

```
<link rel="stylesheet" href="styles.css">
<link rel="stylesheet" href="forms.css">
```

显示和隐藏有效性校验的错误信息

我们能做的更好。

“ 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，但我们不知道错在哪里，或者如何纠正。我们可以借助 `ng-invalid` 类来给出一个更有用的消息。

当用户删除姓名时，显示方式应该是这样的：

要达到这个效果，我们得通过下列方式扩展 `<input>` 标签：

1. 一个 [模板引用变量](#)
2. 将“ is required ”的消息放在附近的一个 `<div>` 元素中，只有当控件无效时，我们才显示它。

下面是我们应该对 **Name** 输入框所要做的：

```
app/hero-form.component.html (节选)

<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>
```

我们需要一个模板引用变量来访问模板中输入框的 Angular 控件。这里，我们创建了一个名叫 `name` 的变量，并且把它赋值为 "ngModel" 。

为什么是 "ngModel" ? 指令的 `exportAs` 属性告诉 Angular 如何把模板引用变量链接到指令中。这里我们把 `name` 设置为 `ngModel` 就是因为 `ngModel` 指令的 `exportAs` 属性设置成了 "ngModel"。

现在，通过把 `div` 元素的 `hidden` 属性绑定到 `name` 控件的属性，我们就可以控制“姓名”字段错误信息的可见性了。

app/hero-form.component.html (节选)

```
1.  <div [hidden]="name.valid || name.pristine"
2.      class="alert alert-danger">
```

这个范例中，当控件是有效或全新 (pristine) 的时，我们要隐藏消息。“全新”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，我们就会只在值有效时隐藏此消息。如果往这个组件中传入一个全新 (空白) 的英雄，或者一个无效的英雄，我们将立刻看到错误信息——虽然我们还啥都没做。

有些人会为这种行为感到不安。它们希望只有在用户做出一个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 [添加一个新英雄](#) 时，我们将看到这种选择的重要性。

英雄的 **第二人格** 是可选项，所以我们不填它。

英雄的 **超能力** 选项是必填的。只要愿意，我们可以往 `<select>` 上添加相同的错误处理。但是这没有那么迫切，因为这个选择框已经足够把“超能力”约束成有效值了。

添加一个英雄，并且重置表单

我们希望在这个表单中添加一个新的英雄。我们在表单的底部放一个“新增英雄”按钮，并且把它的点击事件绑定到一个组件方法上。

app/hero-form.component.html (新增英雄按钮)

```
<button type="button" class="btn btn-default" (click)="newHero()">New Hero</button>
```

app/hero-form.component.ts (新增英雄方法 - v1)

```
newHero() {
  this.model = new Hero(42, '', '');
}
```

再次运行应用，点击 **新增英雄** 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。对三个必填字段来说，这种方式清晰易懂。错误信息是隐藏的，这是因为表单还是全新的，我们还没有修改任何东西。

输入一个名字，并再次点击 **新增英雄** 按钮。这次，我们看到了错误信息！为什么？当我们显示一个新（空白）的英雄时，我们不希望如此。

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。更换了英雄 **并不会重置控件的“全新”状态**。

这反映出，在这种实现方式下，Angular 没办法区分是替换了整个英雄数据还是用程序单独清除了 `name` 属性。Angular 不能作出假设，因此只好让控件保留当前状态——脏状态。

我们不得不使用一个小花招来重置表单控件。我们给组件添加一个 `active` 标记，把它初始化为 `true`。当我们添加一个新的英雄时，它把 `active` 标记设置为 `false`，然后通过一个快速的 `setTimeout` 函数迅速把它设置回 `true`。

app/hero-form.component.ts (新增英雄 - 最终版)

```
active = true;

newHero() {
  this.model = new Hero(42, '', '');
  this.active = false;
  setTimeout(() => this.active = true, 0);
}
```

然后，我们把 form 元素绑定到这个 active 标志上。

app/hero-form.component.html (Form标签)

```
<form *ngIf="active">
```

在通过 NgIf 绑定到 active 标志之后，点击“新增英雄”将从 DOM 中移除这个表单，并在一眨眼的功夫重建它。重新创建的表单处于“全新”状态。错误信息被隐藏了。

这只是一个临时的变通方案，将来我们还会有一个更合适的方案来重置表单。

通过 ngSubmit 来提交表单

在填表完成之后，用户还应该能提交这个表单。“提交”按钮位于表单的底部，它自己不会做任何事，但因为具有特殊的 type 值 (type="submit")，所以它会触发表单提交。

仅仅触发“表单提交”在目前是没用的。要让它有用，我们还要用另一个 Angular 指令更新 <form> 标签—— NgSubmit，并且通过事件绑定机制把它绑定到 HeroFormComponent.submit() 方法上。

```
<form *ngIf="active" (ngSubmit)="onSubmit()" #heroForm="ngForm">
```

最后，我们发现了一些额外的东西！我们定义了一个模板引用变量 `#heroForm`，并且把它初始化为 "ngForm"。

这个 `heroForm` 变量现在引用的是 `NgForm` 指令，它代表的是表单的整体。

NgForm 指令

什么 `NgForm` 指令？我们没有添加过 `NgForm` 指令啊！

Angular 替我们做了。Angular 自动创建了 `NgForm` 指令，并且把它附加到 `<form>` 标签上。

`NgForm` 指令为普通的 `form` 元素扩充了更多特性。它持有我们通过 `ngModel` 指令和 `name` 属性为各个元素创建的那些控件类，并且监视它们的属性变化，包括有效性。它还有自己的 `valid` 属性，只有当 **每一个被包含的控件都有效时**，它才有效。

模板中稍后的部分，通过 `heroForm` 变量，我们把按钮的 `disabled` 属性绑定到了表单的全员有效性。这里是那点 HTML：

```
<button type="submit" class="btn btn-default"
[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，删除 **姓名**。我们违反了“必填姓名”规则，它还是像以前那样显示了错误信息来提醒我们。同时，“提交”按钮也被禁用了。

没想明白？再想一会儿。如果没有 Angular `NgForm` 的帮助，我们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. 定义一个模板引用变量，放在（强化过的）form 元素上

2. 从 50 行之外的按钮上引用这个变量。

切换两个表单区域（额外的荣誉）

现在就提交表单还不够激动人心。

对演示来说，这是一个平淡的收场。老实说，即使让它更出彩，也无法教给我们任何关于表单的新知识。但这是一个锻炼我们新学到的绑定技能的好机会。如果你不感兴趣，可以跳过本章的下面这部分，而不用担心错失任何东西。

我们来实现一些更明显的视觉效果吧。隐藏掉数据输入框，并且显示一些别的东西。

先把表单包裹进 `<div>` 中，并且把它的 `hidden` 属性绑定到 `HeroFormComponent.submitted` 属性上。

app/hero-form.component.html (节选)

```
<div [hidden]="submitted">
  <h1>Hero Form</h1>
  <form *ngIf="active" (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`，直到我们提交了这个表单。来自 `HeroFormComponent` 的代码片段告诉了我们这一点：

```
submitted = false;
```

```
onSubmit() { this.submitted = true; }
```

当我们点击提交按钮时，`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，我们需要显示一些别的东西。在我们刚刚写的 `<div>` 包装下方，添加下列 HTML 块：

app/hero-form.component.html (节选)

```
1.  <div [hidden]="!submitted">
2.    <h2>You submitted the following:</h2>
3.    <div class="row">
4.      <div class="col-xs-3">Name</div>
5.      <div class="col-xs-9 pull-left">{{ model.name }}</div>
6.    </div>
7.    <div class="row">
8.      <div class="col-xs-3">Alter Ego</div>
9.      <div class="col-xs-9 pull-left">{{ model.alterEgo }}</div>
10.     </div>
11.     <div class="row">
12.       <div class="col-xs-3">Power</div>
13.       <div class="col-xs-9 pull-left">{{ model.power }}</div>
14.     </div>
15.     <br>
16.     <button class="btn btn-default"
17.           (click)="submitted=false">Edit</button>
18.   </div>
```

我们的英雄又来了，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。

我们添加了一个“编辑”按钮，它的 `click` 事件被绑定到了一个表达式，它会清除 `submitted` 标志。

当我们点它时，这个只读块消失了，可编辑的表单重新出现了。

现在，它比我们那个刚好够用的版本好玩多了。

结论

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` 装饰器的组件类。
- 用来处理表单提交的 `ngSubmit` 指令。
- 模板引用变量，如 `#heroForm`、`#name` 和 `#power`。
- 用于双向数据绑定、数据验证和变化追踪的 `[(ngModel)]` 语法和 `name` 属性。
- 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，控制提交按钮的禁用状态。
- 对无效控件，定制 CSS 类来给用户提供视觉反馈。

我们最终的项目目录结构看起来是这样：

```
angular-forms
  app
    app.component.ts
    app.module.ts
    hero.ts
    hero-form.component.html
    hero-form.component.ts
    main.ts
  node_modules ...
  index.html
  package.json
  tsconfig.json
```

这里是源码的最终版本：

```
1. import { Component } from '@angular/core';
2.
3. import { Hero }      from './hero';
4.
5. @Component({
6.   moduleId: module.id,
7.   selector: 'hero-form',
8.   templateUrl: 'hero-form.component.html'
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.   // Reset the form with a new hero AND restore 'pristine' class state
22.   // by toggling 'active' flag which causes the form
23.   // to be removed/re-added in a tick via NgIf
24.   // TODO: workaround until NgForm has a reset method (#6822)
25.   active = true;
26.
27.   newHero() {
28.     this.model = new Hero(42, '', '');
29.     this.active = false;
30.     setTimeout(() => this.active = true, 0);
31.   }
32. }
```

下一步

[依赖注入](#)

依赖注入

Angular 的依赖注入系统能够 JIT(刚好及时) 的创建和交付所依赖的服务。

依赖注入 是一个很重要的程序设计模式。 Angular 有自己的依赖注入框架，离开了它，我们几乎没法构建 Angular 应用。 它使用得非常广泛，以至于几乎每个人都会把它简称为 **DI**。

在本章中，我们将学习 DI 是什么，以及我们为什么需要它。 然后，我们将学习在 Angular 应用中该 [如何使用它](#)。

- [为什么依赖注入？](#)
- [Angular 依赖注入](#)
- [注入器提供商](#)
- [依赖注入令牌](#)
- [总结](#)

运行 [在线例子](#)。

为什么需要依赖注入？

我们从下列代码开始：

app/car/car.ts (without DI)

```
1.  export class Car {  
2.    public engine: Engine;  
3.    public tires: Tires;  
4.    public description = 'No DI';
```

```

6.
7.     constructor() {
8.         this.engine = new Engine();
9.         this.tires = new Tires();
10.    }
11.
12.    // Method using the engine and tires
13.    drive() {
14.        return `${this.description} car with ` +
15.            `${this.engine.cylinders} cylinders and ${this.tires.make}
16.            tires.`;
17.    }

```

我们的 `Car` 类会在它的构造函数中亲自创建所需的每样东西。 问题何在？

问题在于，我们这个 `Car` 类过于脆弱、缺乏弹性并且难以测试。

我们的 `Car` 类需要一个 `Engine` 和 `Tires`，它没有去请求一个现成的实例，而是在构造函数中用具体的 `Engine` 和 `Tires` 类新创建了一份只供自己用的副本。

如果 `Engine` 类升级了，并且它的构造函数要求传入一个参数了，该怎么办？我们这个 `Car` 类就被破坏了，而且直到我们把创建引擎的代码重写为 `engine = new Engine(theNewParameter)` 之前，它都是坏的。当我们首次写 `Car` 类时，我们不会在乎 `Engine` 构造函数的参数。现在我们也不想在乎。但是当 `Engine` 类的定义发生变化时，我们就不得不在乎了，`Car` 类也不得不跟着改变。这就会让 `Car` 类过于脆弱。

如果我们想在我们的 `Car` 上用一个不同品牌的轮胎会怎样？太糟了。我们被锁死在 `Tires` 类创建时使用的那个品牌上。这让我们的 `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，但是我们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。我们的车缺乏必要的弹性，无法共享当初给其它消费者创建的车载无线。

当我们给 `Car` 类写测试的时候，我们被它那些隐藏的依赖所摆布。你以为能在测试环境中成功创建一个新的 `Engine` 吗？`Engine` 自己又依赖什么？那些依赖本身又依赖什么？`Engine` 的新实例会发起一个到服务器的异步调用吗？我们当然不想在测试期间这么一层层追下去。

如果我们的 `Car` 应该在轮胎气压低的时候闪动一个警示灯该怎么办？如果我们没法在测试期间换上一个低气压的轮胎，我们该如何确认它能正确的闪警示灯？

我们没法控制这辆车背后隐藏的依赖。当我们不能控制依赖时，类就会变得难以测试。

我们该如何让 `Car` 更强壮、有弹性以及可测试？

答案超级简单。我们把 `Car` 的构造函数改造成使用 DI 的版本：

```
public description = 'DI';

constructor(public engine: Engine, public tires: Tires) { }
```

发生了什么？我们把依赖的定义移到了构造函数中。我们的 `Car` 类不再创建引擎或者轮胎。它仅仅“消费”它们。

我们再次借助 TypeScript 的构造器语法来同时定义参数和属性。

现在，我们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”的定义从 `Car` 类本身解耦开了。只要喜欢，我们就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。

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。消费者必须更新创建这辆车的代码，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。我们很快就来解决消费者的问题。

`Car` 类非常容易测试，因为我们现在对它的依赖有了完全的控制权。在每个测试期间，我们可以往构造函数中传入 mock 对象，做到我们想让它们做的事：

```

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());

```

我们刚刚学到了什么是依赖注入

它是一种编程模式，该模式可以让一个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。我们需要某种机制把这三个部分装配好。

我们可以写一个巨型类来做这件事（不好的模式）：

app/car/car-factory.ts

```

1. import { Engine, Tires, Car } from './car';
2.
3. // BAD pattern!

```

```

4.  export class CarFactory {
5.    createCar() {
6.      let car = new Car(this.createEngine(), this.createTires());
7.      car.description = 'Factory';
8.      return car;
9.    }
10.
11.   createEngine() {
12.     return new Engine();
13.   }
14.
15.   createTires() {
16.     return new Tires();
17.   }
18. }

```

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。这个工厂类将变成一个由相互依赖的工厂方法构成的巨型蜘蛛网。

如果我们能简单的列出我们想建造的东西，而不用定义该把哪些依赖注入到哪些对象中，那该多好！

到了让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。我们使用这个注入器注册一些类，它会指出该如何创建它们。

当我们需要一个 `Car` 时，就简单的找注入器取车就可以了。

```
let car = injector.get(Car);
```

多方皆赢。`Car` 不需要知道如何创建 `Engine` 和 `Tires` 的任何事。消费者不知道如何创建 `Car` 的任何事。我们不需要一个巨大的工厂类来维护它们。`Car` 和消费者只要简单的说出它们想要什么，注入器就会交给它们。

这就是“**依赖注入框架**”存在的原因。

现在，我们知道了依赖注入是什么，以及它的优点是什么。我们再来看看它在 Angular 中是怎么实现的。

Angular 依赖注入

Angular 自带了它自己的依赖注入框架。此框架也能被当做独立模块用于其它应用和框架中。

听起来很好。当我们在 Angular 中构建组件的时候，它到底能为我们做什么？让我们一步一个脚印的看看。

我们从当初在 [英雄指南](#) 中构建过的 `HeroesComponent` 的一个简化版本开始。

```
1. import { Component }           from '@angular/core';
2.
3. @Component({
4.   selector: 'my-heroes',
5.   template: `
6.     <h2>Heroes</h2>
7.     <hero-list></hero-list>
8.   `
9. })
10. export class HeroesComponent { }
```

`HeroesComponent` 是 **英雄** 特性区域的根组件。它管理本区的所有子组件。我们简化后的版本只有一个子组件 `HeroListComponent`，用来显示一个英雄列表。

现在 `HeroListComponent` 从 `HEROES` 获得英雄数据，一个在另一个文件中定义的内存数据集。它在开发的早期阶段可能还够用，但离完美就差得远了。我们一旦开始测试此组件，或者想从远端服务器获得英雄数据，我们就不得不修改 `heroes` 的实现，并要修改每个用到了 `HEROES` 模拟数据的地方。

我们来制作一个服务，把获取英雄数据的代码封装起来。

因为服务是一个 [分离关注点](#)，我们建议你把服务代码放到它自己的文件里。

到 [这个笔记](#) 看更多信息。

app/heroes/hero.service.ts

```
1. import { Injectable } from '@angular/core';
2.
3. import { HEROES }      from './mock-heroes';
4.
5. @Injectable()
6. export class HeroService {
7.   getHeroes() { return HEROES; }
8. }
```

我们的 `HeroService` 暴露了 `getHeroes` 方法，用于返回跟以前一样的模拟数据，但它的消费者不需要知道这一点。

注意服务类上面这个 `@Injectable()` 装饰器。我们 [很快](#) 会讨论它的用途。

我们甚至没有假装这是一个真实的服务。如果我们真的从一个远端服务器获取数据，这个 API 必须是异步的，可能得返回 [ES2015 承诺 \(Promise\)](#)。我们也需要被迫重新处理组件如何消费该服务的方式。通常这个很重要，但是我们目前的故事不需要。

在 Angular 2 中，服务只是一个类。除非我们把它注册进一个 Angular 注入器，否则它没有任何特别之处。

配置注入器

我们并不需要自己创建一个 Angular 注入器。Angular 在启动期间会自动为我们创建一个全应用级注入器。

app/main.ts (excerpt)

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

我们必须先注册 **提供商 Provider** 来配置注入器，这些提供商为我们的应用程序创建所需服务。 我们将在本章的稍后部分解释什么是 **提供商**。在此之前，我们先来看一个在启动期间注册提供商的例子。

我们或者在 **NgModule** 中注册提供商，或者在应用组件中。

在 **NgModule** 中注册提供商。

在 **AppModule** 中，我们注册了 `Logger`、`UserService` 和 `APP_CONFIG` 提供商。

app/app.module.ts

```
@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    AppComponent,
    CarComponent,
    HeroesComponent,
    HeroListComponent,
    InjectorComponent,
    TestComponent,
    ProvidersComponent,
    Provider1Component,
    Provider3Component,
```

```

    Provider4Component,
    Provider5Component,
    Provider6aComponent,
    Provider6bComponent,
    Provider7Component,
    Provider8Component,
    Provider9Component,
    Provider10Component,
],
providers: [
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
bootstrap: [ AppComponent, ProvidersComponent ]
})
export class AppModule { }

```

在组件中注册提供商

下面是更新的 `HeroesComponent`，它注册了 `HeroService`。

app/heroes/heroes.component.ts

```

import { Component }           from '@angular/core';
import { HeroService }         from './hero.service';

@Component({
  selector: 'my-heroes',
  providers: [HeroService],
  template: `
    <h2>Heroes</h2>
    <hero-list></hero-list>
  `
})
export class HeroesComponent { }

```

该用 NgModule 还是应用组件？

一方面，NgModule 中的提供商是被注册到根注入器的。这意味着任何注册到 NgModule 上的提供商都可以被整个应用访问到。

另一方面，注册到应用组件上的只在该组件及其各级子组件中可用。

我们希望 APP_CONFIG 服务能在整个应用中可用，而 HeroService 只需在 **英雄** 特性区可用，而其它地方都不用。

参见 NgModule FAQ 一章的 [我该把“全应用级”提供商加到根模块 AppModule 还是根组件 AppComponent ？](#)

为注入准备 HeroListComponent

HeroListComponent 应该从注入进来的 HeroService 获取英雄数据。遵照依赖注入模式的要求，组件必须在它的构造函数中请求这些服务，就像我们以前解释过的那样。只是个小改动：

```
1. import { Component } from '@angular/core';
2.
3. import { Hero } from './hero';
4. import { HeroService } from './hero.service';
5.
6. @Component({
7.   selector: 'hero-list',
8.   template: `
9.     <div *ngFor="let hero of heroes">
10.       {{hero.id}} - {{hero.name}}
11.     </div>
12.   `
13. })
14. export class HeroListComponent {
15.   heroes: Hero[];
16.
17.   constructor(heroService: HeroService) {
```

```
18.     this.heroes = heroService.getHeroes();
19.
20. }
```

来看构造函数

往构造函数中添加一个参数并不是这里所做的一切。

```
constructor(heroService: HeroService) {
  this.heroes = heroService.getHeroes();
}
```

注意构造函数参数有类型 `HeroService`，并且 `HeroListComponent` 类有一个 `@Component` 装饰器（往上翻可以确认）。另外，记得父级组件（`HeroesComponent`）有 `HeroService` 的 `providers` 信息。

该构造函数类型、`@Component` 装饰器、父级的 `providers` 信息这三个合起来，一起告诉 Angular 的注入器，在任何时候新建一个新的 `HeroListComponent` 的时候，注入一个 `HeroService` 的实例。

显性注入器的创建

当我们在上面介绍注入器的时候，我们展示了如何使用它创建一个新 `Car`。这里，我们也展示一下如何显性的创建这样的注入器：

```
injector = ReflectiveInjector.resolveAndCreate([Car, Engine,
Tires]);
let car = injector.get(Car);
```

但无论在《英雄指南》还是其它范例中，我们都没有发现这样的代码。在必要时，我们可以写 使用显式注入器的代码，但却很少这样做。当 Angular 为我们创建组件时——无论通过像 `<hero-list></hero-list>` 这样的 HTML 标签还是通过 路由 导航到组件——它都会自己管理好注入器的创建和调用。只要让 Angular 做好它自己的工作，我们就能安心享受“自动依赖注入”带来的好处。

单例服务

在一个注入器的范围内，依赖都是单例的。在我们这个例子中，一个单一的 `HeroService` 实例被 `HeroesComponent` 和它的子组件 `HeroListComponent` 共享。

然而，Angular DI 是一个分层的依赖注入系统，这意味着被嵌套的注入器可以创建它们自己的服务实例。要了解更多知识，参见 多级依赖注入器 一章。

测试组件

我们前面强调过，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，在构造函数的参数中列出依赖就是我们需要做的一切。

比如，我们可以使用一个 mock 服务来创建新的 `HeroListComponent` 实例，以便我们可以在测试中操纵它：

```
let expectedHeroes = [{name: 'A'}, {name: 'B'}]
let mockService = <HeroService> {getHeroes: () => expectedHeroes }

it('should have heroes when HeroListComponent created', () => {
  let hlc = new HeroListComponent(mockService);
  expect(hlc.heroes.length).toEqual(expectedHeroes.length);
});
```

要学习更多知识，参见 测试。

服务需要别的服务

我们的 `HeroService` 非常简单。它本身不需要任何依赖。

如果它有依赖呢？如果它需要通过一个日志服务来汇报自己的活动呢？我们同样用 **构造函数注入** 模式，来添加一个带有 `Logger` 参数的构造函数。

下面是在原始类的基础上所做的修改：

```
1. import { Injectable } from '@angular/core';
2.
3. import { HEROES }      from './mock-heroes';
4. import { Logger }      from '../logger.service';
5.
6. @Injectable()
7. export class HeroService {
8.
9.   constructor(private logger: Logger) { }
10.
11.  getHeroes() {
12.    this.logger.log('Getting heroes ...');
13.    return HEROES;
14.  }
15.}
```

现在，这个构造函数会要求一个 `Logger` 类的实例注入进来，并且把它存到一个名为 `logger` 的私有属性中。当别人要求获得英雄数据时，我们会在 `getHeroes` 方法中使用这个属性。

为何 `@Injectable()`？

`@Injectable()` 标志着一个类可以被一个注入器实例化。通常来讲，在试图实例化一个没有被标识为 `@Injectable()` 的类时候，注入器将会报告错误。

在这里，我们可以在我们第一版的 `HeroService` 里面省略 `@Injectable()`，因为它没有注入的参数。但是现在我们必须要有它，因为我们的服务有了一个注

入的依赖。我们需要它，因为 Angular 需要构造函数参数的元数据来注入一个 `Logger`。

建议：为每一个服务类都添加 `@Injectable()`

我们建议为每个服务类都添加 `@Injectable()`，包括那些没有依赖因此严格来说并不需要它的。因为：

- **面向未来**：没有必要记得在后来添加了一个依赖的时候添加 `@Injectable()`。
- **一致性**：所有的服务都遵循同样的规则，并且我们不需要考虑为什么少一个装饰器。

注入器同时负责实例化像 `HerosComponent` 这样的组件。为什么我们不标记 `HerosComponent` 为 `@Injectable()` 呢？

我们可以添加它。但是它不是必需的，因为 `HerosComponent` 已经有 `@Component` 装饰器了，这个装饰器类（和我们随后将会学到的 `@Directive` 和 `@Pipe` 一样）是 `InjectableMetadata` 的子类型。实际上，这个 `InjectableMetadata` 装饰器是把一个类标识为注入器实例化的目标。

在运行时，注入器可以从编译后的 JavaScript 代码中读取类的元数据，并使用构造函数的参数类型信息来决定注入什么。

不是每一个 JavaScript 类都有元数据。TypeScript 编译器默认忽略元数据。如果 `emitDecoratorMetadata` 编译器选项为 `true`（在 `tsconfig.json` 中它应该为 `true`），编译器就会在生成的 JavaScript 中，为 **每一个至少拥有一个装饰器的类添加元数据**。

注入器使用一个类的构造元数据来决定依赖类型，该构造元数据就是构造函数的参数类型所标识的。TypeScript 为任何带有一个装饰器的类生成这样的元数据，任何装饰器都生成。当然，使用一个合适的 `Injectable` 装饰器来标识一个类更加有意义。

总要带着括号

总是使用 `@Injectable()` 的形式，不能只用 `@Injectable`。如果忘了括号，我们的应用就会神不知鬼不觉的失败！

创建和注册日志服务

要把日志服务注入到 `HeroService` 中需要两步：

1. 创建日志服务。
2. 把它注册到应用中。

我们的日志服务很简单：

app/logger.service.ts

```
1. import { Injectable } from '@angular/core';
2.
3. @Injectable()
4. export class Logger {
5.   logs: string[] = []; // capture logs for testing
6.
7.   log(message: string) {
8.     this.logs.push(message);
9.     console.log(message);
10.  }
11. }
```

我们很有可能在应用程序的每个角落都需要日志服务，所以把它放到项目的 `app` 目录，并在应用程序模块 `AppModule` 的元数据中的 `providers` 数组里注册它。

app/app.component.ts (excerpt) (providers-logger)

```
providers: [Logger]
```

如果我们忘了注册这个日志服务，Angular 会在首次查找这个日志服务时，抛出一个异常。

```
EXCEPTION: No provider for Logger! (HeroListComponent -> HeroService  
-> Logger)
```

```
(异常: Logger类没有提供商! (HeroListComponent -> HeroService -> Logger))
```

Angular 这是在告诉我们，依赖注入器找不到日志服务的 **提供商**。在创建 `HeroListComponent` 的新实例时需要创建和注入 `HeroService`，然后 `HeroService` 需要创建和注入一个 `Logger` 实例，Angular 需要这个提供商来创建一个 `Logger` 实例。

这个“创建链”始于 `Logger` 的提供商。这个 **提供商** 就是我们下一节的主题。

注入器的提供商们

提供商 **提供** 所需依赖值的一个具体的运行期版本。注入器依靠 **提供商们** 来创建服务的实例，它会被注入器注入到组件或其它服务中。

我们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此服务。

以前，我们通过 `AppModule` 元数据中的 `providers` 数组注册过 `Logger` 服务，就像这样：

```
providers: [Logger]
```

有很多方式可以 **提供** 一些表现和行为像 `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——它有正确的形态，并且它设计出来就是等着被创建的。但它不是唯一的选项。

我们可以使用其它备选提供商来配置这个注入器，只要它们能交付一个对象，其行为像 `Logger` 就可以了。我们可以提供一个替身类。也就是可以提供一个像 `logger` 的对象。我们可以给它一个提供商，让它调用一个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。

最重要的是：当注入器需要一个 `Logger` 时，它得先有一个提供商。

Provider 类和 provide 对象常量

我们像下面一样写 `providers` 数组：

```
providers: [Logger]
```

这其实是一个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 }]
```

第一个是 **令牌 token**，它作为键值 key 使用，用于定位依赖值，以及注册这个提供商。

第二个是一个 provider definition object。 我们可以把它看做一个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，我们会请求一个不同的类来提供服务。 下列代码告诉注入器：当有人请求一个 `Logger` 时，请返回一个 `BetterLogger`。

```
[{ provide: Logger, useClass: BetterLogger }]
```

带依赖的类提供商

也许一个 `EvenBetterLogger` (更好的日志) 可以在日志消息中显示用户名。 这个日志服务从一个注入进来的 `UserService` 中取得用户， `UserService` 通常也会在应用级被注入。

```

@Injectable()
class EvenBetterLogger extends Logger {
  constructor(private userService: UserService) { super(); }

  log(message: string) {
    let name = this.userService.user.name;
    super.log(`Message to ${name}: ${message}`);
  }
}

```

就像我们在 `BetterLogger` 中那样配置它。

```
[ UserService,
{ provide: Logger, useClass: EvenBetterLogger }]
```

别名类提供商

假设一个老的组件依赖于一个 `OldLogger` 类。`OldLogger` 有和 `NewLogger` 相同的接口，但是由于某些原因，我们不能升级这个老组件并使用它。

当老的 组件想使用 `OldLogger` 记录消息时，我们希望改用 `NewLogger` 的单例对象来记录。

不管组件请求的是新的还是老的日志服务，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` 的一个别名。

我们当然不会希望应用中有两个 `NewLogger` 的不同实例。不幸的是，如果我们尝试通过 `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
let silentLogger = {
  logs: ['silent logger says "Shhhh!". Provided via "useValue"'],
  log: () => {}
};
```

于是我们可以通过 `useValue` 选项来注册一个提供商，它会让这个对象直接扮演 logger 的角色。

```
{ provide: Logger, useValue: silentLogger } ]
```

在 [非类依赖](#) 和 [OpaqueToken](#) 查看更多 `useValue` 的例子。

工厂提供商

有时我们需要动态创建这个依赖值，因为它所需要的信息我们直到最后一刻才能确定。比如，也许这个信息会在浏览器的会话中不停的变化。

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，请呼叫 **工厂提供商**。

我们通过添加一个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**。只有获得授权的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。这个授权可能在一个单一的应用会话中被改变，比如我们改用另一个用户的身份登录时。

和 EvenBetterLogger 不同，我们不能把 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。

为什么？我们也不知道。这样的事经常发生。

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。

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);
}
```

我们可以注入 Logger，但是我们不能注入逻辑型的 isAuthorized。我们不得不通过通过一个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：

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`，但是我们的工厂方法可以。

我们同时把 `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：

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` 属性是一个 提供商令牌 数组。`Logger` 和 `UserService` 类作为它们自身提供商的令牌。注入器解析这些令牌，并且把相应的服务注入到工厂函数中相应的参数中去。

注意，我们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。只要需要，我们就可以使用这个变量注册 `HeroService`，无论在哪。

在这个例子中，我们只在 `HeroesComponent` 中需要它，这里，它代替了元数据 `providers` 数组中原来的 `HeroService` 注册。我们来对比一下新的和老的实现：

```
1. import { Component }           from '@angular/core';
2.
3. import { heroServiceProvider } from './hero.service.provider';
4.
5. @Component({
6.   selector: 'my-heroes',
7.   template: `
8.     <h2>Heroes</h2>
9.     <hero-list></hero-list>
10.    `,
11.   providers: [heroServiceProvider]
12. })
13. export class HeroesComponent { }
```

依赖注入令牌

当我们为注入器注册一个提供商时，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一个依赖时被引用到。令牌就是这个映射表中的键值 key。

在以前的所有范例中，依赖值都是一个类 **实例**，并且类的 **类型** 是它自己的查找键值。这种情况下，我们实际上是直接从注入器中以 `HeroService` 类型作为令牌，来获取一个 `HeroService` 实例。

```
heroservice: HeroService = this.injector.get(HeroService);
```

写一个需要基于类的依赖注入的构造函数对我们来说是很幸运的。我们只要以 `HeroService` 类为类型，定义一个构造函数参数，Angular 就会知道把跟 `HeroService` 类令牌关联的服务注入进来：

```
constructor(heroservice: HeroService)
```

这是一个特殊的规约，因为我们考虑到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？有时候我们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（比如应用程序的标题，或者一个网络 API 终点的地址）定义配置对象，但是这些配置对象不总是类的实例。它们可能是对象，比如下面这个：

app/app-config.ts (excerpt)

```
export interface AppConfig {
  apiEndpoint: string;
  title: string;
}

export const HERO_DI_CONFIG: AppConfig = {
  apiEndpoint: 'api.heroes.com',
  title: 'Dependency Injection'
};
```

我们想让这个 `config` 对象在注入时可用。我们已经知道可以使用一个 [值提供商](#) 来注册一个对象。

但是这种情况下我们要把什么用作令牌呢？我们没办法找一个类来当做令牌，因为没有 `Config` 类。

TypeScript 接口不是一个有效的令牌

`CONFIG` 常量有一个接口：`Config`。不幸的是，我们不能把 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){ }
```

如果我们是在一个强类型的语言中使用依赖注入，这会看起来很奇怪，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 的一个设计期概念。JavaScript 没有接口。在生成 JavaScript 代码时，TypeScript 的接口就消失了。在运行期，没有接口类型信息可供 Angular 查找。

OpaqueToken

解决方案是定义和使用一个 **OpaqueToken**(不透明的令牌)。定义方式类似于这样：

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

export let APP_CONFIG = new OpaqueToken('app.config');
```

我们使用这个 `OpaqueToken` 对象注册依赖的提供商：

```
providers: [{ provide: APP_CONFIG, useValue: HERO_DI_CONFIG }]
```

现在，在 `@Inject` 的帮助下，我们可以把这个配置对象注入到任何需要它的构造函数中：

```
constructor(@Inject(APP_CONFIG) config: AppConfig) {
  this.title = config.title;
}
```

虽然 `Config` `AppConfig` 接口在依赖注入过程中没有任何作用，但它为该类中的配置对象提供了强类型信息。

或者我们在顶级组件 `AppComponent` 中提供并注入这个配置对象。

app/app.module.ts (ngmodule-providers)

```
providers: [
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
```

可选依赖

我们的 `HeroService` 需要一个 `Logger`，但是如果它可以不用一个 `Logger` 就行呢？我们可以通过把构造函数的参数标记为 `@Optional()` 来告诉 Angular 该依赖是可选的：

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

```
constructor(@Optional() private logger: Logger) {
  if (this.logger) {
    this.logger.log(some_message);
  }
}
```

当使用 `@Optional()` 时，我们的代码必须要为一个空值做准备。如果我们不在组件或父级组件中注册一个 `logger` 的话，注入器会设置该 `logger` 的值为空 `null`。

总结

在本章中，我们学习了 Angular 依赖注入的基础。我们可以注册很多种类的提供商，还知道了该如何通过添加构造函数的参数来请求一个被注入对象（比如服务）。

Angular 依赖注入比我们描述的更能干。我们还可以学到它的更多高级特性，从它对嵌套注入器的支持开始，参见 [多级依赖注入](#) 一章。

附录：直接使用注入器工作

这里的 `InjectorComponent` 直接使用了注入器，但我们很少直接使用注入器工作。

app/injector.component.ts

```
1.  @Component({
2.    selector: 'my-injectors',
3.    template: `
4.      <h2>Other Injections</h2>
5.      <div id="car">{{car.drive()}}</div>
6.      <div id="hero">{{hero.name}}</div>
7.      <div id="rodent">{{rodent}}</div>
8.    `,
9.    providers: [Car, Engine, Tires, heroServiceProvider, Logger]
10.  })
11. export class InjectorComponent {
12.   car: Car = this.injector.get(Car);
13.
14.   heroService: HeroService = this.injector.get(HeroService);
15.   hero: Hero = this.heroService.getHeroes()[0];
16.
17.   constructor(private injector: Injector) { }
18.
19.   get rodent() {
20.     let rousDontExist = `R.O.U.S.'s? I don't think they exist!`;
21.     return this.injector.get(ROUS, rousDontExist);
22.   }
23. }
```

`Injector` 本身是一个可注入的服务。

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。然后组件向注入进来的这个注入器请求它所需的服务。

注意，这些服务本身没有被注入到组件中，它们是通过调用 `injector.get` 获得的。

`get` 方法如果解析不出所请求的服务，它就会抛出一个异常。我们还可以带上第二个参数（如果服务没找到，就把它作为默认值返回）调用 `get`，在该例子中，我们获取了一个服务（`ROUS`），它没有在这个注入器或它的任何祖先中注册过。

我们刚描述的这项技术是 [服务定位器模式](#) 的一个范例。

我们要 **避免使用** 此技术，除非我们确实需要它。它会鼓励鲁莽的方法，就像我们在这里看到的。它难以解释、理解和测试。仅通过阅读构造函数，我们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。我们会被迫深入它的实现，才可能明白它都做了啥。

Framework developers may take this approach when they must acquire services generically and dynamically.

附录：为什么我们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。

如果我们蔑视这个建议，并且——比如说——把 `HeroService` 和 `HeroesComponent` 组合在同一个文件里，**就得把组件定义放在后面！** 如果我们把组件定义在了服务的前面，就会在运行时获得一个空指针错误。

在 `forwardRef()` 方法的帮助下，我们实际上也可以先定义组件。它的原理解释在这个 [博客](#) 中。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。

下一步

[模板语法](#)

模板语法

学习如何写模板来显示数据，以及在数据绑定的帮助下响应用户事件。

我们的 Angular 应用管理着用户之所见和所为，并通过 Component 类的实例 (**component**) 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，很多用户都熟悉了组件 / 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。

我们来看看写视图的模板都需要什么。我们将覆盖模板语法中的下列基本元素：

- [HTML](#)
- [插值表达式](#)
- [模板表达式](#)
- [模板语句](#)
- [绑定语法](#)
- [属性绑定](#)
- [HTML 属性、 class 和 style 绑定](#)
- [事件绑定](#)
- [双向数据绑定](#)
- [使用 NgModel 进行双向数据绑定](#)
- [内置指令](#)
 - [NgClass](#)
 - [NgStyle](#)
 - [NgIf](#)
 - [NgSwitch](#)
 - [NgFor](#)
- [* 与 <template>](#)
- [模板引用变量](#)

- 输入输出属性
- 模板表达式操作符
 - 管道
 - “安全导航操作符” (?)

这个 [在线例子](#) 演示了本章中描述的所有语法和代码片段。

HTML

HTML 是 Angular 模板的“语言”。我们的 [“快速起步”](#) 应用就有一个模板是纯 HTML 的：

```
<h1>My First Angular App</h1>
```

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。）

有些合法的 HTML 被用在一个模板中是没有意义的。`<html>`、`<body>` 和 `<base>` 元素在我们的舞台上中并没有扮演有用的角色。基本上所有其它的元素都被一样使用。

我们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。接下来我们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。

我们拿数据绑定的第一种形式——插值表达式——来看看模板的 HTML 可以有多丰富。

插值表达式

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

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

我们使用插值表达式来把计算所得的字符串插入成 HTML 元素标签内的文本或对标签的属性进行赋值。

```
<h3>  
{{title}}
```

```

</h3>
```

在括号之间的“素材”，通常是组件属性的名字。Angular会用组件中同名属性的字符串值，替换这个名字。在这个例子中，Angular计算 title 和 heroImageUrl 属性的值，并把它们填在空白处。首先显示一个粗体的应用标题，然后显示英雄的图片。

更一般化的说法是：括号间的素材是一个 **模板表达式**，Angular首先**对它求值**然后**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：

```
<!-- "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所属组件的方法，就像下面用的 `getVal()`：

```
<!-- "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，它把这个组合出来的插值结果赋给一个 **元素或指令的属性**。

表面上看，我们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，并且这个错误很少给我们带来麻烦。但严格来讲，这是不对的。插值表达式是一个特殊的语法，Angular把它转换成了一个 **属性绑定**，我们后面将会解释这一点。

但是，让我们先仔细审视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，并且把它赋值给绑定目标的一个属性，这个绑定目标可能是一个HTML元素、一个组件或一个指令。

当我们写 `{{1 + 1}}` 时，我们往插值表达式的括号中放进了一个模板表达式。我们会在 **属性绑定**中再次看到模板表达式，它出现在 = 右侧的引号中，看起来像这样：

[property]="expression"。

我们用来写模板表达式的是一个看起来很像 JavaScript 的语言。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
- 自增或自减操作符 (++ 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

也许更让人吃惊的是，模板表达式不能引用全局命名空间中的任何东西。它们不能引用 window 或 document。它们不能调用 console.log 或 Math.max。它们被局限于只能访问来自表达式上下文中的成员。

典型的 表达式上下文 就是 [这个组件的实例](#)，它是各种绑定值的来源。

当看到包裹在双花括号中的 title({{title}}) 时，我们就知道 title 是这个数据绑定组件中的一个属性。当看到 [disabled]="isUnchanged" 中的 isUnchanged 时，我们就知道我们正在引用该组件的 isUnchanged 属性。

组件本身通常就是表达式的 [上下文](#)，这种情况下，模板表达式通常会引用那个组件。

表达式的上下文也包括组件之外的对象。[模板引用变量](#) 就是备选的上下文对象之一。

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。 我们永远不应该担心读取一个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 执行模板表达式比我们想象的频繁。 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。 当计算比较昂贵时，应该考虑缓存那些从其它值计算得出的值。

非常简单

虽然写相当复杂的模板表达式确实是可能的，但我们真的不该那样做。

一个属性名或方法调用属于常态。 偶尔的逻辑取反 (!) 也还凑合。 其它情况下，组件应该自己实现应用逻辑和业务逻辑，它将让开发和测试变得更容易。

幂等性

一个 **幂等的** 表达式更加理想，因为它没有副作用，而且能提升 Angular 变更检测的性能。

用 Angular 的术语说，一个幂等的表达式应该总是返回 **完全相同的东西**，直到它所依赖的值中有一个变了。

在单独的一次消息循环中，被依赖的值不应该改变。 如果一个幂等的表达式返回了一个字符串或数字，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** 对象。

我们将在 [事件绑定](#) 一节看到模板语句，它出现在 `=` 号右侧的引号中，就像这样：

```
(event)="statement" .
```

模板语句 **有副作用**。这就是我们如何从用户的输入来更新应用状态。不然，响应一个事件就没有什么意义了。

响应事件是 Angular 中“单向数据流”的另一面。在事件循环的这一回合中，我们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** 也是一个一个看起来很像 JavaScript 的语言。模板语句解析器和模板表达式解析器有所不同，它的特别之处在于它既支持基本赋值 (`=`) 又支持使用分号 (`;`) 和逗号 (`,`) 把表达式串起来。

但无论如何，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，比如 `+=` 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——典型的就是我们正在绑定事件的那个 **组件的实例** 中的内容。

模板语句无法引用全局命名控件的任何东西。它们不能引用 `window` 或者 `document`。它们不能调用 `console.log` 或者 `Math.max`。

`(click)="onSave()"` 中的 **onSave** 就是数据绑定组件实例中的一个方法。

语句上下文可以包含组件之外的对象。 **模板引用对象** 就是这些备选上下文对象中的一个。在事件绑定语句中，我们将频繁的看到被保留的 `$event` 符号，它代表来自所触发事件的“消息”或“有效载荷”。

语句指南

和表达式一样，要避免写复杂的模板语句。单一函数调用或者单一属性访问应该是常态。

现在，我们对模板表达式和语句有了一点感觉，是时候学习除了插值表达式之外多种多样的数据绑定语法了。

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程序的数据值。虽然我们能往 HTML 推送值或者从 HTML 拉取值，但是如果我们将这些琐事放进数据绑定框架，应用就会更容易写、读以及维护。我们只要简单的在绑定源和目标 HTML 元素之间建立绑定，框架就会完成这项工作。

Angular 提供很多种数据绑定，我们将在本章中逐一讨论它们。首先，我们从高层视角来看看 Angular 数据绑定和它的语法。

根据数据流的方向，我们可以把所有绑定归为三类。每一类都有它独特的语法：

数据流方向	语法	绑定类型
单向 从数据源 到视图目标	<pre>{{expression}} [target] = "expression" bind-target = "expression"</pre>	插值表达式 Property Attribute
类		
样式		
单向 从视图目标 到数据源	<pre>(target) = "statement" on-target = "statement"</pre>	事件

双向

```
[target] =  
"expression"  
bindon-target =  
"expression"
```

双向

译注

由于HTML的Attribute和DOM的Property在中文中都被翻译成了“属性”，无法加以区分，而接下来的部分重点是对它们进行比较的。

我们无法改变历史，因此，在本章的翻译中，我们保留它们的英文形式，而不加翻译，以免混淆。在本章中，如果有只提“属性”的地方，一定是指Property，因为在Angular 2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，我们就仍统一译为“属性”，区分不明显的，我们会加注英文

除了插值表达式之外的绑定类型，在等号左边都有一个 **目标名**，无论是包在括号中（`[]`、`()`）还是用带前缀（`bind-`、`on-`、`bindon-`）的形式。

什么是“目标”？在回答这个问题之前，我们必须先挑战下自我，尝试用另一种方式来审视模板中的 HTML。

一个新的思维模型

数据绑定的威力和允许我们用自定义语言来扩展 HTML 词汇的能力，容易误导我们把模板 HTML 当成 **HTML+**。

也对，它是 **HTML+**。但它也跟我们用过的那些 HTML 有着显著的不同。我们需要一种新的思维模型。

在通常的 HTML 开发过程中，我们使用 HTML 元素创建一个视觉结构，并通过把字符串常量设置给元素的 Attribute 来修改那些元素。

```
<div class="special">Mental Model</div>

<button disabled>Save</button>
```

在 Angular 模板中，我们使用同样的方法来创建一个模板结构和初始化 Attribute 值。

然后，我们学着用包含 HTML 模板的组件创建新元素，并把它们当作原生 HTML 元素在模板中使用。

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

这就是 HTML+。

现在我们开始学习数据绑定。我们碰到的第一种数据绑定看起来是这样的：

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

我们首先会注意到这个怪异的括号注解。接下来，直觉告诉我们，我们正在绑定到这个按钮的 `disabled` Attribute。并且把它设置为组件的 `isUnchanged` Property 的当前值。

但我们的直觉是错的！我们的日常 HTML 思维模式在误导我们。实际上，一旦我们开始数据绑定，我们就不再跟 Attribute 打交道了。我们并不是在设置 Attribute，而是在设置 DOM 元素、组件和指令的 Property。

HTML 的 Attribute vs. DOM 的 Property

要想理解 Angular 绑定如何工作，清楚 HTML Attribute 和 DOM Property 之间的区别是至关重要的。

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。 `id` 就是一个例子。
- 有些 HTML Attribute 没有对应的 Property 。 `colspan` 就是一个例子。
- 有些 DOM Property 没有对应的 Attribute 。 `textContent` 就是一个例子。
- 大量 HTML Attribute 看起来映射到了 Property但却不像我们想象的那样 !

尤其最后一种更让人困惑.....除非我们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 , 它创建了一个对应的 DOM 节点 , 它的 `value` Property 被 **初始化为 “ Bob ”**。

当用户在输入框中输入“ Sally ”时 , DOM 元素的 `value` Property 变成了“ Sally ”。但是这个 HTML `value` Attribute 保持不变。如果我们通过 `input.getAttribute('value')` // 返回 "Bob" 语句获取这个 input 元素的 Attribute , 就会明白这一点。

HTML Attribute `value` 指定了 **初始** 值 ; DOM 的 `value` Property 是 **当前** 值。

`disabled` Attribute 是另一个古怪的例子。按钮的 `disabled` Property 是 `false` , 因为默认情况下按钮是可用的。当我们添加 `disabled` Attribute 时 , 它只要出现就表示按钮的 `disabled` Property 是 `true` , 于是按钮就被禁用了。

添加或删除 `disabled` 时 , Attribute 会禁用或启用这个按钮。但 Attribute 的值无关紧要 , 这就是我们为什么没法通过 `<button disabled="false"> 仍被禁用 </button>` 这种写法来启用一个按钮的原因。

设置按钮的 `disabled` Property(比如 , 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。

这句话很重要 , 我们得再强调一次 :

模板绑定是通过 Property 和 事件 来工作的 , 而不是 Attribute 。

一个没有 ATTRIBUTE 的世界

在 Angular 2 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，我们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。

请把这个思维模型牢牢的印在脑子里，接下来我们学习什么是“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) 一个 Attribute 名。下面是的汇总表：

绑定类型	目标	范例
Property	元素的 Property	<pre> <hero-detail [hero]="currentHero"></hero- detail></pre>
	组件的 Property	<pre><div [ngclass] = "{selected: isselected}"></div></pre>
	指令的 Property	
事件	元素的事件	<pre><button (click) = "onSave()">Save</button></pre>
	组件的事件	<pre><hero-detail (deleteRequest)="deleteHero()"></hero- detail></pre>
	指令的事件	<pre><div (myClick)="clicked=\$event">click me</div></pre>
双向	事件与 Property	<pre><input [(ngModel)]="heroName"></pre>

Attribute

Attribute

例外情况

```
<button [attr.aria-label]="help">help</button>
```

CSS 类

class

property

```
<div [class.special]="isSpecial">Special</div>
```

样式

style

property

```
<button [style.color] = "isSpecial ? 'red' : 'green'">
```

让我们从结构性云层中走出来，看看每种绑定类型的具体情况。

属性绑定

当要把一个视图元素的属性设置为 模板表达式 时，我们就要写模板的 属性绑定 。

最常用的属性绑定是把元素的属性设置为组件中属性的值。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：

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

另一个例子是当组件说它 `isUnchanged` (未改变) 时禁用一个按钮：

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

另一个例子是设置指令的属性：

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

还有另一个例子是设置一个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）：

```
<hero-detail [hero]="currentHero"></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，从组件中的属性数据，流动到目标元素的属性。

我们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。我们只能设置它。

我们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，我们可以通过 **事件绑定** 来监听它们。

如果我们不得不读取目标元素上的属性或调用它的某个方法，我们得用另一种技术。

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 `img` 元素的 `src` 属性。

```
<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**：

```

```

目标的名字总是属性的名字。即使它看起来和别的名字一样。我们看到 `src` 时，可能会把它当做 `Attribute`。不！它不是！它是 `image` 元素的属性名。

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：

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

严格来说，Angular 正在匹配一个指令的 `input` 的名字。这个名字是指令的 `inputs` 数组中所列的名字之一，或者是一个带有 `@Input()` 装饰器的属性。这样的 `inputs` 被映射到了指令自己的属性。

如果名字没有匹配上一个已知指令或元素的属性，Angular 就会报告一个“未知指令”的错误。

消除副作用

正如我们曾讨论过的，计算模板表达式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。我们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，我们的表达式也可能会调用一个具有副作用的属性或方法。但 Angular 没法知道这一点，也没法防止我们误用。

表达式中可以调用像 `getFoo()` 这样的方法。只有我们才知道 `getFoo()` 干了什么。如果 `getFoo()` 改变了什么，而我们把它绑定在什么地方，我们就可能把自己坑了。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，并抛出一个警告型错误。更具普遍性的建议是：只使用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一个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，那我们就要在属性绑定中精确的给它一个 `Hero` 对象：

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

别忘了方括号

括号会告诉 Angular 要计算模板表达式。如果我们忘了括号，Angular 就会把这个表达式当做一个字符串常量看待，并且用该字符串来 **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：

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

一次性字符串初始化

当下列条件满足时，我们 **应该** 省略括号：

- 目标属性接受字符串值。
- 我们想合并到模板中的字符串是一个固定值。
- 这个初始值永不改变。

我们经常这样在标准 HTML 中用这种方式初始化 Attribute，并且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` 属性初始化成了一个固定的字符串，而不是模板表达式。Angular 设置它，然后忘记它。

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

作为对比，这个 `[hero]` 绑定就保持了到组件的 `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？

我们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：

```

<p> is the <i>property bound</i> image.</p>

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

```

在多数情况下，插值表达式是一个更方便的备选项。实际上，在渲染视图之前，Angular 就把这些插值表达式翻译成了对应的属性绑定形式。

没有技术上的理由能决定哪种形式更好。我们倾向于可读性，所以倾向于插值表达式。我们建议建立组织级的代码风格规定，然后选择一种形式，既能遵循规则，又能让手头的任务做起来更自然。

内容安全

假设下面的 **恶毒内容**

```
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。

```

<p><span>"{{evilTitle}}</span> is the <i>interpolated</i> evil title.</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，但是它们两个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

我们可以通过 **Attribute 绑定** 来直接设置 Attribute 的值。

这是“绑定到目标属性”这条规则中唯一的例外。这是唯一的一个能创建和设置 Attribute 的绑定形式。

在本章中，我们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，我们不得不使用 Attribute 绑定。

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

如果我们想写出类似下面这样的东西，这一现状会令我们痛苦：

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

我们会得到这个错误：

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

模板解析错误：不能绑定到'colspan'，因为它不是已知的原生属性

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**，而不是 Attribute。

我们需要 Attribute 绑定来创建和绑定到这样的 Attribute。

Attribute 绑定在语法上类似于属性绑定。但方括号中的部分不是一个元素的属性名，而是由一个 `attr` 前缀，紧跟着一个点（`.`），再跟着 Attribute 的名字组成。我们可以通过一个能求值为字符串的表达式来设置 Attribute 的值。

这里我们把 `[attr.colspan]` 绑定成一个通过计算得到的值：

```
<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 table 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：

```
<!-- 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**，我们可以从元素的 `class` Attribute 上添加和移除 CSS 类名。

CSS 类绑定在语法上类似于属性绑定。但方括号中的部分不是一个元素的属性名，而是包括一个 `class` 前缀，紧跟着一个点（`.`），再跟着 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：

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

我们可以把它改写为一个绑定到所需 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）。

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

最后，我们可以绑定到一个特定的类名。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。

```
<!-- 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>
```

虽然这是一个切换单一类名的好办法，但我们通常更喜欢使用 [NgClass 指令](#) 来同时管理多个类名。

样式绑定

通过 [样式绑定](#)，我们可以设置内联样式。

样式绑定在语法上类似于属性绑定。但方括号中的部分不是一个元素的属性名，而是包括一个 **style** 前缀，紧跟着一个点(.)，再跟着 CSS 样式的属性名。形如： [style.style-property]

。

```
<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，我们可以根据条件用“em”和“%”来设置字体大小的单位。

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

虽然这是一个设置单一样式的好办法，但我们通常更喜欢使用 [NgStyle 指令](#) 来同时设置多个内联样式。

注意，一个 **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](#)，比如 `fontSize`。

事件绑定

我们前面遇到过的那些绑定的数据流都是单向的：[从组件到元素](#)。

用户不会只盯着屏幕看。它们会在输入框中输入文本。它们会从列表中选取条目。它们会点击按钮。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正确的事件，比如击键、鼠标移动、点击和触屏。我们可以通过 Angular 的事件绑定来定义我们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**，和右侧一个引号中的 **模板语句** 组成。下列事件绑定监听按钮的点击事件。无论什么时候，发生点击时，都会调用组件的 `onSave()` 方法。

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

目标事件

圆括号中的名称——比如 `(click)` ——标记出了目标事件。在下面这个例子中，目标是按钮的 `click` 事件。

```
<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**：

```
<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：

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

别名 `input/output 属性` 章节有更多关于该 `myClick` 指令的解释。

如果这个名字没能匹配到元素事件或已知指令的输出型属性，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那些针对事件想作出相应处理的接收器，例如从一个 HTML 控件中取得一个值，并存入一个模型。

这种绑定会通过一个 **名叫 `$event` 的事件对象** 传达关于此事件的信息（包括数据值）。

事件对象的形态取决于目标事件。如果目标事件是一个原生 DOM 元素事件，`$event` 就是一个 **DOM 事件对象**，它有像 `target` 和 `target.value` 这样的属性。

考虑这个范例：

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

我们在把输入框的 `value` 绑定到 `firstName` 属性，并且我们正在通过绑定到输入框的 `input` 事件来监听更改。当用户造成更改时，`input` 事件被触发，并且在一个包含了 DOM 事件对象的 `$event` 的上下文中执行这条语句。

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

如果这个事件属于指令（回想一下：组件是指令的一种），那么 `$event` 便具有指令中生成的形态（译注：比如 `inputs`）。

使用 `EventEmitter` 实现自定义事件

指令使用典型的 Angular `EventEmitter` 来触发自定义事件。指令创建一个 `EventEmitter` 实例，并且把它作为一个属性暴露出来。指令调用 `EventEmitter.emit(payload)` 来触发事件，传进去的消息载荷可以是任何东西。父指令通过绑定到这个属性来监听这个事件，并且通过 `$event` 对象来访问这个载荷。

假设一个 `HeroDetailComponent`，它展示英雄的信息，并响应用户的动作。虽然 `HeroDetailComponent` 有一个删除按钮，但它自己并不知道该如何删除这个英雄。最好的做法是触发一个事件来报告“删除用户”的请求。

这里是来自 `HeroDetailComponent` 的相关摘要：

HeroDetailComponent.ts (template)

```
template: `
<div>
```

```


  {{prefix}} {{hero?.fullName}}
</span>
<button (click)="delete()">Delete</button>
</div>

```

HeroDetailComponent.ts (delete logic)

```

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

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

```

组件定义了一个 `deleteRequest` 属性，它是一个 `EventEmitter` 实例。（译注：
`deleteRequest` 属性是导出 `Output` 属性，是组件与父级组件交互的主要方式之一。参见 [输入和输出属性](#) 和 [父组件监听子组件的事件](#)。我们需要用 `@Output()` 来装饰它，或者把它添加到组件元数据的 `outputs` 数组中，它才能在父级组件可见。）当用户点击 **删除** 时，组件会调用 `delete()` 方法，这个方法告诉 `EventEmitter`，发出一个 `Hero` 对象。

现在，想象作为宿主的父组件绑定到了 `HeroDetailComponent` 的 `deleteRequest` 事件。

```

<hero-detail (deleteRequest)="deleteHero($event)" [hero]="currentHero">
</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` 方法有一个副作用：它删除了一个英雄。模板语句的副作用不仅没问题，反而正是我们所期待的。

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。这都是好事。

双向数据绑定

我们经常需要显示数据属性，并在用户作出更改时，更新数据属性及其显示。

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)]` 语法则结合了 **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，`[(x)]` 语法是最容易被解释的。下面的 `SizerComponent` 符合这个模式。它有 `size` 属性和伴随的 `sizeChange` 事件：

app/sizer.component.ts

```

1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-sizer',
5.   template: `
6.     <div>
7.       <button (click)="dec()" title="smaller">-</button>
8.       <button (click)="inc()" title="bigger">+</button>
9.       <label [style.fontSize.px]="size">FontSize: {{size}}px</label>
10.    </div>`
11. })
12. export class SizerComponent {
13.   @Input() size: number;
14.   @Output() sizechange = new EventEmitter<number>();
15.
16.   dec() { this.resize(-1); }
17.   inc() { this.resize(+1); }
18.
19.   resize(delta: number) {
20.     const size = +this.size + delta;
21.     this.size = Math.min(40, Math.max(8, size));
22.     this.sizeChange.emit(this.size);
23.   }
24. }
```

`size` 的初始值是通过属性绑定得来的输入框的 `value`。点击按钮，在最大 / 最小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`：

```
<my-sizer [(size)]="fontSize"></my-sizer>
<div [style.fontSize.px]="fontSize">Resizable Text</div>
```

`AppComponent.fontSize` 赋予 `SizerComponent.size` 初始值。点击按钮通过双向绑定更新 `AppComponent.fontSize`。被修改的 `AppComponent.fontSize` 通过 **样式** 绑定，改变文本的显示大小。参见 [在线例子](#)。

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：

```
<my-sizer [size]="fontSize" (sizechange)="fontSize=$event"></my-sizer>
```

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

很清楚，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。

我们希望能在像 `<input>` 和 `<select>` 这样的 HTML 元素上使用双向数据绑定。可惜，原生 HTML 元素不自带像 `x` 值和 `xChange` 事件的模式。

幸运的是，Angular 以 **NgModel** 指令为桥梁，让我们可以在表单元素上使用双向数据绑定。

使用 NgModel 进行双向数据绑定

当开发数据输入表单时，我们经常希望能显示数据属性，并在用户做出变更时，更新属性及其显示。

使用 `NgModel` 指令进行双向数据绑定让它变得更加容易。请看下例：

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

要使用 `NGMODEL`，必须导入 `FORMSMODULE`。

要使用 `ngModel` 做双向数据绑定，得先把 `FormsModule` 导入我们的模块并把它加入 `NgModule` 装饰器的 `imports` 数组。

要学习关于 `FormsModule` 和 `ngModel` 的更多知识，参见 [表单](#) 一章。

下面展示了如何导入 `FormsModule`，让 `[(ngModel)]` 变得可用：

app.module.ts (FormsModule import)

```
1. import { NgModule } from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule } from '@angular/forms';
4.
5. import { AppComponent } from './app.component';
6.
7. @NgModule({
8.   imports: [
9.     BrowserModule,
10.    FormsModule
11.   ],
12.   declarations: [
13.     AppComponent
14.   ],
15.   bootstrap: [ AppComponent ]
16. })
17. export class AppModule { }
```

[(ngModel)] 内幕

回顾 `firstName` 的绑定，最值得注意的是，我们可以通过分别绑定 `<input>` 元素的 `value` 属性和 `'input'` 事件来实现同样的效果。

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

这样很笨拙。谁能记住哪个元素属性用于设置，哪个用于汇报用户更改？我们如何从输入框中提取出当前显示的文本，以便更新数据属性？谁想每次都去查一遍？

`ngModel` 指令通过它自己的 `ngModel` 输入属性和 `ngModelChange` 输出属性隐藏了这些繁琐的细节。

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

`ngModel` 数据属性设置元素的 `value` 属性，并为元素 `value` 的变化而监听 `ngModelChange` 事件属性。

每种元素的特点各不相同，所以 `NgModel` 指令只能在一些特定表单元素上使用，例如输入框，因为它们支持 `ControlValueAccessor`。

除非写一个合适的 **值访问器**，否则我们不能把 `[(ngModel)]` 用在我们自己的自定义组件上。但 **值访问器** 技术超出了本章的范围。对于不能控制其 API 的 Angular 组件或者 Web 组件，我们可能需要为其添加合适的 **value accessor**。

但是对于我们能控制的 Angular 组件来说，这么做就完全没有必要了。因为我们可以指定值和事件属性名字来进行基本的 Angular 双向绑定语法，完全不用 `NgModel`。
。

把 `ngModel` 绑定分离开是一个提升，但我们还能做得更好。

我们不应该被迫两次提起数据属性。Angular 应该能捕捉组件的数据属性并用一条声明来设置它——依靠 `[(ngModel)]`，我们可以这么做：

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

[(ngModel)] 就是我们所需的一切吗？有没有什么理由需要仰仗到它的展开形式？

[(ngModel)] 语法规只能 **设置** 一个数据绑定属性。如果需要做更多或不同的事情，我们得自己写它的展开形式。

来做点淘气的事吧，比如强制让输入值变成大写形式：

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

下面是实际操作中的所有变体形式，包括这个大写版本：

NgModel Binding

Result: Hercules

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

内置指令

前一个版本的 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。

在 Angular 中我们不需要那么多指令。使用更强大、更富有表现力的 Angular 绑定系统，我们其实可以达到同样的效果。如果我们能用一个这样的简单绑定达到目的，为什么还需要创建一个指令来处理点击事件？

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

我们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。我们仍会写自己的指令，只是没那么多了。

下面这部分就检阅一下那些最常用的内置指令。

NgClass

我们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`，我们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。

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

当我们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。

绑定到一个 `key:value` 形式的控制对象，是应用 `NgClass` 的好方式。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。

考虑一个像 `setClasses` 这样的组件方法，用于管理三个 CSS 类的状态。

```
setClasses() {
  let classes = {
    saveable: this.canSave,      // true
    modified: !this.isUnchanged, // false
    special: this.isSpecial,    // true
  };
  return classes;
}
```

现在，我们可以添加一个 `NgClass` 属性绑定，它会调用 `setClasses`，并据此设置元素的类：

```
<div [ngClass]="setClasses()">This div is saveable and special</div>
```

NgStyle

我们可以基于组件的状态动态设置内联样式。 绑定到 `NgStyle` 可以让我们同时设置很多内联样式。

样式绑定 是设置 **单一** 样式值的简单方式。

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

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

我们通过把它绑定到一个 `key:value` 控制对象的形式使用 `NgStyle` 。 对象的每个 `key` 是样式名，它的 `value` 就是能用于这个样式的任何值。

考虑一个类似于 `setStyles` 的组件方法，它返回一个定义三种样式的对象：

```
setStyles() {
  let styles = {
    // CSS property names
    'font-style': this.canSave ? 'italic' : 'normal', // italic
    'font-weight': !this.isUnchanged ? 'bold' : 'normal', // normal
    'font-size': this.isSpecial ? '24px' : '8px', // 24px
  };
  return styles;
}
```

现在我们添加一个 `NgStyle` 属性绑定，让它调用 `setStyles` ，并据此设置元素的样式：

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

NgIf

通过把 `NgIf` 指令绑定到一个真值表达式，我们可以把一个元素的子树（元素及其子元素）添加到 DOM 上。

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

不要忘记了 `ngIf` 前面的星号（`*`）。要了解更多，参见 [* 与 <template>](#)。

绑定到一个假值表达式将从 DOM 中移除元素的子树。

```
<!-- because of the ngIf guard
`nullHero.firstName` never has a chance to fail -->
<div *ngIf="nullHero">Hello, {{nullHero.firstName}}</div>

<!-- Hero Detail is not in the DOM because isActive is false-->
<hero-detail *ngIf="isActive"></hero-detail>
```

可见性和 NGIF 不是一回事

我们可以通过 [类绑定](#) 或 [样式绑定](#) 来显示和隐藏一个元素的子树（元素及其子元素）。

```
<!-- 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 -->
<hero-detail [class.hidden]="isSpecial"></hero-detail>

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

隐藏一个子树和用 `NgIf` 排除一个子树是截然不同的。

当我们隐藏一个子树时，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 中实际移除了这个元素的子树。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 / 隐藏技术用在小型元素树上可能还不错。但在隐藏大树时我们得小心；`NgIf` 可能是更安全的选择。但要记住：永远得先测量，再下结论。

NgSwitch

当需要从 **一组** 可能的元素树中根据条件显示 **一个** 时，我们就把它绑定到 `NgSwitch`。Angular 将只把 **选中的** 元素树放进 DOM 中。

下面是例子：

```
<span [ngSwitch]="toeChoice">
  <span *ngSwitchCase="'Eenie'">Eenie</span>
  <span *ngSwitchCase="'Meanie'">Meanie</span>
  <span *ngSwitchCase="'Miney'">Miney</span>
  <span *ngSwitchCase="'Moe'">Moe</span>
  <span *ngSwitchDefault>other</span>
</span>
```

我们把作为父指令的 `NgSwitch` 绑定到一个能返回 **开关值** 的表达式。本例中，这个值是字符串，但它也可能是任何类型的值。

这个例子中，父指令 `NgSwitch` 控制一组 `` 子元素。每个 `` 或者挂在一个 **匹配值** 表达式上，或者被标记为默认情况。

任何时候，这些 `span` 中最多只有一个会出现在 DOM 中。

如果这个 `span` 的 **匹配值** 与 **开关值** 相等，Angular 就把这个 `` 添加到 DOM 中。如果没有任何 `span` 匹配上，Angular 就把默认的 `span` 添加到 DOM 中。Angular 会移除并销毁所有其它的 `span`。

我们可以用任何其它元素代替本例中的 `span`。那个元素可以是一个带有巨大子树的 `<div>`。只有匹配的 `<div>` 和它的子树会显示在 DOM 中，其它的则会被移除。

这里有三个相互合作的指令：

1. `ngSwitch` : 绑定到一个返回开关值的表达式
2. `ngSwitchCase` : 绑定到一个返回匹配值的表达式
3. `ngSwitchDefault` : 一个用于标记出默认元素的 Attribute

不要 在 `ngSwitch` 的前面放星号 (*)，而应该用属性绑定。

要 把星号 (*) 放在 `ngSwitchCase` 和 `ngSwitchDefault` 的前面。要了解更多信息，参见 [* 与 <template>](#)。

NgFor

`NgFor` 是一个 **重复器** 指令——自定义数据显示的一种方式。

我们的目标是展示一个由多个条目组成的列表。我们定义了一个 HTML 块，它规定了单个条目应该如何显示。我们告诉 Angular 把这个块当做模板，渲染列表中的每个条目。

这里是一个例子，它把 `NgFor` 应用在一个简单的 `<div>` 上：

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

我们也可以把 `NgFor` 应用在一个组件元素上，就像这个例子中一样：

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

不要忘了 `ngFor` 前面的星号 (*)。要了解更多，参见 [* 与 <template>](#)

赋值给 `*ngFor` 的文本是一个用于指导重复器如何工作的“操作指南”。

NGFOR 微语法

赋值给 `*ngFor` 的字符串并不是一个 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，把它存在一个局部变量 `hero` 中，并且在每个迭代中让它对模板 HTML 可用

Angular 把这份“操作指南”翻译成一组“元素和绑定”。

在前面的两个例子中，`ngFor` 指令在 `heroes` 变量上进行迭代（它是由父组件的 `heroes` 属性返回的），以其所在的元素为模板“冲压”出很多实例。Angular 为数组中的每个英雄创建了此模板的一个全新实例。

`hero` 前面的 `let` 关键字创建了一个名叫 `hero` 的模板输入变量。

模板输入变量和 [模板引用变量](#) 不是一回事！

我们在模板中使用这个变量来访问英雄的属性，就像在插值表达式中所做的那样。我们也可以把这个变量传给组件元素上的一个绑定，就像我们对 `hero-detail` 所做的那样。

带索引的 NGFOR

`ngFor` 指令支持一个可选的 `index`，它在迭代过程中会从 0 增长到“当前数组的长度”。我们可以通过模板输入变量来捕获这个 `index`，并把它用在模板中。

下一个例子把 `index` 捕获到了一个名叫 `i` 的变量中，使用它“冲压出”像 "1 - Hercules Son of Zeus" 这样的条目。

```
<div *ngFor="let hero of heroes; let i=index">{{i + 1}} - {{hero.fullName}}</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](#)。

NGFORTRACKBY

`ngFor` 指令有时候会性能较差，特别是在大型列表中。对一个条目的一点小更改、移除或添加，都会导致级联的 DOM 操作。

例如，我们可以通过重新从服务器查询来刷新英雄列表。刷新后的列表可能包含很多（如果不是全部的话）以前显示过的英雄。

我们知道这一点，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，所以它没有选择，只能清理老列表、舍弃那些 DOM 元素，并且用新的 DOM 元素来重建一个新列表。

如果我们给它一个 **追踪** 函数，Angular 就可以避免这种折腾。追踪函数告诉 Angular：我们知道两个具有相同 `hero.id` 的对象其实是同一个英雄。下面就是这样一个函数：

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

现在，把 `NgForTrackBy` 指令设置为那个 **追踪** 函数。

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

追踪 函数不会排除所有 DOM 更改。如果用来判断是否同一个英雄的 **属性** 变化了，Angular 就可能不得不更新 DOM 元素。但是如果这个属性没有变化——而且大多数时候它们不会变化——Angular 就能留下这些 DOM 元素。列表界面就会更加平滑，提供更好的响应。

这里是关于 `NgForTrackBy` 效果的一个插图。

NgForTrackBy

Refresh heroes

First hero: **Hercules**

without trackBy

(1) Hercules Son of Zeus
(2) eenie toe
(3) Meanie Toe
(4) Miny Toe
(5) Moe Toe

with trackBy and semi-colon separator

(1) Hercules Son of Zeus
(2) eenie toe
(3) Meanie Toe
(4) Miny Toe
(5) Moe Toe

* 与 <template>

当我们审视 `NgFor`、`Nglf` 和 `NgSwitch` 内置指令时，我们使用了一种古怪的语法：出现在指令名称前面的星号（`*`）。

`*` 是一种语法糖，它让那些需要借助模板来修改 HTML 布局的指令更易于读写。`NgFor`、`Nglf` 和 `NgSwitch` 都会添加或移除元素子树，这些元素子树被包裹在 `<template>` 标签中。

我们没有看到 `<template>` 标签，那是因为这种 `*` 前缀语法让我们忽略了这个标签，而把注意力直接聚焦在所要包含、排除或重复的那些 HTML 元素上。

在这一节，我们将掀起引擎盖，看看 Angular 是怎样替我们扒掉这个 `*`，并且把这段 HTML 展开到 `<template>` 标签中的。

展开 `*ngIf`

我们可以像 Angular 一样，自己把 `*` 前缀语法展开成 template 语法，这里是 `*ngIf` 的一些代码：

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

`currentHero` 被引用了两次，第一次是作为 `NgIf` 的真 / 假条件，第二次把实际的 `hero` 值传给了 `HeroDetailComponent`。

展开的第一步是把 `ngIf`（没有 `*` 前缀）和它的内容传给一个表达式，再赋值给 `template` 指令。

```
<hero-detail template="ngIf:currentHero" [hero]="currentHero"></hero-detail>
```

下一步，也就是最后一步，是把 HTML 包裹进 `<template>` 标签和一个 `[ngIf]` 属性绑定中：

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

注意，`[hero]="currengHero"` 绑定留在了模板中的子元素 `<hero-detail>` 上。

别忘了括号！

不要误写为 `ngIf="currentHero"`！这种语法会把一个字符串 "currentHero" 赋值给 `ngIf`。在 JavaScript 中，非空的字符串是真值，所以 `ngIf` 总会是 `true`，而 Angular 将永远显示 `hero-detail`……即使根本没有 `currentHero`！

展开 `*ngSwitch`

类似的转换也作用于 `*ngSwitch` 上。我们可以自己解开这个语法糖。这里是一个例子，首先是 `*ngSwitchCase` 和 `*ngSwitchDefault`，然后再解出 `<template>` 标签：

```
<span [ngSwitch]="toeChoice">

  <!-- with *NgSwitch -->
  <span *ngSwitchCase="'Eenie'">Eenie</span>
```

```

<span *ngSwitchCase="'Meanie'">Meanie</span>
<span *ngSwitchCase="'Miney'">Miney</span>
<span *ngSwitchCase="'Moe'">Moe</span>
<span *ngSwitchDefault>other</span>

<!-- with <template> -->
<template [ngSwitchCase]="'Eenie'"><span>Eenie</span></template>
<template [ngSwitchCase]="'Meanie'"><span>Meanie</span></template>
<template [ngSwitchCase]="'Miney'"><span>Miney</span></template>
<template [ngSwitchCase]="'Moe'"><span>Moe</span></template>
<template ngSwitchDefault><span>other</span></template>

</span>

```

`*ngSwitchWhen` 和 `*ngSwitchDefault` 用和 `*ngIf` 完全相同的方式展开，把它们以前的元素包裹在 `<template>` 标签中。

现在，我们应该明白为什么 `ngSwitch` 本身不能用星号 (*) 前缀的原因了吧？它没有定义内容，它的任务是控制一组模板。

上面这种情况下，它管理两组 `NgSwitchCase` 和 `NgSwitchDefault` 指令，一次是 (*) 前缀的版本，一次是展开模板后的版本。我们也期待它显示所选模板的值两次。这正是我们在这个例子中看到的：

NgSwitch Binding

Eenie Meanie Miney Moe ???
 | Pick a toe

展开 `*ngFor`

`*ngFor` 要经历类似的转换。我们从一个 `*ngFor` 的例子开始：

```

<hero-detail *ngFor="let hero of heroes; trackBy:trackByHeroes"
[hero]="hero"></hero-detail>

```

这里是在把 `ngFor` 传进 `template` 指令后的同一个例子：

```
<hero-detail template="ngFor let hero of heroes; trackBy:trackByHeroes"
[hero]="hero"></hero-detail>
```

这里，它被进一步扩展成了包裹着原始 `<hero-detail>` 元素的 `<template>` 标签：

```
<template ngFor let-hero [ngForOf]="heroes"
[ngForTrackBy]="trackByHeroes">
<hero-detail [hero]="hero"></hero-detail>
</template>
```

`NgFor` 的代码相对 `Nglif` 更复杂一点，因为一个重复器有更多活动部分需要配置。这种情况下，我们不得不记着为用于标记列表的 `NgForOf` 指令和 `NgForTrackBy` 指令的进行新建和赋值操作。使用 `*ngFor` 语法比直接写这些展开后的 HTML 本身要简单多了。

模板引用变量

模板引用变量 是模板中对 DOM 元素或指令的引用。

它能在原生 DOM 元素中使用，也能用于 Angular 组件——实际上，它可以和任何自定义 Web 组件协同工作。

引用一个模板引用变量

我们可以在同一元素、兄弟元素或任何子元素中引用模板引用变量。

不要在同一个模版中多次定义同一个变量名，否则运行时的值将会不可预测。

这里是关于创建和消费模板引用变量的另外两个例子：

```
<!-- phone refers to the input element; pass its `value` to an event
handler -->
<input #phone placeholder="phone number">
```

```
<button (click)="callPhone(phone.value)">Call</button>

<!-- fax refers to the input element; pass its `value` to an event handler
-->
<input ref=fax placeholder="fax number">
<button (click)="callFax(fax.value)">Fax</button>
```

"phone" 的 (#) 前缀意味着我们将要定义一个 `phone` 变量。

有些人不喜欢使用 `#` 字符，而是使用它的规范形式：`ref-` 前缀。例如，我们既能用 `#phone`，也能用 `ref-phone` 来定义我们的 `phone` 变量。

如何获取变量的值

Angular 把这种变量的值设置为它所在的那个元素。我们在这个 `input` 元素上定义了这些变量。我们把那些 `input` 元素对象传给 `button` 元素，在这里，它们被当做参数传给了事件绑定中的 `call` 方法。

NgForm 和模板引用变量

让我们看看最后一个例子：一个表单，使用模板引用变量的典范。

正如我们在 [表单](#) 一章中所见过的，此表单的 HTML 可以做得相当复杂。下面是一个 **简化过的** 范例——虽然仍算不上多简单。

```
<form (ngSubmit)="onSubmit(theForm)" #theForm="ngForm">
  <div class="form-group">
    <label for="name">Name</label>
    <input class="form-control" name="name" required
[(ngModel)]="currentHero.firstName">
  </div>
  <button type="submit" [disabled]="!theForm.form.valid">Submit</button>
</form>
```

模板引用变量 `theForm` 在这个例子中出现了三次，中间隔着一大段 HTML。

```
<form (ngSubmit)="onSubmit(theForm)" #theForm="ngForm">
  <button type="submit" [disabled]="!theForm.form.valid">Submit</button>
</form>
```

theForm 变量的值是什么？

如果 Angular 没有接管它，那它可能是个 `HTMLFormElement`。实际上它是个 `ngForm`，一个对 Angular 内置指令 `NgForm` 的引用。它包装了原生的 `HTMLFormElement` 并赋予它更多“超能力”，比如跟踪用户输入的有效性。

这解释了我们该如何通过检查 `theForm.form.valid` 来禁用提交按钮，以及如何把一个信息量略大的对象传给父组件的 `onSubmit` 方法（译注：`onSubmit` 方法可能会出发一个事件，被父组件监听，参见下面的 [输入和输出属性](#) 和 [父组件监听子组件的事件](#)。）

输入与输出属性

迄今为止，我们主要聚焦在绑定声明的右侧，学习如何在模板表达式和模板语句中绑定到组件成员上。当一个成员出现在这个位置上，则称之为数据绑定的 **源**。

这一节则专注于绑定到的 **目标**，它位于 **绑定声明中的左侧**。这些指令的属性必须被声明成 **输入** 或 **输出**。

记住：所有 **组件** 皆为 **指令**。

我们要重点突出下绑定 **目标** 和绑定 **源** 的区别。

绑定的 **目标** 是在 `=` 左侧的部分，**源** 则是在 `=` 右侧的部分。

绑定的 **目标** 是绑定符：`[]`、`()` 或 `[]()` 中的属性或事件名，**源** 则是引号 (`""`) 中的部分或插值符号 (`{}{}`) 中的部分。

源 指令中的每个成员都会自动在绑定中可用。我们不需要特别做什么，就能在模板表达式或语句中访问指令的成员。

访问 **目标** 指令中的成员则 **受到限制**。我们只能绑定到那些显式标记为 **输入** 或 **输出** 的属性。

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

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

它们既不是组件的 **输入** 也不是 **输出**。它们是绑定的数据源。

现在，看看 `HeroDetailComponent`，它是 **绑定的目标**。

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

`HeroDetailComponent.hero` 和 `HeroDetailComponent.deleteRequest` 都在绑定声明的 **左侧**。

`HeroDetailComponent.hero` 在方括号中，它是一个属性绑定的目标。

`HeroDetailComponent.deleteRequest` 在圆括号中，它是一个事件绑定的目标。

声明输入和输出属性

目标属性必须被显式的标记为输入或输出。

当我们深入 `HeroDetailComponent` 内部时，就会看到这些属性被装饰器标记成了输入和输出属性。

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

另外，我们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：

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

我们既可以通过装饰器，又可以通过元数据数组来指定输入 / 输出属性。但别同时用！

输入或输出？

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

输入 和 **输出** 这两个词是从目标指令的视角来说的。



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

`HeroDetailComponent.hero` 是一个相对于 `HeroDetailComponent` 视角的 **输入** 属性，因为数据流从模板绑定表达式流向那个属性。

`HeroDetailComponent.deleteRequest` 是一个相对于 `HeroDetailComponent` 视角的 **输出** 属性，因为事件流来自这个属性，并且被模板绑定语句所处理。

输入 / 输出属性别名

有时我们需要让输入 / 输出属性的公开名字不同于内部名字。

这是使用 **属性 (Attribute) 型指令** 时的常见情况。

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

无论如何，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，我们可以使用一个约定俗成的公开名字，同时在内部使用一个不同的名字。在紧上面这个例子中，我们实际上是把 `myClick` 这个别名指向了指令自己的 `clicks` 属性。

通过把别名传进 `@Input/@Output` 装饰器，我们可以为属性指定别名，就像这样：

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

我们也能在 `inputs` 和 `outputs` 数组中为属性指定别名。我们可以写一个冒号 (`:`) 分隔的字符串，**左侧** 是指令中的属性名，**右侧** 则是公开的别名。

```
@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 语法的一个子集，并补充了几个用于特定场景的特殊操作符。这里我们讲其中的两个：**管道** 和 **安全导航操作符**。

管道操作符 (|)

在用到绑定中之前，表达式的结果可能需要一些转换。比如，我们可能希望把一个数字显示成金额、强制文本变成大写，或者过滤一个列表以及排序它。

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。

```
<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。

我们还可以通过多个管道串联出表达式：

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

我们还能对它们使用参数：

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

json 管道是特别设计来帮助我们调试绑定的：

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

它生成的输出是类似于这样的：

```
{
  "firstName": "Hercules",
  "lastName": "Son of Zeus",
  "birthdate": "1970-02-25T08:00:00.000Z",
  "url": "http://www.imdb.com/title/tt0065832/",
  "rate": 325,
  "id": 1
}
```

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。这意味着，当 `currentHero` 为空时，保护视图渲染器，让它免于失败。

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

我们来详细阐述一下这个问题和解决方案：

如果下列数据绑定中 `title` 属性为空，会发生什么？

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

这个视图仍然被渲染出来，但是显示的值是空；我们只能看到“`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。

假设模板表达式需要一个属性路径，在下一个例子中，我们要显示一个空 (`null`) 英雄的 `firstName`。

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

JavaScript 抛出了一个空引用错误，Angular 也是如此：

```
TypeError: Cannot read property 'firstName' 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，我们就可以声称这是一个合理的行为。如果它必须不能为空，但它仍然是空值，我们就制造了一个编程错误，以便它被捕获和修复。这种情况下，抛出一个异常正是我们应该做的。

另一方面，属性路径中的空值可能会时常发生，特别是当我们知道这些数据最终总会到来的时候。

当我们等待数据的时候，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` 属性所做的那样。

不幸的是，当 `currentHero` 为空的时候，我们的应用崩溃了。

我们可以通过写 `Nglf` 代码来解决这个问题。

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

或者我们可以尝试通过 `&&` 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。

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

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一个更加流畅、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，但是应用正常工作，而没有发生错误。

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

在像 `a?.b?.c?.d` 这样的长属性路径中，它工作得很完美。

小结

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

下一步

[Angular 小抄](#)

ANGULAR小抄

一份 Angular 语法的快速指南

Angular for TypeScript 小抄 (v2.0.0)

引导

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

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

使用JIT编译器引导一个
 AppModule模块定义的应用

NgModules

```
@NgModule({ declarations: ..., imports: ...,  
  exports: ..., providers: ..., bootstrap: ...})  
class MyModule {}
```

```
declarations: [MyRedComponent, MyBlueComponent, MyDatePipe]
```

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

```
exports: [MyRedComponent, MyDatePipe]
```

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

定义一个模块，其中包括组件、指令、管道和提供商。

一个数组，包括从属于当前模块的组件、指令和管道。

一个数组，包括被导入到当前模块中的所有模块。来自被导入模块的 declarations 也同样对当前模块有效。

一个数组，包括对导入当前模块的模块可见的组件、指令、管道。

```
providers: [MyService, { provide: ... }]
```

一个数组，包括在对前模块及导入当前模块的模块中的内容物（组件、指令、管道、提供商等）可见的依赖注入提供商。

```
bootstrap: [AppComponent]
```

一个数组，包括由当前模块引导时应该引导的组件

模板语法

```
<input [value]="firstName">
```

把属性 `value` 绑定到表达式 `firstName` 的结果。

```
<div [attr.role]="myAriaRole">
```

把 `role` 这个Attribute绑定到表达式 `myAriaRole` 的结果。

```
<div [class.extra-sparkle]="isDelightful">
```

把元素是否出现CSS类 `extra-sparkle`，绑定到一个表达式 `isDelightful` 的结果是否为真。

```
<div [style.width.px]="mysize">
```

把样式的 `width` 属性绑定到表达式 `mysize` 的结果，以px为单位。这个单位是可选的。

```
<button (click)="readRainbow($event)">
```

当按钮(及其子元素)上的`click`事件被触发时，调用 `readRainbow` 方法，并把事件对象作为参数传入。

```
<div title="Hello {{ponyName}}">
```

把属性绑定到一个插值表达式字符串，比如 "Hello Seabiscuit"。它等价于：
`<div [title]="'Hello ' + ponyName">`

```
<p>Hello {{ponyName}}</p>
```

把文本内容绑定到一个插值表达式，比如 "Hello Seabiscuit"。

```
<my-cmp [(title)]="name">
```

设置双向数据绑定。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`，它提供到 `video` 元素实例的访问，可用于当前模板中的数据绑定和事件绑定表达式中。

```
<p *myUnless="myExpression">...</p>
```

* 符号表示当前元素将被转变成一个内嵌模板。等价于：
`<template
[myUnless]="myExpression">
<p>...</p></template>`

```
<p>Card No.: {{cardNumber | myCardNumberFormatter}}</p>
```

通过名叫 `myCardNumberFormatter` 的管道，转换表达式的当前值 `cardNumber`。

```
<p>Employer: {{employer?.companyName}}</p>
```

安全导航运算符(`?.`)表示 `employer` 字段是可选的，如果它是 `undefined`，表达式剩下的部分将被忽略

```
<svg:rect x="0" y="0" width="100" height="100"/>
```

SVG模板需要在它们的根节点上带一个 `svg:` 前缀，以消除模板中HTML元素和SVG元素的歧义。

```
<svg>  
  <rect x="0" y="0" width="100" height="100"/>  
</svg>
```

`<svg>` 元素在无需前缀的情况下，也能被自动检测为SVG。

内置指令

```
import { CommonModule }  
from '@angular/common';
```

```
<section *ngIf="showSection">
```

基于`showSection`表达式的值移除或重新创建部分DOM树。

```
<li *ngFor="let item of list">
```

把`li`元素及其内容转化成一个模板，并用它来为列表中的每个条目初始化视图。

```
<div [ngSwitch]="conditionExpression">  
  <template [ngSwitchCase]="case1Exp">...</template>  
  <template ngSwitchCase="case2LiteralString">...</template>  
  <template ngSwitchDefault>...</template>  
</div>
```

基于`conditionExpression`的当前值，从内嵌模板中选取一个，有条件的切换`div`的内容。

```
<div [ngClass]="{active: isActive, disabled: isDisabled}">
```

把一个元素上CSS类的出现与否，绑定到一个真值映射表上。右侧的表达式应该返回类似`{class-name: true/false}`的映射表。

表单	<pre>import { FormsModule } from '@angular/forms';</pre>
<input [(ngModel)]="userName">	提供双向绑定，为表单控件提供解析和验证。

类装饰器	<pre>import { Directive, ... } from '@angular/core';</pre>
<code>@Component({...}) class MyComponent() {}</code>	声明当前类是一个组件，并提供关于该组件的元数据。
<code>@Directive({...}) class MyDirective() {}</code>	声明当前类是一个指令，并提供关于该指令的元数据。
<code>@Pipe({...}) class MyPipe() {}</code>	声明当前类是一个管道，并且提供关于该管道的元数据。
<code>@Injectable() class MyService() {}</code>	声明当前类有一些依赖，当依赖注入器创建该类的实例时，这些依赖应该被注入到构造函数中。

指令配置	<pre>@Directive({ property1: value1, ... })</pre>
<code>selector: '.cool-button:not(a)'</code>	指定一个CSS选择器，以便在模板中找出该指令。支持的选择器包括 <code>element</code> , <code>[attribute]</code> , <code>.class</code> , 和 <code>:not()</code> 。不支持父子关系选择器。
<code>providers: [MyService, { provide: ... }]</code>	为当前指令及其子指令提供依赖注入的providers数组。

组件配置	<pre>@Component 扩展了 @Directive，以便 @Directive 中的配置项也能用在组件上</pre>
<code>moduleId: module.id</code>	如果设置了， <code>templateUrl</code> 和 <code>styleUrl</code> 会被解析成相对于组件的。
<code>viewProviders: [MyService, { provide: ... }]</code>	依赖注入provider的数组，局限于当前组件的视图中。
<code>template: 'Hello {{name}}' templateUrl: 'my-component.html'</code>	当前组件视图的内联模板或外部模板地址
<code>styles: ['.primary {color: red}'] styleUrls: ['my-component.css']</code>	内联CSS样式或外部样式表URL的列表，用于给组件的视图添加样式。

供指令类或组件类用的字段装饰器。

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

```
@Input() myProperty;
```

声明一个输入属性，以便我们可以通过属性绑定更新它。(比如：
`<my-cmp [my-property]="someExpression">`
`).`

```
@Output() myEvent = new EventEmitter();
```

声明一个输出属性，以便我们可以通过事件绑定进行订阅。(比如：
`<my-cmp (my-event)="doSomething()">`
`).`

```
@HostBinding('[class.valid]') isValid;
```

把宿主元素的属性(比如CSS类：`valid`)绑定到指令/组件的属性(比如：`isValid`)。

```
@HostListener('click', ['$event']) onClick(e) {...}
```

通过指令/组件的方法(例如`onClick`)订阅宿主元素的事件(例如`click`)，可选传入一个参数(`$event`)。

```
@ContentChild(myPredicate) myChildComponent;
```

把组件内容查询(`myPredicate`)的第一个结果绑定到类的`myChildComponent`属性。

```
@ContentChildren(myPredicate) myChildComponents;
```

把组件内容查询(`myPredicate`)的全部结果，绑定到类的`myChildComponents`属性。

```
@ViewChild(myPredicate) myChildComponent;
```

把组件视图查询(`myPredicate`)的第一个结果绑定到类的`myChildComponent`属性。对指令无效。

```
@ViewChildren(myPredicate) myChildComponents;
```

把组件视图查询(`myPredicate`)的全部结果绑定到类的`myChildComponents`属性。对指令无效。

指令和组件的变更检测与生命周期钩子

(作为类方法实现)

```
constructor(myService: MyService, ...) { ... }
```

类的构造函数会在所有其它生命周期钩子之前调用。使用它来注入依赖，但是要避免用它做较重的工作。

<code>ngOnChanges(changeRecord) { ... }</code>	在输入属性每次变化了之后、开始处理内容或子视图之前被调用。
<code>ngOnInit() { ... }</code>	在执行构造函数、初始化输入属性、第一次调用完 <code>ngOnChanges</code> 之后调用。
<code>ngDoCheck() { ... }</code>	每当检查组件或指令的输入属性是否变化时调用。通过它，可以用自定义的检查方式来扩展变更检测逻辑。
<code>ngAfterContentInit() { ... }</code>	当组件或指令的内容已经初始化、 <code>ngOnInit</code> 完成之后调用。
<code>ngAfterContentChecked() { ... }</code>	在每次检查完组件或指令的内容之后调用。
<code>ngAfterViewInit() { ... }</code>	当组件的视图已经初始化完毕，每次 <code>ngAfterContentInit</code> 之后被调用。只适用于组件。
<code>ngAfterViewChecked() { ... }</code>	每次检查完组件的视图之后调用。只适用于组件。
<code>ngOnDestroy() { ... }</code>	在所属实例被销毁前，只调用一次。

依赖注入配置

```
{ provide: MyService, useClass: MyMockService }
```

把`MyService`类的提供商设置或改写为`MyMockService`。

```
{ provide: MyService, useFactory: myFactory }
```

把`MyService`的提供商设置或改写为`myFactory`工厂函数。

```
{ provide: MyValue, useValue: 41 }
```

把`MyValue`的提供商设置或改写为值`41`。

路由与导航

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

为应用配置路由。支持静态、参数化、重定向和通配符路由。还支持自定义路由数据和解析。

```
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);
```

```
<router-outlet></router-outlet>
<router-outlet name="aux"></router-outlet>
```

```
<a routerLink="/path">
<a [routerLink]="[ '/path', routeParam ]">
<a [routerLink]="[ '/path', { matrixParam: 'value' } ]">
<a [routerLink]="[ '/path' ]" [queryParams]="{ page: 1 }">
<a [routerLink]="[ '/path' ]" fragment="anchor">
```

标记一个位置，用于加载当前激活路由的组件。

```
<a [routerLink]="[ '/path' ]" routerLinkActive="active">
```

当 `routerLink` 被激活时，就把指定的CSS类添加到该元素上。

```
class CanActivateGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | Promise<boolean> | boolean { ... }
}

{ path: ..., canActivate: [CanActivateGuard] }
```

一个用来定义类的接口，路由器会首先调用它来决定是否应该激活该组件。应该返回布尔值或能解析为布尔值的可观察对象(Observable)或承诺(Promise)。

```
class CanDeactivateGuard implements CanDeactivate<T> {
  canDeactivate(
    component: T,
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | Promise<boolean> | boolean { ... }
}

{ path: ..., canDeactivate: [CanDeactivateGuard] }
```

一个用来定义类的接口，路由器在导航后会首先调用它来决定是否应该取消该组件的激活状态。应该返回布尔值或能解析为布尔值的可观察对象(Observable)或承诺(Promise)。

```
class CanActivateChildGuard implements CanActivateChild {
  canActivateChild(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean>|Promise<boolean>|boolean { ... }
}
```

```
{ path: ..., canActivateChild: [CanActivateGuard],
  children: ... }
```

一个用来定义类的接口，路由器会首先调用它来决定是否应该激活该子路由。应该返回布尔值或能解析为布尔值的可观察对象(Observable)或承诺(Promise)。

```
class ResolveGuard implements Resolve<T> {
  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<any>|Promise<any>|any { ... }
}
```

```
{ path: ..., resolve: [ResolveGuard] }
```

一个用来定义类的接口，路由器会在渲染该路由之前先调用它来解析路由数据。应该返回一个值或能解析为值的可观察对象(Observable)或承诺(Promise)。

```
class CanLoadGuard implements CanLoad {
  canLoad(
    route: Route
  ): Observable<boolean>|Promise<boolean>|boolean { ... }
}
```

```
{ path: ..., canLoad: [CanLoadGuard], loadChildren: ... }
```

一个用来定义类的接口，路由器会首先调用它来决定一个惰性加载的模块是否应该被加载。应该返回布尔值或能解析为布尔值的可观察对象(Observable)或承诺(Promise)。

风格指南

如何写 Angular 风格的程序

欢迎光临 Angular 风格指南

目的

如果你在寻找一份关于语法、约定和 Angular 应用程序的组织结构的风格指南，那你就来对了。本风格指南提供了我们所提倡的约定，但最重要的是，解释了为什么。

风格词汇

每个指导原则都会描述好的或者坏的做法，所有指导原则都相互呼应，保持一致。

指导原则中使用的词汇表明我们推荐的强度。

坚持 意味着总是应该遵循的约定。**总是** 可能有点太强。应该**总是**遵循的指导原则非常少见。但是，只有遇到非常不寻常的情况才能打破**坚持**的原则。

考虑 标志着通常应该遵循的指导原则。

如果你能完全理解指导原则背后的含义，并且很好的理由打破它，那就可以打破该指导原则。但是请保持一致。

避免 标志着我们决不应该做的事。需要**避免**的代码范例会有不会被忽视的红色标题。

文件结构约定

在一些代码例子中，有的文件拥有一个或多个相似名字的伴随文件。（比如，hero.component.ts 和 hero.component.html）。

本指南将会使用像 hero.component.ts|html|css|spec 的简写来表示上面描述的多个文件，目的是保持本指南的简洁性，增加文件结构描述时的可读性。

目录

1. 单一职责
2. 命名约定
3. 代码约定
4. 应用程序结构与模块划分
5. 组件
6. 指令
7. 服务
8. 数据服务
9. 生命周期钩子
10. 附录

单一职责

我们遵循 [单一职责原则](#) 来创建的所有组件、服务和其它标志等。这样能帮助我们把应用程序弄的干净整洁，易于阅读、维护和测试。

单一法则

风格 01-01

坚持 每个文件只定义一样东西（比如服务或者组件）。

考虑 把文件大小限制在 400 行代码以内。

为何？ 单组件文件非常容易阅读、维护，并能防止在版本控制系统里与团队冲突。

为何？ 单组件文件可以防止一些隐蔽的程序缺陷，当把多个组件合写在同一个文件中时，可能造成共享变量、创建意外的闭包，或者与依赖之间产生意外耦合等情况。

为何？ 单独的组件通常是该文件默认的输出，这样就可以利用路由器实现按需加载。

最关键的是，可以增强代码可重用性和阅读性，减少出错的可能性。

下面的 **负面** 例子定义了 `AppComponent`，该文件引导了应用程序，定义了 `Hero` 模型对象，并且从服务器加载了英雄 ... 所有都在发生在同一个文件。**不要这么做**。

app/heroes/hero.component.ts

```
1.  /* avoid */
2.
3.  import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';
4.  import { BrowserModule } from '@angular/platform-browser';
5.  import { NgModule, Component, OnInit } from '@angular/core';
6.
7.  class Hero {
8.    id: number;
9.    name: string;
10. }
11.
12. @Component({
13.   selector: 'my-app',
14.   template: `
15.     <h1>{{title}}</h1>
16.     <pre>{{heroes | json}}</pre>
17.   `,
18.   styles: [
19.     `
20.       body {
21.         font-family: sans-serif;
22.       }
23.     `]
```

```

18.     styleUrls: ['app/app.component.css']
19.   })
20. class AppComponent implements OnInit {
21.   title = 'Tour of Heroes';
22.
23.   heroes: Hero[] = [];
24.
25.   ngOnInit() {
26.     getHeroes().then(heroes => this.heroes = heroes);
27.   }
28. }
29.
30. @NgModule({
31.   imports: [ BrowserModule ],
32.   declarations: [ AppComponent ],
33.   exports: [ AppComponent ],
34.   bootstrap: [ AppComponent ]
35. })
36. export class AppModule { }
37.
38. platformBrowserDynamic().bootstrapModule(AppModule);
39.
40. const HEROES: Hero[] = [
41.   {id: 1, name: 'Bombasto'},
42.   {id: 2, name: 'Tornado'},
43.   {id: 3, name: 'Magneta'},
44. ];
45.
46. function getHeroes(): Promise<Hero[]> {
47.   return Promise.resolve(HEROES); // TODO: get hero data from the
48.   server;
}

```

将组件及其支撑部件重新分配到独立的文件中会更好。

```

1. import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';
2.
3. import { AppModule }      from './app/app.module';

```

4.

```
5. platformBrowserDynamic().bootstrapModule(AppModule);
```

随着应用程序的成长，本法则会变得越来越重要。

[回到顶部](#)

小函数

风格 01-02

坚持 定义小函数

考虑 限制在 75 行之内

为何？ 小函数更易于测试，特别是当它们只做一件事，只为一个目的服务的时候。

为何？ 小函数促进了代码的重用。

为何？ 小函数更加易于阅读。

为何？ 小函数更加易于维护。

为何？ 小函数帮助避免一些大函数容易产生的那些与外界共享变量、创建意外的闭包或与依赖之间产生意外耦合等隐蔽的错误。

[回到顶部](#)

命名约定

命名约定对维护性和可读性非常重要。本指南为文件和标志命名推荐了一套命名约定。

总体命名知道原则

风格 02-01

坚持 为所有符号使用一致的命名规则。

坚持 遵循同一个模式来描述符号的特性和类型。推荐的模式为 `feature.type.ts`。

为何？ 命名约定提供了一致的方法来帮助我们一眼锁定内容。在整个项目内保持一致性是至关重要的。在团队内保持一致性也很重要。在公司内部保持一致性可以大幅提高效率。

为何？ 命名约定最直接的目的是：帮我们快速找到代码并让它们更容易理解。

为何？ 目录和文件的名字应该清楚的说明它们的用途。比如 `app/heroes/hero-list.component.ts` 包含了一个用来维护英雄列表的组件。

[回到顶部](#)

使用点和横杠来分隔文件名

风格 02-02

坚持 在描述性名字里面，使用横杠来分隔单词。

坚持 使用点来分隔描述性名字和类型名。

坚持 对所有组件使用一致的类型命名规则，遵循这个模式：先描述组件的特性，再描述它的类型。推荐的模式为 `feature.type.ts`。

坚持 使用惯用的后缀来描述类型，比如 `*.service`、`*.component`、`*.pipe`、`.module`、`.directive`。创建更多类型名，但你必须注意不要创建太多。

为何？ 类型名字提供一致的方法来快速的识别文件是什么。

为何？ 可以让我们利用编辑器或者 IDE 的模糊搜索功能，很容易的找到特定文件。

为何？ 没有被简写的类型名字比如 `.service` 很有描述性，不含糊。简写可能造成混淆，比如 `.srv`、`.svc`，和 `.serv`。

为何？ 与自动化任务的模式匹配。

[回到顶部](#)

符号名与文件名

风格 02-03

坚持 为所有东西使用一致的命名约定：以它们所代表的东西命名。

坚持 使用大写驼峰命名法来命名所有类名。保持符号的名字与它所在的文件名字相同。

坚持 在符号名后面追加约定的类型后缀（比如：`Component`、`Directive`、`Module`、`Pipe`、`Service`）。

坚持 在文件名后面追加约定的类型后缀（比如`.component.ts`、`.directive.ts`、`.module.ts`、`.pipe.ts`、`.service.ts`）。

为何？ 提供一种一致的方式，以快速标识和引用资产。

为何？ 大驼峰命名法是用来识别那些能通过构造函数进行实例化的对象的命名约定。

符号名	文件名
<pre>@Component({ ... }) export class AppComponent { }</pre>	<code>app.component.ts</code>
<pre>@Component({ ... }) export class HeroesComponent { }</pre>	<code>heroes.component.ts</code>
	<code>hero-list.component.ts</code>

```
@Component({ ... })  
export class  
HeroListComponent { }
```

```
@Component({ ... })  
export class  
HeroDetailComponent {  
}
```

```
@Directive({ ... })  
export class  
ValidationDirective {  
}
```

```
@NgModule({ ... })  
export class AppModule
```

```
@Pipe({ name:  
'initCaps' })  
export class  
InitCapsPipe  
implements  
PipeTransform { }
```

```
@Injectable()  
export class  
UserProfileService { }
```

hero-detail.component.ts

validation.directive.ts

app.module.ts

init-caps.pipe.ts

user-profile.service.ts

[回到顶部](#)

服务名

风格 02-04

坚持 使用前后一致的命名规则来命名服务，以它们的特性来命名。

坚持 使用大写驼峰命名法来命名服务。

坚持 当不能从它们的名字里清楚的看出它们是什么的时候（比如它们的名字是名词时），添加 `Service` 后缀。

为何？ 提供前后一致的方法来快速识别和引用服务。

为何？ 清楚的服务名，比如 `Logger` 不需要后缀。

为何？ 如果服务名字是名词时，比如 `Credit`，需要一个后缀。当名字不能很明显的标示出它是服务还是其它东西的时候，应该添加后缀。

符号名

```
@Injectable()  
export class  
HeroDataService { }
```

hero-data.service.ts

```
@Injectable()  
export class  
CreditService { }
```

credit.service.ts

```
@Injectable()  
export class Logger { }
```

logger.service.ts

[回到顶部](#)

引导

风格 02-05

坚持 把应用的引导程序和平台相关的逻辑放到名为 `main.ts` 的文件里。

坚持 在“引导”逻辑中包含错误处理代码。

避免 把应用逻辑放在 `main.ts` 中，而应该考虑把它们放进组件或服务里。

为何？ 遵循前后一致的约定来处理应用的启动逻辑。

为何？ 这是从其它技术平台借鉴的一个常用约定。

main.ts

```
1. import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2.
3. import { AppModule } from './app/app.module';
4.
5. platformBrowserDynamic().bootstrapModule(AppModule)
6.   .then(success => console.log(`Bootstrap success`))
7.   .catch(err => console.error(err));
```

[回到顶部](#)

指令的选择器

风格 02-06

坚持 使用小驼峰命名法来命名指令的选择器。

为何？ 保持指令里定义的属性名字与它们绑定的视图 HTML 属性名字一致。

为何？ Angular HTML 解析器是大小写敏感的，它识别小写驼峰写法。

[回到顶部](#)

为组件自定义前缀

风格 02-07

坚持 使用带连字符的小写元素选择器的值（比如 `admin-users`）。

坚持 为组件选择器使用自定义前缀。比如 `toh` 前缀表示 Tour of Heroes（英雄指南），而前缀`admin 表示管理特性区。

坚持 使用前缀来识别特性区域或者应用程序本身。

为何？ 防止与来自其它应用中的组件和原生 HTML 元素发生命名冲突。

为何？ 把我们的组件推广和共享到其它应用中会更容易。

为何？ 组件在 DOM 中更容易被区分出来。

app/heroes/hero.component.ts

```
1.  /* avoid */
2.
3.  // HeroComponent is in the Tour of Heroes feature
4.  @Component({
5.    selector: 'hero'
6.  })
7.  export class HeroComponent {}
```

app/users/users.component.ts

```
1.  /* avoid */
2.
3.  // UsersComponent is in an Admin feature
4.  @Component({
5.    selector: 'users'
6.  })
```

```
7. export class UsersComponent {}
```

app/heroes/hero.component.ts

```
1. @Component({
2.   selector: 'toh-hero'
3. })
4. export class HeroComponent {}
```

app/users/users.component.ts

```
1. @Component({
2.   selector: 'admin-users'
3. })
4. export class UsersComponent {}
```

为指令添加自定义前缀

风格 02-08

坚持 为指令的选择器使用自定义前缀（比如前缀 `toh` 来自 `Tour of Heroes`）。

坚持 用小驼峰形式拼写非元素选择器，除非该选择器就是要用来匹配某个原生 HTML 属性的。

为何？ 防止名字冲突。

为何？ 指令更加容易被识别。

app/shared/validate.directive.ts

```
1. /* avoid */
```

```
2.  
3. @Directive({  
4.   selector: '[validate]'  
5. })  
6. export class ValidateDirective {}
```

app/shared/validate.directive.ts

```
1. @Directive({  
2.   selector: '[tohvalidate]'  
3. })  
4. export class ValidateDirective {}
```

[回到顶部](#)

管道名

风格 02-09

坚持 为所有管道使用前后一致的命名约定，用它们的特性来命名。

为何？ 提供一致的方法快速识别和引用管道。

符号名

文件名

```
@Pipe({ name:  
'ellipsis' })  
export class  
EllipsisPipe  
implements  
PipeTransform { }
```

ellipsis.pipe.ts

```
@Pipe({ name:  
  'initCaps' })  
export class  
InitCapsPipe  
implements  
PipeTransform { }
```

init-caps.pipe.ts

[回到顶部](#)

单元测试文件名

风格02-10

坚持 测试规范文件的名字应该和被测试的组件名字一样。

坚持 测试配置文件命名应该跟随后缀 `.spec` 。

为何？ 提供一致的方法来快速识别测试。

为何？ 提供一个与 `karma` 或者其它测试运行器相配的命名模式。

符号名

文件名

组件

heroes.component.spec.ts

hero-list.component.spec.ts

hero-detail.component.spec.ts

服务

logger.service.spec.ts

hero.service.spec.ts

filter-text.service.spec.ts

管道

ellipsis.pipe.spec.ts

init-caps.pipe.spec.ts

[回到顶部](#)

端到端测试文件名

风格 02-11

坚持 端到端测试配置文件应该和它们所测试的特性同名，并加上后缀 `.e2e-spec`。

为何？ 提供一致的方法快速识别端到端测试文件。

为何？ 提供一个与测试运行器和构建自动化相配的模式。

符号名

文件名

端到端测试

app.e2e-spec.ts

heroes.e2e-spec.ts

[回到顶部](#)

Angular NgModule 命名

STYLE 02-12

坚持 为符号名追加 `Module` 后缀

坚持 为文件名添加 `.module.ts` 扩展名。

坚持 用特性名和所在目录命名模块。

为何？ 提供一个一致的方式来快速的标出和引用模块。

为何？ 大驼峰命名法是一种命名约定，用来标出可用构造函数实例化的对象。

为何？ 很容易就能看出这个模块是同名特性的跟模块。

坚持 为 **RoutingModule** 类名添加 `RoutingModule` 后缀。

坚持 **RoutingModule** 的文件名用 `-routing.module.ts` 结尾。

为何？ `RouterModule` 是一种专门用来配置 Angular 路由器的模块。“让类名和文件名保持一致”这项约定可以让这些模块易于跟踪和验证。

Symbol Name	File Name
<pre>@NgModule({ ... }) export class AppModule { }</pre>	app.module.ts
<pre>@NgModule({ ... }) export class HeroesModule { }</pre>	heroes.module.ts
<pre>@NgModule({ ... }) export class VillainsModule { }</pre>	villains.module.ts
<pre>@NgModule({ ... }) export class AppRoutingModule { }</pre>	app-routing.module.ts
	heroes-routing.module.ts

```
@NgModule({ ... })  
export class  
HeroesRoutingModule {  
}
```

[回到顶部](#)

编程约定

坚持一套前后一致的编程、命名和空格的约定。

类

风格03-01

坚持 使用大写驼峰命名法来命名类。

为何？ 遵循类命名传统约定。

为何？ 类可以被实例化和构造出实例。根据约定，用大写驼峰命名法来标示可被构造出来的东西。

app/shared/exception.service.ts

```
1.  /* avoid */  
2.  
3.  export class exceptionService {  
4.    constructor() { }  
5.  }
```

app/shared/exception.service.ts

```

1.  export class ExceptionService {
2.    constructor() { }
3.  }

```

[回到顶部](#)

常量

风格 03-02

坚持 用 `const` 声明变量，除非它们的值在应用的生命周期内会发生变化。

为何？ 告诉读者这个值是不可变的。

为何？ TypeScript 会要求在声明时立即初始化，并阻止对其再次赋值，以确保达成我们的意图。

考虑 把常量名拼写为小驼峰格式。

为何？ 小驼峰变量名（`heroRoutes`）比传统的“大写蛇形命名法”（`HERO_ROUTES`）更容易阅读和理解。

为何？ 把常量命名为大写蛇形命名法的传统源于现代 IDE 出现之前，以便阅读时可以快速发现那些 `const` 定义。而 TypeScript 本身就能够防止意外赋值。

坚持 容许 **现存的** `const` 变量沿用大写蛇形命名法。

为何？ 传统的大写蛇形命名法（`UPPER_SNAKE_CASE`）仍然很流行、很普遍，特别是在第三方模块中。修改它们带不来多大价值，同时还会有破坏现有代码和文档的风险。

app/shared/data.service.ts

```

1.  export const mockHeroes   = ['Sam', 'Jill']; // prefer
2.  export const heroesurl   = 'api/heroes';      // prefer

```

```
3. export const VILLAINS_URL = 'api/villains'; // tolerate
```

[回到顶部](#)

接口

风格 03-03

坚持 使用大写驼峰命名法来命名接口。

考虑 不要在接口名字前面加 `I` 前缀。

考虑 用类代替接口。

为何？ TypeScript 指导原则 不建议使用“`I`”前缀。

为何？ 单独一个类的代码量小于 **类 + 接口**。

为何？ 类实际上就是接口（只是要用 `implements` 代替 `extends` 而已）。

为何？ 在 Angular 依赖注入系统中，接口类可以作为服务提供商的查阅令牌。

app/shared/hero-collector.service.ts

```
1. /* avoid */
2.
3. import { Injectable } from '@angular/core';
4.
5. import { IHero } from './hero.model.avoid';
6.
7. @Injectable()
8. export class HeroCollectorService {
9.   hero: IHero;
10.
11.   constructor() { }
12. }
```

app/shared/hero-collector.service.ts

```
1. import { Injectable } from '@angular/core';
2.
3. import { Hero } from './hero.model';
4.
5. @Injectable()
6. export class HeroCollectorService {
7.   hero: Hero;
8.
9.   constructor() { }
10. }
```

[回到顶部](#)

属性和方法

样式 03-04

坚持 使用小写驼峰命名法来命名属性和方法。

避免 使用下划线为前缀来命名私有属性和方法。

为何？ 遵循传统属性和方法的命名约定。

为何？ JavaScript 不支持真正的私有属性和方法。

为何？ TypeScript 工具让识别私有或公有属性和方法变得很简单。

app/shared/toast.service.ts

```
1. /* avoid */
2.
3. import { Injectable } from '@angular/core';
4.
5. @Injectable()
6. export class ToastService {
```

```
7.     message: string;
8.
9.     private _toastCount: number;
10.
11.    hide() {
12.        this._toastCount--;
13.        this._log();
14.    }
15.
16.    show() {
17.        this._toastCount++;
18.        this._log();
19.    }
20.
21.    private _log() {
22.        console.log(this.message);
23.    }
24. }
```

app/shared/toast.service.ts

```
1. import { Injectable } from '@angular/core';
2.
3. @Injectable()
4. export class ToastService {
5.     message: string;
6.
7.     private toastCount: number;
8.
9.     hide() {
10.         this.toastCount--;
11.         this.log();
12.     }
13.
14.     show() {
15.         this.toastCount++;
16.         this.log();
17.     }
18.
19.     private log() {
```

```
20.     console.log(this.message);
21. }
22. }
```

[回到顶部](#)

导入语句中的空行

风格 03-06

坚持 在第三方导入和自己代码的导入之间留一个空行。

考虑 按模块名字的字母顺排列导入行。

考虑 在解构表达式中按字母顺序排列导入的东西。

为何？ 空行可以让阅读和定位本地导入变得更加容易。

为何？ 按字母顺序排列可以让阅读和定位本地导入更加容易。

app/heroes/shared/hero.service.ts

```
1. /* avoid */
2.
3. import { ExceptionService, SpinnerService, ToastService } from
   '.../../core';
4. import { Http, Response } from '@angular/http';
5. import { Injectable } from '@angular/core';
6. import { Hero } from './hero.model';
```

app/heroes/shared/hero.service.ts

```
1. import { Injectable } from '@angular/core';
2. import { Http, Response } from '@angular/http';
3.
```

```
4. import { Hero } from './hero.model';
5. import { ExceptionService, SpinnerService, ToastService } from
'../../core';
```

[回到顶部](#)

应用程序的结构与 Angular 模块划分

准备一个短期和一个长期的实施方案。从零开始，但要时刻考虑应用程序接下来要走的路。

把所有应用程序的源代码都放到名叫 `app` 的目录里。所有特性区都在自己的文件夹中，带有他们自己的 Angular 模块。

所有内容都遵循每个文件单个特性的原则。每个组件、服务和管道都在自己的文件里。所有第三方程序包都被保存到其它目录里而不是 `app` 目录里，我们不会修改它们，所以不希望它们弄乱我们的应用程序。使用本指南介绍的文件命名约定。

[回到顶部](#)

LIFT

风格 04-01

坚持 组织应用的结构，达到这些目的：快速定位 (`Locate`) 代码、一眼识别 (`Identify`) 代码、尽量保持扁平结构 (`Flatten`) 和尝试 `Try` 遵循不重复自己 DRY - Do Not Repeat Yourself 原则。

坚持 遵循四个基本指导原则来定义文件结构，上面四个基本原则是按重要顺序排列的。

为何？ LIFT 提供了前后一致的结构，它具有扩展性强、模块化的特性。因为容易快速锁定代码，所以提高了开发者的效率。另外，检查应用结构是否合理的方法是问问自己：我们能快速打开与此特性有关的文件并开始工作吗？

[回到顶部](#)

定位

风格 04-02

坚持 直观、简单和快速的定位我们的代码。

为何？ 要想高效的工作，就必须能迅速找到文件，特别是当不知道（或不记得）文件名时。把相关的文件一起放在一个直观的位置可以节省时间。富有描述性的目录结构会让你和后面的维护者眼前一亮。

[回到顶部](#)

识别

风格 04-03

坚持 命名文件到这个程度：可以从名字立刻知道它包含了什么，代表了什么。

坚持 文件名要具有说明性，并保证文件中只包含一个组件。

避免 创建包含很多组件、服务或者混合体的文件。

为何？ 花费更少的时间来查找和琢磨代码，就会变得更有效率。较长的文件名远胜于较短却容易混淆的缩写名。

当你有一组小型、紧密相关的特性时，违反**一物一文件**的规则可能会更好，这种情况下单一文件可能会比多个文件更容易发现和理解。注意这个例外。

[回到顶部](#)

扁平

风格 04-04

坚持 尽可能保持扁平的目录结构。

考虑 当同一目录下达到 7 个或更多个文件时创建子目录。

考虑 配置 IDE，以隐藏无关的文件，比如生成出来的 `.js` 文件和 `.js.map` 文件等。

为何？ 没人想要在超过七层的目录中查找文件。扁平的结构有利于搜索。

另一方面，**心理学家们相信**，当关注的事物超过 9 个时，人类就会开始感到吃力。所以，当一个文件夹中有 10 个或更多个文件时，可能就是创建子目录的时候了。

还是根据你自己的舒适度而定吧。除非创建新文件夹能有显著的价值，否则尽量使用扁平结构。

[回到顶部](#)

T-DRY (尝试遵循不重复自己 DRY 的约定)

风格 04-05

坚持 不要重复自己 (DRY)

避免 过度 DRY，以致牺牲了阅读性。

为何？ 虽然 DRY（不要重复你自己）很重要，但如果要以牺牲 LIFT 的其它原则为代价，那就不值得了。这也就是为什么它被称为 **T-DRY**。比如，把组件命名为 `hero-view.component.html` 是多余的，因为组件显然就是一个视图（view）。但如果它不是这么显著，或不符合常规，那就把它写出来。

[回到顶部](#)

总体结构指导原则

风格 04-06

坚持 从零开始，但要时刻考虑应用程序接下来要走的路。

坚持 有一个短期和一个长期的实施方案。

坚持 把所有源代码都放到名为 `app` 的目录里。

坚持 如果组件具有多个相互合作的文件 (`.ts`、`.html`、`.css` 和 `.spec`)，那就为它创建一个文件夹。

为何？ 在早期阶段能够帮助保持应用的结构小巧而且易于维护，这样当应用增长时就会更容易进化了。

为何？ 组件通常有四个文件（例如 `*.html`、`*.css`、`*.ts` 和 `*.spec.ts`），它们很容易把一个目录弄乱。

目录和文件结构

```
<project root>
  └── app
    ├── core
    │   ├── core.module.ts
    │   ├── exception.service.ts|spec.ts
    │   └── user-profile.service.ts|spec.ts
    ├── heroes
    │   ├── hero
    │   │   └── hero.component.ts|html|css|spec.ts
    │   ├── hero-list
    │   │   └── hero-list.component.ts|html|css|spec.ts
    │   └── shared
    │       ├── hero-button.component.ts|html|css|spec.ts
    │       ├── hero.model.ts
    │       └── hero.service.ts|spec.ts
    └── heroes.module.ts
```

```
└ heroes-routing.module.ts  
  └ shared  
    └ shared.module.ts  
    └ init-caps.pipe.ts|spec.ts  
    └ text-filter.component.ts|spec.ts  
    └ text-filter.service.ts|spec.ts  
  └ villains  
    └ villain  
      └ ...  
    └ villain-list  
      └ ...  
  └ shared  
    └ ...  
  └ villains.component.ts|html|css|spec.ts  
  └ villains.module.ts  
  └ villains-routing.module.ts  
  └ app.component.ts|html|css|spec.ts  
  └ app.module.ts  
  └ app-routing.module.ts  
  └ main.ts  
  └ index.html  
  ...
```

把组件放在专用目录中的方式广受欢迎，对于小型应用，还可以保持组件扁平化（而不是放在专用目录中）。这样会把四个文件放在现有目录中，但是也会减少目录的嵌套。无论你如何选择，请保持一致。

[回到顶部](#)

按特性组织的目录结构

风格 04-07

坚持 根据特性区的名字创建目录。

为何？ 开发人员可以定位代码，扫一眼就能知道每个文件代表什么，目录尽可能保持扁平，既没有重复也没有多余的名字。

为何？ LIFT 原则中包含了所有这些。

为何？ 通过精心组织内容并让它们遵循 LIFT 原则，可以避免应用变得杂乱无章。

为何？ 当有很多文件时（比如 10 个以上），在“专用目录”型结构中定位它们会比在扁平结构中更容易。

坚持 为每个特性区创建一个 Angular 模块。

为何？ Angular 模块能让对可路由的特性进行惰性加载变得更容易。

为何？ Angular 模块能让我们更容易隔离、测试盒复用特性。

[点这里查看目录和文件结构的范例](#)

[回到顶部](#)

应用的根模块

风格 04-08

坚持 在应用的根部创建一个 Angular 模块（比如 `/app/`）。

为何？ 每个应用都至少需要一个根 Angular 模块。

考虑 把根模块命名为 `app.module.ts`。

为何？ 能让定位和识别根模块变得更容易。

app/app.module.ts

```

1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3.
4. import { AppComponent }  from './app.component';
5. import { HeroesComponent } from './heroes/heroes.component';
6.
7. @NgModule({
8.   imports: [
9.     BrowserModule,
10.   ],
11.   declarations: [
12.     AppComponent,
13.     HeroesComponent
14.   ],
15.   exports: [ AppComponent ],
16.   entryComponents: [ AppComponent ]
17. })
18. export class AppModule {}

```

[回到顶部](#)

特性模块

风格 04-09

坚持 为应用中的每个明显的特性创建一个 Angular 模块。

坚持 把特性模块放在与特性区同名的目录中（如 `app/heroes` ）。

坚持 特性模块的名字应该能反映出特性区的名字和目录（如 `app/heroes/heroes.module.ts` ）。

坚持 特性模块的符号名应该能反映出特性区、目录和文件的名字（如在 `app/heroes/heroes.module.ts` 中定义 `HeroesModule` ）。

为何？ 特性模块可以对其它模块暴露或隐藏自己的实现。

为何？ 特性模块标记出组成该特性分区的相关组件集合。

为何？ 特性模块能很容易的被路由器加载——无论使用主动加载还是惰性加载的方式。

为何？ 特性模块在特定的功能和其它应用特性之间定义了清晰的边界。

为何？ 特性模块帮助澄清开发职责，以便于把这些职责指派给不同的开发组。

为何？ 特性模块易于隔离，以便测试。

[回到顶部](#)

共享特性模块

风格 04-10

坚持 在 `shared` 目录中创建名叫 `SharedModule` 的特性模块（比如在 `app/shared/shared.module.ts` 中定义 `SharedModule` ）。

坚持 把可能被应用的其它特性模块使用的公共资产（如组件、指令和管道）放在 `SharedModule` 中，这些资产倾向于共享自己的新实例（而不是单例）。

坚持 在 `SharedModule` 中导入所有模块都需要的资产（比如：`CommonModule` 和 `FormsModule` ）。

为何？ `SharedModule` 中包含的组件、指令和管道可能需要来自其它公共模块的特性（比如来自 `CommonModule` 中的 `ngFor` 指令）。

坚持 在 `SharedModule` 中声明所有组件、指令和管道。

坚持 从 `SharedModule` 中导出其它特性模块所需的全部符号。

为何？ `SharedModule` 的存在，能让常用的组件、指令和管道在很多其它模块的组件模板中都自动可用。

避免 在 `SharedModule` 中指定应用级的单例服务提供商。但如果是故意设计的单例也可以，不过还是要小心。

为何？ 惰性加载的特性模块如果导入了这个共享模块，就会创建一份自己的服务副本，这可能会导致意料之外的后果。

为何？对于单例服务，你是不会希望每个模块都有自己的一份单独实例的。而如果 `SharedModule` 提供了一个服务，那就会发生这种情况。



```
1. import { NgModule }           from '@angular/core';
2. import { CommonModule }      from '@angular/common';
3. import { FormsModule }       from '@angular/forms';
4.
5. import { FilterTextComponent } from './filter-text/filter-
text.component';
6. import { FilterTextService }   from './filter-text/filter-
text.service';
7. import { InitCapsPipe }        from './init-caps.pipe';
8.
9. @NgModule({
10.   imports: [CommonModule, FormsModule],
11.   declarations: [
12.     FilterTextComponent,
```

```
13.     InitCapsPipe
14.   ],
15.   providers: [FilterTextService],
16.   exports: [
17.     CommonModule,
18.     FormsModule,
19.     FilterTextComponent,
20.     InitCapsPipe
21.   ]
22. })
23. export class SharedModule { }
```

[回到顶部](#)

核心特性模块

风格 04-11

坚持 把那些“只用一次”的类收集到 `CoreModule` 中，并且对外隐藏它们的实现细节。简化的 `AppModule` 会导入 `CoreModule`，并且把它作为整个应用的总指挥。

坚持 在 `core` 目录下创建一个名叫 `CoreModule` 的特性模块（比如在 `app/core/core.module.ts` 中定义 `CoreModule`）。

坚持 把一个要共享给整个应用的单例服务放进 `CoreModule` 中（比如 `ExceptionService` 和 `LoggerService`）。

坚持 导入 `CoreModule` 中的资产所需要的全部模块（比如 `CommonModule` 和 `FormsModule`）。

为何？ `CoreModule` 提供了一个或多个单例服务。Angular 使用应用的根注入器注册这些服务提供商，让每个服务的这个单例对象对所有需要它们的组件都是可用的，而不用管该组件是通过主动加载还是惰性加载的方式加载的。

为何？ `CoreModule` 将包含一些单例服务。而如果惰性加载模块导入这些服务，它就会得到一个新实例，而不是所期望的全应用级单例。

坚持 把应用级、只用一次的组件收集到 `CoreModule` 中。 只在应用启动时从 `AppModule` 中导入它一次，以后再也不要导入它（比如 `NavController` 和 `SpinnerComponent` ）。

为何？ 真实世界中的应用会有很多只用一次的组件（比如：加载动画、消息浮层、模态框等），它们只会在 `AppComponent` 的模板中出现，而不会出现在其它地方，所以它们没有被共享的价值。然而它们又太大了，放在根目录中就会显得乱七八糟的。

避免 在 `AppModule` 之外的任何地方导入 `CoreModule` 。

为何？ 如果惰性加载的特性模块直接导入 `CoreModule`，就会创建它自己的服务副本，并导致意料之外的后果。

为何？ 主动加载的特性模块已经准备好了访问 `AppModule` 的注入器，因此也能取得 `CoreModule` 中的服务。

坚持 从 `CoreModule` 中导出所有符号，`AppModule` 会导入它们，并让它们能在所有特性模块中可用。

为何？ `CoreModule` 的存在就能让常用的单例服务在所有其它模块中可用。

为何？ 你会希望整个应用都使用这个单例服务。你不会希望每个模块都有这个单例服务的单独的实例。然而如果 `CoreModule` 中提供了一个服务，就可能偶尔导致这种后果。

```
src
  --app
    --core
      --core.module.ts
      --logger.service.ts|spec.ts
    --nav
      --nav.component.ts|html|css|spec.ts
    --spinner
      spinner.component.ts|html|css|spec.ts
      spinner.service.ts|spec.ts
```

```
└─ app.component.ts|html|css|spec.ts  
  └─ app.module.ts  
  └─ app-routing.module.ts  
  └─ main.ts  
  └─ index.html  
  ...
```

```
1. import { NgModule }      from '@angular/core';  
2. import { BrowserModule } from '@angular/platform-browser';  
3.  
4. import { AppComponent }  from './app.component';  
5. import { HeroesComponent } from './heroes/heroes.component';  
6. import { CoreModule }    from './core/core.module';  
7.  
8. @NgModule({  
9.   imports: [  
10.     BrowserModule,  
11.     CoreModule,  
12.   ],  
13.   declarations: [  
14.     AppComponent,  
15.     HeroesComponent  
16.   ],  
17.   exports: [ AppComponent ],  
18.   entryComponents: [ AppComponent ]  
19. })  
20. export class AppModule {}
```

`AppModule` 变得更小了，因为很多应用根部的类都被移到了其它模块中。
`AppModule` 变得稳定了，因为你将会往其它模块中添加特性组件和服务提供商，而不是这个 `AppModule`。 `AppModule` 把工作委托给了导入的模块，而不是亲力亲为。`AppModule` 聚焦在它自己的主要任务上：作为整个应用的总指挥。

[回到顶部](#)

防止多次导入 CoreModule

风格 04-12

应该只有 `AppModule` 才能导入 `CoreModule`。

坚持 防范多次导入 `CoreModule`，并通过添加守卫逻辑来尽快失败。

为何？ 守卫可以阻止对 `CoreModule` 的多次导入。

为何？ 守卫会禁止创建单例服务的多个实例。

```
1. export function throwIfAlreadyLoaded(parentModule: any, moduleName: string) {
2.   if (parentModule) {
3.     throw new Error(`#${moduleName} has already been loaded. Import Core modules in the AppModule only.`);
4.   }
5. }
```

[回到顶部](#)

惰性加载的目录

样式 04-13

某些边界清晰的应用特性或工作流可以做成 **惰性加载** 或 **按需加载** 的，而不用总是随着应用启动。

坚持 把惰性加载特性下的内容放进 **惰性加载目录** 中。典型的 **惰性加载目录** 包含 **路由组件** 及其子组件以及与它们有关的那些资产和模块。

为何？ 这种目录让标识和隔离这些特性内容变得更轻松。

[回到顶部](#)

永远不要直接导入惰性加载的目录

样式 04-14

避免 让兄弟模块和父模块直接导入 **惰性加载特性** 中的模块。

为何？ 直接导入并使用此模块会主动加载它，而我们原本的设计意图是按需加载它。

[回到顶部](#)

组件

组件选择器命名

风格 05-02

坚持 使用 **中线 (dashed) 命名法** 或 **烤串 (kebab) 命名法** 来命名组件中的元素选择器。

为何？ 保持元素命名与 **自定义元素** 命名规范一致。

app/heroes/shared/hero-button/hero-button.component.ts

```
1.  /* avoid */
2.
3.  @Component({
4.    selector: 'tohHeroButton',
```

```

5.     templateUrl: 'hero-button.component.html'
6.   })
7.   export class HeroButtonComponent {}
```

```

1.   @Component({
2.     selector: 'toh-hero-button',
3.     templateUrl: 'hero-button.component.html'
4.   })
5.   export class HeroButtonComponent {}
```

[回到顶部](#)

把组件当做元素

风格 05-03

坚持 通过选择器把组件定义为元素。

为何？ 组件有很多包含 HTML 以及可选 Angular 模板语法的模板。它们的功能多数都与把内容放进页面有关，而这和 HTML 元素的设计意图很相似。

为何？ 组件表示页面上的可视元素。把该选择器定义成 HTML 元素标签可以与原生 HTML 和 WebComponent 保持一致。

为何？ 查看组件是否有模板 HTML 文件，是最简单的识别一个符号是组件还是指令的方法。

app/heroes/hero-button/hero-button.component.ts

```

1.   /* avoid */
2.
3.   @Component({
4.     selector: '[tohHeroButton]',
5.     templateUrl: 'hero-button.component.html'
```

```

6.      })
7.      export class HeroButtonComponent {}
```

app/app.component.html

```

1.      <!-- avoid -->
2.
3.      <div tohHeroButton></div>
```

```

1.      @Component({
2.        selector: 'toh-hero-button',
3.        templateUrl: 'hero-button.component.html'
4.      })
5.      export class HeroButtonComponent {}
```

[回到顶部](#)

把模板和样式提取到它们自己的文件

风格 05-04

坚持 当超过三行的时候，把模板和样式提取到一个单独的文件。

坚持 把模板文件命名为 `[component-name].component.html`，这里的 `[component-name]` 就是组件名。

坚持 把样式文件命名为 `[component-name].component.css`，这里的 `[component-name]` 就是组件名。

为何？ 在 `(.js 和 .ts)` 代码里面内联模板时，一些编辑器不支持语法提示。

为何？ 当没有与内联模板和样式混合的时候，组件文件里的逻辑更加易于阅读。

app/heroes/heroes.component.ts

```
1.  /* avoid */
2.
3.  @Component({
4.    selector: 'toh-heroes',
5.    template: `
6.      <div>
7.        <h2>My Heroes</h2>
8.        <ul class="heroes">
9.          <li *ngFor="let hero of heroes">
10.            <span class="badge">{{hero.id}}</span> {{hero.name}}
11.          </li>
12.        </ul>
13.        <div *ngIf="selectedHero">
14.          <h2>{{selectedHero.name | uppercase}} is my hero</h2>
15.        </div>
16.      </div>
17.    `,
18.    styleUrls: [
19.      '.heroes {
20.        margin: 0 0 2em 0; list-style-type: none; padding: 0; width:
21.        15em;
22.      }
23.      .heroes li {
24.        cursor: pointer;
25.        position: relative;
26.        left: 0;
27.        background-color: #EEE;
28.        margin: .5em;
29.        padding: .3em 0;
30.        height: 1.6em;
31.        border-radius: 4px;
32.      }
33.      .heroes .badge {
34.        display: inline-block;
35.        font-size: small;
36.        color: white;
37.        padding: 0.8em 0.7em 0 0.7em;
38.        background-color: #607D8B;
39.        line-height: 1em;
40.        position: relative;
41.      }
42.    ]
43.  }
44.  `,
45.  styleUrls: [
46.    './heroes.component.css'
47.  ]
48.}
```

```

40.         left: -1px;
41.         top: -4px;
42.         height: 1.8em;
43.         margin-right: .8em;
44.         border-radius: 4px 0 0 4px;
45.     }
46.   `]
47. )
48. export class HeroesComponent implements OnInit {
49.   heroes: Hero[];
50.   selectedHero: Hero;
51.
52.   ngOnInit() {}
53. }
```

```

1.  @Component({
2.    selector: 'toh-heroes',
3.    templateUrl: 'heroes.component.html',
4.    styleUrls: ['heroes.component.css']
5.  })
6.  export class HeroesComponent implements OnInit {
7.    heroes: Hero[];
8.    selectedHero: Hero;
9.
10.   ngOnInit() { }
11. }
```

[回到顶部](#)

内联 Input 和 Output 属性装饰器

风格 05-12

坚持使用 `@Directive` 和 `@Component` 装饰器的 `@Input` 和 `@Output`，而非 `inputs` 和 `outputs` 属性：

坚持把 `@Input()` 或者 `@Output()` 放到它们装饰的属性的同一行。

为何？ 这样易于在类里面识别哪个属性是 `inputs` 或 `outputs`。

为何？ 如果你需要重命名属性或者 `@Input` 或者 `@Output` 关联的事件名字，你可以在一个位置修改。

为何？ 依附到指令的元数据声明会比较简短，更易于阅读。

为何？ 把装饰器放到同一行可以精简代码，同时更易于识别输入或输出属性。

app/heroes/shared/hero-button/hero-button.component.ts

```
1.  /* avoid */
2.
3.  @Component({
4.    selector: 'toh-hero-button',
5.    template: `<button></button>`,
6.    inputs: [
7.      'label'
8.    ],
9.    outputs: [
10.      'change'
11.    ]
12.  })
13.  export class HeroButtonComponent {
14.    change = new EventEmitter<any>();
15.    label: string;
16.  }
```

app/heroes/shared/hero-button/hero-button.component.ts

```
1.  @Component({
2.    selector: 'toh-hero-button',
3.    template: `<button>{{label}}</button>`
4.  })
5.  export class HeroButtonComponent {
6.    @Output() change = new EventEmitter<any>();
7.    @Input() label: string;
```

```
8. }
```

[回到顶部](#)

避免重命名输入和输出

风格 05-13

避免 重命名输入和输出。

为何？可能导致混乱，造成指令的输入或输出属性的名字与导出的公共 API 名字不一样。

app/heroes/shared/hero-button/hero-button.component.ts

```
1. /* avoid */
2.
3. @Component({
4.   selector: 'toh-hero-button',
5.   template: `<button>{{label}}</button>`
6. })
7. export class HeroButtonComponent {
8.   @Output('changeEvent') change = new EventEmitter<any>();
9.   @Input('labelAttribute') label: string;
10. }
```

app/app.component.html

```
1. <!-- avoid -->
2.
3. <toh-hero-button labelAttribute="OK" (changeEvent)="doSomething()">
4. </toh-hero-button>
```

```

1.  @Component({
2.    selector: 'toh-hero-button',
3.    template: `<button>{{label}}</button>`
4.  })
5.  export class HeroButtonComponent {
6.    @Output() change = new EventEmitter<any>();
7.    @Input() label: string;
8.  }

```

[回到顶部](#)

成员顺序

风格 05-14

坚持 把属性成员放到顶部，方法成员紧随其后。

坚持 先放公共成员，再放私有成员，并按照字母顺序排列。

为何？ 把类的成员按照统一的顺序排列，可以让它们更易于阅读，这能帮我们立即识别出组件的那个成员服务于何种目的（比如是实现还是接口）。

app/shared/toast/toast.component.ts

```

1.  /* avoid */
2.
3.  export class ToastComponent implements OnInit {
4.
5.    private defaults = {
6.      title: '',
7.      message: 'May the Force be with You'
8.    };
9.    message: string;
10.   title: string;
11.   private toastElement: any;
12.

```

```

13.     ngOnInit() {
14.       this.toastElement = document.getElementById('toh-toast');
15.     }
16.
17.     // private methods
18.     private hide() {
19.       this.toastElement.style.opacity = 0;
20.       window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);
21.     }
22.
23.     activate(message = this.defaults.message, title =
24.       this.defaults.title) {
25.       this.title = title;
26.       this.message = message;
27.       this.show();
28.     }
29.     private show() {
30.       console.log(this.message);
31.       this.toastElement.style.opacity = 1;
32.       this.toastElement.style.zIndex = 9999;
33.
34.       window.setTimeout(() => this.hide(), 2500);
35.     }
36.   }

```

app/shared/toast/toast.component.ts

```

1.   export class ToastComponent implements OnInit {
2.     // public properties
3.     message: string;
4.     title: string;
5.
6.     // private fields
7.     private defaults = {
8.       title: '',
9.       message: 'May the Force be with You'
10.    };
11.    private toastElement: any;
12.

```

```

13.     // public methods
14.     activate(message = this.defaults.message, title =
15.       this.defaults.title) {
16.       this.title = title;
17.       this.message = message;
18.       this.show();
19.
20.     ngOnInit() {
21.       this.toastElement = document.getElementById('toh-toast');
22.     }
23.
24.     // private methods
25.     private hide() {
26.       this.toastElement.style.opacity = 0;
27.       window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);
28.     }
29.
30.     private show() {
31.       console.log(this.message);
32.       this.toastElement.style.opacity = 1;
33.       this.toastElement.style.zIndex = 9999;
34.       window.setTimeout(() => this.hide(), 2500);
35.     }
36.   }

```

[回到顶部](#)

把逻辑放到服务里

风格 05-15

坚持 把组件类中的逻辑限制到只有视图需要的逻辑。所有其它逻辑都应该被放到服务。

坚持 把可以重复使用的逻辑放到服务里，保持组件简单并聚焦于它们预期目的。

为何？当逻辑被放置到服务里并以函数的形式暴露时，它可以被多个组件重复使用。

为何？在单元测试时，服务里的逻辑更加容易被隔离。在组件里调用它的逻辑也很容易被模仿 Mock。

为何？从组件移除依赖并隐藏实施细节。

为何？保持组件苗条、精简和聚焦

app/heroes/hero-list/hero-list.component.ts

```
1.  /* avoid */
2.
3.  import { OnInit } from '@angular/core';
4.  import { Http, Response } from '@angular/http';
5.  import { Observable } from 'rxjs/Observable';
6.
7.  import { Hero } from '../shared/hero.model';
8.
9.  const heroesUrl = 'http://angular.io';
10.
11. export class HeroListComponent implements OnInit {
12.   heroes: Hero[];
13.   constructor(private http: Http) {}
14.   getHeroes() {
15.     this.heroes = [];
16.     this.http.get(heroesUrl)
17.       .map((response: Response) => <Hero[]>response.json().data)
18.       .catch(this.catchBadResponse)
19.       .finally(() => this.hideSpinner())
20.       .subscribe((heroes: Hero[]) => this.heroes = heroes);
21.   }
22.   ngOnInit() {
23.     this.getHeroes();
24.   }
25.
26.   private catchBadResponse(err: any, source: Observable<any>) {
27.     // log and handle the exception
28.     return new Observable();
29.   }
30.
```

```

31.     private hideSpinner() {
32.         // hide the spinner
33.     }
34. }
```

app/heroes/hero-list/hero-list.component.ts

```

1.  import { Component, OnInit } from '@angular/core';
2.
3.  import { Hero, HeroService } from '../shared';
4.
5.  @Component({
6.     selector: 'toh-hero-list',
7.     template: `...
8. })
9. export class HeroListComponent implements OnInit {
10.    heroes: Hero[];
11.    constructor(private heroService: HeroService) {}
12.    getHeroes() {
13.        this.heroes = [];
14.        this.heroService.getHeroes()
15.            .subscribe(heroes => this.heroes = heroes);
16.    }
17.    ngOnInit() {
18.        this.getHeroes();
19.    }
20. }
```

[回到顶部](#)

不要给输出属性加前缀

风格 05-16

坚持 命名事件时，不要带前缀 `on`。

坚持 把事件处理器方法命名为 `on` 前缀之后紧跟着事件名。

为何？ 与内置事件命名一致，比如按钮点击。

为何？ Angular 允许 **另一种备选语法** `on-*`。如果事件的名字本身带有前缀 `on`，那么绑定的表达式可能是 `on-onEvent`。

app/heroes/hero.component.ts

```
1.  /* avoid */
2.
3.  @Component({
4.    selector: 'toh-hero',
5.    template: `...
6.  })
7.  export class HeroComponent {
8.    @Output() onSaveedTheDay = new EventEmitter<boolean>();
9. }
```

app/app.component.html

```
1.  <!-- avoid -->
2.
3.  <toh-hero (onSavedTheDay)="onSavedTheDay($event)"></toh-hero>
```

```
1.  export class HeroComponent {
2.    @Output() savedTheDay = new EventEmitter<boolean>();
3. }
```

[回到顶部](#)

把展示逻辑放到组件类里

风格 05-17

坚持 把表现层逻辑放进组件类中，而不要放在模板里。

为何？ 逻辑应该只出现在一个地方（组件类里）而不应分散在两个地方。

为何？ 将组件的展示逻辑放到组件类而非模板里，可以增强测试性、维护性和重用使用性。

app/heroes/hero-list/hero-list.component.ts

```

1.  /* avoid */
2.
3.  @Component({
4.    selector: 'toh-hero-list',
5.    template: `
6.      <section>
7.        Our list of heroes:
8.        <hero-profile *ngFor="let hero of heroes" [hero]="hero">
9.        </hero-profile>
10.       Total powers: {{totalPowers}}<br>
11.       Average power: {{totalPowers / heroes.length}}
12.     </section>
13.   `
14. })
15. export class HeroListComponent {
16.   heroes: Hero[];
17.   totalPowers: number;
18. }
```

app/heroes/hero-list/hero-list.component.ts

```

1.  @Component({
2.    selector: 'toh-hero-list',
3.    template: `
4.      <section>
5.        Our list of heroes:
6.        <toh-hero *ngFor="let hero of heroes" [hero]="hero">
7.        </toh-hero>
```

```

8.     Total powers: {{totalPowers}}<br>
9.     Average power: {{avgPower}}
10.    </section>
11.    .
12.  })
13. export class HeroListComponent {
14.   heroes: Hero[];
15.   totalPowers: number;
16.
17.   get avgPower() {
18.     return this.totalPowers / this.heroes.length;
19.   }
20. }

```

[回到顶部](#)

指令

使用指令来加强已有元素

风格 06-01

坚持 当你需要有无模板的展示逻辑时，使用属性型指令。

为何？ 属性型指令没有配套的模板。

为何？ 一个元素可能使用多个属性型指令。

app/shared/highlight.directive.ts

```

1.  @Directive({
2.   selector: '[toHighlight]'
3. })
4. export class HighlightDirective {
5.   @HostListener('mouseover') onMouseEnter() {
6.     // do highlight work

```

```
7.      }
8.    }
```

app/app.component.html

```
<div toHighlight>Bombasta</div>
```

[回到顶部](#)

使用 HostListener 和 HostBinding 类装饰器

风格 06-03

考虑 优先使用 `@HostListener` 和 `@HostBinding`，而不是 `@Directive` 和 `@Component` 装饰器的 `host` 属性。

坚持 让你的选择保持一致。

为何？ 对于关联到 `@HostBinding` 的属性或关联到 `@HostListener` 的方法，要改名时只要在指令类中修改一次就行了。如果使用元数据属性 `host`，你就得在组件类中修改属性声明的同时修改相关的元数据。

app/shared/validator.directive.ts

```
1.  import { Directive, HostBinding, HostListener } from '@angular/core';
2.
3.  @Directive({
4.    selector: '[tohvalidator]'
5.  })
6.  export class ValidatorDirective {
7.    @HostBinding('attr.role') role = 'button';
8.    @HostListener('mouseenter') onMouseEnter() {
9.      // do work
10.    }
11.  }
```

与不推荐的方式（`host` 元数据）比较一下。

为何？ `host` 元数据只是一个便于记忆的名字而已，并不需要额外的 ES 导入。

app/shared/validator2.directive.ts

```
1. import { Directive } from '@angular/core';
2.
3. @Directive({
4.   selector: '[tohvalidator2]',
5.   host: {
6.     'attr.role': 'button',
7.     '(mouseenter)': 'onMouseEnter()'
8.   }
9. })
10. export class Validator2Directive {
11.   role = 'button';
12.   onMouseEnter() {
13.     // do work
14.   }
15. }
```

[回到顶部](#)

服务

在同一个注入器中，服务总是单例的

风格 07-01

坚持 在同一个注入器内，把服务当做单例使用。使用它们来共享数据和功能。

为何？ 服务是在一个特性范围或一个应用内理想的共享方法的理想载体。

为何？服务是共享状态性内存数据的理想方法。

app/heroes/shared/hero.service.ts

```
1.  export class HeroService {
2.    constructor(private http: Http) { }
3.
4.    getHeroes() {
5.      return this.http.get('api/heroes')
6.        .map((response: Response) => <Hero[]>response.json().data);
7.    }
8.  }
```

[回到顶部](#)

单一职责

风格 07-02

坚持 新建单一职责的服务，把它封装在自己的环境内。

坚持 当服务成长到超出单一用途时，新建一个服务。

为何？ 当服务有多个职责时，它很难被测试。

为何？ 当某个服务有多个职责时，每个注入它的组件或服务都会承担这些职责的全部开销。

[回到顶部](#)

提供一个服务

风格 07-03

坚持 在被共享范围内的顶级组件里，将服务提供到 Angular 2 的注入器里。

为何？ Angular 注入器是层次性的。

为何？ 在顶层组件提供服务时，该服务实例在所有该顶级组件的子级组件中可见并共享。

为何？ 服务是共享方法或状态的理想方法。

为何？ 当不同的两个组件需要一个服务的不同的实例时，上面的方法这就不理想了。在这种情况下，我们最好在需要崭新和单独服务实例的组件里提供服务。

```
1. import { Component } from '@angular/core';
2.
3. import { HeroService } from './heroes';
4.
5. @Component({
6.   selector: 'toh-app',
7.   template: `
8.     <toh-heroes></toh-heroes>
9.   `,
10.  providers: [HeroService]
11. })
12. export class AppComponent {}
```

[回到顶部](#)

使用 `@Injectable()` 类装饰器

风格 07-04

坚持 当使用类型作为令牌来注入服务的依赖时，使用 `@Injectable` 类装饰器，而非 `@Inject` 参数装饰器。

为何？ Angular 的 DI 机制会基于在服务的构造函数中所声明的类型来解析这些服务的依赖。

为何？当服务只接受类型令牌相关的依赖时，比起在每个构造函数参数上使用`@Inject()`，`@Injectable()`的语法简洁多了。

app/heroes/shared/hero-arena.service.ts

```
1.  /* avoid */
2.
3.  export class HeroArena {
4.    constructor(
5.      @Inject(HeroService) private heroService: HeroService,
6.      @Inject(Http) private http: Http) {}
7. }
```

app/heroes/shared/hero-arena.service.ts

```
1.  @Injectable()
2.  export class HeroArena {
3.    constructor(
4.      private heroService: HeroService,
5.      private http: Http) {}
6. }
```

[回到顶部](#)

数据服务

分离数据调用

风格 08-01

坚持把数据操作和数据互动重构到服务里。

坚持让数据服务来负责 XHR 调用、本地储存、内存储存或者其它数据操作。

为何？ 组件的职责是为视图展示或收集信息。它不应该理会如何得到数据，它只需要知道向谁要数据。把如何取得数据的逻辑移动到数据服务里，简化了组件，让其聚焦于视图。

为何？ 在测试使用数据服务的组件时，可以让数据调用更容易被测试（模仿或者真实）。

为何？ 数据服务的实现可能有非常具体的代码来处理数据仓库，包括数据头(headers)、如何与数据交谈或者其它服务（比如 Http）。把逻辑分离到数据服务可以把该逻辑封装到一个地方，对外部使用者（比如组件）隐藏具体的实施细节。

[回到顶部](#)

生命周期钩子

使用生命周期钩子来介入到 Angular 暴露的重要事件里。

[回到顶部](#)

实现生命周期钩子接口

风格 09-01

坚持 实现生命周期钩子接口。

为何？ 如果使用强类型的方法签名，那么编译器和编辑器可以帮你揪出拼写错误。

app/heroes/shared/hero-button/hero-button.component.ts

```
1.  /* avoid */
2.
3.  @Component({
4.    selector: 'toh-hero-button',
5.    template: `<button>OK</button>`
6.  })
7.  export class HeroButtonComponent {
```

```
8.     ngOnInit() { // misspelled
9.       console.log('The component is initialized');
10.    }
11. }
```

app/heroes/shared/hero-button/hero-button.component.ts

```
1. @Component({
2.   selector: 'toh-hero-button',
3.   template: `<button>OK</button>`
4. })
5. export class HeroButtonComponent implements OnInit {
6.   ngOnInit() {
7.     console.log('The component is initialized');
8.   }
9. }
```

[回到顶部](#)

附录

有用的 Angular 工具和小提示

[回到顶部](#)

Codelyzer

风格 A-01

坚持 使用 `codelyzer` 来实施本指南。

考虑 调整 `codelyzer` 的规则来满足你的需求。

[回到顶部](#)

文档模板和代码片段

风格 A-02

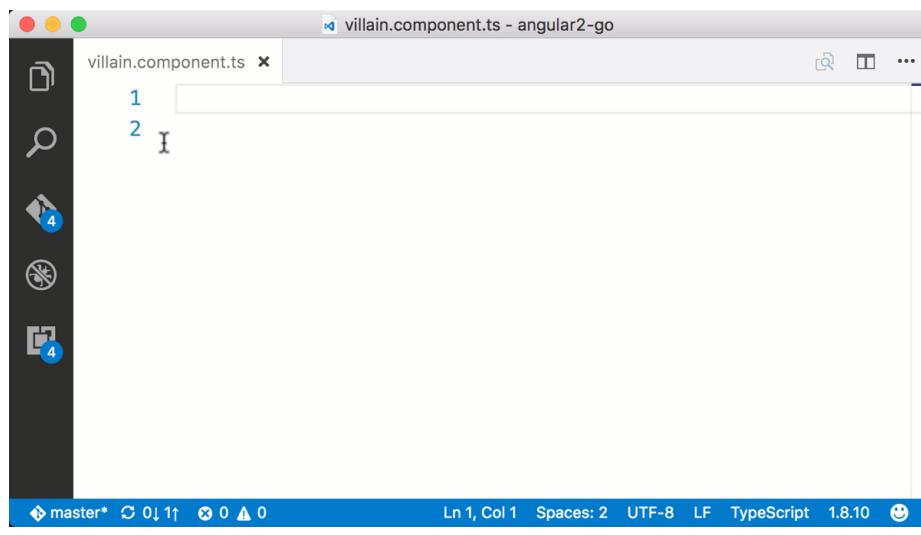
坚持 使用文件模板或代码片段来帮助实现一致的风格和模式。下面是为一些网络开发编辑器和 IDE 准备的模板和 / 或代码片段：

考虑 使用 Atom 的 [代码片段](#) 来实施本风格指南。

考虑 使用 Sublime Text 的 [代码片段](#) 来实施本风格指南。

考虑 使用 Vim 的 [代码片段](#) 来实施本风格指南。

考虑 使用 Visual Studio Code 的 [代码片段](#) 来实施本风格指南。



[回到顶部](#)

词汇表

Angular 中最重要的词汇的简要定义

Angular 词汇表

Angular 2 有自己的词汇表。 虽然大多数的 Angular 2 短语都是日常用语，但是在 Angular 体系中，它们有特别的含义。

该词汇表列出了常用词和少量具有独特或反直觉含义的罕用词。

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Ahead of Time (AOT) 预编译

开发者可以在构造时 (build-time) 编译 Angular 应用程序。通过 [Compiler-cli](#) - [ngc](#) 编译应用程序，应用可以从一个模块工厂 (Module Factory) 直接启动，意思是不再需要把 Angular 编译器添加到 JavaScript 包中。预编译的应用程序将加载迅速，并具有更高的性能。

Angular 模块

帮助我们将一个应用程序组织成拼合的功能模块群。一个 Angular 模块标识了被应用程序使用的组件、指令和管道等，它同时包含了应用程序需要的外来

Angular 模块的列表，比如 `FormsModule`。

每个 Angular 应用程序都有一个应用程序根模块类。按规约这个类的名字为 `AppModule`，存放在名为 `app.module.ts` 的文件。

要了解详情和范例，参见 [Angular 模块](#) 页。

注解 (Annotation)

[装饰器 \(Decoration\)](#) 在实践中的同义词。

属性型指令 (Attribute Directive)

指令 (Directive) 分类中的一种。它允许监听或修改其它 HTML 元素、属性、组件的行为。它们通常用作 HTML 属性，就像它的名字所暗示的那样。

`ngClass` 指令就是典型的属性型指令。它可以添加或移除 CSS 类名。

封装桶 (Barrel)

封装桶是把一系列模块中的 **导出结果** 归纳进一个单一的快捷模块的一种方式。封装桶本身是一个模块文件，它重新导出 **选中的** 导出，这些导入来自其它文件。

设想在 `heroes` 目录下有三个 ES2015 的模块：

```
// heroes/hero.component.ts  
export class HeroComponent {}  
  
// heroes/hero.model.ts  
export class Hero {}  
  
// heroes/hero.service.ts  
export class HeroService {}
```

假如没有封装桶，消费者就需要三条 import 语句：

```
import { HeroComponent } from '../heroes/hero.component.ts';  
import { Hero } from '../heroes/hero.model.ts';  
import { HeroService } from '../heroes/hero.service.ts';
```

在 `heroes` 目录下添加一个封装桶（按规约叫做 `index`），它导出所有这三条：

```
export * from './hero.model.ts'; // re-export all of its  
exports  
export * from './hero.service.ts'; // re-export all of its  
exports  
export { HeroComponent } from './hero.component.ts'; // re-  
export the named thing
```

现在，消费者就可以从这个封装桶中导入它需要的东西了。

```
import { Hero, HeroService } from '../heroes'; // index is  
implied
```

Angular 的每个 范围化包 (Scoped Package) 都有一个叫做 `index` 的封装桶。

这就是为什么可以这样写：

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

注意，你可以利用 Angular 模块 达到同样的目的。

绑定 (Binding)

几乎都是指的 [数据绑定 \(Data Binding\)](#) 和将一个 HTML 对象属性绑定到一个数据对象属性的行为。

有可能指的是 [依赖注入 \(Dependency Injection\)](#) 在一个令牌 (Token) (也叫键值 Key) 和一个依赖的 [提供商 \(Provider\)](#) 之间的绑定。这种用法很少，而且一般都会在上下文中写清楚。

启动 / 引导 (Bootstrap)

通过一个名叫 `bootstrap` 的方法来引导 Angular 应用程序。这个 `bootstrap` 方法会识别应用程序的顶级“根” [组件 \(Component\)](#)，并可能通过 [依赖注入体系 \(Dependency Injection System\)](#) 注册服务的 [提供商 \(Provider\)](#)。要了解详情，参见 [快速起步](#)。

你可以在同一个 `index.html` 中引导多个应用，每个应用都有它自己的顶级根组件。

驼峰式命名法 (camelCase)

驼峰式命名法是指除了首字母要小写外，每个单词或缩写都以大写字母开头的形式来书写复合词或短语的一种实践。

函数、属性和方法命名一般都使用驼峰式拼写法。比如 `square`，`firstName` 和 `getHeroes` 等。注意这里的 `square` 只是一个例子，用来示范如何用驼峰式命

名法表示单一词。

这种形式也被叫做 **小驼峰式命名法 (lower camel case)** , 以区别于 **大驼峰式命名法 (也叫 Pascal 命名法 (PascalCase))** 。 在文档中提到“驼峰式命名法 (camelCase) ”的时候 , 我们所指的都是小驼峰命名法。

组件 (Component)

组件是一种用来把数据展示到 **视图 (View)** , 并处理几乎所有的视图显示和交互逻辑的 Angular 类。

组件是 Angular 系统中最重要的基本构造块之一。 它其实是一个拥有 **模板 (Template)** 的 **指令 (Directive)** 。

开发人员使用 **@Component** 来装饰一个组件类 , 也就是把这个核心组件的元数据附加到类上。 Angular 会利用这个元数据信息创建一个组件实例 , 并把组件的模板作为视图渲染出来。

如果你熟悉 "MVC" 和 "MVVM" 架构模式 , 就会意识到“组件”充当了“控制器 (Controller) ”和“视图模型 (View Model) ”的角色。

中线命名法 (dash-case)

使用中线 (-) 分隔每个单词来书写词汇或短语的方法叫做中线命名法。 这种命名法也被称为 **烤串命名法 (kebab-case)** 。

指令 的选择器 (例如 `my-app`) 和文件名 (比如 `'hero-list.component.ts'`) 通常都是通过中线命名法来命名的。

数据绑定 (Data Binding)

应用程序会将数据展示给用户，并对用户的操作（点击、触屏、按键）做出回应。

以前的手动操作是：将数据推送到 HTML 页面中、添加事件监听器、从屏幕获取变化后的数据，并更新应用中的值。现在，你可以声明 HTML 小部件和数据源之间的关系，并让框架来处理各种细节。

Angular 有一个非常强大的数据绑定框架，它带有很多种数据绑定操作，并支持声明式语法。

到 [模板语法](#) 页了解更多的绑定形式：

- [插值表达式 \(Interpolation\)](#)。
- [属性绑定 \(Property Binding\)](#)。
- [事件绑定 \(Event Binding\)](#)。
- [Attribute 绑定 \(Attribute Binding\)](#)。
- [CSS 类绑定 \(Class Binding\)](#)。
- [样式绑定 \(Style Binding\)](#)。
- [基于 ngModel 的双向数据绑定 \(Two-way data binding with ngModel\)](#)。

装饰器 (Decorator | Decoration)

装饰器是一个 [函数](#)，这个函数将元数据添加到类、类成员（属性、方法）和函数上。

装饰器是一个 JavaScript 的语言 [特性](#)，装饰器在 TypeScript 里面已经采纳并实现了，并被推荐到了 ES2016(也就是 ES7)。

要想应用装饰器，就把装饰器放到被装饰对象的上面或左边。

Angular 使用自己的一套装饰器来实现应用程序各部分之间的相互操作。下面的例子中使用了 `@Component` 装饰器来将一个类标记为一个 Angular 组件 (Component) , 并用 `@Input` 装饰器来装饰该组件的 `name` 属性。
`@Component` 装饰器中省略的参数对象还可以包含和组件有关的元数据。

```
1.  @Component({...})
2.  export class AppComponent {
3.    constructor(@Inject('SpecialFoo') public foo:Foo) {}
4.    @Input() name:string;
5.  }
```

一个装饰器的作用域会被限制在它所装饰的东西上，这是一个语言级特性。在上面这个例子中，就算别的类在同一个文件中紧跟着上面的类也不会有任何装饰器“泄露”到其它类。

永远别忘了在装饰器后面加括号 `()` 。

依赖注入 (Dependency Injection)

依赖注入既是设计模式，同时又是一种机制：当应用程序的一些部件需要另一些部件的时候，使用依赖注入机制来创建被请求的部件并将其注入到发出请求的部件中。

Angular 开发者构建应用程序时的首选方法是：定义许多精简的小部件，每个小部件只做一件事并做好它，然后在运行期把这些精简小部件装配在一起组成应用程序。

这些小部件通常会依赖其它小部件。一个 Angular 组件 (Component) 可能依赖一个“服务”部件来获取数据或处理运算。如果部件 A 要靠另一个部件 B 才能工作，那么 A “依赖于” B ， B 是 A 的“依赖”。

我们可以要求“依赖注入系统”为我们创建一个部件 A 并处理所有 A 的“依赖”。如果 A 需要 B , B 需要 C , 这个系统便解析这个依赖链，返回一个完全准备好的 A 实例。

Angular 提供并使用自己精心设计的 [依赖注入 \(Dependency Injection\)](#) 系统来组装和运行应用程序：在要用时，将一些部件“注入”到另一些部件里面。

在 Angular 内核中有一个 [注入器 \(Injector\)](#)，这个注入器根据需要返回被依赖部件。`injector.get(token)` 方法返回与该 token(令牌) 参数相关的依赖部件。

令牌是一个 Angular 中的类型 ([OpaqueToken](#))。我们很少需要直接接触令牌。绝大多数类方法都接受类名 (`Foo`) 或字符串 ("foo") , Angular 会把这些类名称和字符串转换成令牌。当调用 `injector.get(Foo)` 时，注入器返回用 `Foo` 类生成的令牌所对应的依赖值，该依赖值通常是 `Foo` 类的实例。

Angular 在创建 [组件 \(Component\)](#) 以供显示的过程中，会在内部执行很多类似的依赖注入请求。

注入器 (`Injector`) 维护一个令牌与相应依赖值的对照表 (map)。如果注入器不能找到一个令牌对应的依赖值，它就会使用提供商 (`Provider`) 来创建一个依赖值。

[提供商 \(Provider\)](#) 是创建依赖实例的“菜谱”之一，这个实例会与一个特定的令牌关联起来。

只有当注入器内部的提供商注册表中存在与令牌对应的提供商时，注入器才能为这个令牌创建一个依赖值。所以注册提供商是一个非常关键的准备步骤。

Angular 会为每个注册器注册很多 Angular 内置提供商。我们也可以注册自己的提供商。通常注册提供商的最佳时间是在应用程序开始 [引导 \(Bootstrap\)](#) 的时候。当然，我们也有其它很多机会注册提供商。

要了解关于依赖注入的更多知识，请参见 [依赖注入 \(Dependency Injection\)](#) 页。

指令 (Directive)

指令是一个 Angular 类，这个类负责创建和重塑浏览器 DOM 中的 HTML 元素，同时负责与 HTML 元素的互动。指令是 Angular 中最基本的特性之一。

指令几乎都是关联到 HTML 元素或属性 (Attribute) 的。我们通常把这些关联到的 HTML 元素或者属性 (Attribute) 当做指令本身。当 Angular 在 HTML 模板中遇到一个指令的时候，它就会找出一个与该指令相匹配的类，创建此类的实例，然后把浏览器中这部分 DOM 的控制权交给它。

你可以为自定义指令指定自定义的 HTML 标签（比如 `<my-directive>`），然后，就可以像写原生 HTML 一样把这些自定义标签放到 HTML 模板里了。这样，指令就变成了 HTML 本身的拓展。

指令包括了一下三个类别：

1. **组件 (Component)**: 用来把程序逻辑和 HTML 模板组合起来，渲染出应用程序的视图。组件一般表示成 HTML 元素的形式，它们是构建 Angular 应用程序的基本单元。可以预见，开发人员将会写很多很多组件。
2. **属性型指令 (Attribute Directive)** : 可以监控和修改其它 HTML 元素、HTML 属性 (Attribute)、DOM 属性 (Property)、组件等行为等等。它们一般表示为 HTML 元素的属性 (Attribute)，故名。
3. **结构型指令 (Structural Directive)** : 负责塑造或重塑 HTML 布局。这一般是通过添加、删除或者操作 HTML 元素及其子元素来实现的。

ECMAScript

官方 JavaScript 语言规范

最新的被认可的 JavaScript 版本是 **ECMAScript 2015**，(也叫“ES2015”或“ES6”)。Angular 2 的开发者要么使用这个版本的 JavaScript，要么使用与这个版本兼容的语言，比如 **TypeScript**。

目前，几乎所有现代浏览器都只支持上老版本的“ ECMAScript 5 ” (也就是 ES5) 标准。使用 ES2015 ， ES2016 或者其它兼容语言开发的应用程序，都必须被“ [转译 \(Transpile\)](#) ”成 ES5 JavaScript 。

Angular 的开发者也可以选择直接使用 ES5 编程。

ES 2015

[ECMAScript 2015](#) 的缩写。

ES6

[ECMAScript 2015](#) 的简写。

ES5

“ ECMAScript 5 ”的缩写，大部分现代浏览器使用的 JavaScript 版本。参见 [ECMAScript](#) 。

注入器 (Injector)

Angular 依赖注入系统 (Dependency Injection System) 中的一个对象，它可以在自己的缓存中找到一个“有名字的依赖”或者利用一个已注册的 提供商 (Provider) 来创建这样一个依赖。

输入属性 (Input)

指令属性可以作为 属性绑定 的目标。数据值会从模板表达式等号右侧的数据源中，流入这个属性。

参见 模板语法 页的 输入与输出属性 部分。

插值表达式 (Interpolation)

属性数据绑定 (Property Data Binding) 的形式之一：位于双大括号中的 模板表达式 (Template Expression) 会被渲染成文本。在被赋值给元素属性或者显示在元素标签中之前，这些文本可能会先与周边的文本合并，参见下面的例子。

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

要学习关于插值表达式的更多知识，参见 模板语法 一章。

即时 (Just-in-time (JiT)) 编译

Angular 的即时编译在浏览器中启动并编译所有的组件和模块，并动态运行应用程序。它很适合在开发期使用。但是在产品发布时，推荐采用 预编译 (

Ahead of Time) 模式。

烤串命名法 (kebab-case)

参见 [中线命名法 \(dash-case\)](#)。

生命周期钩子 (Lifecycle Hook)

指令 (Directive) 和 组件 (Component) 具有生命周期，它们由 Angular 在创建、更新和销毁它们的过程中进行管理。

你可以通过实现一个或多个“生命周期钩子”接口，切入到生命周期中的关键时间点。

每个接口只有一个钩子方法，方法名一般是接口的名字加前缀 `ng`。比如，`OnInit` 接口的钩子类的方法名为 `ngOnInit`。

Angular 会按照下面的顺序调用钩子类方法：

- `ngOnChanges` - 在 输入属性 (Input)/ 输出属性 (Output) 的绑定值发生变化的时候调用。
- `ngOnInit` - 在第一轮 `ngOnChanges` 完成后调用。
- `ngDoCheck` - 开发者自定义变化监测器。
- `ngAfterContentInit` - 在组件初始化以后调用。
- `ngAfterContentChecked` - 在检查每个组件内容后调用。
- `ngAfterViewInit` - 在组件视图初始化后调用。
- `ngAfterViewChecked` - 在检查每个组件视图后调用
- `ngOnDestroy` - 在指令销毁前调用。

要了解更多，参见 [生命周期钩子 \(Lifecycle Hook\)](#) 一章。

模块 (Module)

在 Angular 里有两种模块：

- [Angular 模块](#)。到 [Angular Module](#) 页查看详情和例子。
- 本节描述的 ES2015 模块。

Angular 应用程序是模块化的。

一般来说，我们用模块来组装应用程序，这些模块包含我们自己编写的模块和从其它地方获取的模块。

典型的模块，是具有单一用途的内聚代码块。

模块代码中通常会 **导出 (export)** 一些东西，最典型的就是类。模块如果需要什么东西，那就 **导入 (import)** 它。

Angular 的模块结构和输出 / 导入语法是基于 [ES2015 模块化标准](#) 的。

采用这个标准的应用程序需要一个模块加载器来按需加载模块并解析模块的依赖关系。Angular 不包含任何模块加载器，也不偏爱哪一个第三方库（虽然几乎所有例子都使用 SystemJS）。你可以自行选择任何与这个标准兼容的模块化库。

模块一般与它定义导出物的文件同名。比如，Angular 的 [日期管道 \(DatePipe\)](#) 类属于名叫 `date_pipe` 的特性模块，位于文件 `date_pipe.ts` 中。

你很少需要直接访问 Angular 的特性模块。而通常会从一个 Angular 的 [范围化包 \(Scoped Package\)](#) 中导入它们，比如 `@angular/core`。

Observable

一个 `Observable` 是一个数组，它包含的元素在一段时间内异步到达。

`Observable` 帮助我们管理异步数据，比如来自后台服务的数据。Angular 自身使用了 `Observable`，包括 Angular 的事件系统和它的 http 客户端服务。

为了利用 `Observable`，Angular 使用了名为 Reactive Extensions (RxJS) 的第三方包。在下个版本的 JavaScript - ES 2016 中，`Observable` 是建议的功能之一。

输出属性 (Output)

输出属性是指令的一种属性，它可作为 [事件绑定](#) 的 **目标**。事件流可以通过这个属性，流出到接收者（模板表达式等号的右边就是接收者）。

参见 [模板语法](#) 中的 [输入与输出属性](#) 部分。

Pascal 命名法 (PascalCase)

遵循“每个单词都用大写开头”的规则的书写单词、复合词或短语的命名方法叫做 Pascal 命名法。类名一般都采用 Pascal 命名法。比如 `Person` 和 `Customer`

这种命名法也叫 **大驼峰式命名法**，以区别于 **小驼峰式命名法** 或 **驼峰式命名法 (camelCase)**。在本教程中，“Pascal 命名法”都是指的 * 大驼峰式命名法”，“驼峰式命名法”指的都是“小驼峰式命名法”

管道 (Pipe)

Angular 的管道是一个函数，用于把输入值转换成输出值以供 **视图 (View)** 显示。使用 **Pipe** 来把管道函数关联到它的名字上。然后，就可以在 HTML 中用它的名字来声明该如何把输入值转换为显示值了。

下面这个例子中就用内置的 **currency** 管道把数字值显示成了本地货币格式。

```
<label>Price: </label>{{product.price | currency}}
```

Read more in the page on [pipes](#).

提供商 (Provider)

依赖注入系统依靠提供商来创建依赖的实例。它把一个供查阅用的令牌和代码（有时也叫“配方”）关联到一起，以便创建依赖值。

动态表格 (Reactive Forms)

通过组件代码来构建 Angular 表单的方法。

构建动态表单是：

- “真理来源”于组件。表单验证在组件代码中定义。
- 每个控制器都是在组件类中使用 **new FormControl()** 或者 **FormBuilder** 显性的创建的。
- 模板中的 **input** 元素 **不** 使用 **ngModel** 。

- 相关联的 Angular 指令全部以 `Form` 开头，比如 `FormGroup`、`FormControl` 和 `FormControlName`。

动态表单非常强大、灵活，它在复杂数据输入的场景下尤其好用，比如动态的生成表单控制器。

路由器 (Router)

大部分应用程序包含多个屏或 [视图 \(View\)](#)。用户通过点击链接、按钮和其它类似动作，在它们之间穿梭，这样应用程序就会从一个视图变换到另一个视图。

Angular 的 [路由器 \(Component Router\)](#) 是一个特性丰富的机制，它可以配置和管理整个导航过程，包括建立和销毁视图。

多数情况下，组件会通过 `RouterConfig` 中定义的路由到视图的对照表来附加到 [路由器](#) 上。

[路由组件](#) 的模板中带有一个 `RouterOutlet` 元素，它用来显示由路由器生成的视图。

应用中的其它视图中某些 A 标签或按钮上带有 `RouterLink` 指令，用户可以点击它们来导航到这里。

要了解更多，请参见 [路由与导航](#) 页。

Router module

一个独立的 [Angular 模块](#) 是用来提供导航时所需的必备服务提供商和指令的。

要了解更多，请参见 [路由与导航](#) 页。

路由组件 (Routing Component)

带有 RouterOutlet 的 Angular 组件 基于路由器导航来显示视图。

要了解更多，请参见 [路由与导航](#) 页。

范围化包 (Scoped Package)

Angular 模块是用一系列 范围化包 的形式发布的，比如 `@angular/core` 、
`@angular/common` 、 `@angular/platform-browser-dynamic` 、 `@angular/http`
和 `@angular/router` 。

范围化包 (Scoped Package) 是对相关 npm 包进行分组的一种方式。

使用和导入 普通 包相同的方式导入范围化包。 从消费者的视角看，唯一的不同是那些包的名字是用 Angular 的 范围化包名 `@angular` 开头的。

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

下划线命名法 (蛇形命名法)

用下划线分隔复合词或词组的命名方法称为 **下划线命名法** 。

服务

组件很强大很好 ... 但是，我们该如何处理那些不与任何特定视图相关的数据和逻辑？又如何在组件之间共享这些数据和逻辑？我们创建服务！

应用程序经常需要服务，比如英雄数据服务或者日志服务。组件依赖这些服务来做一些繁重的工作。

服务是一个具有特定功能的类。我们经常创建服务来实现不依赖任何特定视图的特征、在组件之间提供共享数据或逻辑，或者封装外部互动等。

要了解更多，请参见 [英雄指南](#) 教程中的 [服务](#) 页。

结构型指令 (Structural Directive)

结构型指令是一种可以通过添加、删除或操作元素和其各级子元素来塑造或重塑 HTML 布局的 [指令 \(Directive\)](#)，

要了解更多，请参见 [结构型指令](#) 页。

模板 (Template)

模板是一大块 HTML。Angular 会在 [指令 \(Directive\)](#) 特别是 [组件 \(Component\)](#) 的支持和持续指导下，用它来渲染 [视图 \(View\)](#)。

模板驱动表单

这是一项在视图中使用 HTML 表单和输入类元素构建 Angular 表单的技术。它的替代方案是 [响应式表单](#)。

当构建模板驱动表单时：

- “信任之源”是模板。验证规则是用属性（Attribute）的形式定义在独立输入控件上的。
- 使用 `ngModel` 的 双向绑定 负责组件模型和用户输入之间的同步。
- 在幕后，Angular 为每个带有 `name` 属性和双向绑定的 `input` 元素创建了一个新的控件。
- 相关的 Angular 指令都带有 `ng` 前缀，比如 `ngForm`、`ngModel` 和 `ngModelGroup`。

模板驱动的表单便捷、快速、简单，是很多基础型数据输入表单的最佳选择。

要学习如何构建模板驱动型表单，请参见 [表单](#) 页。

模板表达式 (Template Expression)

Angular 用来在 [数据绑定 \(Data Binding\)](#) 内求值的、类似 JavaScript 语法的表达式。

到 [模板语法](#) 一章中了解更多模板表达式的知识。

转译 (Transpile)

把用 JavaScript 的某种形态（比如 TypeScript）编写的程序转换成另一个形式的 JavaScript（例如 [ES5](#)）的过程。

TypeScript

一种支持了几乎所有 ECMAScript 2015 语言特性和一些未来版本可能有的特性（比如 [装饰器 \(Decorator\)](#)）的 JavaScript 语言。

TypeScript 还以它的可选类型系统而著称。该类型系统提供了编译期类型检查和强大的工具支持（比如“Intellisense”，自动代码补齐，重构和智能搜索等）。许多程序编辑器和开发环境都自带了 TypeScript 支持或通过插件提供支持。

TypeScript 是 Angular 的首选语言，当然，我们也欢迎你使用其它 JavaScript 语言，比如 [ES5](#)。

到 [TypeScript 官方网站](#) 了解更多知识。

视图 (View)

视图是屏幕中一小块，用来显示信息并回应用户动作，比如点击、移动鼠标和按键等。

Angular 在一个或多个 [指令 \(Directive\)](#) 的控制下渲染视图，尤其是 [组件 \(Component\)](#) 型指令及其 [模板 \(Template\)](#)。组件扮演着非常重要的角色，我们甚至经常会为了方便，直接用“视图”作为组件的代名词。

视图一般包含其它视图，在用户在应用程序中导航的时候，任何视图都可能被动态加载或卸载，这一般会在 [路由器 \(Router\)](#) 的控制下进行。

区域 (Zones)

区域是一种用来封装和截听 JavaScript 应用程序异步动作的机制。

浏览器中的 DOM 和 JavaScript 之间常会有一些数量有限的异步活动，比如 DOM 事件（如点击）、[承诺 \(Promise\)](#)、和通过 [XHR](#) 查询远程服务等。

区域能截听所有这些活动，并让“区域的客户”有机会在异步活动完成之前和之后采取行动。

Angular 会在一个 Zone 区域中运行应用程序，在这个区域中，它可以对异步事件做出反应，可以通过检查数据变更、利用 [数据绑定 \(Data Bindings\)](#) 来更新信息显示。

到 [Brian Ford 的视频](#) 学习更多关于区域的知识。

更新记录

最新文档更新历史记录。

文档变更日志

我们将持续不断的更新和改进 Angular 文档。本日志记录了近期最重要的变更。

与 Angular v.2.1.1 同步 (2016-10-21)

使用 Angular v.2.1.1 更新和测试所有文档和代码例子。

使用 npm 的 `@types` 包替换 `typings` (2016-10-20)

文档例子现在从 npm 的 `@types` 第三方库获取 TypeScript 类型信息，不再使用 `typings`。删除 `typings.json` 文件。

"[从 1.x 升级](#)" 指南反映了这个变化。`package.json` 安装 `@types/angular` 和一些 `@types/angular-...` 包来支持升级。它们在纯 Angular 2 开发中是不需要的。

"模板语法" 添加了双向数据绑定语法的解释 (2016-10-20)

展示了如何在自定义 Angular 组件中双向数据绑定，用基础 `[()` 重新解释 `[(ngModel)]`。

破坏性变化： `in-memory-web-api` (v.0.1.11) 以 esm umd 的形式发布 (2016-10-19)

这个变化支持 ES6 开发者，并与典型的 Angular 库看齐。它不会影响模块的 API，但是它改变了加载和导入它的方式。参见 [in-memory-web-api](#) 库的 [变更记录](#)。

"路由器" 预加载 语法和 :enter/:leave 动画 (2016-10-19)

路由器可以在应用启动 **之后** 和用户导航到惰性加载模块 **之前**，预先加载惰性模块，以增强性能。

新 `:enter` 和 `:leave` 语法，让动画更加自然。

与 Angular v.2.1.0 同步 (2016-10-12)

使用 Angular v.2.1.0 更新和测试所有文档和代码例子。

添加了新的“预编译 (AoT)” 烹饪书 (2016-10-11)

全新 [预编译 \(AoT\)](#) 烹饪书介绍了什么是 AoT 编译和为何你需要它。它以 **快速开始** 应用程序开始讲解，接着介绍了编译和构建 [英雄指南](#) 的更高级的注意事项。

与 Angular v.2.0.2 同步 (2016-10-6)

使用 Angular v.2.0.2 更新和测试所有文档和代码例子。

在“路由和导航”向导中添加 路由模块 (2016-10-5)

[Routing and Navigation](#) 现在在 [路由模块](#) 中设置路由配置。[路由模块](#) 替换之前的 [路由对象](#)，使用了 `ModuleWithProviders`。

路由与导航

所有使用路由的例子都使用 [路由模块](#)，相关内容也被更新。更新最多的是 [Angular 模块 \(NgModule\)](#) 章和 [Angular 模块常见问题](#) 烹饪书。

全新“国际化”烹饪书 (2016-09-30)

添加了新的 [国际化 \(i18n\)](#) 烹饪书，展示了如何使用 Angular 的“ i18n ”工具来讲模板文本翻译到多种语言。

重命名“ angular-in-memory-web-api ”包 (2016-09-27)

许多例子使用了 `angular-in-memory-web-api` 来模拟远程服务器。这个库在你拥有服务器之前的早期开发阶段也很有用。

这个包的名字从“ `angular2-in-memory-web-api` ”（仍然有效，但不再更新了）重新命名了。新的“ `angular-in-memory-web-api` ”有新的功能。[到 github 获得更多详情](#) .

“风格指南”中添加了 NgModules(2016-09-27)

[StyleGuide](#) 解释了我们为 Angular 模块（`NgModule`）而推荐的约定。

现在，封装桶不再那么重要，风格指南已经移除了它们。它们仍然很有价值，但是它们与 Angular 风格无关。我们同时对 [不推荐使用 `@Component.host` 属性](#) 的规则有所放宽。

moduleId : 到处添加 `module.id`(2016-09-25)

在所有使用 `templateUrl` 或者 `styleUrls` 来获取模板或样式的例子组件都被转换为 [相对模块](#) 的 URL 。我们添加了 `moduleId: module.id` 到它们的 `@Component` 元数据。

当应用像例子当前使用的方法一样 - 使用 SystemJS 加载模块时，本更新是 AoT 编译器的前提条件。

简化了“生命周期钩子”章 (2016-09-24)

[生命周期钩子](#) 章现在更加简短，并且对强调了 Angular 是以什么顺序来调用钩子方法的。

CHANGE LOG

An annotated history of recent documentation improvements.

The Angular documentation is a living document with continuous improvements. This log calls attention to recent significant changes.

Sync with Angular v.2.2.0 (2016-11-14)

Docs and code samples updated and tested with Angular v.2.2.0

UPDATE: NgUpgrade Guide for the AoT friendly `upgrade/static` module (2016-11-14)

The updated [NgUpgrade Guide](#) guide covers the new AoT friendly `upgrade/static` module released in v.2.2.0, which is the recommended facility for migrating from Angular 1 to Angular 2. The documentation for the version prior to v.2.2.0 has been removed.

ES6 described in "TypeScript to JavaScript" (2016-11-14)

The updated "TypeScript to JavaScript" cookbook now explains how to write apps in ES6/7 by translating the common idioms in the TypeScript documentation examples (and elsewhere on the web) to ES6/7 and ES5.

Sync with Angular v.2.1.1 (2016-10-21)

Docs and code samples updated and tested with Angular v.2.1.0

API 参考

TYPE: ALL

STATUS: ALL

 Filter

@angular/common

K APP_BASE_HREF	P AsyncPipe
C CommonModule	P CurrencyPipe
P DatePipe	P DecimalPipe
C HashLocationStrategy	P I18nPluralPipe
P I18nSelectPipe	P JsonPipe
C Location	I LocationChangeEvent
I LocationChangeListener	C LocationStrategy
P LowerCasePipe	D NgClass
D NgFor	D NgIf
C NgLocalization	D NgPlural
D NgPluralCase	D NgStyle
D NgSwitch	D NgSwitchCase
D NgSwitchDefault	D NgTemplateOutlet
C PathLocationStrategy	P PercentPipe
C PlatformLocation	P SlicePipe
P UppercasePipe	

@angular/common/testing

C MockLocationStrategy	C SpyLocation
------------------------	---------------

@angular/core

K	ANALYZE_FOR_ENTRY_COMPONENT_BOOTSTRAP_LISTENER
K	APP_ID
K	AUTO_STYLE
C	AfterContentInit
C	AfterViewInit
C	AnimationEntryMetadata
C	AnimationKeyframesSequenceMetadata
C	AnimationPlayer
C	AnimationStateDeclarationMetadata
C	AnimationStateTransitionMetadata
C	AnimationTransitionEvent
C	ApplicationInitStatus
C	ApplicationRef
K	COMPILER_OPTIONS
E	ChangeDetectionStrategy
F	Class
I	ClassProvider
C	Compiler
T	CompilerOptions
C	ComponentFactory
C	ComponentRef
@	ContentChildren
C	DebugNode
@	Directive
C	ElementRef
C	ErrorHandler
I	ExistingProvider
I	ForwardRefFn
@	Host
I	HostListener
K	APP_INITIALIZER
C	AfterContentChecked
C	AfterViewChecked
C	AnimationAnimateMetadata
C	AnimationGroupMetadata
C	AnimationMetadata
C	AnimationStateMetadata
C	AnimationStyleMetadata
C	AnimationWithStepsMetadata
C	ApplicationModule
I	Attribute
K	CUSTOM_ELEMENTS_SCHEMA
C	ChangeDetectorRef
I	ClassDefinition
C	CollectionChangeRecord
C	CompilerFactory
@	Component
C	ComponentFactoryResolver
@	ContentChild
C	DebugElement
C	DefaultIterableDiffer
C	DoCheck
C	EmbeddedViewRef
C	EventEmitter
I	FactoryProvider
I	GetTestability
I	HostBinding
@	Inject

@ Injectable	C Injector
I Input	I IterableDiffer
I IterableDifferFactory	C IterableDifers
C KeyValueChangeRecord	I KeyValueDiffer
I KeyValueDifferFactory	C KeyValueDifers
K LOCALE_ID	C ModuleWithComponentFactories
I ModuleWithProviders	K NO_ERRORS_SCHEMA
I NgModule	C NgModuleFactory
C NgModuleFactoryLoader	C NgModuleRef
C NgZone	C OnChanges
C OnDestroy	C OnInit
C OpaqueToken	@ Optional
I Output	K PACKAGE_ROOT_URL
K PLATFORM_INITIALIZER	I Pipe
I PipeTransform	C PlatformRef
T Provider	C Query
C QueryList	C ReflectiveInjector
C ReflectiveKey	C RenderComponentType
C Renderer	C ResolvedReflectiveFactory
I ResolvedReflectiveProvider	C RootRenderer
C Sanitizer	I SchemaMetadata
E SecurityContext	@ Self
C SimpleChange	I SimpleChanges
@ SkipSelf	C SystemJsNgModuleLoader
C SystemJsNgModuleLoaderCon	K TRANSLATIONS
K TRANSLATIONS_FORMAT	C TemplateRef
C Testability	C TestabilityRegistry
I TrackByFn	@ Type
I TypeProvider	I ValueProvider
@ ViewChild	@ ViewChildren
C ViewContainerRef	E ViewEncapsulation
C ViewRef	C WrappedValue

I WtfScopeFn	F animate
F asNativeElements	F assertPlatform
F createPlatform	F createPlatformFactory
F destroyPlatform	F enableProdMode
F forwardRef	F getDebugNode
F getModuleFactory	F getPlatform
F group	F isDevMode
F keyframes	K platformCore
F resolveForwardRef	F sequence
F setTestabilityGetter	F state
F style	F transition
F trigger	K wtfCreateScope
K wtfEndTimeRange	K wtfLeave
K wtfStartTimeRange	

@angular/core/testing

C ComponentFixture	K ComponentFixtureAutoDetect
K ComponentFixtureNoNgZone	C InjectSetupWrapper
T MetadataOverride	C TestBed
C TestComponentRenderer	T TestModuleMetadata
F async	F discardPeriodicTasks
F fakeAsync	F flushMicrotasks
F get TestBed	F inject
F resetFakeAsyncZone	F tick
F withModule	

@angular/forms

C AbstractControl	C AbstractControlDirective
C AbstractFormGroupDirective	I AsyncValidatorFn
D CheckboxControlValueAccesso	C ControlContainer

I ControlValueAccessor	D DefaultValueAccessor
I Form	C FormArray
D FormArrayName	C FormBuilder
C FormControl	D FormControlDirective
D FormControlName	C FormGroup
D FormGroupDirective	D FormGroupName
C FormsModule	D MaxLengthValidator
D MinLengthValidator	K NG_ASYNC_VALIDATORS
K NG_VALIDATORS	K NG_VALUE_ACCESSOR
C NgControl	D NgControlStatus
D NgControlStatusGroup	D NgForm
D NgModel	D NgModelGroup
D NgSelectOption	D PatternValidator
D RadioControlValueAccessor	C ReactiveFormsModule
D RequiredValidator	D SelectControlValueAccessor
D SelectMultipleControlValueAccessor	I Validator
I ValidatorFn	C Validators

@angular/http

C BaseRequestOptions	C BaseResponseOptions
C BrowserXhr	C Connection
C ConnectionBackend	C CookieXSRFStrategy
C Headers	C Http
C HttpModule	C JSONPBackend
C JSONPConnection	C Jsonp
C JsonpModule	C QueryEncoder
E ReadyState	C Request
E RequestMethod	C RequestOptions
I RequestOptionsArgs	C Response
E ResponseContentType	C RequestOptions

T ResponseOptionsArgs	E ResponseType
C URLSearchParams	C XHRBackend
C XHRConnection	C XSRFStrategy

@angular/http/testing

C MockBackend	C MockConnection
--	---

@angular/platform-browser

C AnimationDriver	C BrowserModule
C By	K DOCUMENT
C DomSanitizer	K EVENT_MANAGER_PLUGINS
C EventManager	K HAMMER_GESTURE_CONFIG
C HammerGestureConfig	C NgProbeToken
I SafeHtml	I SafeResourceUrl
I SafeScript	I SafeStyle
I SafeUrl	C Title
F disableDebugTools	F enableDebugTools
K platformBrowser	

@angular/platform-browser/testing

C BrowserTestingModule	K platformBrowserTesting
---	---

@angular/platform-browser-dynamic

K RESOURCE_CACHE_PROVIDERS	K platformBrowserDynamic
---	---

@angular/platform-browser-dynamic/testing

C BrowserDynamicTestingModule	K platformBrowserDynamicTesting
--	--

@angular/platform-server

C ServerModule	K platformDynamicServer
K platformServer	

@angular/platform-server/testing

C ServerTestingModule	K platformServerTesting
---	---

@angular/platform-webworker

C ClientMessageBroker	C ClientMessageBrokerFactory
C FnArg	C MessageBus
I MessageBusSink	I MessageBusSource
K PRIMITIVE	C ReceivedMessage
C ServiceMessageBroker	C ServiceMessageBrokerFactory
C UiArguments	K WORKER_APP_LOCATION_PROVIDERS
K WORKER_UI_LOCATION_PROVIDER	C WorkerAppModule
F bootstrapWorkerUi	K platformWorkerApp
K platformWorkerUi	

@angular/platform-webworker-dynamic

K platformWorkerAppDynamic
--

@angular/router

I ActivatedRoute	I ActivatedRouteSnapshot
I CanActivate	I CanActivateChild
I CanDeactivate	I CanLoad
T Data	C DefaultUrlSerializer
T Event	I ExtraOptions
T LoadChildren	T LoadChildrenCallback

C	NavigationCancel	C	NavigationEnd
C	NavigationError	I	NavigationExtras
C	NavigationStart	C	NoPreloading
K	PRIMARY_OUTLET	T	Params
C	PreloadAllModules	C	PreloadingStrategy
I	Resolve	T	ResolveData
I	Route	C	Router
D	RouterLink	D	RouterLinkActive
D	RouterLinkWithHref	C	RouterModule
D	RouterOutlet	C	RouterOutletMap
I	RouterState	I	RouterStateSnapshot
T	Routes	C	RoutesRecognized
C	UrlSegment	C	UrlSerializer
I	UrlTree	F	provideRoutes

@angular/router/testing

C	RouterTestingModule	C	SpyNgModuleFactoryLoader
F	setupTestingRouter		

@angular/upgrade

C	UpgradeAdapter	C	UpgradeAdapterRef
-------------------------------------	----------------	-------------------------------------	-------------------

教程: 英雄指南

英雄指南教程带我们一步步使用 TypeScript 创建 Angular 应用。

《英雄指南》目标概览

我们的终极计划是构建一个程序，来帮助招聘公司管理一群英雄。即使英雄们也需要找工作。

当然，在本教程中，我们只完成一小步。我们这次构建的应用会涉及很多特性：获得并显示英雄列表，编辑所选英雄的详情，并在英雄数据的多个视图之间建立导航。这些特性，在成熟的、数据驱动的应用中经常见到。

这篇《英雄指南》覆盖了 Angular 的核心原理。我们将使用内置指令来显示 / 隐藏元素，并且显示英雄数据的列表。我们将创建一个组件来显示英雄的详情，另一个组件则用来显示英雄列表。我们将对只读数据使用单向数据绑定。我们将添加一些可编辑字段，并通过双向数据绑定更新模型。我们将把组件上的方法绑定到用户事件上，比如按键和点击。我们将学习从主列表视图中选择一个英雄，然后在详情视图中编辑它。我们将通过管道对数据进行格式化。我们将创建一个共享服务来管理我们的英雄们。我们将使用路由在不同的视图及其组件之间进行导航。

完成本教程后，我们将学习足够的 Angular 核心技术，并确信 Angular 确实能做到我们需要它做的。我们将覆盖大量入门级知识，同时我们也会看到大量链接，指向更深入的章节。

运行 [live example](#)。

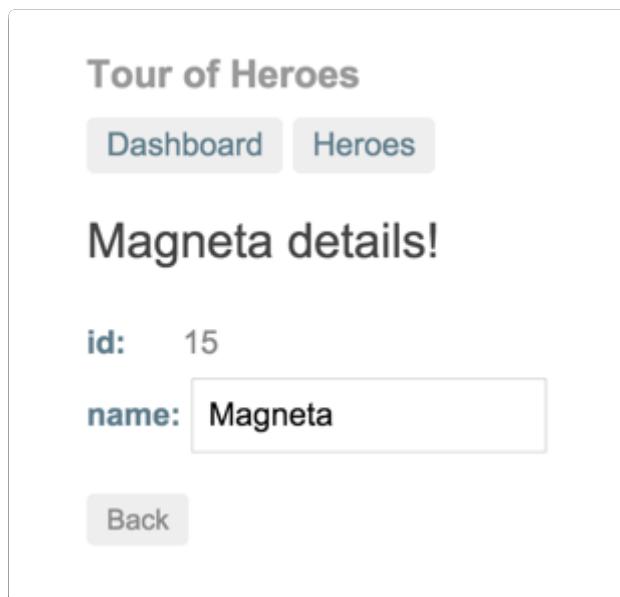
游戏的终点

下面是本教程关于界面的构想：开始是“ Dashboard (控制台) ”视图，来展示我们最勇敢的英雄。



控制台顶部中有两个链接：“ Dashboard (控制台) ”和“ Heroes (英雄列表) ”。我们将点击它们在“控制台”和“英雄列表”视图之间导航。

当我们点击控制台上名叫“ Magneta ”的英雄时，路由将把我们带到这个英雄的详情页，在这里，我们可以修改英雄的名字。



点击“ Back (后退) ”按钮将返回到“ Dashboard (控制台) ”。顶部的链接可以把我们带到任何一个主视图。如果我们点击“ Heroes (英雄列表) ”链接，应用将把我们带到“英雄”主列表视图。

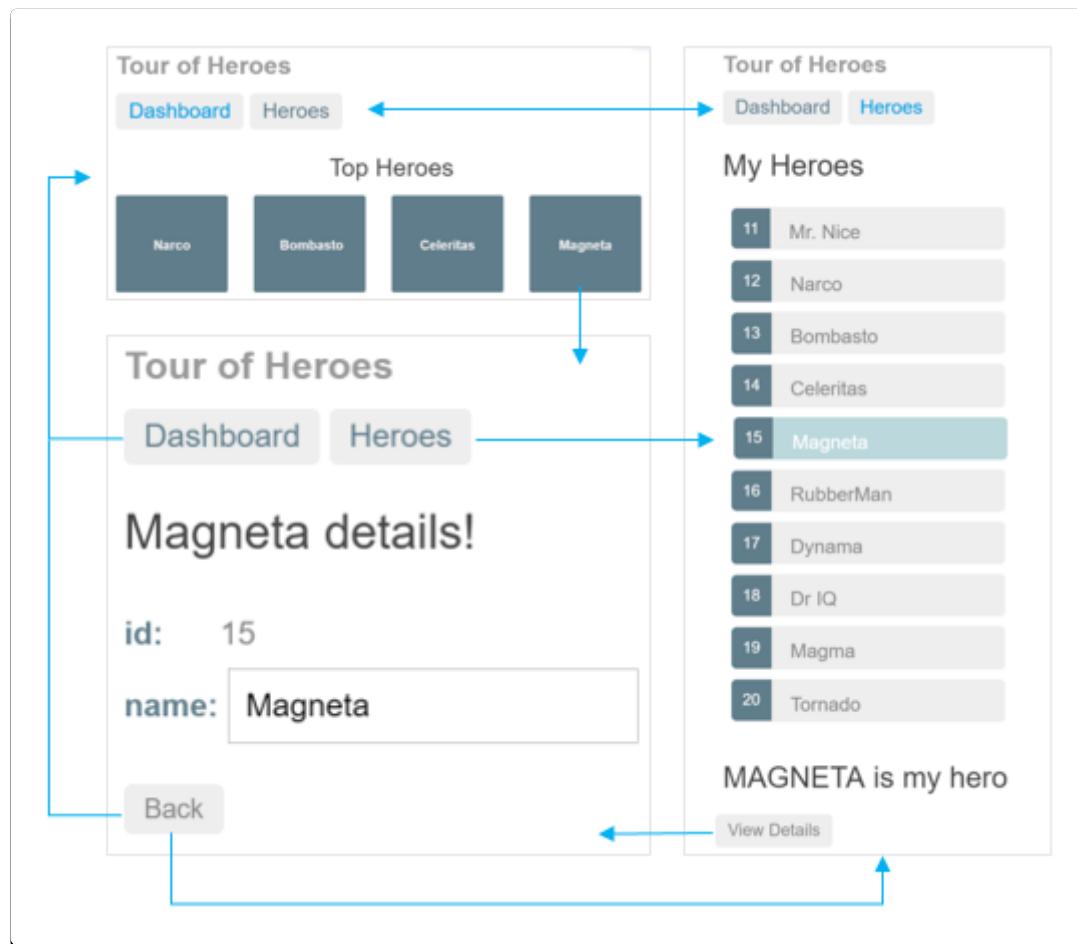
The screenshot shows a web application interface titled "Tour of Heroes". At the top, there are two tabs: "Dashboard" and "Heroes", with "Heroes" being the active one. Below the tabs, the heading "My Heroes" is centered. A vertical list of heroes is presented in a card-based format, each with a number and a name. The numbers are 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20. The names correspond to the numbers: Mr. Nice (11), Narco (12), Bombasto (13), Celeritas (14), Magneta (15), RubberMan (16), Dynama (17), Dr IQ (18), Magma (19), and Tornado (20). Each hero card has a dark blue header bar containing the number.

英雄编号	英雄名称
11	Mr. Nice
12	Narco
13	Bombasto
14	Celeritas
15	Magneta
16	RubberMan
17	Dynama
18	Dr IQ
19	Magma
20	Tornado

当我们点击另一位英雄时，一个只读的“微型详情视图”会显示在列表下方，以体现我们的选择。

我们可以点击“View Details（查看详情）”按钮进入所选英雄的编辑视图。

下面这张图汇总了我们所有可能的导航路径。



下图演示了我们应用中的所有操作。

The screenshot shows a web application titled "Tour of Heroes". At the top, there are two tabs: "Dashboard" and "Heroes", with "Heroes" being the active tab. Below the tabs, the title "Narco details!" is displayed. Underneath the title, there are two input fields: one for "id" containing "12" and another for "name" containing "Narco". A "Back" button is located at the bottom left of the form area.

接下来

让我们一起一步步构建出《英雄指南》。正如我们在无数应用遇到那样，每一步都由一个需求驱动。毕竟做任何事都要有个理由。

这一路上，我们将遇到很多 Angular 核心原理。

勇敢的前进吧！

下一步

[英雄编辑器](#)

英雄编辑器

我们构建一个简单的英雄编辑器

很久很久以前

每一个故事，都有一个起点。而我们的故事则开始于“[快速起步](#)”的结尾处。

运行这部分的 [在线例子](#)。

创建一个名为 `angular2-tour-of-heroes` 的文件夹，并且遵循“[快速起步](#)”中的步骤初始化它——环境准备、目录结构，并放进我们《英雄指南》的核心文件。

你还可以 [下载“快速起步”的源码](#)作为起步。

我们应该有如下目录结构：

```
angular-tour-of-heroes
  app
    app.component.ts
    app.module.ts
    main.ts
  node_modules ...
  index.html
  package.json
```

```
|- styles.css  
|- systemjs.config.js  
└ tsconfig.json
```

保持应用不断转译和运行

我们要启动 TypeScript 编译器，它会监视文件变更，并且启动开发服务器。我们只要敲：

```
npm start
```

这个命令会在监视模式下运行编译器，启动开发服务器，在浏览器中启动我们的应用，并在我们构建《英雄指南》的时候让应用得以持续运行。

显示我们的英雄

我们要在应用中显示英雄数据

我们来为 `AppComponent` 添加两个属性：`title` 属性表示应用的名字，而 `hero` 属性表示一个名叫“Windstorm”的英雄。

app.component.ts (AppComponent类)

```
export class AppComponent {  
  title = 'Tour of Heroes';  
  hero = 'Windstorm';  
}
```

现在，我们为这些新属性建立数据绑定，以更新 `@Component` 装饰器中指定的模板

```
template: '<h1>{{title}}</h1><h2>{{hero}} details!</h2>'
```

保存后，浏览器应该会自动刷新，并显示我们的标题和英雄。

这里的“双大括号”会告诉应用：从组件中读取 `title` 和 `hero` 属性，并且渲染它们。这就是单向数据绑定的“插值表达式”形式。

要了解插值表达式的更多知识，参阅 “[显示数据](#)”一章。

Hero 对象

此时此刻，我们的英雄还只有一个名字。显然，它 / 她应该有更多属性。让我们把 `hero` 从一个字符串字面量换成一个类。

创建一个 `Hero` 类，它具有 `id` 和 `name` 属性。现在，把下列代码放在 `app.component.ts` 的顶部，仅次于 `import` 语句。

app.component.ts (Hero类)

```
export class Hero {  
  id: number;  
  name: string;  
}
```

现在，有了一个 `Hero` 类，我们就要把组件 `hero` 属性的类型换成 `Hero` 了。然后以 `1` 为 `id`、以“Windstorm”为名字，初始化它。

app.component.ts (hero property)

```
hero: Hero = {  
  id: 1,  
  name: 'windstorm'  
};
```

我们把 `hero` 从一个字符串换成了对象，所以也得更新模板中的绑定表达式，来引用 `hero` 的 `name` 属性。

```
template: '<h1>{{title}}</h1><h2>{{hero.name}} details!</h2>'
```

浏览器自动刷新，并继续显示这位英雄的名字。

添加更多的 HTML

能显示名字虽然不错，但我们还想看到这位英雄的所有属性。我们将添加一个 `<div>` 来显示英雄的 `id` 属性，用另一个 `<div>` 来显示英雄的 `name` 属性。

```
template: '<h1>{{title}}</h1><h2>{{hero.name}} details!</h2><div>
<label>id: </label>{{hero.id}}</div><div><label>name: </label>
{{hero.name}}</div>'
```

啊哦！我们的模板字符串已经太长了。我们最好小心点，免得在模板中出现拼写错误。

多行模板字符串

我们可以通过字符串加法来制作更可读的模板，但这样仍然太难看了——难于阅读，容易拼错。这样不行！我们要借助 ES2015 和 TypeScript 提供的模板字符串来保持清爽。

把模板的双引号改成反引号，并且让 `<h1>`，`<h2>` 和 `<div>` 标签各占一行。

app.component.ts (AppComponent的 模板)

```
1.   template: `
2.     <h1>{{title}}</h1>
3.     <h2>{{hero.name}} details!</h2>
4.     <div><label>id: </label>{{hero.id}}</div>
5.     <div><label>name: </label>{{hero.name}}</div>
6.   `
```

编辑我们的英雄

我们想在一个文本框中编辑英雄的名字。

把英雄的名字从单纯的 `<label>` 重构成 `<label>` 和 `<input>` 元素的组合，就像下面这样：

app.component.ts (input元素)

```
1. template: ` 
2.   <h1>{{title}}</h1>
3.   <h2>{{hero.name}} details!</h2>
4.   <div><label>id: </label>{{hero.id}}</div>
5.   <div>
6.     <label>name: </label>
7.     <input value="{{hero.name}}" placeholder="name">
8.   </div>
9. `
```

在浏览器中，我们看到英雄的名字显示成一个 `<input>` 文本框。但看起来还是有点不太对劲。当修改名字时，我们的改动并没有反映到 `<h2>` 中。使用单向数据绑定，我们没法实现所期望的这种行为。

双向绑定

我们的期望是：在 `<input>` 中显示英雄的名字，修改它，并且在所有绑定到英雄名字的地方看到这些修改。简而言之，我们需要双向数据绑定。

在我们让 **表单输入** 支持双向数据绑定之前，我们得先导入 `FormsModule` 模块。只要把它添加到 `NgModule` 装饰器的 `imports` 数组中就可以了，该数组是应用中用到的外部模块列表。这样我们就引入了表单包，其中包含了 `ngModel`。

app.module.ts (FormsModule import)

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
```

```
4.  
5. import { AppComponent } from './app.component';  
6.  
7. @NgModule({  
8.   imports: [  
9.     BrowserModule,  
10.    FormsModule  
11.   ],  
12.   declarations: [  
13.     AppComponent  
14.   ],  
15.   bootstrap: [ AppComponent ]  
16. })  
17. export class AppModule {}
```

要学习关于 `FormsModule` 和 `ngModel` 的更多知识，参见 [表单](#) 和 [模板语法](#) 两章

接下来更新模板，加入用于双向绑定的内置指令 `ngModel`。

把 `<input>` 替换为下列 HTML

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

浏览器刷新。又见到我们的英雄了。我们可以编辑英雄的名字，也能看到这个改动立刻体现在 `<h2>` 中。

我们已经走过的路

我们来盘点一下已经构建完成的部分。

- 我们的《英雄指南》使用双大括号插值表达式（单向数据绑定的一种形式）来显示应用的标题和 `Hero` 对象的属性。
- 我们使用 ES2015 的模板字符串写了一个多行模板，使我们的模板更具可读性。
- 为了同时显示和修改英雄的名字，我们还使用了内置的 `ngModel` 指令，往 `<input>` 元素上添加了双向数据绑定。
- `ngModel` 指令将这些修改传播到每一个对 `hero.name` 的其它绑定。

运行这部分的 [在线例子](#)。

完整的 `app.component.ts` 是这样的：

app.component.ts

```
1. import { Component } from '@angular/core';
2.
3. export class Hero {
4.   id: number;
5.   name: string;
6. }
7.
8. @Component({
9.   selector: 'my-app',
10.  template: `
11.    <h1>{{title}}</h1>
12.    <h2>{{hero.name}} details!</h2>
13.    <div><label>id: </label>{{hero.id}}</div>
14.    <div>
15.      <label>name: </label>
16.      <input [(ngModel)]="hero.name" placeholder="name">
17.    </div>
18.    `
19.  })
20. export class AppComponent {
21.   title = 'Tour of Heroes';
22.   hero: Hero = {
23.     id: 1,
24.     name: 'Windstorm'
25.   };
}
```

26. }

前方的路

我们的《英雄指南》只显示了一个英雄，而我们真正要显示的是一个英雄列表。我们还希望允许用户选择一个英雄，并且显示它 / 她的详情。我们将在 [教程的下一章](#) 中学习如何获取列表数据，并将把它们绑定到模板，以及允许用户选择其中一个英雄。

下一步

[主从结构](#)

主从结构

我们构建一个主从结构的页面，用于展现英雄列表

显示多个英雄

我们需要管理多个英雄。让我们来扩展《英雄指南》应用，让它显示一个英雄列表，并允许用户选择一个英雄，查看该英雄的详细信息。

运行这部分的 [在线例子](#)。

我们来盘点一下显示英雄列表都需要些什么。首先，需要一份英雄列表数据。还要把这些英雄显示到一个视图的模板中，所以，我们需要用某种途径来做到这一点。

延续上一步教程

在继续《英雄指南》的第二部分之前，先来检查一下，完成 [第一部分](#) 之后，你是否已经有了如下目录结构。如果没有，你得先回到第一部分，看看错过了哪里。

```
angular-tour-of-heroes
  app
    app.component.ts
    app.module.ts
    main.ts
  node_modules ...
  index.html
  package.json
```

```
|- styles.css  
|- systemjs.config.js  
└ tsconfig.json
```

让应用代码保持转译和运行

我们要启动 TypeScript 编译器，它会监视文件变更，并启动开发服务器。我们只要敲：

```
npm start
```

这个命令会在我们构建《英雄指南》的时候让应用得以持续运行。

显示我们的英雄

创建英雄

我们先创建一个由十位英雄组成的数组。

app.component.ts (hero array)

```
1.  const HEROES: Hero[] = [  
2.    { id: 11, name: 'Mr. Nice' },  
3.    { id: 12, name: 'Narco' },  
4.    { id: 13, name: 'Bombasto' },  
5.    { id: 14, name: 'Celeritas' },  
6.    { id: 15, name: 'Magneta' },  
7.    { id: 16, name: 'RubberMan' },  
8.    { id: 17, name: 'Dynamite' },  
9.    { id: 18, name: 'Dr IQ' },  
10.   { id: 19, name: 'Magma' },  
11.   { id: 20, name: 'Tornado' }  
12. ];
```

`HEROES` 变量是一个由 `Hero` 类的实例构成的数组，我们在第一部分定义过它。我们当然希望从一个 Web 服务中获取这个英雄列表，但别急，我们得把步子迈得小一点——先用一组 Mock(模拟) 出来的英雄。

导出英雄们

我们在 `AppComponent` 上创建一个公共属性，用来暴露这些英雄，以供绑定。

app.component.ts (hero array property)

```
heroes = HEROES;
```

我们并不需要明确定义 `heroes` 属性的数据类型，TypeScript 能从 `HEROES` 数组中推断出来。

我们已经把英雄列表定义在了这个组件类中。但显然，我们最终还是得从一个数据服务中获取这些英雄。正因如此，一开始就应该把英雄数据隔离到一个类中来实现。

在一个模板中显示英雄

我们的组件有了 `heroes` 属性，我们再到模板中创建一个无序列表来显示它们。我们将在标题和英雄详情之间，插入下面这段 HTML 代码。

app.component.ts (heroes template)

```
1.  <h2>My Heroes</h2>
2.  <ul class="heroes">
3.    <li>
4.      <!-- each hero goes here -->
5.    </li>
6.  </ul>
```

现在，我们有了一个模板。接下来，就用英雄们的数据来填充它。

通过 ngFor 来显示英雄列表

我们想要把组件中的 `heroes` 数组绑定到模板中，迭代并逐个显示它们。这下，我们就得借 Angular 的帮助来完成它了。我们来一步步实现它！

首先，修改 `` 标签，往上添加内置指令：`*ngFor`。

app.component.ts (ngFor)

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

ngFor 的前导星号 (*) 是此语法的重要组成部分。

ngFor 的 * 前缀表示 `` 及其子元素组成了一个主控模板。

ngFor 指令在 `AppComponent.heroes` 属性返回的 `heroes` 数组上迭代，并输出此模板的实例。

引号中赋值给 ngFor 的那段文本表示“从 `heroes` 数组中取出每个英雄，存入一个局部的 `hero` 变量，并让它在相应的模板实例中可用”。

`hero` 前的 `let` 关键字表示 `hero` 是一个模板输入变量。在模板中，我们可以引用这个变量来访问一位英雄的属性。

要学习更多关于 ngFor 和模板输入变量的知识，参见 [显示数据](#) 和 [模板语法](#) 两章。

现在，我们在 `` 标签中插入一些内容，以便使用模板变量 `hero` 来显示英雄的属性。

app.component.ts (ngFor模板)

```
<li *ngFor="let hero of heroes">
  <span class="badge">{{hero.id}}</span> {{hero.name}}
</li>
```

当浏览器刷新时，我们就看到了英雄列表。

给我们的英雄们“美容”

我们的英雄列表看起来实在是稀松平常。但当用户的鼠标划过英雄或选中了一个英雄时，我们得让它 / 她看起来醒目一点。

要想给我们的组件添加一些样式，请把 `@Component` 装饰器的 `styles` 属性设置为下列 CSS 类：

app.component.ts (styles)

```
styles: [
  .selected {
    background-color: #CFD8DC !important;
    color: white;
  }
  .heroes {
    margin: 0 0 2em 0;
    list-style-type: none;
    padding: 0;
    width: 15em;
  }
  .heroes li {
    cursor: pointer;
    position: relative;
    left: 0;
    background-color: #EEE;
    margin: .5em;
    padding: .3em 0;
    height: 1.6em;
    border-radius: 4px;
  }
  .heroes li.selected:hover {
```

```
background-color: #BBD8DC !important;
color: white;
}

.heroes li:hover {
color: #607D8B;
background-color: #DDD;
left: .1em;
}

.heroes .text {
position: relative;
top: -3px;
}

.heroes .badge {
display: inline-block;
font-size: small;
color: white;
padding: 0.8em 0.7em 0 0.7em;
background-color: #607D8B;
line-height: 1em;
position: relative;
left: -1px;
top: -4px;
height: 1.8em;
margin-right: .8em;
border-radius: 4px 0 0 4px;
}

`]
```

注意，我们又使用了反引号语法来书写多行字符串。

这里有很多种样式！我们可以像上面那样把它们内联在组件中，或者把样式移到单独的文件中——这样能让编写组件变得更容易。我们会后面的章节中使用独立样式文件，现在我们先不管它。

当我们为一个组件指定样式时，它们的作用域将仅限于该组件。上面的例子中，这些样式只会作用于 `AppComponent` 组件，而不会“泄露”到外部 HTML 中。

用于显示英雄们的这个模板看起来像这样：

app.component.ts (styled heroes)

```

1.  <h2>My Heroes</h2>
2.  <ul class="heroes">
3.    <li *ngFor="let hero of heroes">
4.      <span class="badge">{{hero.id}}</span> {{hero.name}}
5.    </li>
6.  </ul>

```

选择英雄

在我们的应用中，已经有了英雄列表以及一个单独的英雄。但列表和单独的英雄之间还没有任何关联。我们希望用户在列表中选中一个英雄，然后让这个被选中的英雄出现在详情视图中。这种 UI 布局模式，通常被称为“主从结构”。在这个例子中，主视图是英雄列表，从视图则是被选中的英雄。

我们通过组件中的一个 `selectedHero` 属性来连接主从视图，它被绑定到了 `click` 事件上。

click 事件

我们往 `` 元素上插入一句 Angular 事件绑定代码，绑定到它的 `click` 事件。

app.component.ts (template excerpt)

```

1.  <li *ngFor="let hero of heroes" (click)="onSelect(hero)">
2.    <span class="badge">{{hero.id}}</span> {{hero.name}}
3.  </li>

```

事件绑定详解

```
(click)="onSelect(hero)"
```

圆括号表示 `` 元素上的 `click` 事件就是我们要绑定的目标。等号右边的表达式调用 `AppComponent` 的 `onSelect()` 方法，并把模板输入变量 `hero` 作为参数传进去。它和

我们前面在 `ngFor` 中定义的 `hero` 变量是同一个。

要学习关于事件绑定的更多知识，参见：[用户输入](#) 和 [模板语法](#) 两章。

添加 click 事件处理器

我们的事件绑定引用了 `onSelect` 方法，但它还不存在。我们现在就把它添加到组件上。

这个方法该做什么？它应该把组件中被选中的英雄设置为用户刚刚点击的那个。

我们的组件还没有用来表示“当前选中的英雄”的变量，我们就从这一步开始。

导出“当前选中的英雄”

在 `AppComponent` 中，我们不再需要一个固定的 `hero` 属性。那就直接把它改为 `selectedHero` 属性。

app.component.ts (selectedHero)

```
selectedHero: Hero;
```

我们已决定：在用户选取之前，不会默认选择任何英雄，所以，我们不用像 `hero` 一样去初始化 `selectedHero` 变量。

现在，[添加一个 `onSelect` 方法](#)，用于将用户点击的英雄赋给 `selectedHero` 属性。

app.component.ts (onSelect)

```
1.  onSelect(hero: Hero): void {
2.      this.selectedHero = hero;
3.  }
```

我们将把所选英雄的详细信息显示在模板中。目前，它仍然引用的是以前的 `hero` 属性。我们这就修改模板，让它绑定到新的 `selectedHero` 属性。

app.component.ts (template excerpt)

```
1.  <h2>{{selectedHero.name}} details!</h2>
2.  <div><label>id: </label>{{selectedHero.id}}</div>
3.  <div>
4.    <label>name: </label>
5.    <input [(ngModel)]="selectedHero.name" placeholder="name"/>
6.  </div>
```

使用 `ngIf` 隐藏空的详情

当应用刚加载时，我们会看到一个英雄列表，但还没有任何英雄被选中。`selectedHero` 属性是 `undefined`。因此，我们会看到浏览器控制台中出现下列错误：

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

别忘了我们要在模板中显示的是 `selectedHero.name`。显然，这个 `name` 属性是不存在的，因为 `selectedHero` 本身还是 `undefined` 呢。

要处理这个问题，我们可以先让英雄详情不要出现在 DOM 中，直到有英雄被选中。

我们把模板中的“英雄详情”内容区用放在一个 `<div>` 中。然后往上添加一个 `ngIf` 内置指令，然后把 `ngIf` 的值设置为组件的 `selectedHero` 属性。

app.component.ts (ngIf)

```
1.  <div *ngIf="selectedHero">
2.    <h2>{{selectedHero.name}} details!</h2>
3.    <div><label>id: </label>{{selectedHero.id}}</div>
4.    <div>
5.      <label>name: </label>
6.      <input [(ngModel)]="selectedHero.name" placeholder="name"/>
```

```
7.      </div>
8.    </div>
```

记住，`ngIf` 前面的星号（*）是语法中的重要组成部分。

当没有 `selectedHero` 时，`ngIf` 指令会从 DOM 中移除表示英雄详情的这段 HTML。没有了表示英雄详情的元素，也就不用担心绑定问题。

当用户选取了一个英雄，`selectedHero` 变成了“真”值，于是 `ngIf` 把“英雄详情”加回 DOM 中，并且计算它所嵌套的各种绑定。

`ngIf` 和 `ngFor` 被称为“结构型指令”，因为它们可以修改 DOM 的部分结构。换句话说，它们让 Angular 在 DOM 中显示内容的方式结构化了。

要了解更多 `ngIf`，`ngFor` 和其它结构型指令，请参阅：[结构型指令和模板语法章节](#)。

浏览器刷新了，我们看到了一个英雄列表，但是还没有选中的英雄详情。当 `selectedHero` 是 `undefined` 时，`ngIf` 会保证英雄详情不出现在 DOM 中。当我们从列表中点击一个英雄时，选中的英雄被显示在英雄详情里。正如我们所预期的那样。

给所选英雄添加样式

我们在下面的详情区看到了选中的英雄，但是我们还是没法在上面的列表区快速的找到这位英雄。通过把 CSS 类 `selected` 添加到主列表的 `` 元素上，我们可以解决这个问题。比如，当我们在列表区选中了 Magneta 时，我们可以通过设置一个轻微的背景色来让它略显突出。



我们将通过一个在 `class` 属性上的绑定，来把 `selected` 类添加到模板上。我们把这个绑定表达式设置为 `selectedHero` 和 `hero` 的比较结果，

关键是 CSS 类的名字：`selected`。当两位英雄一致时，它为 `true`，否则为 `false`。也就是说：“**当两位英雄匹配时，应用上 `selected` 类，否则不应用**”。

app.component.ts (setting the CSS class)

```
[class.selected]="hero === selectedHero"
```

注意，模板中的 `class.selected` 是括在一对方括号中的。这就是“属性绑定”的语法，实现从数据源 (`hero === selectedHero` 表达式) 到 `class` 属性的单向数据流动。

app.component.ts (styling each hero)

```
<li *ngFor="let hero of heroes"
    [class.selected]="hero === selectedHero"
    (click)="onSelect(hero)"
    <span class="badge">{{hero.id}}</span> {{hero.name}}
</li>
```

学习关于 [属性绑定](#) 的更多知识，参见“模板语法”一章。

浏览器重新加载了我们的应用。我们选中英雄 Magneta，于是它通过背景色的变化被清晰的标记了出来。

Tour of Heroes

My Heroes

11 Mr. Nice

12 Narco

13 Bombasto

14 Celeritas

15 Magneta

16 RubberMan

17 Dynama

18 Dr IQ

19 Magma

20 Tornado

我们选择了另一个英雄，于是色标也跟着移到了这位英雄上。

完整的 `app.component.ts` 文件如下：

app.component.ts

```
1. import { Component } from '@angular/core';
2.
3. export class Hero {
4.   id: number;
5.   name: string;
6. }
7.
8. const HEROES: Hero[] = [
9.   { id: 11, name: 'Mr. Nice' },
10.  { id: 12, name: 'Narco' },
11.  { id: 13, name: 'Bombasto' },
```

```
12.     { id: 14, name: 'Celeritas' },
13.     { id: 15, name: 'Magneta' },
14.     { id: 16, name: 'RubberMan' },
15.     { id: 17, name: 'Dynamite' },
16.     { id: 18, name: 'Dr IQ' },
17.     { id: 19, name: 'Magma' },
18.     { id: 20, name: 'Tornado' }
19. ];
20.
21. @Component({
22.   selector: 'my-app',
23.   template: `
24.     <h1>{{title}}</h1>
25.     <h2>My Heroes</h2>
26.     <ul class="heroes">
27.       <li *ngFor="let hero of heroes"
28.           [class.selected]="hero === selectedHero"
29.           (click)="onSelect(hero)">
30.             <span class="badge">{{hero.id}}</span> {{hero.name}}
31.           </li>
32.         </ul>
33.         <div *ngIf="selectedHero">
34.           <h2>{{selectedHero.name}} details!</h2>
35.           <div><label>id: </label>{{selectedHero.id}}</div>
36.           <div>
37.             <label>name: </label>
38.             <input [(ngModel)]="selectedHero.name" placeholder="name"/>
39.           </div>
40.         </div>
41.       `,
42.       styles: [
43.         .selected {
44.           background-color: #CFD8DC !important;
45.           color: white;
46.         }
47.         .heroes {
48.           margin: 0 0 2em 0;
49.           list-style-type: none;
50.           padding: 0;
51.           width: 15em;
52.         }
53.         .heroes li {
54.           cursor: pointer;
```

```
55.         position: relative;
56.         left: 0;
57.         background-color: #EEE;
58.         margin: .5em;
59.         padding: .3em 0;
60.         height: 1.6em;
61.         border-radius: 4px;
62.     }
63.     .heroes li.selected:hover {
64.         background-color: #BBD8DC !important;
65.         color: white;
66.     }
67.     .heroes li:hover {
68.         color: #607D8B;
69.         background-color: #DDD;
70.         left: .1em;
71.     }
72.     .heroes .text {
73.         position: relative;
74.         top: -3px;
75.     }
76.     .heroes .badge {
77.         display: inline-block;
78.         font-size: small;
79.         color: white;
80.         padding: 0.8em 0.7em 0 0.7em;
81.         background-color: #607D8B;
82.         line-height: 1em;
83.         position: relative;
84.         left: -1px;
85.         top: -4px;
86.         height: 1.8em;
87.         margin-right: .8em;
88.         border-radius: 4px 0 0 4px;
89.     }
90.     `]
91. })
92. export class AppComponent {
93.     title = 'Tour of Heroes';
94.     heroes = HEROES;
95.     selectedHero: Hero;
96.
97.     onSelect(hero: Hero): void {
```

```
98.     this.selectedHero = hero;
99.
100. }
```

已走的路

在本章中，我们达成了这些：

- 我们的《英雄指南》现在显示一个可选英雄的列表
- 我们添加了选择英雄的能力，并且会显示这个英雄的详情
- 我们学会了如何在组件模板中使用内置的 `ngIf` 和 `ngFor` 指令

运行这部分的 [在线例子](#)。

前方的路

我们的《英雄指南》长大了，但还远远不够完善。我们显然不能把整个应用都放进一个组件中。我们需要把它拆分成一系列子组件，然后教它们协同工作——就像我们将在[下一章](#)学到的那样。

下一步

[多个组件](#)

多个组件

我们把主从结构的页面重构为多个组件

我们的应用正在成长中。现在又有新的用例：重复使用组件，传递数据给组件并创建更多可复用的资源。我们来把英雄详情从英雄列表中拆分出来，让这个英雄详情组件可以被复用。

运行这部分的 [在线例子](#)。

延续上一步教程

在继续《英雄指南》之前，先来检查一下，你是否已经有了如下目录结构。如果没有，你得先回上一章，看看错过了哪里。

```
angular-tour-of-heroes
```

```
  app
    ├── app.component.ts
    ├── app.module.ts
    └── main.ts
  node_modules ...
  index.html
  package.json
  styles.css
  systemjs.config.js
  tsconfig.json
```

让应用代码保持转译和运行

我们要启动 TypeScript 编译器，它会监视文件变更，并启动开发服务器。只要敲：

```
npm start
```

这个命令会在我们构建《英雄指南》的时候让应用得以持续运行。

制作英雄详情组件

我们的英雄列表和英雄详情目前位于同一个文件的同一个组件中。现在它们还很小，但很快它们都会长大。我们将来肯定会收到新需求：针对这一个，却不能影响另一个。然而，每一个更改都会给这两个组件带来风险，并且带来双倍的测试负担，却没有任何好处。如果我们需要在应用的其它地方复用英雄详情组件，英雄列表组件也会跟着混进去。

我们当前的组件违反了 [单一职责原则](#)。虽然这只是一个教程，但我们还是得坚持做正确的事——况且，做正确的事这么容易，我们何乐而不为呢？别忘了，我们正在学习的就是如何构建真正的 Angular 应用。

我们来把英雄详情拆分成一个独立的组件。

拆分英雄详情组件

在 `app` 目录下添加一个名叫 `hero-detail.component.ts` 的文件，并且创建 `HeroDetailComponent`。代码如下：

app/hero-detail.component.ts (initial version)

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

@Component({
  selector: 'my-hero-detail',
})
export class HeroDetailComponent {
```

命名约定

我们希望一眼就能看出哪些类是组件，以及哪些文件包含组件。

你会注意到，在名叫 `app.component.ts` 的文件中有一个 `AppComponent` 组件，在名叫 `hero-detail.component.ts` 的文件中有一个 `HeroDetailComponent` 组件。

我们的所有组件名都以 `Component` 结尾。所有组件的文件名都以 `.component` 结尾。

这里我们使用小写 **中线命名法** (也叫 **烤串命名法**) 拼写文件名，所以不用担心它在服务器或者版本控制系统中出现大小写问题。

一开始，我们要先从 Angular 中导入 `Component` 和 `Input` 装饰器，因为马上就会用到它们。

我们使用 `@Component` 装饰器创建元数据。在元数据中，我们指定选择器的名字，用以标记此组件的元素。然后，我们导出这个组件类，以便其它组件可以使用它。

做完这些，我们把它导入 `AppComponent` 组件，并创建相应的 `<my-hero-detail>` 元素。

英雄详情模板

目前，`AppComponent` 的 **英雄列表** 和 **英雄详情** 视图被组合在同一个模板中。让我们从 `AppComponent` 中 **剪切** 出 **英雄详情** 的内容，并且粘贴到 `HeroDetailComponent` 组件的 `template` 属性中。

以前我们绑定到了 `AppComponent` 的 `selectedHero.name` 属性中。

`HeroDetailComponent` 组件将会有一个 `hero` 属性，而不是 `selectedHero` 属性。所以，我们要把模板中的所有 `selectedHero` 替换为 `hero`。只改这些就够了。最终结果如下所示：

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

```
template: `

<div *ngIf="hero">
  <h2>{{hero.name}} details!</h2>
  <div><label>id: </label>{{hero.id}}</div>
  <div>
    <label>name: </label>
    <input [(ngModel)]="hero.name" placeholder="name"/>
  </div>
</div>

`
```

现在，我们的英雄详情布局只存在于 `HeroDetailComponent` 组件中了。

添加 HERO 属性

我们这就添加刚刚所说的 `hero` 属性到组件类中。

```
hero: Hero;
```

啊哦！我们定义的 `hero` 属性是 `Hero` 类型的，但是我们的 `Hero` 类还在 `app.component.ts` 文件中。我们有了两个组件，它们都有各自的文件，并且都需要引用 `Hero` 类。

要解决这个问题，我们也得从 `app.component.ts` 文件中把 `Hero` 类移到属于它自己的 `hero.ts` 文件中。

app/hero.ts

```
export class Hero {
  id: number;
  name: string;
}
```

我们得从 `hero.ts` 中导出 `Hero` 类，因为我们要从那些组件文件中引用它。在 `app.component.ts` 和 `hero-detail.component.ts` 的顶部添加下列 import 语句：

```
import { Hero } from './hero';
```

HERO 是一个输入属性

还得告诉 `HeroDetailComponent` 显示哪个英雄。谁告诉它呢？自然是父组件 `AppComponent` 了！

`AppComponent` 确实知道该显示哪个英雄——用户从列表中选中的那个。而这个英雄就是 `selectedHero` 属性的值。

我们马上升级 `AppComponent` 的模板，以便把该组件的 `selectedHero` 属性绑定到 `HeroDetailComponent` 组件的 `hero` 属性上。绑定看起来 可能 是这样的：

```
<my-hero-detail [hero]="selectedHero"></my-hero-detail>
```

注意，在等号 (=) 左边方括号中的这个 `hero` 是属性绑定的 目标。

Angular 希望我们把 **目标属性** 定义成组件的 **输入属性**，否则，Angular 会拒绝绑定，并且抛出一个错误。

我们在 [这里](#) 详细解释了输入属性，以及为什么 **目标属性** 需要“显式定义”这样的特殊待遇，而 **源属性** 却不需要。

我们有几种方式把 `hero` 声明成 **输入属性**。这里我们采用 **首选** 的方式：使用我们前面导入的 `@Input` 装饰器，为 `hero` 属性加上注解。

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

要了解 `@Input()` 装饰器的更多知识，参见 [属性型指令](#) 一章。

更新 AppModule

回到 `AppModule`，该应用的根模块，我们要教它使用 `HeroDetailComponent` 组件。

我们先导入 `HeroDetailComponent` 组件，好让我们可以引用它。

```
import { HeroDetailComponent } from './hero-detail.component';
```

接下来，添加 `HeroDetailComponent` 到 `NgModule` 装饰器中的 `declarations` 数组。这个数组包含了所有属于本应用模块的，由我们亲自创建的组件、管道和指令。

```
1.  @NgModule({
2.    imports: [
3.      BrowserModule,
4.      FormsModule
5.    ],
6.    declarations: [
7.      AppComponent,
8.      HeroDetailComponent
9.    ],
10.   bootstrap: [ AppComponent ]
11. )
12. export class AppModule { }
```

更新 AppComponent

找到我们刚刚从模板中移除 **英雄详情** 的地方，放上用来表示 `HeroDetailComponent` 组件的 HTML 标签。

```
<my-hero-detail></my-hero-detail>
```

`my-hero-detail` 是我们在 `HeroDetailComponent` 元数据中的 `selector` 属性所指定的名字。

这两个组件目前还不能协同工作，直到我们把 `AppComponent` 组件的 `selectedHero` 属性和 `HeroDetailComponent` 组件的 `hero` 属性绑定在一起，就像这样：

```
<my-hero-detail [hero]="selectedHero"></my-hero-detail>
```

`AppComponent` 的模板是这样的：

app.component.ts (template)

```
template: `
  <h1>{{title}}</h1>
  <h2>My Heroes</h2>
  <ul class="heroes">
    <li *ngFor="let hero of heroes"
        [class.selected]="hero === selectedHero"
        (click)="onSelect(hero)">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </li>
  </ul>
  <my-hero-detail [hero]="selectedHero"></my-hero-detail>
`,
```

感谢数据绑定机制，`HeroDetailComponent` 组件应该能从 `AppComponent` 组件中获取英雄数据，并且在列表的下方显示英雄的详情了。每当用户选中一个新的英雄时，详情信息应该随之更新。

搞定！

当在浏览器中查看应用时，我们看到了英雄列表。当选中一个英雄时，还能看到所选英雄的详情。

值得关注的进步是：我们可以在应用中的任何地方使用这个 `HeroDetailComponent` 组件来显示英雄详情了。

我们创建了第一个可复用组件！

回顾应用结构

来验证下吧，在本章中，经过这些漂亮的重构，我们应该得到了下列结构：

```
angular-tour-of-heroes
├── app
│   ├── app.component.ts
│   ├── app.module.ts
│   ├── hero.ts
│   └── hero-detail.component.ts
├── main.ts
└── node_modules ...
    ...
index.html
package.json
tsconfig.json
```

下面就是我们在本章讨论过的源码文件：

```
1. import { Component, Input } from '@angular/core';
2.
3. import { Hero } from './hero';
4.
5. @Component({
6.   selector: 'my-hero-detail',
```

```
7.   template: ``
8.     <div *ngIf="hero">
9.       <h2>{{hero.name}} details!</h2>
10.      <div><label>id: </label>{{hero.id}}</div>
11.      <div>
12.        <label>name: </label>
13.        <input [(ngModel)]="hero.name" placeholder="name"/>
14.      </div>
15.    </div>
16.  `
17.  })
18. export class HeroDetailComponent {
19.   @Input()
20.   hero: Hero;
21. }
```

走过的路

来盘点一下我们已经构建完的部分。

- 我们创建了一个可复用组件
- 我们学会了如何让一个组件接收输入
- 我们学会了在 Angular 模块中声明该应用所需的指令——只要把这些指令列在 `NgModule` 装饰器的 `declarations` 数组中就可以了。
- 我们学会了把父组件绑定到子组件。

运行这部分的 [在线例子](#)。

前方的路

通过抽取共享组件，我们的《英雄指南》变得更有复用性了。

但在 `AppComponent` 中，我们仍然使用着 mock 数据。显然，这种方式不能“可持续发展”。我们要把数据访问逻辑抽取到一个独立的服务中，并在需要数据的组件之间共享。

在 [下一步](#)，我们将学习如何创建服务。

下一步

[服务](#)

服务

我们创建一个可复用的服务来调用英雄的数据

服务

《英雄指南》继续前行。接下来，我们准备添加更多的组件。

将来会有更多的组件访问英雄数据，但我们不想一遍又一遍的复制粘贴同样的代码。我们的替代方案是，创建一个单一的、可复用的数据服务，然后学着把它注入到那些想用它的组件中去。

我们将重构数据访问代码，把它隔离到一个独立的服务中去，让组件尽可能保持精简，专注于为视图提供支持。在这种方式下，借助 mock 服务来对组件进行单元测试也会更容易。

因为数据服务通常都是异步的，我们将在本章创建一个基于 **承诺（Promise）** 的数据服务。

运行这部分的 [在线例子](#)。

延续上一步教程

在继续《英雄指南》之前，先来检查一下，你是否已经有了如下目录结构。如果没有，你得先回上一章，看看错过了哪里。

```
angular-tour-of-heroes
  └─ app
```

```
|- app.component.ts  
|- app.module.ts  
|- hero.ts  
|- hero-detail.component.ts  
|- main.ts  
|- node_modules ...  
-- index.html  
-- package.json  
-- styles.css  
-- systemjs.config.js  
-- tsconfig.json
```

让应用代码保持转译和运行

打开 terminal/console 窗口，启动 TypeScript 编译器，它会监视文件变更，并启动开发服务器。只要敲：

```
npm start
```

当我们继续构建《英雄指南》时，应用会自动运行和更新。

创建英雄服务

客户向我们描绘了本应用更大的目标。它们说，想要在不同的页面中用多种方式显示英雄。现在我们已经能从列表中选择一个英雄了，但这还不够。很快，我们将添加一个仪表盘来显示表现最好的英雄，并且创建一个独立视图来编辑英雄的详情。所有这些视图都需要英雄的数据。

目前，`AppComponent` 显示的是 mock 英雄数据。我们可改进的地方至少有两个：首先，定义英雄的数据不该是组件的任务。其次，想把这份英雄列表的数据共享给其它组件和视图可不那么容易。

我们可以把获取英雄数据的任务重构为一个单独的服务，它将提供英雄数据，并且把这个服务在所有需要英雄数据的组件之间共享。

创建 HeroService

在 `app` 目录下创建一个名叫 `hero.service.ts` 的文件。

我们遵循的文件命名约定是：服务名称的小写形式（基本名），加上 `.service` 后缀。如果服务名称包含多个单词，我们就把基本名部分写成中线形式（`dash-case`，也被称作烤串形式 `kebab-case`）。比如，`SpecialSuperHeroService` 服务应该被定义在 `special-super-hero.service.ts` 文件中。

我们把这个类命名为 `HeroService`，并且导出它，以供别人使用。

app/hero.service.ts (starting point)

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

@Injectable()
export class HeroService {
```

可注入的服务

注意，我们引入了 Angular 的 `Injectable` 函数，并通过 `@Injectable()` 装饰器使用这个函数。

不要忘了写圆括号！ 如果忘了写，就会导致一个很难诊断的错误。

当 TypeScript 看到 `@Injectable()` 装饰器时，就会记下本服务的元数据。如果 Angular 需要往这个服务中注入其它依赖，就会使用这些元数据。

此刻，`HeroService` 还没有任何依赖，但我们还是得加上这个装饰器。作为一项最佳实践，无论是出于提高统一性还是减少变更的目的，都应该从一开始就加上`@Injectable()` 装饰器。

获取英雄

添加一个名叫`getHeros` 的桩方法。

app/hero.service.ts (getHeroes stub)

```
@Injectable()
export class HeroService {
  getHeroes(): void {} // stub
}
```

在这个实现上暂停一下，我们先来讲一个重点。

数据使用者并不知道本服务会如何获取数据。我们的`HeroService` 服务可以从任何地方获取英雄的数据。它可以从网络服务器获取，可以从浏览器的局部存储区获取，也可以是 mock 的数据源。

这就是从组件中移除数据访问代码的美妙之处。这样我们可以随时改变数据访问的实现方式，而无需对使用英雄的组件作任何改动。

Mock 英雄数据

我们曾在`AppComponent` 组件中写过 mock 版的英雄数据。它不该在那里，但也不该在 **这里**！我们得把 mock 数据移到它自己的文件中去。

从`app.component.ts` 文件中剪切`HEROS` 数组，并把它粘贴到`app` 目录下一个名叫`mock-heroes.ts` 的文件中。我们还要把`import {Hero}...` 语句拷贝过来，因为我们的英雄数组用到了`Hero` 类。

app/mock-heroes.ts

```
1. import { Hero } from './hero';
2.
```

```
3.  export const HEROES: Hero[] = [
4.    {id: 11, name: 'Mr. Nice'},
5.    {id: 12, name: 'Narco'},
6.    {id: 13, name: 'Bombasto'},
7.    {id: 14, name: 'Celeritas'},
8.    {id: 15, name: 'Magneta'},
9.    {id: 16, name: 'RubberMan'},
10.   {id: 17, name: 'Dynamite'},
11.   {id: 18, name: 'Dr IQ'},
12.   {id: 19, name: 'Magma'},
13.   {id: 20, name: 'Tornado'}
14. ];
```

我们导出了 `HEROES` 常量，以便在其它地方导入它——比如 `HeroService` 服务。

同时，回到刚剪切出 `HEROES` 数组的 `app.component.ts` 文件，我们留下了一个尚未初始化的 `heroes` 属性：

app/app.component.ts (heroes property)

```
heroes: Hero[];
```

返回模拟的英雄数据

回到 `HeroService`，我们导入 `HEROES` 常量，并在 `getHeroes` 方法中返回它。我们的 `HeroService` 服务现在看起来是这样：

app/hero.service.ts

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

import { Hero } from './hero';
import { HEROES } from './mock-heroes';

@Injectable()
export class HeroService {
  getHeroes(): Hero[] {
```

```
    return HEROES;
}
}
```

使用 HeroService 服务

我们可以在多个组件中使用 HeroService 服务了，先从 AppComponent 开始。

通常，我们会从导入要用的东西开始，比如 `HeroService`。

```
import { HeroService } from './hero.service';
```

导入这个服务让我们可以在代码中引用它。`AppComponent` 该如何在运行中获得一个具体的 `HeroService` 实例呢？

我们要自己 new 出这个 HeroService 吗？不！

固然，我们可以使用 `new` 关键字来创建 `HeroService` 的实例，就像这样：

```
heroService = new HeroService(); // don't do this
```

但这不是个好主意，有很多理由，比如：

- 我们的组件将不得不弄清楚该如何创建 `HeroService`。如果有一天我们修改了 `HeroService` 的构造函数，我们不得不找出创建过此服务的每一处代码，并修改它。而给代码打补丁的行为容易导致错误，并增加了测试的负担。
- 我们每次使用 `new` 都会创建一个新的服务实例。如果这个服务需要缓存英雄列表，并把这个缓存共享给别人呢？怎么办？没办法，做不到。
- 我们把 `AppComponent` 锁死在 `HeroService` 的一个特定实现中。我们很难在别的场景中把它换成别的实现。比如，能离线操作吗？能在测试时使用不同的模拟版本吗？这可不容易。

- 如果……如果……嘿！这下我们可有得忙了！

有办法了，真的！这个办法真是简单得不可思议，它能解决这些问题，你就再也没有犯错误的借口了。

注入 HeroService

用这两行代码代替用 `new` 时的一行：

1. 我们添加了一个构造函数，同时还定义了一个私有属性。
2. 我们添加了组件的 `providers` 元数据

下面就是这个构造函数：

```
app/app.component.ts (constructor)  
  
constructor(private heroService: HeroService) { }
```

构造函数自己什么也不用做，它在参数中定义了一个私有的 `heroService` 属性，并把它标记为注入 `HeroService` 的靶点。

现在，Angular 将会知道，当它创建 `AppComponent` 实例时，需要先提供一个 `HeroService` 的实例。

要了解关于依赖注入的更多知识，请参见 [依赖注入](#) 一章。

注入器 还不知道该如何创建 `HeroService`。如果现在运行我们的代码，Angular 就会失败，并报错：

```
EXCEPTION: No provider for HeroService! (AppComponent -> HeroService)  
(异常: 没有HeroService的提供商! (AppComponent -> HeroService))
```

我们还得注册一个 `HeroService` 提供商，来告诉注入器如何创建 `HeroService`。要做到这一点，我们应该在 `@Component` 组件的元数据底部添加 `providers` 数组属性如下：

```
providers: [HeroService]
```

`providers` 数组告诉 Angular，当它创建新的 `AppComponent` 组件时，也要创建一个 `HeroService` 的新实例。`AppComponent` 会使用那个服务来获取英雄列表，在它组件树中的每一个子组件也同样如此。

AppComponent 中的 getHeroes

我们已经获得了此服务，并把它存入了私有变量 `heroService` 中。我们这就开始使用它。

停下来想一想。我们可以在同一行内调用此服务并获得数据。

```
this.heroes = this.heroService.getHeroes();
```

在真实的世界中，我们并不需要把一行代码包装成一个专门的方法，但无论如何，我们在演示代码中先这么写：

```
getHeroes(): void {
  this.heroes = this.heroService.getHeroes();
}
```

ngOnInit 生命周期钩子

毫无疑问，`AppComponent` 应该获取英雄数据并显示它。我们该在哪里调用 `getHeroes` 方法呢？在构造函数中吗？不！

多年的经验和惨痛的教训教育我们，应该把复杂的逻辑扔到构造函数外面去，特别是那些需要从服务器获取数据的逻辑更是如此。

构造函数是为了简单的初始化工作而设计的，比如把构造函数的参数赋值给属性。它的负担不应该过于沉重。我们应该能在测试中创建一个组件，而不用担心它会做实际的工作——比如和服务器通讯，直到我们主动要求它做这些。

如果不在构造函数中，总得有地方调用 `getHeroes` 吧。

这也不难。只要我们实现了 Angular 的 **ngOnInit 生命周期钩子**，Angular 就会主动调用这个钩子。Angular 提供了一些接口，用来介入组件生命周期的几个关键时间点：刚创建时、每次变化时，以及最终被销毁时。

每个接口都有唯一的一个方法。只要组件实现了这个方法，Angular 就会在合适的时机调用它。

要了解关于生命周期钩子的更多知识，请参见 [生命周期钩子](#) 一章。

这是 `OnInit` 接口的基本轮廓：

```
app/app.component.ts (ngOnInit stub)

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

export class AppComponent implements OnInit {
  ngOnInit(): void {
  }
}
```

我们写下带有初始化逻辑的 `ngOnInit` 方法，然后留给 Angular，供其在正确的时机调用。在这个例子中，我们通过调用 `getHeroes` 来完成初始化。

```
ngOnInit(): void {
  this.getHeroes();
}
```

}

我们的应用将会像期望的那样运行，显示英雄列表，并且在我们点击英雄的名字时，显示英雄的详情。

我们就快完成了，但还有点事情不太对劲。

异步服务与承诺

我们的 `HeroService` 立即返回一个模拟的英雄列表，它的 `getHeroes` 函数签名是同步的。

```
this.heroes = this.heroService.getHeroes();
```

请求英雄数据，并直接在结果中返回它。

将来，我们会从远端服务器上获取英雄数据。我们还没调用 `http`，但在未来的章节中我们会希望这么做。

那时候，我们不得不等待服务器返回，并且在等待时，我们没法阻塞 UI 响应，即使我们想这么做（也不应这么做）也做不到，因为浏览器不会阻塞。

我们不得不使用一些异步技术，而这将改变 `getHeroes` 方法的签名。

我们将使用 **承诺**。

`HeroService` 会生成一个承诺

承诺 就是……好吧，它就是一个承诺——在有了结果时，它承诺会回调我们。我们请求一个异步服务去做点什么，并且给它一个回调函数。它会去做（在某个地方），一旦完成，它就会调用我们的回调函数，并通过参数把工作结果或者错误信息传给我们。

这里只是粗浅的说说，要了解更多 ES2015 Promise 的信息，请参见 [这里](#) 或在 Web 上搜索其它学习资源。

把 `HeroService` 的 `getHeroes` 方法改写为返回承诺的形式：

app/hero.service.ts (excerpt)

```
getHeroes(): Promise<Hero[]> {
  return Promise.resolve(HEROES);
}
```

我们继续使用模拟数据。我们通过返回一个 **立即解决的承诺** 的方式，模拟了一个超快、零延迟的超级服务器。

基于承诺的行动

回到 `AppComponent` 和它的 `getHeroes` 方法，我们看到它看起来还是这样的：

app/app.component.ts (getHeroes - old)

```
getHeroes(): void {
  this.heroes = this.heroService.getHeroes();
}
```

在修改了 `HeroService` 之后，我们还要把 `this.heroes` 替换为一个承诺，而不再是一个英雄数组。

我们得修改这个实现，把它变成 **基于承诺** 的，并在承诺的事情被解决时再行动。一旦承诺的事情被成功解决，我们就会显示英雄数据。

我们把回调函数作为参数传给承诺对象的 `then` 函数：

app/app.component.ts (getHeroes - revised)

```
getHeroes(): void {
  this.heroService.getHeroes().then(heroes => this.heroes = heroes);
}
```

回调中所用的 [ES2015 箭头函数](#) 比等价的函数表达式更加简洁，能优雅的处理 `this` 指针。

在回调函数中，我们把由服务返回的英雄数组赋值给组件的 `heroes` 属性。是的，这就搞定了。

我们的程序仍在运行，仍在显示英雄列表，在选择英雄时，仍然会把它 / 她显示在详情页面中。

查看附件中的“[慢一点](#)”一节，来了解在较差的网络连接中这个应用会是什么样的。

回顾本应用的结构

再检查下，经历了本章的所有重构之后，我们应该有了下列文件结构：

```
angular-tour-of-heroes
  app
    app.component.ts
    app.module.ts
    hero.ts
    hero-detail.component.ts
```

```
hero.service.ts  
main.ts  
mock-heroes.ts  
node_modules ...  
index.html  
package.json  
— styles.css  
— systemjs.config.js  
— tsconfig.json
```

这就是我们在本章讨论过的这些源码文件：

```
1. import { Injectable } from '@angular/core';  
2.  
3. import { Hero } from './hero';  
4. import { HEROES } from './mock-heroes';  
5.  
6. @Injectable()  
7. export class HeroService {  
8.   getHeroes(): Promise<Hero[]> {  
9.     return Promise.resolve(HEROES);  
10.  }  
11. }
```

走过的路

来盘点一下我们已经构建完的部分。

- 我们创建了一个能被多个组件共享的服务类。
- 我们使用 `ngOnInit` 生命周期钩子，以便在 `AppComponent` 激活时获取英雄数据。
- 我们把 `HeroService` 定义为 `AppComponent` 的一个提供商。

- 我们创建了一个模拟的英雄数据，并把它导入我们的服务中。
- 我们把服务设计为返回承诺，组件从承诺中获取数据。

运行这部分的 [在线例子](#)。

前方的路

通过使用共享组件和服务，我们的《英雄指南》更有复用性了。我们还要创建一个仪表盘，要添加在视图间路由的菜单链接，还要在模板中格式化数据。随着我们应用的进化，我们还会学到如何进行设计，让它更易于扩展和维护。

我们将在 [下一章](#) 学习 Angular 组件路由，以及在视图间导航的知识。

附件：慢一点

我们可以模拟慢速连接。

导入 `Hero` 类，并且在 `HeroService` 中添加如下的 `getHeroesSlowly` 方法：

app/hero.service.ts (getHeroesSlowly)

```
getHeroesSlowly(): Promise<Hero[]> {
  return new Promise<Hero[]>(resolve =>
    setTimeout(resolve, 2000) // delay 2 seconds
    .then(() => this.getHeroes());
}
```

像 `getHeroes` 一样，它也返回一个承诺。但是，这个承诺会在提供模拟数据之前等待两秒钟。

回到 `AppComponent`，用 `heroService.getHeroesSlowly` 替换掉 `heroService.getHeroes`，并观察本应用的行为。

下一步

路由

路由

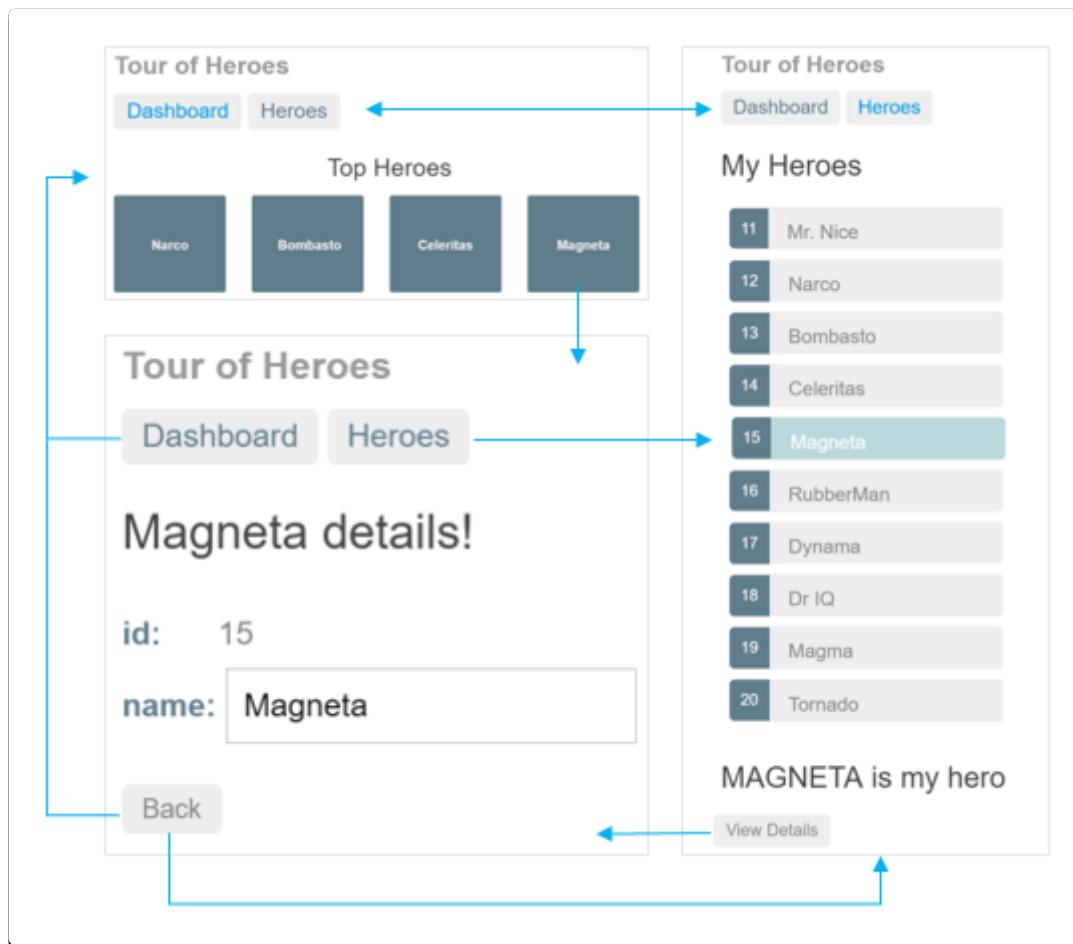
我们添加 Angular 路由，并且学习在视图之间导航

应用中的路由

我们收到了一份《英雄指南》的新需求：

- 添加一个 **控制台** 视图。
- 在 **英雄列表** 和 **控制台** 视图之间导航。
- 无论在哪个视图中点击一个英雄，都会导航到该英雄的详情页。
- 在邮件中点击一个 **深链接**，会直接打开一个特定英雄的详情视图。

完成时，用户就能像这样浏览一个应用：



我们将把 Angular 路由器 加入应用中，以满足这些需求。（译注：硬件领域中的路由器是用来帮你找到另一台网络设备的，而这里的路由器用于帮你找到一个组件）

[路由和导航](#) 章节介绍了更多关于路由的细节。

运行这部分的 [在线例子](#)。

注意看浏览器地址栏中的 URL 变化，点击右上角的蓝色 'X' 按钮，弹出预览窗口。

[Launch the preview in a separate window](#)

我们在哪

在继续《英雄指南》之前，先来检查一下，在添加了英雄服务和英雄详情组件之后，你是否已经有了如下目录结构。如果没有，你得先回上一章，再照做一遍。

```
angular-tour-of-heroes
└── app
    ├── app.component.ts
    ├── app.module.ts
    └── hero.service.ts
    ├── hero.ts
    ├── hero-detail.component.ts
    ├── main.ts
    ├── mock-heroes.ts
    ├── node_modules ...
    ├── index.html
    ├── package.json
    ├── styles.css
    ├── systemjs.config.js
    └── tsconfig.json
```

让应用代码保持转译和运行

打开 terminal/console 窗口，运行下列命令启动 TypeScript 编译器，它会监视文件变更，并启动开发服务器：

```
npm start
```

我们继续构建《英雄指南》，应用也会保持运行并自动更新。

行动计划

下面是我们的计划：

- 把 `AppComponent` 变成应用程序的“壳”，它只处理导航，
- 把现在由 `AppComponent` 关注的 **英雄们** 移到一个独立的 `HeroesComponent` 中，
- 添加路由
- 添加一个新的 `DashboardComponent` 组件
- 把 **控制台** 加入导航结构中。

路由 是导航的另一个名字。 **路由器** 就是从一个视图导航到另一个视图的机制。

拆分 `AppComponent`

现在的应用会加载 `AppComponent` 组件，并且立刻显示出英雄列表。

我们修改后的应用将提供一个壳，它会选择 **控制台** 和 **英雄列表** 视图之一，然后默认显示它。

`AppComponent` 组件应该只处理导航。 我们来把 **英雄列表** 的显示职责，从 `AppComponent` 移到 `HeroesComponent` 组件中。

HeroesComponent

`AppComponent` 的职责已经被移交给 `HeroesComponent` 了。 与其把 `AppComponent` 中所有的东西都搬过去，不如索性把它改名为 `HeroesComponent`，然后单独创建一个新的 `AppComponent` 壳。

改名的步骤如下：

- 把 `app.component.ts` 文件改名为 `heroes.component.ts`
- 把 `AppComponent` 类改名为 `HeroesComponent`

- 把 `my-app` 选择器改名为 `my-heroes`

app/heroes.component.ts (showing renamings only)

```
@Component({
  selector: 'my-heroes',
})
export class HeroesComponent implements OnInit {
```

创建 AppComponent

新的 `AppComponent` 将成为应用的“壳”。它将在顶部放一些导航链接，并且把我们要导航到的页面放在下面的显示区中。

这些起始步骤是：

- 添加支持性的 `import` 语句。
- 创建一个名叫 `app.component.ts` 的新文件。
- 定义一个导出的 `AppComponent` 类。
- 在类的上方添加 `@Component` 元数据装饰器，装饰器中带有 `my-app` 选择器。
- 将下面的项目从 `HeroesComponent` 移到 `AppComponent`：
 - `title` 类属性
 - 在模板中添加一个 `<h1>` 标签，包裹着到 `title` 属性的绑定。
- 在模板中添加 `<my-heroes>` 标签，以便我们仍能看到英雄列表。
- 添加 `HeroesComponent` 组件到根模块的 `declarations` 数组中，以便 Angular 能认识 `<my-heroes>` 标签。

- 添加 `HeroService` 到根模块的 `providers` 数组中，因为我们的每一个视图都需要它。
- 从 `HeroesComponent` 的 `providers` 数组中移除 `HeroService``，因为它被提到模块了。
- 导入 `AppComponent`。

我们的第一个草稿版就像这样：

```

1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
5.   template: `
6.     <h1>{{title}}</h1>
7.     <my-heroes></my-heroes>
8.   `
9. })
10. export class AppComponent {
11.   title = 'Tour of Heroes';
12. }
```

从 `HEROESCOMPONENT` 的 `PROVIDERS` 中移除 `HEROSERVICE`

回到 `HeroesComponent`，并从 `providers` 数组中 **移除 `HeroService`**。我们要把它从 `HeroesComponent` **提升** 到根 `NgModule` 中。我们可不希望在应用的两个不同层次上存在它的 **两个副本**。

应用仍然在运行，并显示着英雄列表。我们把 `AppComponent` 重构成了一个新的 `AppComponent` 和 `HeroesComponent`，它们工作得很好！我们毫发无损的完成了这次重构。

添加路由

我们已准备好开始下一步。与其自动显示英雄列表，我们更希望在用户点击按钮之后才显示它。换句话说，我们希望“导航”到英雄列表。

我们需要 Angular 的 **路由器**。

Angular 路由器是一个可选的外部 Angular NgModule，名叫 `RouterModule`。路由器包含了多种服务 (`RouterModule`)、多种指令 (`RouterOutlet`, `RouterLink`, `RouterLinkActive`)、和一套配置 (`Routes`)。我们将先配置路由。

设置 base 标签

打开 `index.html` 并且在 `<head>` 区的顶部添加 `<base href="/">` 语句。

index.html (base-href)

```
<head>
<base href="/">
```

BASE REF 是不可或缺的

查看 [路由器](#) 一章的 **base href** 部分，了解为何如此。

配置路由

本应用还没有路由。我们来为应用的路由新建一个配置。

路由 告诉路由器，当用户点击链接或者把 URL 粘贴到浏览器地址栏时，应该显示哪个视图。

我们的第一个路由是指向 `HeroesComponent` 的。

app/app.module.ts (heroes route)

```
import { RouterModule } from '@angular/router';

RouterModule.forRoot([
{
```

```

    path: 'heroes',
    component: HeroesComponent
  }
])

```

这个 `RouteConfig` 是一个 **路由定义** 的数组。此刻我们只有一个路由定义，但别急，后面还会添加更多。

“路由定义”包括几个部分：

- **path:** 路由器会用它来匹配路由中指定的路径和浏览器地址栏中的当前路径，如 `/heroes`。
- **component:** 导航到此路由时，路由器需要创建的组件，如 `HeroesComponent`。

到 [路由](#) 章节学习更多关于使用路由库来定义路由的知识。

让路由器可用

我们设置了初始路由配置。现在把它添加到 `AppModule` 里。添加配置好的 `RouterModule` 到 `AppModule` 的 `imports` 数组中。

app/app.module.ts (app routing)

```

import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { RouterModule }  from '@angular/router';

import { AppComponent }   from './app.component';
import { HeroDetailComponent } from './hero-detail.component';
import { HeroesComponent } from './heroes.component';
import { HeroService }    from './hero.service';

@NgModule({
  ...
})

```

```

imports: [
  BrowserModule,
  FormsModule,
  RouterModule.forRoot([
    {
      path: 'heroes',
      component: HeroesComponent
    }
  ])
],
declarations: [
  AppComponent,
  HeroDetailComponent,
  HeroesComponent
],
providers: [
  HeroService
],
bootstrap: [ AppComponent ]
})
export class AppModule {
}

```

这里使用了 `forRoot` 方法，因为我们在应用根部提供配置的路由器。`forRoot` 方法提供了路由需要的路由服务提供商和指令，并基于当前浏览器 URL 初始化导航。

路由插座 (Outlet)

如果我们把路径 `/heroes` 粘贴到浏览器的地址栏，路由器会匹配到 '`'Heroes'`' 路由，并显示 `HeroesComponent` 组件。但问题是，该把它显示在哪呢？

我们必须 **告诉它位置**，所以我们把 `<router-outlet>` 标签添加到模板的底部。

`RouterOutlet` 是 `RouterModule` 提供的 指令之一。当我们在应用中导航时，路由器就把激活的组件显示在 `<router-outlet>` 里面。

路由器链接

我们当然不会真让用户往地址栏中粘贴路由的 URL，而应该在模板中的什么地方添加一个 a 链接标签。点击时，就会导航到 `HeroesComponent` 组件。

修改过的模板是这样的：

app/app.component.ts (template-v2)

```
template:
<h1>{{title}}</h1>
<a routerLink="/heroes">Heroes</a>
<router-outlet></router-outlet>
```

注意，a 标签中的 `[routerLink]` 绑定。我们把 `RouterLink` 指令（`ROUTER_DIRECTIVES` 中的另一个指令）绑定到一个字符串。它将告诉路由器，当用户点击这个链接时，应该导航到哪里。

由于这个链接不是动态的，我们只要用 **一次性绑定** 的方式绑定到路由的路径（`path`）就行了。回来看路由配置表，我们清楚的看到，这个路径—— `'/heroes'` 就是指向 `HeroesComponent` 的那个路由的路径。

参阅 [路由](#) 章学习更多动态路由器链接和 **链接参数数组** 的知识。

刷新浏览器。我们只看到了应用标题和英雄链接。英雄列表到哪里去了？

浏览器的地址栏显示的是 `/`。而到 `HeroesComponent` 的路由中的路径是 `/heroes`，不是 `/`。我们没有任何路由能匹配当前的路径 `/`，所以，自然没啥可显示的。接下来，我们就修复这个问题。

我们点击“英雄列表 (Heroes) ”导航链接，浏览器地址栏更新为 /heroes，并且看到了英雄列表。我们终于导航过去了！

在这个阶段， AppComponent 看起来是这样的：

```
app/app.component.ts (v2)

1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
5.   template: `
6.     <h1>{{title}}</h1>
7.     <a routerLink="/heroes">Heroes</a>
8.     <router-outlet></router-outlet>
9.
10.`)
11. export class AppComponent {
12.   title = 'Tour of Heroes';
13. }
```

AppComponent 现在加上了路由器，并能显示路由到的视图了。因此，为了把它从其它种类的组件中区分出来，我们称这类组件为 **路由器组件**。

添加一个 控制台

当我们有多个视图的时候，路由才有意义。所以我们需要另一个视图。

先创建一个 DashboardComponent 的占位符，让我们可以导航到它或从它导航出来。

```
app/dashboard.component.ts (v1)

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

@Component({
  selector: 'my-dashboard',
  template: '<h3>My Dashboard</h3>'
```

```
})  
export class DashboardComponent { }
```

我们先不实现它，稍后，我们再回来，给这个组件一些实际用途。

配置控制台路由

回到 `app.module.ts`，我们得教它如何导航到这个控制台。

先导入 `DashboardComponent` 类，然后把下列路由的定义添加到 `Routes` 数组中。

app/app.module.ts (Dashboard route)

```
{  
  path: 'dashboard',  
  component: DashboardComponent  
},
```

还得把 `DashboardComponent` 添加到根模块的 `declarations` 数组中。

app/app.module.ts (dashboard)

```
declarations: [  
  AppComponent,  
  DashboardComponent,  
  HeroDetailComponent,  
  HeroesComponent  
,
```

REDIRECT TO

我们希望在应用启动的时候就显示控制台，而且我们希望在浏览器的地址栏看到一个好看的 URL，比如 `/dashboard`。记住，浏览器启动时，在地址栏中使用的路径是 `/`。我们可以使用一个重定向路由来达到目的。

可以使用重定向路由来实现它。添加下面的内容到路由定义的数组中：

app/app.module.ts (redirect)

```
{  
  path: '',  
  redirectTo: '/dashboard',  
  pathMatch: 'full'  
},
```

要学习关于 **重定向** 的更多知识，参见 [路由与导航](#) 一章。

添加导航到模版中

最后，在模板上添加一个到控制台的导航链接，就放在 **英雄 (Heroes)** 链接的上方。

app/app.component.ts (template-v3)

```
template: `  
  <h1>{{title}}</h1>  
  <nav>  
    <a routerLink="/dashboard">Dashboard</a>  
    <a routerLink="/heroes">Heroes</a>  
  </nav>  
  <router-outlet></router-outlet>
```

我们在 `<nav>` 标签中放了两个链接。它们现在还没有作用，但稍后，当我们对这些链接添加样式时，会显得比较方便。

刷新浏览器。应用显示出了控制台，并可以在控制台和英雄列表之间导航了。

控制台上的顶级英雄

我们想让控制台更有趣，比如：一眼就能看到四个顶级英雄。

把元数据中的 `template` 属性替换为 `templateUrl` 属性，它将指向一个新的模板文件。

设置 `moduleId` 属性到 `module.id`，相对模块加载 `templateUrl`。

app/dashboard.component.ts (metadata)

```
@Component({
  moduleId: module.id,
  selector: 'my-dashboard',
  templateUrl: 'dashboard.component.html',
})
```

使用下列内容创建文件：

app/dashboard.component.html

```
1. <h3>Top Heroes</h3>
2. <div class="grid grid-pad">
3.   <div *ngFor="let hero of heroes" class="col-1-4">
4.     <div class="module hero">
5.       <h4>{{hero.name}}</h4>
6.     </div>
7.   </div>
8. </div>
```

我们又一次使用 `*ngFor` 来在英雄列表上迭代，并显示它们的名字。还添加了一个额外的 `<div>` 元素，来帮助稍后的美化工作。

共享 HeroService

我们想要复用 `HeroService` 来存放组件的 `heroes` 数组。

回忆一下，在前面的章节中，我们从 `HeroesComponent` 的 `providers` 数组中移除了 `HeroService` 服务，并把它添加到顶级组件 `AppModule` 的 `providers` 数组中。

这个改动创建了一个 `HeroService` 的单例对象，它对应用中的 **所有** 组件都有效。在 `DashboardComponent` 组件中，Angular 会把 `HeroService` 注入进来，我们就能在 `DashboardComponent` 中使用它了。

获取英雄数组

打开 `dashboard.component.ts` 文件，并把必备的 `import` 语句加进去。

app/dashboard.component.ts (imports)

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

import { Hero } from './hero';
import { HeroService } from './hero.service';
```

我们得实现 `OnInit` 接口，因为我们要在 `ngOnInit` 方法中初始化英雄数组——就像上次一样。我们需要导入 `Hero` 类和 `HeroService` 类来引用它们的数据类型。

我们现在就实现 `DashboardComponent` 类，像这样：

app/dashboard.component.ts (class)

```
export class DashboardComponent implements OnInit {

  heroes: Hero[] = [];

  constructor(private heroService: HeroService) { }

  ngOnInit(): void {
    this.heroService.getHeroes()
      .then(heroes => this.heroes = heroes.slice(1, 5));
  }
}
```

在 `HeroesComponent` 之前，我们也看到过类似的逻辑：

- 创建一个 `heroes` 数组属性
- 把 `HeroService` 注入构造函数中，并且把它保存在一个私有的 `heroService` 字段中。
- 在 Angular 的 `ngOnInit` 生命周期钩子里面调用服务来获得英雄列表。

值得注意的区别是：我们用 `Array.slice` 方法提取了四个英雄（第 2、3、4、5 个）。

刷新浏览器，在这个新的控制台中就看到了四个英雄。

导航到英雄详情

虽然我们在 `HeroesComponent` 组件的底部显示了所选英雄的详情，但我们还没有 **导航** 到 `HeroDetailComponent` 组件过——我们曾在需求中指定过三种：

1. 从 **控制台 (Dashboard)** 导航到一个选定的英雄。
2. 从 **英雄列表 (Heroes)** 导航到一个选定的英雄。
3. 把一个指向该英雄的“深链接” URL 粘贴到浏览器的地址栏。

添加 `hero-detail` 路由，是一个显而易见的起点。

路由到一个英雄详情

我们将在 `app.module.ts` 中添加一个到 `HeroDetailComponent` 的路由，也就是配置其它路由的地方。

新路由的不寻常之处在于，我们必须告诉 `HeroDetailComponent` **该显示哪个英雄**。以前的 `HeroesComponent` 组件和 `DashboardComponent` 组件还未要求我们告诉它任何东西。

现在，父组件 `HeroesComponent` 通过数据绑定来把一个英雄对象设置为组件的 `hero` 属性。就像这样：

```
<my-hero-detail [hero]="selectedHero"></my-hero-detail>
```

显然，在我们的任何一个路由场景中它都无法工作。不仅如此，我们也没法把一个完整的 `hero` 对象嵌入到 URL 中！不过我们本来也不想这样。

参数化路由

我们可以把英雄的 `id` 添加到 URL 中。当导航到一个 `id` 为 11 的英雄时，我们期望的 URL 是这样的：

```
/detail/11
```

URL 中的 `/detail/` 部分是固定不变的，但结尾的数字 `id` 部分会随着英雄的不同而变化。我们要把路由中可变的那部分表示成一个 **参数 (parameter)** 或 **Token**，用以获取英雄的 `id`。

配置带参数的路由

下面是我们将使用的 **路由定义**。

app/app.module.ts (hero detail)

```
{
  path: 'detail/:id',
  component: HeroDetailComponent
},
```

路径中的冒号 (`:`) 表示 `:id` 是一个占位符，当导航到这个 `HeroDetailComponent` 组件时，它将被填入一个特定英雄的 `id`。

我们已经做好了该应用的路由。

我们没有往模板中添加一个 '英雄详情'，这是因为用户不会直接点击导航栏中的链接去查看一个特定的英雄。它们只会通过在英雄列表或者控制台中点击来显示一个英雄。

稍后我们会响应这些 **英雄** 的点击事件。现在对它们做什么都还没有意义——除非 `HeroDetailComponent` 已经做好了，并且能够被导航过去。

那就需要对 `HeroDetailComponent` 做一次大修。

修改 HeroDetailComponent

在重写 `HeroDetailComponent` 之前，我们先看看它现在的样子：

app/hero-detail.component.ts (current)

```
1. import { Component, Input } from '@angular/core';
2. import { Hero } from './hero';
3.
4. @Component({
5.   selector: 'my-hero-detail',
6.   template: `
7.     <div *ngIf="hero">
8.       <h2>{{hero.name}} details!</h2>
9.       <div>
10.         <label>id: </label>{{hero.id}}
11.       </div>
12.       <div>
13.         <label>name: </label>
14.         <input [(ngModel)]="hero.name" placeholder="name"/>
15.       </div>
16.     </div>
17.   `
18. })
19. export class HeroDetailComponent {
20.   @Input() hero: Hero;
21. }
```

模板不用修改，我们会用原来的方式显示英雄。导致这次大修的原因是如何获得这个英雄的数据。

我们不会再从父组件的属性绑定中取得英雄数据。新的 `HeroDetailComponent` 应该从 `ActivatedRoute` 服务的可观察对象 `params` 中取得 `id` 参数，并通过 `HeroService` 服务获取具有这个指定 `id` 的英雄数据。

首先，添加需要的导入项目：

```
// Keep the Input import for now, we'll remove it later:
import { Component, Input, OnInit } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
import { Location } from '@angular/common';

import { HeroService } from './hero.service';
```

然后注入 `ActivatedRoute` 和 `HeroService` 服务到构造函数中，将它们的值保存到私有变量中：

app/hero-detail.component.ts (constructor)

```
constructor(
  private heroService: HeroService,
  private route: ActivatedRoute,
  private location: Location
) {}
```

我们告诉这个类，我们要实现 `OnInit` 接口。

```
export class HeroDetailComponent implements OnInit {
```

在 `ngOnInit` 生命周期钩子中，从 `RouteParams` 服务中提取 `id` 参数，并且使用 `HeroService` 来获得具有这个 `id` 的英雄数据。

app/hero-detail.component.ts (ngOnInit)

```
ngOnInit(): void {
  this.route.params.forEach((params: Params) => {
    let id = +params['id'];
    this.heroService.getHero(id)
      .then(hero => this.hero = hero);
  });
}
```

注意我们提取 `id` 的方法：调用 `forEach` 方法，它会提供一个路由参数的数组。

英雄的 `id` 是数字，而路由参数的值 **总是字符串**。所以我们需要通过 JavaScript 的 `(+)` 操作符把路由参数的值转成数字。

添加 HeroService.getHero

这段代码的问题在于 `HeroService` 并没有一个叫 `getHero` 的方法，我们最好在别人报告应用出问题之前赶快修复它。

打开 `HeroService`，并添加一个 `getHero` 方法，用来通过 `id` 从 `getHeros` 过滤英雄列表：

app/hero.service.ts (getHero)

```
getHero(id: number): Promise<Hero> {
  return this.getHeroes()
    .then(heroes => heroes.find(hero => hero.id === id));
}
```

回到 `HeroDetailComponent` 来完成收尾工作。

回到原路

我们能用多种方式导航到 `HeroDetailComponent`。但当我们完工时，我们该导航到那里呢？

现在用户可以点击 `AppComponent` 中的两个链接，或点击浏览器的“后退”按钮。 我们来添加第三个选项：一个 `goBack` 方法，来根据浏览器的历史堆栈，后退一步。

app/hero-detail.component.ts (goBack)

```
goBack(): void {
  this.location.back();
}
```

回退太多步会跑出我们的应用。 在 Demo 中，这算不上问题。但在真实的应用中，我们需要对此进行防范。也许你该用 `CanDeactivate guard`。

然后，我们通过一个事件绑定把此方法绑定到模板底部的 **后退 (Back)** 按钮上。

```
<button (click)="goBack()">Back</button>
```

修改模板，添加这个按钮以提醒我们还要做更多的改进，并把模板移到独立的 `hero-detail.component.html` 文件中去。

app/hero-detail.component.html

```
<div *ngIf="hero">
  <h2>{{hero.name}} details!</h2>
  <div>
    <label>id: </label>{{hero.id}}</div>
  <div>
    <label>name: </label>
    <input [(ngModel)]="hero.name" placeholder="name" />
  </div>
  <button (click)="goBack()">Back</button>
</div>
```

然后更新组件的元数据，用一个 `templateUrl` 属性指向我们刚刚创建的模板文件。

app/hero-detail.component.ts (metadata)

```
@Component({
  moduleId: module.id,
  selector: 'my-hero-detail',
  templateUrl: 'hero-detail.component.html',
})
```

下面是(几乎)完成的 `HeroDetailComponent`：

app/hero-detail.component.ts (latest)

```
import { Component, OnInit }      from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
import { Location }             from '@angular/common';

import { Hero }                 from './hero';
import { HeroService }          from './hero.service';
@Component({
  moduleId: module.id,
  selector: 'my-hero-detail',
  templateUrl: 'hero-detail.component.html',
})
export class HeroDetailComponent implements OnInit {
  hero: Hero;

  constructor(
    private heroService: HeroService,
    private route: ActivatedRoute,
    private location: Location
  ) {}

  ngOnInit(): void {
    this.route.params.forEach((params: Params) => {
      let id = +params['id'];
      this.heroService.getHero(id)
        .then(hero => this.hero = hero);
    });
  }
}
```

```
}

goBack(): void {
  this.location.back();
}

}
```

选择一个 控制台 中的英雄

当用户从控制台中选择了一位英雄时，本应用要导航到 `HeroDetailComponent` 以查看和编辑所选的英雄。

虽然控制台英雄被显示为像按钮一样的方块，但是它们的行为应该像锚标签一样。当鼠标移动到一个英雄方块上时，目标 URL 应该显示在浏览器的状态条上，用户应该能拷贝链接或者在新的浏览器标签中打开英雄详情视图。

要达到这个效果，再次打开 `dashboard.component.html`，将用来迭代的 `<div *ngFor...>` 替换为 `<a>`，就像这样：

app/dashboard.component.html (repeated <a> tag)

```
<a *ngFor="let hero of heroes" [routerLink]=["'/detail', hero.id]"
class="col-1-1-4">
```

注意 `[routerLink]` 绑定。

`AppComponent` 模板 中的顶级导航有一些路由器链接被设置固定的路径，例如 `"/dashboard"` 和 `"/heroes"`。

这次，我们绑定了一个包含 **链接参数数组** 的表达式。该数组有两个元素，目的路由的路径和一个用来设置当前英雄的 **路由参数**。

这两个数组项目与我们之前在 `app.module.ts` 中添加的 `path` 和 `:id` 为代号被参数化的英雄详情路由的配置对象对应。

app/app.module.ts (hero detail)

```
{
  path: 'detail/:id',
  component: HeroDetailComponent
},
```

刷新浏览器，并从控制台中选择一位英雄，应用就会直接导航到英雄的详情。

重构路由为一个 路由模块

`AppModule` 中有将近 20 行代码是用来配置四个路由的。绝大多数应用有更多路由，并且它们还有 `守卫服务` 来保护不希望或未授权的导航。路由的配置可能迅速占领这个模块，并掩盖其主要目的，即为 Angular 编译器设置整个应用的关键配置。

我们应该重构路由配置到它自己的类。什么样的类呢？当前的 `RouterModule.forRoot()` 产生一个 Angular `ModuleWithProviders`，所以这个路由类应该是一种模块类。它应该是一个 **路由模块**。

按约定，**路由模块** 的名字应该包含“`Routing`”，并与导航到的组件所在的模块的名称看齐。

在 `app.module.ts` 所在目录创建 `app-routing.module.ts` 文件。将下面从 `AppModule` 类提取出来的代码拷贝进去：

app/app-routing.module.ts

```

1. import { NgModule }           from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3.
4. import { DashboardComponent } from './dashboard.component';
5. import { HeroesComponent }    from './heroes.component';
6. import { HeroDetailComponent } from './hero-detail.component';
7.
8. const routes: Routes = [
9.   { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
10.  { path: 'dashboard', component: DashboardComponent },
```

```

11.      { path: 'detail/:id', component: HeroDetailComponent },
12.      { path: 'heroes',      component: HeroesComponent }
13. ];
14.
15. @NgModule({
16.   imports: [ RouterModule.forRoot(routes) ],
17.   exports: [ RouterModule ]
18. })
19. export class AppRoutingModule {}

```

典型 路由模块 值得注意的有：

- 将路由抽出到一个变量中。你可能将来会导出它。而且它让 **路由模块** 模式更加明确。
- 添加 `RouterModule.forRoot(routes)` 到 `imports` .
- 添加 `RouterModule` 到 `exports` , 这样关联模块的组件可以访问路由的声明，比如 `RouterLink` 和 `RouterOutlet` 。
- 无 `Declarations` ! 声明是关联模块的任务。
- 如果你有守卫服务，添加模块 `providers` ; 本例子无守卫服务。

更新 AppModule

现在，删除 `AppModule` 中的路由配置，并导入 `AppRoutingModule` (同时用 ES 导入语句和将它添加到 `NgModule.imports` 数组) 。

下面是修改后的 `AppModule` , 与重构前的对比：

```

1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { AppComponent }   from './app.component';
6. import { DashboardComponent } from './dashboard.component';

```

```

7. import { HeroDetailComponent } from './hero-detail.component';
8. import { HeroesComponent }      from './heroes.component';
9. import { HeroService }         from './hero.service';
10.
11. import { AppRoutingModule }    from './app-routing.module';
12.
13. @NgModule({
14.   imports: [
15.     BrowserModule,
16.     FormsModule,
17.     AppRoutingModule
18.   ],
19.   declarations: [
20.     AppComponent,
21.     DashboardComponent,
22.     HeroDetailComponent,
23.     HeroesComponent
24.   ],
25.   providers: [ HeroService ],
26.   bootstrap: [ AppComponent ]
27. })
28. export class AppModule { }

```

它更简单，专注于确定应用的关键部分。

在 HeroesComponent 中选择一位英雄

之前我们添加了从控制台选择一个英雄的功能。我们现在要做的事和 `HeroesComponent` 中很像。

那个组件的当前模板展示了一个主从风格的界面：上方是英雄列表，底下是所选英雄的详情。

app/heroes.component.ts (当前的模板)

```

template: `
<h1>{{title}}</h1>
<h2>My Heroes</h2>

```

```

<ul class="heroes">
  <li *ngFor="let hero of heroes"
    [class.selected]="hero === selectedHero"
    (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
<my-hero-detail [hero]="selectedHero"></my-hero-detail>
,

```

我们要做的是将详情组建移动到它自己的视图，并在用户决定编辑一个英雄时导航到它。

删除顶部的 `<h1>` (在从 `AppComponent` 转到 `HeroesComponent` 期间可以先忘掉它) 。

删除模板最后带有 `<my-hero-detail>` 标签的那一行。

这里我们不再展示完整的 `HeroDetailComponent` 了。 我们要在它自己的页面中显示英雄详情，并像我们在控制台中所做的那样路由到它。

但是，我们要做一点小小的改动。 当用户从这个列表中选择一个英雄时，我们 **不会** 再跳转到详情页。 而是在本页中显示一个 **Mini 版英雄详情**，并等用户点击一个按钮时，才导航到 **完整版英雄详情** 页。

添加 Mini 版英雄详情

在模板底部原来放 `<my-hero-detail>` 的地方添加下列 HTML 片段：

```

<div *ngIf="selectedHero">
  <h2>
    {{selectedHero.name | uppercase}} is my hero
  </h2>
  <button (click)="gotoDetail()">View Details</button>
</div>

```

点击一个英雄，用户将会在英雄列表的下方看到这些：

MR. NICE is my hero

[View Details](#)

使用UpperCasePipe 格式化

注意，英雄的名字全被显示成大写字母。那是 `uppercase` 管道的效果，借助它，我们能插手“插值表达式绑定”的过程。去管道操作符 (`|`) 后面找它。

```
 {{selectedHero.name | uppercase}} is my hero
```

管道擅长做下列工作：格式化字符串、金额、日期和其它显示数据。Angular 自带了好几个管道，而且我们还可以写自己的管道。

要学习关于管道的更多知识，参见 [管道](#) 一章。

把内容移出组件文件

这还没完。当用户点击 [查看详情](#) 按钮时，要让它能导航到 `HeroDetailComponent`，我们仍然不得不修改组件类。

这个组件文件太大了。它大部分都是模板或 CSS 样式。要想在 HTML 和 CSS 的噪音中看清组件的工作逻辑太难了。

在做更多修改之前，我们先把模板和样式移到它们自己的文件中去：

1. 把模板内容 **剪切并粘贴** 到新的 `heroes.component.html` 文件。
2. 把样式内容 **剪切并粘贴** 到新的 `heroes.component.css` 文件。

3. 设置 组件元数据的 `templateUrl` 和 `styleUrls` 属性，来分别引用这两个文件。

4. 设置 `moduleId` 属性为 `module.id`，将 `templateUrl` 和 `styleUrls` 路径设置为相对组件的路径。

`styleUrls` 属性是一个由样式文件的文件名（包括路径）组成的数组。我们还可以列出来自多个不同位置的样式文件。

app/heroes.component.ts (revised metadata)

```
@Component({
  moduleId: module.id,
  selector: 'my-heroes',
  templateUrl: 'heroes.component.html',
  styleUrls: [ 'heroes.component.css' ]
})
```

更新 HeroesComponent 类

点击按钮时，`HeroesComponent` 导航到 `HeroesDetailComponent`。该按钮的 **点击** 事件被绑定到 `gotoDetail` 方法，它使用命令式的导航，告诉路由器去哪儿。

该方法需要组件类做些变化：

1. 从 Angular 路由器库导入 `router`
2. 在构造函数中注入 `router`（与 `HeroService` 一起）
3. 实现 `gotoDetail`，调用 `router.navigate` 方法

app/heroes.component.ts (gotoDetail)

```
gotoDetail(): void {
  this.router.navigate(['/detail', this.selectedHero.id]);
```

}

注意，我们将一个双元素的 **链接参数数组** ——路径和路由参数——传递到 `router.navigate`，与之前在 `DashboardComponent` 中使用 `[routerLink]` 绑定一样。

app/heroes.component.ts (class)

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

  constructor(
    private router: Router,
    private heroService: HeroService) { }

  getHeroes(): void {
    this.heroService.getHeroes().then(heroes => this.heroes = heroes);
  }

  ngOnInit(): void {
    this.getHeroes();
  }

  onSelect(hero: Hero): void {
    this.selectedHero = hero;
  }

  gotoDetail(): void {
    this.router.navigate(['/detail', this.selectedHero.id]);
  }
}
```

刷新浏览器，并开始点击。我们能在应用中导航：从控制台到英雄详情再回来，从英雄列表到 Mini 版英雄详情到英雄详情，再回到英雄列表。我们可以在控制台和英雄列表之间跳来跳去。

我们已经满足了在本章开头设定的所有导航需求。

美化本应用

应用在功能上已经正常了，但还太丑。我们富有创意的设计师团队提供了一些 CSS 文件，能让它变得好看一些。

具有样式的控制台

设计师认为我们应该把控制台的英雄们显示在一排方块中。它们给了我们大约 60 行 CSS 来实现它，包括一些简单的媒体查询语句来实现响应式设计。

如果我们把这 60 来行 CSS 粘贴到组件元数据的 `styles` 中，它们会完全淹没组件的工作逻辑。不能这么做。在一个独立的 `*.css` 文件中编辑 CSS 当然会更简单。

把 `dashboard.component.css` 文件添加到 `app` 目录下，并在组件元数据的 `styleUrls` 数组属性中引用它。就像这样：

```
app/dashboard.component.ts (styleUrls)
```

```
styleUrls: [ 'dashboard.component.css' ]
```

美化英雄详情

设计师还给了我们 `HeroDetailComponent` 特有的 CSS 风格。

在 `app` 目录下添加 `hero-detail.component.css` 文件，并且在 `styleUrls` 数组中引用它——就像当初在 `DashboardComponent` 中做过的那样。同时删除 `hero``@Input` 装饰器属性和它的导入语句。

上述组件的 CSS 文件内容如下：

```
1.  label {
2.    display: inline-block;
3.    width: 3em;
4.    margin: .5em 0;
5.    color: #607D8B;
6.    font-weight: bold;
```

```
7.      }
8.      input {
9.        height: 2em;
10.       font-size: 1em;
11.       padding-left: .4em;
12.     }
13.     button {
14.       margin-top: 20px;
15.       font-family: Arial;
16.       background-color: #eee;
17.       border: none;
18.       padding: 5px 10px;
19.       border-radius: 4px;
20.       cursor: pointer; cursor: hand;
21.     }
22.     button:hover {
23.       background-color: #cf8dc;
24.     }
25.     button:disabled {
26.       background-color: #eee;
27.       color: #ccc;
28.       cursor: auto;
29.     }
```

美化导航链接

设计师还给了我们一些 CSS，用于让 `AppComponent` 中的导航链接看起来更像可被选择的按钮。要让它们协同工作，我们得把那些链接包含在 `<nav>` 标签中。

在 `app` 目录下添加一个 `app.component.css` 文件，内容如下：

app/app.component.css (navigation styles)

```
h1 {
  font-size: 1.2em;
  color: #999;
  margin-bottom: 0;
}
h2 {
  font-size: 2em;
```

```
margin-top: 0;
padding-top: 0;
}
nav a {
  padding: 5px 10px;
  text-decoration: none;
  margin-top: 10px;
  display: inline-block;
  background-color: #eee;
  border-radius: 4px;
}
nav a:visited, a:link {
  color: #607D8B;
}
nav a:hover {
  color: #039be5;
  background-color: #CFD8DC;
}
nav a.active {
  color: #039be5;
}
```

routerLinkActive 指令

Angular 路由器提供了 `routerLinkActive` 指令，我们可以用它来为匹配了活动路由的 HTML 导航元素自动添加一个 CSS 类。我们唯一要做的就是为它定义样式。真好！

app/app.component.ts (active router links)

```
template: `
<h1>{{title}}</h1>
<nav>
  <a routerLink="/dashboard"
  routerLinkActive="active">Dashboard</a>
  <a routerLink="/heroes"
  routerLinkActive="active">Heroes</a>
</nav>
```

```
<router-outlet></router-outlet>
`,
```

设置 `AppComponent` 的 `styleUrls` 属性，指向这个 CSS 文件。

app/app.component.ts (styleUrls)

```
styleUrls: ['app.component.css'],
```

应用的全局样式

当我们把样式添加到组件中时，我们假定组件所需的一切——HTML、CSS、程序代码——都在紧邻的地方。这样，无论是把它们打包在一起还是在别的组件中复用它都会很容易。

我们也可以在所有组件之外创建 **应用级** 样式。

我们的设计师提供了一组基础样式，这些样式应用到的元素横跨整个应用。这些和我们前面引入的主样式全集是一样的（参见 [快速起步，“添加一些样式”](#)）。下面是摘录：

styles.css (excerpt)

```
/* Master styles */
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
```

```
}

body, input[text], button {
  color: #888;
  font-family: Cambria, Georgia;
}

/* . . . */
/* everywhere else */
* {
  font-family: Arial, Helvetica, sans-serif;
}
```

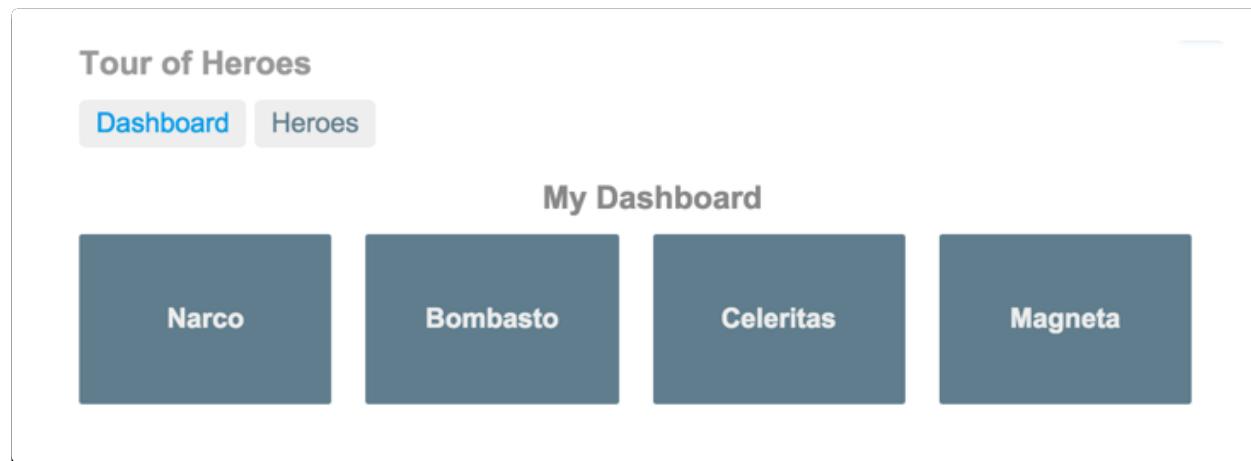
如果在根目录下没有一个名叫 `styles.css` 的文件，就添加它。 确保它包含 [这里给出的主样式](#)。

如有必要，也可以编辑 `index.html` 来引用这个样式表。

index.html (link ref)

```
<link rel="stylesheet" href="styles.css">
```

看看现在的应用！我们的控制台、英雄列表和导航链接都漂亮多了！



应用结构和代码

在 [在线例子](#) 中回顾本章的范例代码。 验证我们是否已经得到了如下结构：

```
angular-tour-of-heroes
├── app
│   ├── app.component.css
│   ├── app.component.ts
│   ├── app.module.ts
│   ├── app-routing.module.ts
│   ├── dashboard.component.css
│   ├── dashboard.component.html
│   ├── dashboard.component.ts
│   ├── hero.service.ts
│   ├── hero.ts
│   ├── hero-detail.component.css
│   ├── hero-detail.component.html
│   ├── hero-detail.component.ts
│   ├── heroes.component.css
│   ├── heroes.component.html
│   ├── heroes.component.ts
│   ├── main.ts
│   └── mock-heroes.ts
├── node_modules ...
├── index.html
└── package.json
```

```
├── styles.css
└── systemjs.config.js
```

```
tsconfig.json
```

总结

走过的路

在本章中，我们往前走了很远：

- 添加了 Angular 路由器 在各个不同组件之间导航。
- 学会了如何创建路由链接来表示导航栏的菜单项。
- 使用路由链接参数来导航到用户所选的英雄详情。
- 在多个组件之间共享了 `HeroService` 服务。
- 把 HTML 和 CSS 从组件中移出来，放到了它们自己的文件中。
- 添加了 `uppercase` 管道，来格式化数据
- 将路由重构为路由模块，并导入它。

前方的路

我们有了很多用于构建应用的基石。但我们仍然缺少很关键的一块：远程数据存取。

在下一章，我们将从硬编码模拟数据改为使用 http 服务从服务器获取数据。

下一步

[HTTP](#)

HTTP

把我们的服务和组件改为用 Angular 的 HTTP 服务实现

获取与保存数据

客户对我们的进展很满意！现在，它们想要从服务器获取英雄数据，然后让用户添加、编辑和删除英雄，并且把这些修改结果保存回服务器。

在这一章中，我们要让应用程序学会通过 HTTP 调用来访问远程服务器上相应的 Web API。

运行这部分的 [在线例子](#)。

延续上一步教程

在 [前一章](#) 中，我们学会了在仪表盘和固定的英雄列表之间导航，并编辑选定的英雄。这也就是本章的起点。

保持应用的转译与运行

打开 terminal/console 窗口，输入下列命令来启动 TypeScript 编译器，它会启动开发服务器，并监视文件变更：

```
npm start
```

当我们继续构建《英雄指南》时，应用会运行并自动更新。

准备 HTTP 服务

HttpModule 并不是 Angular 的核心模块。它是 Angular 用来进行 Web 访问的一种可选方式，并通过 Angular 包中一个名叫 `@angular/http` 的独立附属模块发布了出来。

幸运的是，`systemjs.config` 中已经配置好了 **SystemJS**，并在必要时加载它，因此我们已经为从 `@angular/http` 中导入它做好了准备。

注册 HTTP 服务

我们的应用将会依赖于 Angular 的 `http` 服务，它本身又依赖于其它支持类服务。来自 `@angular/http` 库中的 `HttpModule` 保存着这些 HTTP 相关服务提供商的全集。

我们要能从本应用的任何地方访问这些服务，就要把 `HttpModule` 添加到 `AppModule` 的 `imports` 列表中。这里同时也是我们引导应用及其根组件 `AppComponent` 的地方。

app/app.module.ts (v1)

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4. import { HttpModule }    from '@angular/http';

5.

6. import { AppRoutingModule } from './app-routing.module';
7.

8. import { AppComponent }      from './app.component';
9. import { DashboardComponent } from './dashboard.component';
10. import { HeroesComponent }   from './heroes.component';
11. import { HeroDetailComponent } from './hero-detail.component';
12. import { HeroService }       from './hero.service';

13.

14. @NgModule({
15.   imports: [
16.     BrowserModule,
17.     FormsModule,
18.     HttpModule,
19.     AppRoutingModule
20.   ],
21.   declarations: [
22.     AppComponent,
```

```

23.      DashboardComponent,
24.      HeroDetailComponent,
25.      HeroesComponent,
26.    ],
27.    providers: [ HeroService ],
28.    bootstrap: [ AppComponent ]
29.  })
30. export class AppModule { }

```

注意，现在 `HttpModule` 已经是根模块 `AppModule` 的 `imports` 数组的一部分了。

模拟 Web API

我们建议在根模块 `AppModule` 的 `providers` 数组中注册全应用级的服务。

我们的应用正处于开发的早期阶段，并且离进入产品阶段还很远。我们甚至都还没有一个用来处理英雄相关请求的 Web 服务器，在此之前，**我们将不得不伪造一个**。

我们要**要点小花招**，让 http 客户端从一个 Mock 服务（**内存 (in-memory)Web API**）中获取和保存数据。

这个版本的 `app/app.module.ts` 就是用来实现这个小花招的：

app/app.module.ts (v2)

```

import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { HttpModule }    from '@angular/http';

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

// Imports for loading & configuring the in-memory web api
import { InMemorywebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService }  from './in-memory-data.service';

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

```

```

import { HeroesComponent }      from './heroes.component';
import { HeroDetailComponent } from './hero-detail.component';
import { HeroService }         from './hero.service';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    InMemoryWebApiModule.forRoot(InMemoryDataService),
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    DashboardComponent,
    HeroDetailComponent,
    HeroesComponent,
  ],
  providers: [ HeroService ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }

```

导入 `InMemoryWebApiModule` 并将其加入到模块的 `imports` 数组。

`InMemoryWebApiModule` 将 `Http` 客户端默认的后端服务（这是一个辅助服务，负责与远程服务器对话）替换了 **内存 Web API** 服务：

```
InMemoryWebApiModule.forRoot(InMemoryDataService),
```

`forRoot` 配置方法需要 `InMemoryDataService` 类实例，用来向内存数据库填充数据：

app/in-memory-data.service.ts

```

import { InMemoryDbService } from 'angular-in-memory-web-api';
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    let heroes = [
      {id: 11, name: 'Mr. Nice'},

```

```

        {id: 12, name: 'Narco'},
        {id: 13, name: 'Bombasto'},
        {id: 14, name: 'Celeritas'},
        {id: 15, name: 'Magneta'},
        {id: 16, name: 'RubberMan'},
        {id: 17, name: 'Dynamite'},
        {id: 18, name: 'Dr IQ'},
        {id: 19, name: 'Magma'},
        {id: 20, name: 'Tornado'}
    ];
    return {heroes};
}
}

```

该文件代替了 `mock-heroes.ts` 文件，它现在可以安全的删除了。

本章主要是介绍 Angular 的 HTTP 库，不要因为这种“替换后端”的细节而分心。先不要管为什么，只管照着这个例子做就可以了。

要学习关于 **内存 Web API** 的更多知识，请参阅 [HTTP 客户端](#) 一章。记住，**内存 Web API** 主要用于开发的早期阶段或《英雄指南》这样的演示程序。如果你已经有了一个真实的 Web API 服务器，请尽管跳过它。

英雄与 HTTP

来看看我们目前的 `HeroService` 的实现

```

getHeroes(): Promise<Hero[]> {
    return Promise.resolve(HEROES);
}

```

我们返回一个承诺（Promise），它用 mock 版的英雄列表进行解析。它当时可能看起来显得有点过于复杂，不过我们早就预料到总有这么一天会通过一个 HTTP 客户端来获取英雄数据，而且我们知道，那一定是一个异步操作。

这一天到来了！我们把 `getHeroes()` 换成用 HTTP。

app/hero.service.ts (updated getHeroes and new class members)

```
private heroesUrl = 'app/heroes'; // URL to web api

constructor(private http: Http) { }

getHeroes(): Promise<Hero[]> {
  return this.http.get(this.heroesUrl)
    .toPromise()
    .then(response => response.json().data as Hero[])
    .catch(this.handleError);
}
```

更新后的导入声明如下：

app/hero.service.ts (updated imports)

```
import { Injectable }      from '@angular/core';
import { Headers, Http }  from '@angular/http';

import 'rxjs/add/operator/toPromise';

import { Hero }           from './hero';
```

刷新浏览器后，英雄数据就会从 Mock 服务器被成功读取。

Http 承诺 (Promise)

我们仍然返回一个承诺，但是用不同的方法来创建它。

Angular 的 `http.get` 返回一个 RxJS 的 `Observable` 对象。 **Observable(可观察对象)** 是一个管理异步数据流的强力方式。 后面我们还会进一步学习 **可观察对象**。

现在，我们先利用 `toPromise` 操作符把 `Observable` 直接转换成 `Promise` 对象，回到已经熟悉的地盘。

```
.toPromise()
```

不幸的是，Angular 的 `Observable` 并没有一个 `toPromise` 操作符 ... 没有打包在一起发布。Angular 的 `Observable` 只是一个骨架实现。

有很多像 `toPromise` 这样的操作符，用于扩展 `Observable`，为其添加有用的能力。如果我们希望得到那些能力，就得自己添加那些操作符。那很容易，只要从 RxJS 库中导入它们就可以了，就像这样：

```
import 'rxjs/add/operator/toPromise';
```

在 `then` 回调中提取出数据

在 `promise` 的 `then` 回调中，我们调用 HTTP 的 `Reponse` 对象的 `json` 方法，以提取出其中的数据。

```
.then(response => response.json().data as Hero[])
```

这个由 `json` 方法返回的对象只有一个 `data` 属性。这个 `data` 属性保存了 **英雄** 数组，这个数组才是调用者真正想要的。所以我们取得这个数组，并且把它作为承诺的值进行解析。

仔细看看这个由服务器返回的数据的形态。这个 **内存 Web API** 的范例中所做的是返回一个带有 `data` 属性的对象。你的 API 也可以返回其它东西。请调整这些代码以匹配 **你的 Web API**。

调用者并不知道这些实现机制，它仍然像以前那样接收一个包含 **英雄数据** 的承诺。它也不知道我们已经改成了从服务器获取英雄数据。它也不必了解把 HTTP 响应转换成英雄数据时所作的这些复杂变换。看到美妙之处了吧，这正是将数据访问委托给一个服务的目的。

错误处理

在 `getHeroes()` 的最后，我们 `catch` 了服务器的失败信息，并把它们传给了错误处理器：

```
.catch(this.handleError);
```

这是一个关键的步骤！我们必须预料到 http 请求会失败，因为有太多我们无法控制的原因可能导致它们频繁出现各种错误。

```
private handleError(error: any): Promise<any> {
  console.error('An error occurred', error); // for demo purposes
  only
  return Promise.reject(error.message || error);
}
```

在这个范例服务中，我们把错误记录到控制台中；在真实世界中，我们应该做得更好。

我们还要通过一个被拒绝 (rejected) 的承诺 (promise) 来把该错误用一个用户友好的格式返回给调用者，以便调用者能把一个合适的错误信息显示给用户。

getHeroes API 没变

虽然我们对 `getHeroes()` 做了一些重要的 **内部** 修改，该方法公开的函数签名却没有任何变化。我们返回的仍然是一个承诺，不用被迫更新任何一个调用了 `getHeroes()` 的组件。

我们的客户很欣赏这种富有弹性的 API 集成方式。现在它们想增加创建和删除英雄的功能。

我们来看看在更新英雄的详情时会发生什么。

更新英雄详情

我们已经可以在英雄详情中编辑英雄的名字了。来试试吧。在输入的时候，页头上的英雄名字也会随之更新。不过当我们点了 Back (后退) 按钮时，这些修改就丢失了。

以前是不会丢失更新的，现在是怎么回事？当该应用使用 mock 出来的英雄列表时，修改的是一份全局共享的英雄列表，而现在改成了从服务器获取数据。如果我们希望这些更改被持久化，我们就得把它们写回服务器。

保存英雄详情

我们先来确保对英雄名字的编辑不会丢失。先在英雄详情模板的底部添加一个保存按钮，它绑定了一个 click 事件，事件绑定会调用组件中一个名叫 save 的新方法：

app/hero-detail.component.html (save)

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

save 方法使用 hero 服务的 update 方法来持久化对英雄名字的修改，然后导航回前一个视图：

app/hero-detail.component.ts (save)

```
save(): void {
  this.heroService.update(this.hero)
    .then(() => this.goBack());
}
```

hero 服务的 update 方法

update 方法的大致结构与 getHeroes 类似，不过我们使用 HTTP 的 put 方法来把修改持久化到服务端：

app/hero.service.ts (update)

```

private headers = new Headers({'Content-Type': 'application/json'});

update(hero: Hero): Promise<Hero> {
  const url = `${this.heroesUrl}/${hero.id}`;
  return this.http
    .put(url, JSON.stringify(hero), {headers: this.headers})
    .toPromise()
    .then(() => hero)
    .catch(this.handleError);
}

```

我们通过一个编码在 URL 中的英雄 id 来告诉服务器应该更新哪个英雄。 put 的 body 是该英雄的 JSON 字符串，它是通过调用 `JSON.stringify` 得到的。并且在请求头中标记出的 body 的内容类型（`application/json`）。

刷新浏览器试一下，对英雄名字的修改确实已经被持久化了。

添加英雄

要添加一个新的英雄，我们得先知道英雄的名字。我们使用一个 `input` 元素和一个添加按钮来实现。

把下列代码插入 `heroes` 组件的 HTML 中，放在标题的下面：

app/heroes.component.html (add)

```

<div>
  <label>Hero name:</label> <input #heroName />
  <button (click)="add(heroName.value); heroName.value=''">
    Add
  </button>
</div>

```

当 `click` 事件触发时，我们调用组件的 `click` 处理器，然后清空这个输入框，以便用来输入另一个名字。

app/heroes.component.ts (add)

```
add(name: string): void {
  name = name.trim();
  if (!name) { return; }
  this.heroService.create(name)
    .then(hero => {
      this.heroes.push(hero);
      this.selectedHero = null;
    });
}
```

当指定的名字不为空的时候， click 处理器就会委托 hero 服务来创建一个具有此名字的英雄，并把这个新的英雄添加到我们的数组中。

最后，我们在 `HeroService` 类中实现这个 `create` 方法。

app/hero.service.ts (create)

```
create(name: string): Promise<Hero> {
  return this.http
    .post(this.heroesUrl, JSON.stringify({name: name}), {headers:
  this.headers})
    .toPromise()
    .then(res => res.json().data)
    .catch(this.handleError);
}
```

刷新浏览器，并创建一些新的英雄！

删除一个英雄

英雄太多了？我们在英雄列表视图中来为每个英雄添加一个删除按钮吧。

把这个 `button` 元素添加到英雄列表组件的 HTML 中，把它放在 `` 标签中的英雄名的后面：

```
<button class="delete"
(click)="delete(hero); $event.stopPropagation()">x</button>
```

`` 元素应该变成了这样：

app/heroes.component.html (li-element)

```
<li *ngFor="let hero of heroes" (click)="onSelect(hero)"
[class.selected]="hero === selectedHero">
{{hero.id}}
{{hero.name}}
<button class="delete"
(click)="delete(hero); $event.stopPropagation()">x</button>
</li>
```

除了调用组件的 `delete` 方法之外，这个 `delete` 按钮的 `click` 处理器还应该阻止 `click` 事件向上冒泡——我们并不希望触发 `` 的事件处理器，否则它会选中我们要删除的这位英雄。

`delete` 处理器的逻辑略复杂：

app/heroes.component.ts (delete)

```
delete(hero: Hero): void {
  this.heroService
    .delete(hero.id)
    .then(() => {
      this.heroes = this.heroes.filter(h => h !== hero);
      if (this.selectedHero === hero) { this.selectedHero = null; }
    });
}
```

当然，我们仍然把删除英雄的操作委托给了 `hero` 服务，不过该组件仍然负责更新显示：它从数组中移除了被删除的英雄，如果删除的是正选中的英雄，还会清空选择。

我们希望删除按钮被放在英雄条目的最右边。于是 CSS 变成了这样：

app/heroes.component.css (additions)

```
button.delete {
  float:right;
  margin-top: 2px;
  margin-right: .8em;
  background-color: gray !important;
  color:white;
}
```

hero 服务的 `delete` 方法

hero 服务的 `delete` 方法使用 HTTP 的 `delete` 方法来从服务器上移除该英雄：

app/hero.service.ts (delete)

```
delete(id: number): Promise<void> {
  const url = `${this.heroesUrl}/${id}`;
  return this.http.delete(url, {headers: this.headers})
    .toPromise()
    .then(() => null)
    .catch(this.handleError);
}
```

刷新浏览器，并试一下这个新的删除功能。

可观察对象 (Observable)

`Http` 服务中的每个方法都返回一个 `HTTP Response` 对象的 `Observable` 实例。

我们的 `HeroService` 中把那个 `Observable` 对象转换成了 `Promise` (承诺)，并把这个承诺返回给了调用者。这一节，我们将学会直接返回 `Observable`，并且讨论何时以及为何那样做会更好。

背景

一个 **可观察对象** 是一个事件流，我们可以用数组型操作符（函数）来处理它。

Angular 内核中提供了对可观察对象的基本支持。而我们这些开发人员可以自己从 RxJS **可观察对象** 库中引入操作符和扩展。我们会简短的讲解下如何做。

快速回忆一下 `HeroService`，它在 `http.get` 返回的 `Observable` 后面串联了一个 `toPromise` 操作符。该操作符把 `Observable` 转换成了 `Promise`，并且我们把这个承诺返回给了调用者。

转换成承诺通常是更好地选择，我们通常会要求 `http.get` 获取单块数据。只要接收到数据，就算完成。使用承诺这种形式的结果是让调用方更容易写，并且承诺已经在 JavaScript 程序员中被广泛接受了。

但是请求并非总是“一次性”的。我们可以开始一个请求，并且取消它，再开始另一个不同的请求——在服务器对第一个请求作出回应之前。像这样一个 **请求 - 取消 - 新请求** 的序列用 **承诺** 是很难实现的，但接下来我们会看到，它对于 **可观察对象** 却很简单。

按名搜索

我们要为《英雄指南》添加一个 **英雄搜索** 特性。当用户在搜索框中输入一个名字时，我们将不断发起 HTTP 请求，以获得按名字过滤的英雄。

我们先创建 `HeroSearchService` 服务，它会把搜索请求发送到我们服务器上的 Web API

。

app/hero-search.service.ts

```
1. import { Injectable }      from '@angular/core';
2. import { Http, Response } from '@angular/http';
3. import { Observable } from 'rxjs';
4.
5. import { Hero }           from './hero';
6.
7. @Injectable()
8. export class HeroSearchService {
9.
10.   constructor(private http: Http) {}
11.
```

```

12.     search(term: string): Observable<Hero[]> {
13.       return this.http
14.         .get(`app/heroes/?name=${term}`)
15.         .map((r: Response) => r.json().data as Hero[]);
16.     }
17.   }

```

`HeroSearchService` 中的 `http.get()` 调用和 `HeroService` 中的很相似，只是这次带了查询字符串。显著的不同是：我们不再调用 `toPromise`，而是直接返回 可观察对象。

HeroSearchComponent

我们再创建一个新的 `HeroSearchComponent` 来调用这个新的 `HeroSearchService`。

组件模板很简单，就是一个输入框和一个显示匹配的搜索结果的列表。

app/hero-search.component.html

```

1.  <div id="search-component">
2.    <h4>Hero Search</h4>
3.    <input #searchBox id="search-box" (keyup)="search(searchBox.value)"
4.      />
5.    <div *ngFor="let hero of heroes | async"
6.          (click)="gotoDetail(hero)" class="search-result" >
7.      {{hero.name}}
8.    </div>
9.  </div>
10. </div>

```

我们还要往这个新组件中添加样式。

app/hero-search.component.css

```

1.  .search-result{
2.    border-bottom: 1px solid gray;

```

```

3. border-left: 1px solid gray;
4. border-right: 1px solid gray;
5. width:195px;
6. height: 20px;
7. padding: 5px;
8. background-color: white;
9. cursor: pointer;
10. }
11.
12. #search-box{
13.   width: 200px;
14.   height: 20px;
15. }
```

当用户在搜索框中输入时，一个 `keyup` 事件绑定会调用该组件的 `search` 方法，并传入新的搜索框的值。

`*ngFor` 从该组件的 `heroes` 属性重复获取 `hero` 对象。这也没啥特别的。

但是，接下来我们看到 `heroes` 属性现在是英雄列表的 `Observable` 对象，而不再只是英雄数组。`*ngFor` 不能用可观察对象做任何事，除非我们在它后面跟一个 `async` pipe (`AsyncPipe`)。这个 `async` 管道会订阅到这个可观察对象，并且为 `*ngFor` 生成一个英雄数组。

该创建 `HeroSearchComponent` 类及其元数据了。

app/hero-search.component.ts

```

1. import { Component, OnInit } from '@angular/core';
2. import { Router } from '@angular/router';
3. import { Observable } from 'rxjs/Observable';
4. import { Subject } from 'rxjs/Subject';
5.
6. import { HeroSearchService } from './hero-search.service';
7. import { Hero } from './hero';
8.
9. @Component({
10.   moduleId: module.id,
11.   selector: 'hero-search',
```

```
12.     templateUrl: 'hero-search.component.html',
13.     styleUrls: [ 'hero-search.component.css' ],
14.     providers: [HeroSearchService]
15.   })
16.
17.   export class HeroSearchComponent implements OnInit {
18.     heroes: Observable<Hero[]>;
19.     private searchTerms = new Subject<string>();
20.
21.     constructor(
22.       private heroSearchService: HeroSearchService,
23.       private router: Router) {}
24.
25.     // Push a search term into the observable stream.
26.     search(term: string): void {
27.       this.searchTerms.next(term);
28.     }
29.
30.     ngOnInit(): void {
31.       this.heroes = this.searchTerms
32.         .debounceTime(300)           // wait for 300ms pause in events
33.         .distinctUntilChanged()    // ignore if next search term is same
34.         as previous
35.         .switchMap(term => term   // switch to new observable each time
36.           // return the http search observable
37.           ? this.heroSearchService.search(term)
38.             // or the observable of empty heroes if no search term
39.             : Observable.of<Hero[]>([]))
40.         .catch(error => {
41.           // TODO: real error handling
42.           console.log(error);
43.           return Observable.of<Hero[]>([]);
44.         });
45.     }
46.
47.     gotoDetail(hero: Hero): void {
48.       let link = ['/detail', hero.id];
49.       this.router.navigate(link);
50.     }
51.   }
```

搜索词

仔细看下这个 `searchTerms` :

```
private searchTerms = new Subject<string>();

// Push a search term into the observable stream.
search(term: string): void {
  this.searchTerms.next(term);
}
```

`Subject` (主题) 是一个 **可观察的** 事件流中的生产者。`searchTerms` 生成一个产生字符串的 `Observable`，用作按名称搜索时的过滤条件。

每当调用 `search` 时都会调用 `next` 来把新的字符串放进该主题的 **可观察** 流中。

初始化 HEROES 属性 (NGONINIT)

`Subject` 也是一个 `Observable` 对象。我们要把搜索词的流转换成 `Hero` 数组的流，并把结果赋值给 `heroes` 属性。

```
heroes: Observable<Hero[]>;

ngOnInit(): void {
  this.heroes = this.searchTerms
    .debounceTime(300)          // wait for 300ms pause in events
    .distinctUntilChanged()     // ignore if next search term is same
    as previous
    .switchMap(term => term    // switch to new observable each time
      // return the http search observable
      ? this.heroSearchService.search(term)
      // or the observable of empty heroes if no search term
      : Observable.of<Hero[]>([]))
    .catch(error => {
      // TODO: real error handling
      console.log(error);
      return Observable.of<Hero[]>([]);
    });
}
```

如果我们直接把每一次用户按键都直接传给 `HeroSearchService`，就会发起一场 HTTP 请求风暴。这可不好玩。我们不希望占用服务器资源，也不想耗光蜂窝移动网络的流量。

幸运的是，我们可以在字符串的 `Observable` 后面串联一些 `Observable` 操作符，来归并这些请求。我们将对 `HeroSearchService` 发起更少的调用，并且仍然获得足够及时的响应。做法如下：

- 在传出最终字符串之前，`debounceTime(300)` 将会等待，直到新增字符串的事件暂停了 300 毫秒。我们实际发起请求的间隔永远不会小于 300ms。
- `distinctUntilChanged` 确保只在过滤条件变化时才发送请求，这样就不会重复请求同一个搜索词了。
- `switchMap` 会为每个从 `debounce` 和 `distinctUntilChanged` 中通过的搜索词调用搜索服务。它会取消并丢弃以前的搜索可观察对象，只保留最近的。

`switchMap` 操作符（以前叫 "flatMapLatest"）是非常智能的。

每次符合条件的按键事件都会触发一次对 `http` 方法的调用。即使在发送每个请求前都有 300 毫秒的延迟，我们仍然可能同时拥有多个在途的 HTTP 请求，并且它们返回的顺序未必就是发送时的顺序。

`switchMap` 保留了原始的请求顺序，并且只返回最近一次 `http` 调用返回的可观察对象。这是因为以前的调用都被取消或丢弃了。

如果搜索框为空，我们还可以短路掉这次 `http` 方法调用，并且直接返回一个包含空数组的可观察对象。

注意，取消 `HeroSearchService` 的可观察对象并不会实际中止（`abort`）一个未完成的 HTTP 请求，除非服务支持这个特性，这个问题我们以后再讨论。目前我们的做法只是丢弃不希望的结果。

- `catch` 拦截失败的可观察对象。这个简单的例子中只是把错误信息打印到控制台（但实际的应用需要做更多事），然后返回一个包含空数组的可观察对象，以清空搜索结果。

导入 RxJS 操作符

Angular 的基本版 `Observable` 实现中，RxJS 操作符是不可用的。我们得 **导入** 它们，以扩展 `Observable`。

通过在本文件的顶部写上适当的 `import` 语句，我们可以为 `Observable` 扩展出这里用到的那些操作符。

有很多权威人士建议我们这样做。

在这个例子中，我们使用一些不同的方法。我们把整个应用中要用的那些 RxJS `Observable` 扩展组合在一起，放在一个单独的 RxJS 导入文件中。

app/rxjs-extensions.ts

```
// Observable class extensions
import 'rxjs/add/observable/of';
import 'rxjs/add/observable/throw';

// Observable operators
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/switchMap';
```

我们在顶级的 `AppModule` 中导入 `rxjs-extensions` 就可以一次性加载它们。

app/app.module.ts (rxjs-extensions)

```
import './rxjs-extensions';
```

为仪表盘添加搜索组件

将表示“英雄搜索”组件的 HTML 元素添加到 `DashboardComponent` 模版的最后面。

app/dashboard.component.html

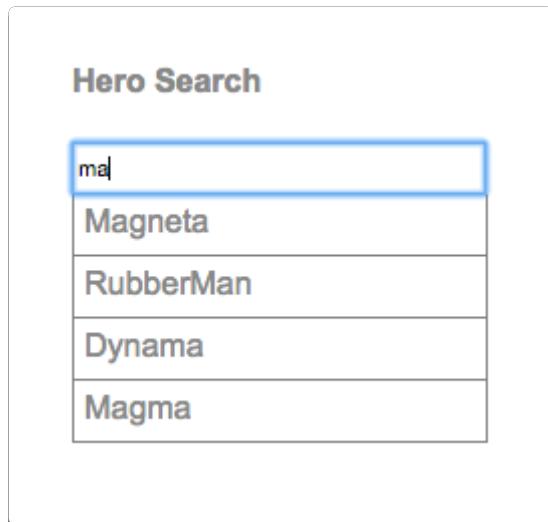
```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" [routerLink]=["'/detail', hero.id]"
    class="col-1-4">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>
<hero-search></hero-search>
```

最后，从 `hero-search.component.ts` 中导入 `HeroSearchComponent` 并将其添加到 `declarations` 数组中。

app/app.module.ts (search)

```
declarations: [
  AppComponent,
  DashboardComponent,
  HeroDetailComponent,
  HeroesComponent,
  HeroSearchComponent
],
```

再次运行该应用，跳转到 **Dashboard**，并在英雄下方的搜索框里输入一些文本。看起来就像这样：



应用的结构与代码

回顾一下本章 [在线例子](#) 中的范例代码。验证我们是否得到了如下结构：

```
angular-tour-of-heroes
  └── app
      ├── app.component.ts
      ├── app.component.css
      ├── app.module.ts
      ├── app-routing.module.ts
      ├── dashboard.component.css
      ├── dashboard.component.html
      ├── dashboard.component.ts
      ├── hero.ts
      ├── hero-detail.component.css
      ├── hero-detail.component.html
      ├── hero-detail.component.ts
      └── hero-search.component.html (new)
          └── hero-search.component.css (new)
```

```
hero-search.component.ts (new)
hero-search.service.ts (new)
rxjs-extensions.ts
hero.service.ts
heroes.component.css
heroes.component.html
heroes.component.ts
main.ts
in-memory-data.service.ts (new)
node_modules ...
index.html
package.json
styles.css
systemjs.config.js
tsconfig.json
```

最后冲刺

旅程即将结束，不过我们已经收获颇丰。

- 我们添加了在应用程序中使用 HTTP 的必备依赖。
- 我们重构了 `HeroService`，以通过 web API 来加载英雄数据。
- 我们扩展了 `HeroService` 来支持 `post`、`put` 和 `delete` 方法。
- 我们更新了组件，以允许用户添加、编辑和删除英雄。
- 我们配置了一个内存 Web API。
- 我们学会了如何使用“可观察对象”。

下面是我们 **添加或修改** 之后的文件汇总。

```

1. import { Component }           from '@angular/core';
2.
3. @Component({
4.   moduleId: module.id,
5.   selector: 'my-app',
6.   template: `
7.     <h1>{{title}}</h1>
8.     <nav>
9.       <a routerLink="/dashboard"
routerLinkActive="active">Dashboard</a>
10.      <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
11.    </nav>
12.    <router-outlet></router-outlet>
13.  `,
14.   styleUrls: ['app.component.css']
15. })
16. export class AppComponent {
17.   title = 'Tour of Heroes';
18. }

```

```

1. import { Injectable }         from '@angular/core';
2. import { Http, Response } from '@angular/http';
3. import { Observable } from 'rxjs';
4.
5. import { Hero }              from './hero';
6.
7. @Injectable()
8. export class HeroSearchService {
9.
10.   constructor(private http: Http) {}
11.
12.   search(term: string): Observable<Hero[]> {
13.     return this.http
14.       .get(`app/heroes/?name=${term}`)
15.       .map((r: Response) => r.json().data as Hero[]);
16.   }
17. }

```


ANGULAR模块（NgModule）

用 @NgModule 定义应用中的模块

Angular 模块 能帮你把应用组织成多个紧密相关的功能块。

Angular 模块是带有 `@NgModule` 装饰器函数的 **类**。`@NgModule` 接收一个元数据对象，该对象告诉 Angular 如何编译和运行模块代码。它标记出该模块 **拥有** 的组件、指令和管道，并把它们的一部分公开出去，以便外部组件使用它们。它可以向应用的依赖注入器中添加服务提供商。本章还会涉及到更多选项。

本章将会讲解如何 **创建** `NgModule` 类，以及如何加载它们——可以在程序启动时主动加载，也可以稍后由 **路由器** 按需加载。

目录

- [Angular 模块化](#)
- [应用的根模块](#)
- [引导 根模块](#)
- [声明](#)
- [提供商](#)
- [导入](#)
- [解决冲突](#)
- [特性模块](#)
- [通过路由器 惰性加载模块](#)
- [共享模块](#)
- [核心模块](#)
- [用 `forRoot` 配置核心服务](#)
- [禁止重复导入 `CoreModule`](#)
- [NgModule 元数据的属性](#)

在线例子

本章通过一个基于《英雄指南》的渐进式例子解释了 Angular 的模块。这里是例子演化过程中一些关键节点的在线例子。

- [最小化的 NgModule 应用](#)
- [第一个联系人模块](#)
- [修改过的联系人模块](#)
- [添加 SharedModule 之前](#)
- [最终版](#)

常见问题

本章涵盖了英雄指南下的 Angular 模块概念。

烹饪宝典中的 [Angular 模块常见问题](#) 为一些与设计和实现有关的问题提供了答案。不过在阅读常见问题之前，要先阅读本章。

Angular 模块化

模块是组织应用程序和使用外部程序库的最佳途径。

很多 Angular 库都是模块，比如：`FormsModule`、`HttpModule`、`RouterModule`。

很多第三方库也封装成了 Angular 模块，比如：[Material Design](#)、[Ionic](#)、[AngularFire2](#)。

Angular 模块把组件、指令和管道打包成内聚的功能块，每块聚焦于一个特性分区、业务领域、工作流，或一组通用的工具。

模块还能用来把服务加到应用程序中。这些服务可能是内部研发的，比如应用日志服务；也可能是外部资源，比如 Angular 路由和 Http 客户端。

模块可能在应用启动时主动加载，也可能由路由器进行异步 [惰性加载](#)。

Angular 模块是一个由 `@NgModule` 装饰器提供元数据的类，元数据包括：

- 声明哪些组件、指令、管道属于该模块。
- 公开某些类，以便其它的组件模板可以使用它们。
- 导入其它模块，从其它模块总获得本模块所需的组件、指令和管道。
- 在应用程序级提供服务，以便应用中的任何组件都能使用它。

每个 Angular 应用至少有一个模块类——**根模块**，我们将通过引导根模块来启动应用。

对于组件很少的简单应用来说，只用一个 **根模块** 就足够了。随着应用规模的增长，我们逐步从 **根模块** 中重构出一些 **特性模块**，它们用来实现一组密切相关的功能。然后，我们在 **根模块** 中导入它们。

稍后我们就会看到怎么做。不过还是先从 **根模块** 开始吧！

AppModule - 应用的根模块

每个 Angular 应用都有一个 **根模块** 类。按照约定，它的类名叫做 `AppModule`，被放在 `app.module.ts` 文件中。

最小化的 `AppModule` 是这样的：

```
app/app.module.ts (minimal)

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

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

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

`@NgModule` 装饰器用来为模块定义元数据。 我们先凭直觉来理解一下元数据，接下来再逐步深入细节。

这个元数据只导入了一个辅助模块，`BrowserModule`，每个运行在浏览器中的应用都必须导入它。

`BrowserModule` 注册了一些关键的应用服务提供商。 它还包括了一些通用的指令，比如`NgIf` 和 `NgFor`，所以这些指令在该模块的任何组件模板中都是可用的。

`declarations` 列出了该应用程序中唯一的组件（根组件），这是该应用的顶层组件，而不会暴露出整个组件树。

下面范例 `AppComponent` 显示被绑定的标题：

app/app.component.ts (minimal)

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

@Component({
  selector: 'my-app',
  template: '<h1>{{title}}</h1>',
})
export class AppComponent {
  title = 'Minimal NgModule';
}
```

最后，`@NgModule.bootstrap` 属性把这个 `AppComponent` 标记为 **引导 (bootstrap) 组件**。 当 Angular 引导应用时，它会在 DOM 中渲染 `AppComponent`，并把结果放进 `index.html` 的 `<my-app>` 元素标记内部。

在 main.ts 中引导

在 `main.ts` 文件中，我们通过引导 `AppModule` 来启动应用。

针对不同的平台，Angular 提供了很多引导选项。 本章我们只讲两个选项，都是针对浏览器平台的。

通过即时（ JIT ）编译器动态引导

先看看 `dynamic` 选项， Angular 编译器 在浏览器中编译并引导该应用。

```
app/main.ts (dynamic)

// The browser platform with a compiler
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

// The app module
import { AppModule } from './app.module';

// Compile and launch the module
platformBrowserDynamic().bootstrapModule(AppModule);
```

这里的例子演示进行动态引导的方法。

Angular 2 Example - Minimal NgModule

Project

app

- app.component.0.ts
- app.module.0.ts
- main.0.ts
- index.html
- README.md
- styles.css
- systemjs.config.js
- tsconfig.json

Preview

Minimal
NgModule

Click to run

Plunker © (2016) is created by Geoffrey Goodman

使用预编译器（ AoT - Ahead-Of-Time ）进行静态引导

静态方案可以生成更小、启动更快的应用，建议优先使用它，特别是在移动设备或高延迟网络下。

使用 **static** 选项，Angular 编译器作为构建流程的一部分提前运行，生成一组类工厂。它们的核心就是 `AppModuleNgFactory`。

引导预编译的 `AppModuleNgFactory` 的语法和动态引导 `AppModule` 类的方式很相似。

app/main.ts (static)

```
// The browser platform without a compiler
import { platformBrowser } from '@angular/platform-browser';

// The app module factory produced by the static offline compiler
import { AppModuleNgFactory } from './app.module.ngfactory';

// Launch with the app module factory.
platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

由于整个应用都是预编译的，所以我们不用把 **Angular 编译器** 一起发到浏览器中，也不用在浏览器中进行编译。

下载到浏览器中的应用代码比动态版本要小得多，并且能立即执行。引导的性能可以得到显著提升。

无论是 JIT 还是 AOT 编译器都会从同一份 `AppModule` 源码中生成一个 `AppModuleNgFactory` 类。JIT 编译器动态、在内存中、在浏览器中创建这个工厂类。AOT 编译器把工厂输出成一个物理文件，也就是我们在静态版本 `main.ts` 中导入的那个。

通常，`AppModule` 不必关心它是如何被引导的。

虽然 `AppModule` 会随着应用而演化，但是 `main.ts` 中的引导代码不会变。这将是我们最后一次关注 `main.ts` 了。

声明指令和组件

应用继续演进。首先加入的是 `HighlightDirective`，一个 **属性型指令**，它会设置所在元素的背景色。

`app/highlight.directive.ts`

```
import { Directive, ElementRef, Renderer } from '@angular/core';

@Directive({ selector: '[highlight]' })
/** Highlight the attached element in gold */
export class HighlightDirective {
  constructor(renderer: Renderer, el: ElementRef) {
    renderer.setStyle(el.nativeElement, 'backgroundColor', 'gold');
    console.log(`* AppRoot highlight called for ${el.nativeElement.tagName}`);
  }
}
```

我们更新 `AppComponent` 的模板，来把该指令附加到标题上：

```
template: '<h1 highlight>{{title}}</h1>'
```

如果我们现在就运行该应用，Angular 将无法识别 `highlight` 属性，并且忽略它。我们必须在 `AppModule` 中声明该指令。

导入 `HighlightDirective` 类，并把它加入该模块的 `declarations` 数组中，就像这样：

```
declarations: [
  AppComponent,
  HighlightDirective,
],
```

添加组件

我们决定将标题重构到独立的 `TitleComponent` 中。该组件的模板绑定到了组件的 `title` 和 `subtitle` 属性中，就像这样：

app/title.component.html

```
<h1 highlight>{{title}} {{subtitle}}</h1>
```

app/title.component.ts

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

@Component({
  moduleId: module.id,
  selector: 'app-title',
  templateUrl: 'title.component.html',
})
export class TitleComponent {
  @Input() subtitle = '';
  title = 'Angular Modules';
}
```

我们重写了 `AppComponent` 来把这个新的 `TitleComponent` 显示到 `<app-title>` 元素中，并使用一个输入型绑定来设置 `subtitle`。

app/app.component.ts (v1)

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

@Component({
  selector: 'my-app',
  template: '<app-title [subtitle]="subtitle"></app-title>'
})
export class AppComponent {
  subtitle = '(v1)';
}
```

除非我们在 `AppModule` 中声明过，否则 Angular 无法识别 `<app-title>` 标签。导入 `TitleComponent` 类，并把它加到模块的 `declarations` 中：

```
declarations: [
  AppComponent,
  HighlightDirective,
  TitleComponent,
],
```

服务提供商

模块是为模块中的所有组件提供服务的最佳途径。

[依赖注入](#) 一章中讲过 Angular 的层次化依赖注入系统，以及如何在组件树的不同层次上通过 [提供商](#) 配置该系统。

模块可以往应用的“根依赖注入器”中添加提供商，让那些服务在应用中到处可用。

很多应用都需要获取当前登录的用户的信息，并且通过 `user` 服务来访问它们。该范例中有一个 `UserService` 的伪实现。

app/user.service.ts

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

@Injectable()
/** Dummy version of an authenticated user service */
export class UserService {
  userName = 'Sherlock Holmes';
}
```

该范例应用会在标题下方为已登录用户显示一条欢迎信息。更新 `TitleComponent` 的模板来显示它。

app/title.component.html

```
<h1 highlight>{{title}} {{subtitle}}</h1>
<p *ngIf="user">
  <i>welcome, {{user}}</i>
<p>
```

更新 `TitleComponent`，为它加入一个构造函数，注入 `UserService` 类，并把组件的 `user` 属性设置为它的实例。

app/title.component.ts

```
import { Component, Input } from '@angular/core';
import { UserService } from './user.service';

@Component({
  moduleId: module.id,
  selector: 'app-title',
  templateUrl: 'title.component.html',
})
export class TitleComponent {
  @Input() subtitle = '';
  title = 'Angular Modules';
  user = '';

  constructor(userService: UserService) {
    this.user = userService.userName;
  }
}
```

我们已经 **定义** 并 **使用了** 该服务。现在，我们通过把它加入 `AppModule` 元数据的 `providers` 属性中，来把它 **提供** 给所有组件使用。

app/app.module.ts (providers)

```
providers: [ UserService ],
```

导入“支持模块”

在没有当前用户时，显然不应该显示欢迎界面。

注意，在修改过的 `TitleComponent` 中，有一个 `*ngIf` 指令在“守卫着”该消息。如果没有当前用户，就没有任何消息。

app/title.component.html (ngIf)

```
<p *ngIf="user">
  <i>Welcome, {{user}}</i>
<p>
```

虽然 `AppModule` 没有声明过 `NgIf` 指令，但该应用仍然能正常编译和运行。为什么这样没问题呢？Angular 的编译器遇到不认识的 HTML 时应该不是忽略就是报错才对。

Angular 能识别 `NgIf` 指令，是因为我们以前导入过它。最初版本的 `AppModule` 就导入了 `BrowserModule`。

app/app.module.ts (imports)

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

导入 `BrowserModule` 会让该模块公开的所有组件、指令和管道在 `AppModule` 下的任何组件模板中直接可用，而不需要额外的繁琐步骤。

更准确的说，`NgIf` 是在来自 `@angular/common` 的 `CommonModule` 中声明的。

`CommonModule` 提供了很多应用程序中常用的指令，包括 `NgIf` 和 `NgFor` 等。

`BrowserModule` 导入了 `CommonModule` 并且 **重新导出** 了它。最终的效果是：只要导入 `BrowserModule` 就自动获得了 `CommonModule` 中的指令。

很多熟悉的 Angular 指令并不属于 `CommonModule`。例如，`NgModel` 和 `RouterLink` 分别属于 Angular 的 `FormsModule` 模块和 `RouterModule` 模块。在使用那些指令之前，我们也必须 **导入** 那些模块。

要解释这一点，我们可以再加入 `ContactComponent` 组件，它是一个表单组件，从 Angular 的 `FormsModule` 中导入了表单支持。

添加 ContactComponent

Angular 表单 是用来管理用户数据输入的最佳方式之一。

`ContactComponnet` 组件展现“联系人编辑器”，它是用 **模板驱动式表单** 实现的。

ANGULAR 表单的风格

我们写 Angular 表单组件时，可以使用 **模板驱动式表单**，也可以使用 **响应式表单**。

该例子中从 `@angular/forms` 中导入了 `FormsModule`，这是因为 `ContactComponent` 组件用的是 **模板驱动式表单**。那些带有 **响应式表单** 组件的模块，应该转而导入 `ReactiveFormsModule`。

`ContactComponent` 的选择器会去匹配名叫 `<app-contact>` 的元素。在 `AppComponent` 模板中 `<app-title>` 的下方添加一个具有此名字的元素：

app/app.component.ts (template)

```
template: `<app-title [subtitle]="subtitle"></app-title>
<app-contact></app-contact>`
```

ContactComponent 还有很多事要做。表单组件通常都是很复杂的。本组件具有它自己的 ContactService 和 **自定义管道** Awesome，以及 HighlightDirective 的另一个版本。

为了方便管理，我们把所有与联系人相关的编程元素都放进 app/contact 目录，并把该组件分解成三个基本成分：HTML、TypeScript 和 CSS 文件：

```
1.  <h2>Contact of {{user_name}}</h2>
2.  <div *ngIf="msg" class="msg">{{msg}}</div>
3.
4.  <form *ngIf="contacts" (ngSubmit)="onSubmit()" #contactForm="ngForm">
5.    <h3 highlight>{{ contact.name | awesome }}</h3>
6.    <div class="form-group">
7.      <label for="name">Name</label>
8.      <input type="text" class="form-control" required
9.        [(ngModel)]="contact.name"
10.       name="name" #name="ngModel" >
11.       <div [hidden]="name.valid" class="alert alert-danger">
12.         Name is required
13.       </div>
14.     </div>
15.     <br>
16.     <button type="submit" class="btn btn-default"
17.       [disabled]="!contactForm.form.valid">Save</button>
18.     <button type="button" class="btn" (click)="next()"
19.       [disabled]="!contactForm.form.valid">Next Contact</button>
20.     <button type="button" class="btn" (click)="newContact()>New
21.       Contact</button>
22.   </form>
```

先来看组件模板。注意模板中部的双向数据绑定 [(ngModel)]。ngModel 是 NgModel 指令的选择器。

虽然 NgModel 是 Angular 指令，但 **Angular 编译器** 并不会识别它，这是因为：(a) AppModule 没有声明过它，并且 (b) 它也没有通过 BrowserModule 被导入过。

退一步说，即使 Angular 有办法识别 ngModel，ContactComponent 也不会表现的像 Angular 表单，因为本组件表单的表单相关的特性（比如有效性验证）还不可用。

导入 FormsModule

把 `FormsModule` 加到 `AppModule` 元数据中的 `imports` 列表中：

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

一旦我们声明了这些新组件、管道和指令，`[(ngModel)]` 绑定就会正常工作，用户的输入也能被 Angular 表单验证了。

不要 把 `NgModel`（或 `FORMS_DIRECTIVES`）加到 `AppModule` 元数据的 `declarations`` 数据中！

这些指令属于 `FormsModule`。组件、指令和管道 **只能** 属于一个模块。

永远不要再次声明属于其它模块的类。

声明联系人的组件、指令和管道

如果我们没有声明该联系人模块的组件、指令和管道，该应用就会失败。更新 `AppModule` 中的 `declarations` 元数据，就像这样：

app/app.module.ts (declarations)

```
declarations: [
  AppComponent,
  HighlightDirective,
  TitleComponent,

  AwesomePipe,
  ContactComponent,
  ContactHighlightDirective
],
```

如果有两个同名指令，都叫做 `HighlightDirective`，该怎么办呢？

我们只要在 import 时使用 `as` 关键字来为第二个指令创建个别名就可以了。

```
import {  
  HighlightDirective as ContactHighlightDirective  
} from './contact/highlight.directive';
```

这解决了在文件中使用指令 **类型** 时的冲突问题，但是还有另一个问题没有解决，我们将在 [后面](#) 讨论它。

提供 ContactService

`ContactComponent` 显示从 `ContactService` 服务中获取的联系人信息，该服务是被 Angular 注入到组件的构造函数中的。

我们必须在某个地方提供该服务。在 `ContactComponent` 中 **可以** 提供它。但是那样一来，它的作用范围就会 **仅** 局限于该组件及其子组件。而我们希望让该服务与其它和联系人有关的组件中共享，稍后我们就会添加哪些组件。

在此应用中，我们选择把 `ContactService` 添加到 `AppModule` 元数据的 `providers` 列表中：

app/app.module.ts (providers)

```
providers: [ ContactService, UserService ],
```

现在，`ContactService` 服务就能被注入进该应用中的任何组件了，就像 `UserService` 一样。

全应用范围的提供商

`ContactService` 的提供商是 **全应用** 范围的，这是因为 Angular 使用该应用的 **根注入器** 注册模块的 `providers`。

从架构上看，`ContactService` 属于“联系人”这个业务领域。 **其它** 领域中的类并不需要知道 `ContactService`，也不会要求注入它。

我们可能会期待 Angular 提供一种 **模块** 范围内的机制来保障此设计。但它没有。与组件不同，Angular 的模块实例并没有它们自己的注入器，所以它们也没有自己的供应商范围。

Angular 是故意这么设计的。Angular 的模块设计，主要目的是扩展应用程序，丰富其模块化能力。

在实践中，服务的范围很少会成为问题。联系人之外的组件不会意外注入 `ContactService` 服务。要想注入 `ContactService`，你得先导入它的 **类型**。而只有联系人组件才会导入 `ContactService`) 类型 _。

参见 [关于此问题的 FAQ](#)，那里有非常详细的讲解。

运行该应用

一切就绪，可以运行该应用及其联系人编辑器了。

应用的文件结构是这样的：

```
app
├── app.component.ts
├── app.module.ts
├── highlight.directive.ts
└── main.ts
├── title.component.(html|ts)
└── user.service.ts
└── contact
    └── awesome.pipe.ts
```

```

  contact.component.(css|html|ts)
  contact.service.ts
  highlight.directive.ts

```

试试这个例子：

The screenshot shows a Plunker project titled "Angular 2 Example - Contact NgModule v.1". The left sidebar displays the project structure:

- Project
 - app
 - contact
 - app.component.1b.ts
 - app.module.1b.ts
 - highlight.directive.ts
 - main.1b.ts
 - title.component.html
 - title.component.ts
 - user.service.ts
 - index.html
 - README.md
 - styles.css
 - systemjs.config.js
 - tsconfig.json

The right panel shows a preview of the application. It features a large play button icon. The main title is "Angular Modules (v1)". Below it, the text "Welcome, Sherlock Holmes" and "Contact of Sherlock" is displayed. A prominent button says "Click to run". Below the button, there is a form with a teal header containing the text "Awesome Sam Spade". The form has a "Name" field containing "Sam Spade", and buttons for "Save", "Next Contact", and "New Contact". At the bottom, it says "Plunker © (2016) is created by Geoffrey Goodman" with social sharing icons.

解决指令冲突

以前我们在声明联系人的 `HighlightDirective` 指令时遇到了问题，因为在应用程序一级已经有了一个 `HighlightDirective` 类。

两个指令都用同一个名字让人不爽。

在查找它们的选择器时，它们都试图用不同的颜色来高亮所依附的元素。

```
1. import { Directive, ElementRef, Renderer } from '@angular/core';
```

```

2.
3. @Directive({ selector: '[highlight]' })
4. /**
5.  * Highlight the attached element in gold */
6. export class HighlightDirective {
7.   constructor(renderer: Renderer, el: ElementRef) {
8.     renderer.setStyle(el.nativeElement, 'backgroundColor',
9.       'gold');
10.    console.log(
11.      `* AppRoot highlight called for ${el.nativeElement.tagName}`);
12.  }
13. }

```

Angular 会只用它们中的一个吗？不会。所有指令都声明在该模块中，所以 **这两个指令都会被激活**。

当两个指令在同一个元素上争相设置颜色时，后声明的那个会胜出，因为它对 DOM 的修改覆盖了前一个。在该例子中，联系人的 `HighlightDirective` 把应用标题的文本染成了蓝色，而我们原本期望它保持金色。

真正的问题在于，有 **两个不同的类** 试图做同一件事。

多次导入 **同一个** 指令是没问题的，Angular 会移除重复的类，而只注册一次。

但是这里实际上有两个不同的类，定义在不同的文件中，只是恰好有相同的名字。

从 Angular 的角度看，两个类并没有重复。Angular 会同时保留这两个指令，并让它们依次修改同一个 HTML 元素。

至少，应用仍然编译通过了。如果我们使用相同的选择器定义了两个不同的组件类，并指定了同一个元素标记，编译器就会报错说它无法在同一个 DOM 位置插入两个不同的组件。

真乱！

我们可以通过创建特性模块来消除组件与指令的冲突。 特性模块可以把来自一个模块中的声明和来自另一个的区隔开。

特性模块

该应用还不大，但是已经在受结构方面的问题困扰了。

- 随着一个个类被加入应用中，根模块 `AppModule` 变大了，并且还会继续变大。
- 我们遇到了指令冲突。

联系人模块的 `HighlightDirective` 在 `AppModule` 中声明的 `HighlightDirective` 的基础上进行了二次上色。 并且，它染了应用标题文字的颜色，而不仅仅是 `ContactComponent` 中的。

- 该应用在联系人和其它特性区之间缺乏清晰的边界。 这种缺失，导致难以在不同的开发组之间分配职责。

我们用 **特性模块** 技术来缓解此问题。

特性模块

特性模块 是带有 `@NgModule` 装饰器及其元数据的类，就像根模块中一样。 特性模块的元数据和根模块的元数据携带的属性是一样的。

根模块和特性模块还共享着相同的执行环境。 它们共享着同一个依赖注入器，这意味着某个模块中定义的服务在所有模块中也都能用。

它们在技术上有两个显著的不同点：

1. 我们 **引导** 根模块来 **启动** 应用，但 **导入** 特性模块来 **扩展** 应用。
2. 特性模块可以对其他模块暴露或隐藏自己的实现。

此外，特性模块主要还是从它的设计意图上来区分。

特性模块用来提供一组紧密相关的功能。 聚焦于应用的某个业务领域、用户的某个工作流、某个基础设施（表单、HTTP、路由），或一组相互关联的工具。

虽然这些都能在根模块中做，但特性模块可以帮助我们把应用切分成具有特定关注点和目标的不同区域。

特性模块通过自己提供的服务和它决定对外共享的那些组件、指令、管道来与根模块等其它模块协同工作。

在下一节，我们从根模块中把与联系人有关的功能切分到专门的特性模块中。

把联系人做成特性模块

把与联系人有关的这些元素重构到“联系人”特性模块中很简单。

1. 在 `app/contact` 目录下创建 `ContactModule`。
2. 把联系人相关的元素从 `AppModule` 移到 `ContactModule` 中。
3. 把导入 `BrowserModule` 改为导入 `CommonModule`。
4. 在 `AppModule` 中导入 `ContactModule`。

`AppModule` 是唯一有改变的 **已经存在** 的类，不过我们还会添加一个新文件。

添加 ContactModule

下面是新的 `ContactModule`

app/contact/contact.module.ts

```

1. import { NgModule }           from '@angular/core';
2. import { CommonModule }       from '@angular/common';
3. import { FormsModule }        from '@angular/forms';
4.
5. import { AwesomePipe }        from './awesome.pipe';
6.
7. import
8.   { ContactComponent }      from './contact.component';

```

```

9. import { ContactService }      from './contact.service';
10. import { HighlightDirective } from './highlight.directive';
11.
12. @NgModule({
13.   imports:      [ CommonModule, FormsModule ],
14.   declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],
15.   exports:       [ ContactComponent ],
16.   providers:    [ ContactService ]
17. })
18. export class ContactModule { }

```

我们从 `AppModule` 中把与联系人有关的 `import` 语句和 `@NgModule` 中与联系人有关的内容拷贝到 `ContactModule` 中。

我们还 **导入** 了 `FormsModule`，来满足“联系人”组件的需求。

当前模块不会继承其它模块中对组件、指令或管道的访问权。`AppModule` 中的 `imports` 与 `ContactModule` 的 `imports` 互不相干。如果 `ContactComponent` 要绑定到 `[(ngModel)]`，它所在的 `ContactModule` 必需导入 `FormsModule`。

我们还用 `CommonModule` 替换了 `BrowserModule`，其中缘由参见 [这条常见问题](#)。

我们在该模块的 `declarations` 中 **声明** 了“联系人”的组件、指令和管道。

我们 **导出了** `ContactComponent`，这样其它模块只要导入了 `ContactModule`，就可以在其组件模板中使用来自 `ContactModule` 中的组件了。

“联系人”中声明的所有其它类默认都是私有的。`AwesomePipe` 和 `HighlightDirective` 对应用的其它部分则是不可见的。所以 `HighlightDirective` 不能把 `AppComponent` 的标题文字染色。

重构 AppModule

返回 `AppModule`，移除所有与联系人有关的内容。

删除属于联系人的 `import` 语句。删除联系人的 `declarations` 和 `providers` 。从 `imports` 列表中移除 `FormsModule` （`AppComponent` 并不需要它）。只保留本应用的根一级需要的那些类。

然后，导入 `ContactModule`，以便应用仍然可以显示从这里导出的 `ContactComponent`。

下面是 `AppModule` 重构完的版本与之前版本的一对一对比。

```

1. import { NgModule }             from '@angular/core';
2. import { BrowserModule }       from '@angular/platform-browser';
3.
4. /* App Root */
5. import
6.   { AppComponent }            from './app.component';
7. import { HighlightDirective }  from './highlight.directive';
8. import { TitleComponent }     from './title.component';
9. import { UserService }        from './user.service';
10.
11. /* Contact Imports */
12. import
13.   { ContactModule }          from './contact/contact.module';
14.
15. @NgModule({
16.   imports:      [ BrowserModule, ContactModule ],
17.   declarations: [ AppComponent, HighlightDirective, TitleComponent ],
18.   providers:    [ UserService ],
19.   bootstrap:   [ AppComponent ],
20. })
21. export class AppModule { }
```

增强功能

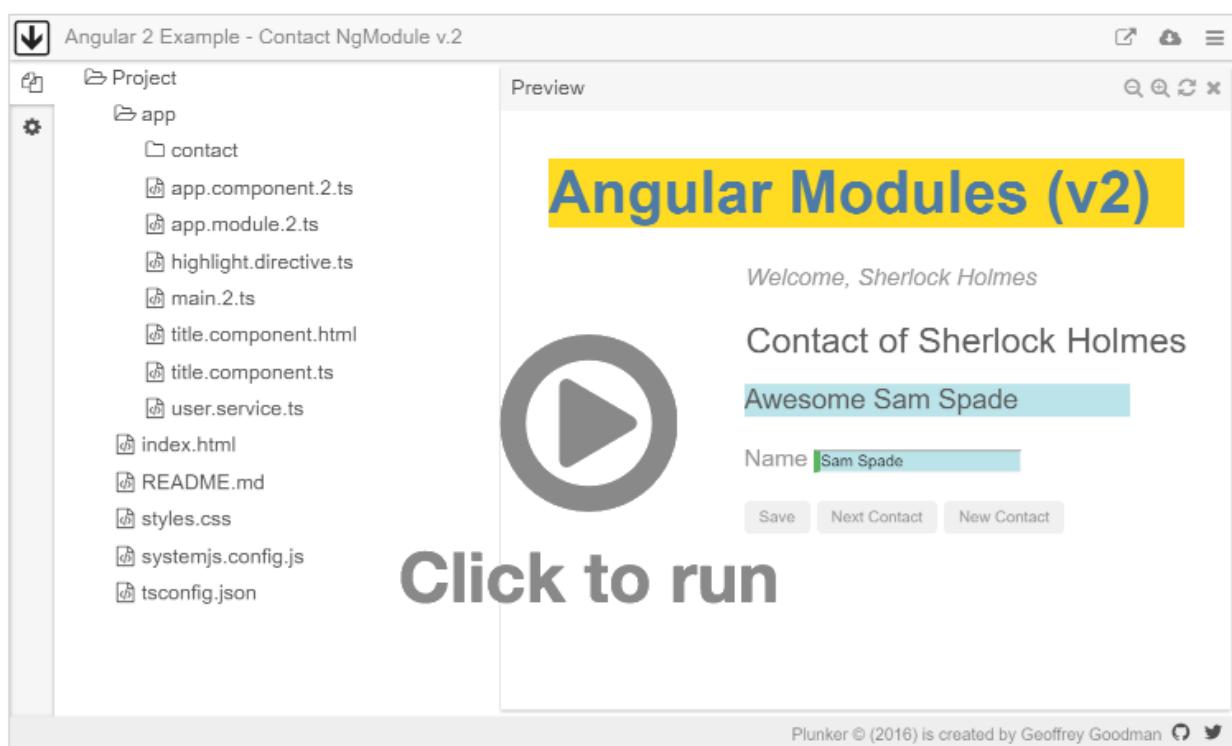
修改后的 `AppModule` 有很令人喜欢的特征：

- 它不会再随着 **联系人** 的领域扩张而修改。
- 只有当添加新模块时才需要修改它。

- 它也变得简单了：

- 更少的 `import` 语句
- 不再导入 `FormsModule`
- 没有与联系人有关的声明
- 没有 `ContactService` 提供商
- 没有 `HighlightDirective` 冲突

试试范例的 `ContactModule` 版。



用路由器实现延迟（Lazy）加载

“英雄管理局”这个例子应用继续成长。它又增加了两个模块，一个用来管理雇佣的英雄，另一个用来匹配英雄与危机。这两个模块都还处于前期开发阶段。它们对于整个故事来说无关紧要，这里我们就不逐行讨论了。

到 [在线例子](#) 试用并下载当前版本的完整代码。

当前应用中还有一些方面值得深入探讨。

- 该应用有三个特性模块：联系人（Contact）、英雄（Hero）和危机（Crisis）。
- Angular 路由器帮助用户在这些模块之间导航。
- ContactComponent 组件是应用启动时的默认页。
- ContactModule 仍然会在应用启动时被主动加载。
- HeroModule 和 CrisisModule 会被惰性加载。

我们从这个 AppComponent 新模板的顶部看起：标题、三个链接和 <router-outlet> 。

app/app.component.ts (v3 - Template)

```
template: `
  <app-title [subtitle]="subtitle"></app-title>
  <nav>
    <a routerLink="contact" routerLinkActive="active">Contact</a>
    <a routerLink="crisis" routerLinkActive="active">Crisis
      Center</a>
    <a routerLink="heroes" routerLinkActive="active">Heroes</a>
  </nav>
  <router-outlet></router-outlet>
`
```

<app-contact> 元素不见了，改成了路由到 联系人 页。

AppModule 被的修改很大：

app/app.module.ts (v3)

```
1. import { NgModule }           from '@angular/core';
2. import { BrowserModule }     from '@angular/platform-browser';
```

```
3.  
4.  /* App Root */  
5.  import { AppComponent }      from './app.component.3';  
6.  import { HighlightDirective } from './highlight.directive';  
7.  import { TitleComponent }    from './title.component';  
8.  import { UserService }      from './user.service';  
9.  
10. /* Feature Modules */  
11. import { ContactModule }   from './contact/contact.module.3';  
12.  
13. /* Routing Module */  
14. import { AppRoutingModule } from './app-routing.module.3';  
15.  
16. @NgModule({  
17.   imports:      [  
18.     BrowserModule,  
19.     ContactModule,  
20.     AppRoutingModule  
21.   ],  
22.   providers:    [ UserService ],  
23.   declarations: [ AppComponent, HighlightDirective, TitleComponent ],  
24.   bootstrap:    [ AppComponent ]  
25. })  
26. export class AppModule { }
```

有些文件名带有 `.3` 扩展名，用来和以前 / 以后的版本区分开。我们会在适当的时机解释它们的差异。

该模块仍然要导入 `ContactModule` 模块，以便在应用启动时加载它的路由和组件。

该模块不用导入 `HeroModule` 或 `CrisisModule`。它们将在用户导航到其中的某个路由时，被异步获取并加载。

与第二版相比，最值得注意的修改是 `imports` 中那个额外的 `AppRoutingModule` 模块。`AppRoutingModule` 是一个 **路由模块** 用来处理应用的路由。

应用路由

app/app-routing.module.ts

```

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

export const routes: Routes = [
  { path: '', redirectTo: 'contact', pathMatch: 'full' },
  { path: 'crisis', loadChildren:
    'app/crisis/crisis.module#CrisisModule' },
  { path: 'heroes', loadChildren: 'app/hero/hero.module#HeroModule' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

路由器有 [专门的章节](#) 做了深入讲解，所以这里我们跳过细节，而是专注于它和 Angular 模块的协作。

该文件定义了三个路由。

第一个路由把空白 URL（比如：`http://host.com/`）重定向到了另一个路径为 `contact` 的路由（比如：`http://host.com/contact`）。

`contact` 路由并不是在这里定义的，而是定义在 [联系人](#) 特性区自己的路由文件 `contact.routing.ts` 中。对于带有路由组件的特性模块，其标准做法就是让它们定义自己的路由。稍后我们就会看到这些。

另外两个路由使用惰性加载语法来告诉路由器要到哪里去找这些模块。

```

{ path: 'crisis', loadChildren:
  'app/crisis/crisis.module#CrisisModule' },
{ path: 'heroes', loadChildren: 'app/hero/hero.module#HeroModule' }
```

惰性加载模块的位置是 **字符串** 而不是 **类型**。在本应用中，该字符串同时标记出了模块 **文件** 和模块 **类**，两者用 `#` 分隔开。

RouterModule.forRoot

`RouterModule` 类的 `forRoot` 静态方法和提供的配置，被添加到 `imports` 数组中，提供该模块的路由信息。

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

该方法返回的 `AppRoutingModule` 类是一个 `Routing Module`，它同时包含了 `RouterModule` 指令和用来生成配置好的 `Router` 的依赖注入提供商。

这个 `AppRoutingModule` **仅仅** 是给应用程序的 **根** 模块使用的。

永远不要在特性路由模块中调用 `RouterModule.forRoot`！

回到根模块 `AppModule`，把这个 `routing` 对象添加到根模块的 `imports` 列表中，该应用就可以正常导航了。

app/app.module.ts (imports)

```
imports: [
  BrowserModule,
  ContactModule,
  AppRoutingModule
],
```

路由到特性模块

`app/contact` 目录中也有一个新文件 `contact-routing.module.ts`。它定义了我们前面提到过的 `contact` 路由，并提供了 `ContactRoutingModule`，就像这样：

app/contact/contact-routing.module.ts (routing)

```
@NgModule({
  imports: [RouterModule.forChild([
    { path: 'contact', component: ContactComponent }
  ]),
  exports: [RouterModule]
})
export class ContactRoutingModule {}
```

这次我们要把路由列表传给 `RouterModule` 的 `forChild` 方法。该方法会为特性模块生成另一种对象。

总是在特性路由模块中调用 `RouterModule.forChild`。

当需要为根模块和特性模块分别提供不同的 `import` 值时，`forRoot` 和 `forChild` 也可以作为约定俗成的方法名。虽然 Angular 无法识别它们，但是 Angular 开发人员可以。

当你要写类似的模块，来为根模块和特性模块分别导出一些 **声明** 和服务时，请 **遵循这个约定**。

`ContactModule` 已经做了两个微小但重要的细节改动：

```
1.  @NgModule({
2.    imports:      [ CommonModule, FormsModule, ContactRoutingModule ],
3.    declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],
4.    providers:    [ ContactService ]
5.  })
```

```
6. export class ContactModule { }
```

1. 它从 `contact-routing.module.ts` 中导入了 `ContactRoutingModule` 对象
2. 它不再导出 `ContactComponent`

现在我们改成了通过路由器导航到 `ContactComponent`，所以也就没有理由公开它了。它也不再需要选择器（`selector`）。也没有模板会再引用 `ContactComponent`。它从 `AppComponent` 模板中彻底消失了。

路由到惰性加载的模块

惰性加载的 `HeroModule` 和 `CrisisModule` 与其它特性模块遵循同样的规则。它们和“主动加载”的 `ContactModule` 看上去没有任何区别。

`HeroModule` 比 `CrisisModule` 略复杂一些，因此更适合用作范例。它的文件结构如下：

```
hero
├── hero-detail.component.ts
├── hero-list.component.ts
├── hero.component.ts
└── hero.module.ts
    └── hero-routing.module.ts
        └── hero.service.ts
    └── highlight.directive.ts
```

如果你读过 [路由](#) 章，那么对这个子路由的场景应该觉得很熟悉。`HeroComponent` 是本特性区的顶级组件和路由宿主。模板带有 `<router-outlet>` 指令，它或者显示英雄列表（`HeroList`）或者显示所选英雄的编辑器（`HeroDetail`）。这两个组件都把获取和保存数据的任务委托给 `HeroService` 执行。

还有 [另一个](#) `HighlightDirective` 指令，它用另一种方式为元素染色。我们还应该 [做点什么](#) 来消除这种不必要的重复和不一致性。不过目前先暂时容忍这个问题。

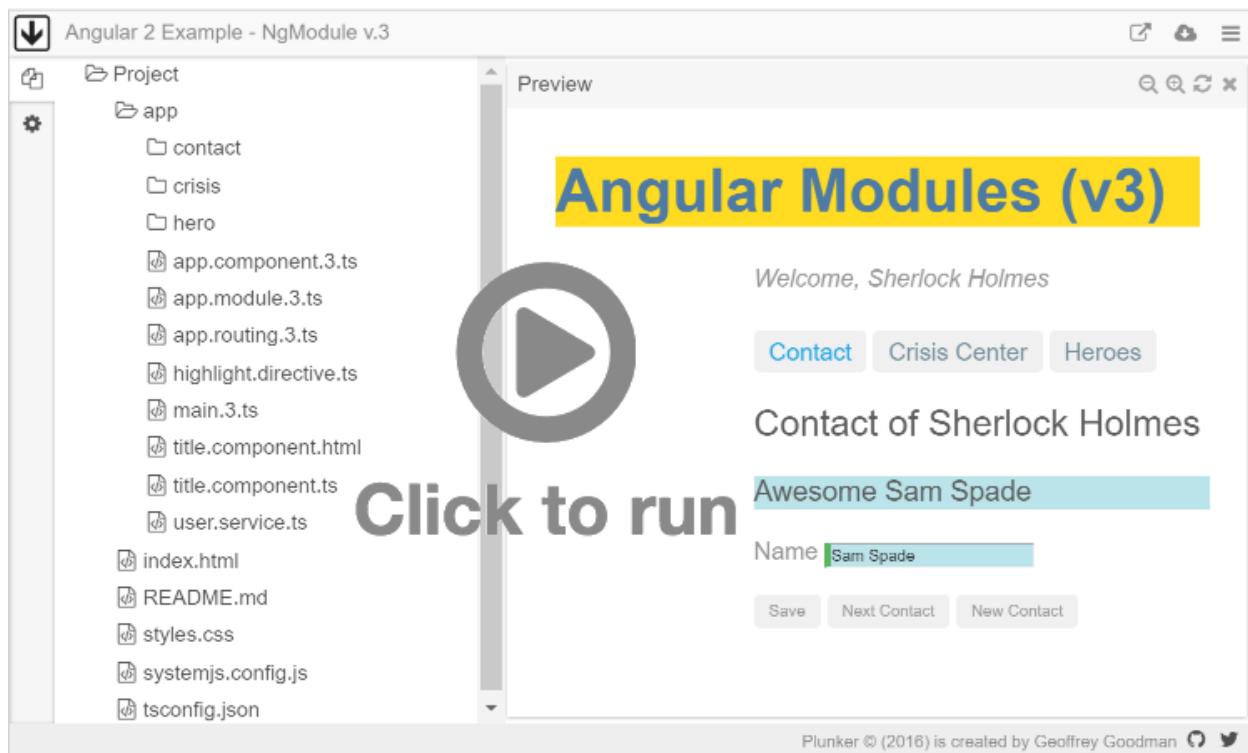
HeroModule 是特性模块，与其它的没什么不同。

app/hero/hero.module.ts (class)

```
@NgModule({
  imports: [ CommonModule, FormsModule, HeroRoutingModule ],
  declarations: [
    HeroComponent, HeroDetailComponent, HeroListComponent,
    HighlightDirective
  ]
})
export class HeroModule { }
```

它导入了 `FormsModule`，因为 `HeroDetailComponent` 的模板中绑定到了 `[(ngModel)]`。像 `ContactModule` 和 `CrisisModule` 中一样，它还从 `hero-routing.module.ts` 中导入了 `HeroRoutingModule`。

`CrisisModule` 和本模块非常像，我们不再赘述。



共享模块

本应用在继续演进中。让我们感到不爽的是：这里有 `HighlightDirective` 的三个不同版本。还有一大堆其它乱七八糟的东西堆在 `app` 目录这一级，我们得把它们清出去。

我们添加 `SharedModule` 来存放这些公共组件、指令和管道，并且共享给那些想用它们的模块。

- 创建 `app/shared` 目录
- 把 `AwesomePipe` 和 `HighlightDirective` 从 `app/contact` 移到 `app/shared` 中。
- 从 `app/` 和 `app/heroes` 目录中删除 `HighlightDirective` 类
- 创建 `SharedModule` 类来管理这些共享的素材
- 更新启动特性模块，让它们导入 `SharedModule`

这些都是普通的任务，也容易解决。`SharedModule` 的代码如下：

app/app/shared/shared.module.ts

```

1. import { NgModule }           from '@angular/core';
2. import { CommonModule }       from '@angular/common';
3. import { FormsModule }        from '@angular/forms';
4.
5. import { AwesomePipe }        from './awesome.pipe';
6. import { HighlightDirective } from './highlight.directive';
7.
8. @NgModule({
9.   imports:      [ CommonModule ],
10.  declarations: [ AwesomePipe, HighlightDirective ],
11.  exports:      [ AwesomePipe, HighlightDirective,
12.                  CommonModule, FormsModule ]
13. })
14. export class SharedModule { }
```

值得注意的有：

- 它导入了 `CommonModule`，这是因为它的组件需要这些公共指令。
- 正如我们所期待的，它声明并导出了工具性的管道、指令和组件类。
- 它重新导出了 `CommonModule` 和 `FormsModule`

重新导出其它模块

当回顾应用程序时，我们注意到很多需要 `SharedModule` 的组件也同时用到了来自 `CommonModule` 的 `NgIf` 和 `NgFor` 指令，并且还通过来自 `FormsModule` 的 `[(ngModel)]` 指令绑定到了组件的属性。那些声明这些组件的模块将不得不同时导入 `CommonModule`、`FormsModule` 和 `SharedModule`。

通过让 `SharedModule` 重新导出 `CommonModule` 和 `FormsModule` 模块，我们可以消除这种重复。于是导入 `SharedModule` 的模块也同时 **免费** 获得了 `CommonModule` 和 `FormsModule`。

实际上，`SharedModule` 本身所声明的组件没绑定过 `[(ngModel)]`，那么，严格来说 `SharedModule` 并不需要导入 `FormsModule`。

这时 `SharedModule` 仍然可以导出 `FormsModule`，而不需要先把它列在 `imports` 中。

为什么 TitleComponent 没有被共享

设计 `SharedModule` 的目的在于让常用的组件、指令和管道可以被用在 **很多** 其它模块的组件模板中。

而 `TitleComponent` **只被** `AppComponent` 用了一次，因此没必要共享它。

为什么 UserService 没有被共享

虽然很多组件都共享着同一个服务 **实例**，但它们是靠 Angular 的依赖注入体系实现的，而不是模块体系。

例子中的很多组件都注入了 `UserService`。在整个应用程序中，**只应该有一个** `UserService` 的实例，并且它 **只应该有一个** 提供商。

`UserService` 是全应用级单例。我们不希望每个模块都各自有它的实例。而如果由 `SharedModule` 提供 `UserService`，就会导致 **铁板钉钉的危险**。

不要 在共享模块中把应用级单例添加到 `providers` 中。否则如果一个惰性加载模块导入了此共享模块，就会导致它自己也生成一份此服务的实例。

核心（Core）模块

现在，我们的根目录下只剩下 `UserService` 和 `TitleComponent` 这两个被根组件 `AppComponent` 用到的类没有清理了。但正如我们已经解释过的，它们无法被包含在 `SharedModule` 中。

不过，我们可以把它们收集到单独的 `CoreModule` 中，并且 **只在应用启动时导入它一次，而不会在其它地方导入它**。

步骤：

- 创建 `app/core` 文件夹
- 把 `UserService` 和 `TitleComponent` 从 `app` 移到 `app/core` 中
- 创建 `CoreModule` 类来管理这些核心素材
- 更新 `AppRoot` 模块，使其导入 `CoreModule` 模块

这些都是一些熟悉的普通任务。最有趣的是 `CoreModule`：

app/app/core/core.module.ts

```
1. import {
2.   ModuleWithProviders, NgModule,
3.   optional, skipSelf }      from '@angular/core';
4.
5. import { CommonModule }      from '@angular/common';
6.
7. import { TitleComponent }    from './title.component';
8. import { UserService }       from './user.service';
9. @NgModule({
10.   imports:      [ CommonModule ],
11.   declarations: [ TitleComponent ],
12.   exports:       [ TitleComponent ],
13.   providers:     [ UserService ]
14. })
15. export class CoreModule {  
16. }
```

我们正在从 Angular 核心库中导入一些从未用到的符号，稍后我们会接触它们。

我们对 `@NgModule` 的元数据应该很熟悉。由于该模块 **拥有** `TitleComponent`，所以我们声明了它。由于 `AppComponent`（位于 `AppModule` 模块）在模板中显示了这个标题，所以我们导出了它。由于 `TitleComponent` 需要用到 Angular 的 `NgIf` 指令，所以我们导入了 `CommonModule`。

`CoreModule` **提供了** `UserService`。Angular 在该应用的“根注入器”中注册了它的提供商，导致这份 `UserService` 的实例在每个需要它的组件中都是可用的，无论那个组件时主动加载的还是惰性加载的。

没必要？

这个场景设计的是有点生硬。该应用太小了，所以其实并不需要拆分出单独的服务文件和小型的、一次性的组件。

把 `TitleComponent` 放在根目录中其实也无所谓。即使我们决定把 `UserService` 文件挪到 `app/core` 目录中，根 `AppModule` 也仍然可以自己注册 `UserService`（就像现在这样）。

但真实的应用要考虑很多。它们有只用于 `AppComponent` 的模板中的一些一次性的组件（比如：加载动画、消息浮层和模态对话框等）。我们不用在其它地方导入它们，因此没必要 **共享** 它们。然而如果把它们留在根目录，还是显得太大、太乱了。

应用通常还有很多像这里的 `UserService` 这样的单例服务。当程序启动时，每个服务都只能在应用的“根注入器”中 **注册一次**。

当很多组件在它们的构造函数中注入这些服务时（因此也需要用 JavaScript 的 `import` 语句来导入它们的符号），任何组件或模块自身都不应该定义或重新创建这些服务。因为它们的 **提供商** 不是共享的。

因此我们建议把这些一次性的类收集到 `CoreModule` 中，并且隐藏它们的实现细节。简化之后的根模块 `AppModule` 导入 `CoreModule` 来获取其能力。记住，根模块是整个应用的总指挥，不应该插手过多细节。

清理

我们已经重构完 `CoreModule` 和 `SharedModule`，现在开始清理其它模块。

清理 AppModule

这里是更新后的 `AppModule` 与其第三版本的对比：

```
1. import { NgModule }           from '@angular/core';
2. import { BrowserModule }    from '@angular/platform-browser';
3.
4. /* App Root */
5. import { AppComponent }      from './app.component';
6.
7. /* Feature Modules */
8. import { ContactModule }    from './contact/contact.module';
9. import { CoreModule }        from './core/core.module';
10.
11. /* Routing Module */
12. import { AppRoutingModule } from './app-routing.module';
13.
14. @NgModule({
15.   imports: [
16.     BrowserModule,
17.     ContactModule,
18.     CoreModule,
19.     AppRoutingModule
20.   ],
21.   declarations: [ AppComponent ],
22.   bootstrap:    [ AppComponent ]
23. })
24. export class AppModule { }
```

注意 `AppModule` 已经变得：

- 更小了。因为很多 `app/root` 下的类被移到了其它模块中。
- 更稳定了。因为我们以后会在其它模块中添加组件和服务提供商，而不是这里。
- 导入其它模块并把任务委托给它们，而不是亲力亲为。
- 聚焦于自己的主要任务：总指挥整个应用程序。

清理 ContactModule

这里是新的 `ContactModule` 与以前版本的对比：

```
1. import { NgModule }           from '@angular/core';
2. import { SharedModule }       from '../shared/shared.module';
3.
4. import { ContactComponent }   from './contact.component';
5. import { ContactService }    from './contact.service';
6. import { ContactRoutingModule } from './contact-routing.module';
7.
8. @NgModule({
9.   imports:      [ SharedModule, ContactRoutingModule ],
10.  declarations: [ ContactComponent ],
11.  providers:    [ ContactService ]
12. })
13. export class ContactModule { }
```

注意：

- `AwesomePipe` 和 `HighlightDirective` 不见了。
- 导入 `SharedModule`，不再导入 `CommonModule` 和 `FormsModule`。
- 这个新版本更加精简和干净了。

用 CoreModule.forRoot 配置核心服务

那些为应用添加服务提供商的模块，也可以同时提供配置那些提供商的功能。

按照约定，模块的静态方法 `forRoot` 可以同时提供并配置服务。它接收一个服务配置对象，并返回一个 `ModuleWithProviders`。这个简单对象具有两个属性：

- `ngModule` - `CoreModule` 类
- `providers` - 配置好的服务提供商

根模块 `AppModule` 会导入 `CoreModule` 类并把它的 `providers` 添加到 `AppModule` 的服务提供商中。

更精确的说法是，Angular 会先累加所有导入的提供商，**然后才** 把它们追加到 `@NgModule.providers` 中。这样可以确保我们显式添加到 `AppModule` 中的那些提供商总是优先于从其它模块中导入的提供商。

现在添加 `CoreModule.forRoot` 方法，以便配置核心中的 `UserService`。

我们曾经用一个可选的、被注入的 `UserServiceConfig` 服务扩展过核心的 `UserService` 服务。如果有 `UserServiceConfig`，`UserService` 就会据此设置用户名。

app/core/user.service.ts (constructor)

```
constructor(@Optional() config: UserServiceConfig) {
  if (config) { this._userName = config.userName; }
}
```

这里的 `CoreModule.forRoot` 接收 `UserServiceConfig` 对象：

app/core/core.module.ts (forRoot)

```
static forRoot(config: UserServiceConfig): ModuleWithProviders {
  return {
    providers: [
      { provide: 'UserService', useValue: config }
    ]
  }
}
```

```

@NgModule: CoreModule,
providers: [
  {provide: UserServiceConfig, useValue: config }
]
};

}

```

最后，我们在 `AppModule` 的 `imports` 列表中调用它。

app/app.module.ts (imports)

```

imports: [
  BrowserModule,
  ContactModule,
  CoreModule.forRoot({userName: 'Miss Marple'}),
  AppRoutingModule
],

```

该应用不再显示默认的“ Sherlock Holmes ”，而是用“ Miss Marple ”作为用户名称。

只在应用的根模块 `AppModule` 中调用 `forRoot` 。 如果在其它模块（特别是惰性加载模块）中调用它则违反了设计意图，并会导致运行时错误。

别忘了导入 其返回结果，而且不要把它添加到 `@NgModule` 的其它任何列表中。

禁止多次导入 CoreModule

只有根模块 `AppModule` 才能导入 `CoreModule` 。 如果惰性加载模块导入了它，就会出**问题**。

我们可以**祈祷**任何开发人员都不会犯错。但是最好还是对它进行一些保护，以便让它“尽快出错”。只要把下列代码添加到 `CoreModule` 的构造函数中就可以了。

```
constructor (@Optional() @SkipSelf() parentModule: CoreModule) {
  if (parentModule) {
    throw new Error(
      'CoreModule is already loaded. Import it in the AppModule only');
  }
}
```

这个构造函数会要求 Angular 把 `CoreModule` 注入自身。这看起来像一个危险的循环注入。

确实，如果 Angular 在 **当前** 注入器中查阅 `CoreModule`，这确实会是一个循环引用。不过，`@SkipSelf` 装饰器意味着“在当前注入器的所有祖先注入器中寻找 `CoreModule`。”

如果该构造函数在我们所期望的 `AppModule` 中运行，就没有任何祖先注入器能够提供 `CoreModule` 的实例，于是注入器会放弃查找。

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。

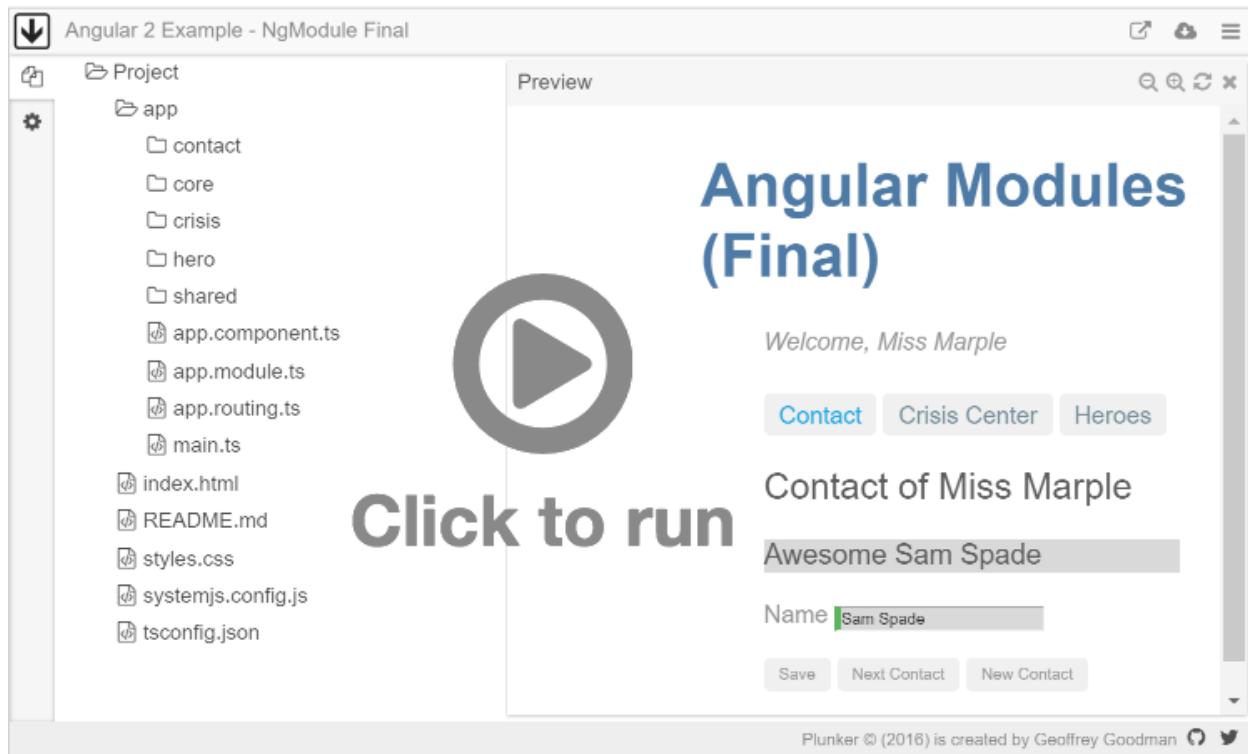
如果我们错误的把 `CoreModule` 导入了一个惰性加载模块（比如 `HeroModule`）中，那就不一样了。

Angular 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。

总结

完工！你可以到下面的在线例子中试验它，并下载最终版本的全部源码。



常见问题 (FAQ)

现在，你已经理解了 Angular 的模块。可能你还会对烹饪宝典中的 [Angular 模块常见问题](#) 感兴趣，它解答了很多关于设计和实现方面的问题。

动画

Angular 动画系统指南。

动画是现代 Web 应用设计中一个很重要的方面。我们希望用户界面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。

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。我们也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。

至于其它浏览器，就需要一个填充库 (polyfill) 了。你可以 [从这里获取](#) `web-animations.min.js`，并把它加入你的页面中。

目录

- [范例：在两个状态之间进行转场 \(Transition\)](#)
- [状态与转场](#)
- [范例：进场与离场](#)
- [范例：从其它状态进场与离场](#)
- [可动的 \(Animatable\) 属性与单位](#)

- 自动属性值计算
- 动画时间线 (Timing)
- 基于关键帧 (Keyframes) 的多阶段动画
- 并行动画组 (Group)
- 动画回调

本章中引用的这个例子可以到 [在线例子](#) 去体验。

快速起步范例：在两个状态间转场

我们来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。在添加动画之前，先引入一些与动画有关的函数：

```
import {  
  Component,  
  Input,  
  trigger,  
  state,  
  style,  
  transition,  
  animate  
} from '@angular/core';
```

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。

```

  animations: [
    trigger('heroState', [
      state('inactive', style({
        backgroundColor: '#eee',
        transform: 'scale(1)'
      })),
      state('active', style({
        backgroundColor: '#cf8dc',
        transform: 'scale(1.1)'
      })),
      transition('inactive => active', animate('100ms ease-in')),
      transition('active => inactive', animate('100ms ease-out'))
    ])
  ]

```

在这个例子中，我们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。

我们刚刚定义了一个动画，但它还没有被用到任何地方。要想使用它，可以在模板中用 `[@triggerName]` 语法来把它附加到一个或多个元素上。

```

template: `
<ul>
  <li *ngFor="let hero of heroes"
    [@heroState]="hero.state"
    (click)="hero.toggleState()">
    {{hero.name}}
  </li>
</ul>
`,

```

这里，我们把该动画触发器添加到了由 `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`。这个值应该或者是 `inactive` 或者是 `active`，因为我们刚刚为它们俩定义过动画状态。

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：

```
1. import {
2.   Component,
3.   Input,
4.   trigger,
5.   state,
6.   style,
7.   transition,
8.   animate
9. } from '@angular/core';
10.
11. import { Heroes } from './hero.service';
12.
13. @Component({
14.   moduleId: module.id,
15.   selector: 'hero-list-basic',
16.   template: `
17.     <ul>
18.       <li *ngFor="let hero of heroes"
19.           [@herostate]="hero.state"
20.           (click)="hero.toggleState()">
21.         {{hero.name}}
22.       
23.     </ul>
24.   `,
25.   styleUrls: ['hero-list.component.css'],
26.   animations: [
27.     trigger('herostate', [
28.       state('inactive', style({
29.         backgroundColor: '#eee',
30.         transform: 'scale(1)'
31.       })),
32.       state('active', style({
33.         backgroundColor: '#cf8dc',
34.         transform: 'scale(1.1)'
35.       })),
36.     ])
37.   ]
38. }
```

```

36.         transition('inactive => active', animate('100ms ease-in')),
37.         transition('active => inactive', animate('100ms ease-out'))
38.     ])
39.   ]
40. }
41. export class HeroListBasicComponent {
42.   @Input() heroes: Heroes;
43. }

```

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，基于英雄对象的逻辑状态，我们使用了 `'active'` 和 `'inactive'` 这两种状态。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，我们得能从组件模板中读取它。

我们可以为每个动画状态定义了**一组样式**：

```

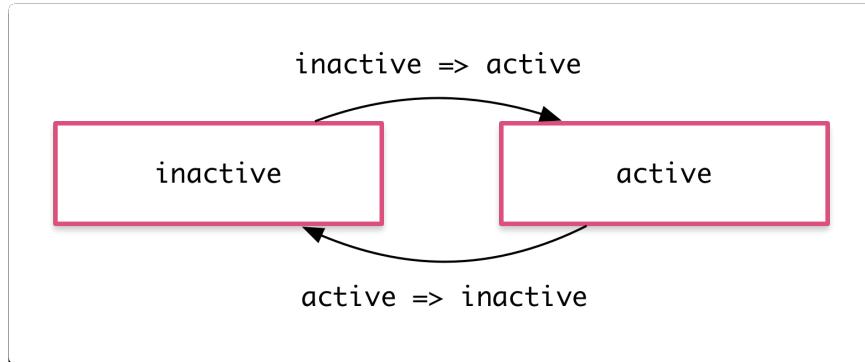
state('inactive', style({
  backgroundColor: '#eee',
  transform: 'scale(1)'
}),
state('active', style({
  backgroundColor: '#cf8dc',
  transform: 'scale(1.1)'
}),

```

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：

```
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：

```
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：

```
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。

```
transition('inactive => active', [
  style({
    backgroundColor: '#cf8dc',
    transform: 'scale(1.3)'
  }),
  animate('80ms ease-in', style({
    backgroundColor: '#eee',
    transform: 'scale(1)'
```

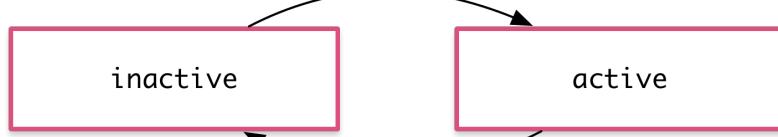
```
    }})
  ],
}
```

* (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。

```
inactive => active
inactive => *
*      => active
*      => *
```

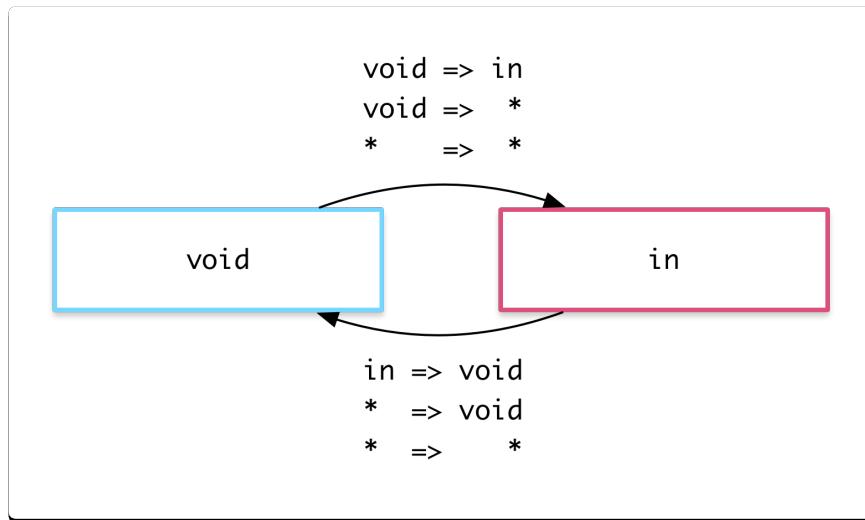


```
active => inactive
active => *
*      => inactive
*      => *
```

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

```

  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：

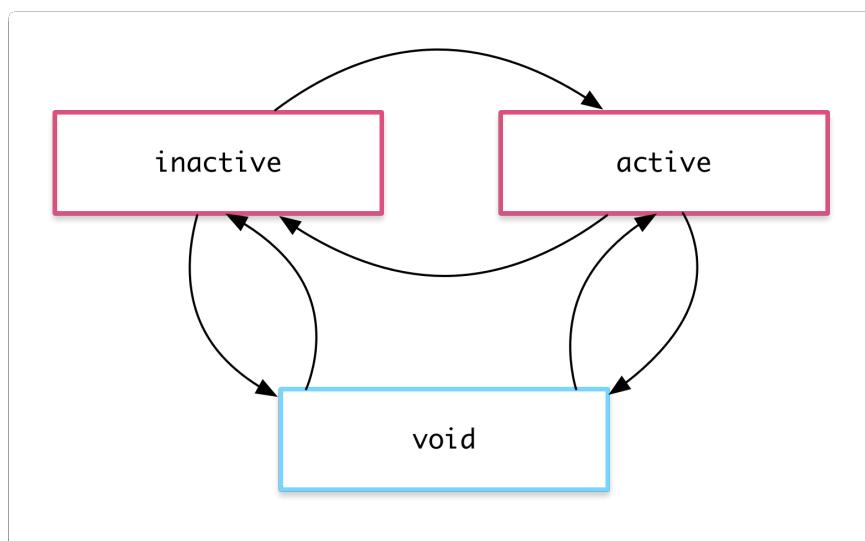
```
transition(':enter', [ ... ]); // void => *
transition(':leave', [ ... ]); // * => void
```

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。这让我们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：



```
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，我们想在动画中使用的尺寸类样式，它的值在开始运行之前都是不可知的。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：

```
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，但它们有不同的缓动函数。“离场”动画会在轻微的延迟之后开始：

```
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 10 ease-out', style({
        opacity: 0,
        transform: 'translateX(100%)'
      }))
    ])
]
```

```

        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那个时间点。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。

在这个例子中，我们使用关键帧来为进场和离场动画添加一些“反弹效果”：

```

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})
      ]))
    ])
  ])
]
```

```
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)

我们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！

但我们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，我们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：

```
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 0.1s ease', style({
          transform: 'translateX(50px)'
        }))
      ])
    ])
  ])
]
```

```

        })),
        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，我们有一个叫做 `@flyInOut` 的 `trigger`。在那里我们可以挂钩到那些回调，比如：

```

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。

属性型指令

属性型指令把行为添加到现有元素上。

属性 型指令用于改变一个 DOM 元素的外观或行为。

目录

- [指令概览](#)
- [创建简单的属性型指令](#)
- [把这个属性型指令应用到模板中的元素](#)
- [响应用户引发的事件](#)
- [使用数据绑定把值传到指令中](#)
- [绑定第二个属性](#)

试试[在线例子](#)。

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 格局的指令
3. 属性型指令 - 改变元素显示和行为的指令。

组件 是这三种指令中最常用的，我们在构建应用程序时会写大量组件。参见 [快速开始 第三步](#)，了解更多创建组件的信息。

结构型 指令会通过添加 / 删除 DOM 元素来更改 DOM 树布局。 NgFor 和 NgIf 就是两个最熟悉的例子。

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

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选择器，用于指出与此指令相关联的属性名字。控制器类实现了指令需要具备的行为。

本章展示了如何创建简单的属性型指令，在用户鼠标悬浮在一个元素上时，改变它的背景色

实际上，指令并不一定只是简单的设置背景颜色。样式绑定可以像下面这样设置样式：

```
<p [style.background]="'lime'">I am green with envy!</p>
```

参见 [模板语法 章的 样式绑定](#)。

作为一个简单的例子，它展示了属性型指令是如何工作的。

编写指令代码

创建一个项目文件夹 (`attribute-directives`) 并按照 [快速起步](#) 中的步骤进行初始化。

你还可以 [下载“快速起步”的源码](#)作为起步。

在指定的文件夹下创建下列源码文件：

app/highlight.directive.ts

```
1. import { Directive, ElementRef, Input, Renderer } from
   '@angular/core';
2.
3. @Directive({ selector: '[myHighlight]' })
4. export class HighlightDirective {
5.     constructor(el: ElementRef, renderer: Renderer) {
6.         renderer.setStyle(el.nativeElement, 'backgroundColor',
   'yellow');
7.     }
8. }
```

`import` 语句指定了从 Angular 的 `core` 库导入的一些符号。

1. `Directive` 提供 `@Directive` 装饰器功能。
2. `ElementRef` 注入 到指令构造函数中。

这样代码可以访问 DOM 元素。

1. `Input` 将数据从绑定表达式传达到指令中。
2. `Renderer` 让代码可以改变 DOM 元素的样式。

然后， `@Directive` 装饰器函数以“配置对象”参数的形式，包含了指令的元数据。

属性型指令的 `@Directive` 装饰器需要一个 css 选择器，以便从模板中识别出关联到这个指令的 HTML。

css 中的 attribute 选择器 就是属性名称加方括号。

这里，指令的选择器是 `[myHighlight]`，Angular 将会在模板中找到带有 `myHighlight` 这个属性的元素。

为什么不直接叫做 "highlight" ?

理论上，`highlight` 是一个比 `myHighlight` 更好的名字，而且在这里它确实能工作。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 **不会** 给自己的 `highlight` 指令添加 `ng` 前缀。那个前缀属于 Angular，使用它可能导致无法检测的问题。比如，这个简短的前缀 `my` 可以帮助你区分自定义指令。

`@Directive` 元数据的后面就是指令的控制器类，叫做 `HighlightDirective`，它包括了指令的工作逻辑。Exporting `HighlightDirective` makes it accessible to other components. 导出 `HighlightDirective` 以便让它可以被其它组件访问。

Angular 会为每个被指令匹配上的元素创建一个该指令控制器类的实例，并把 Angular 的 `ElementRef` 和 `Renderer` 注入进它的构造函数。`ElementRef` 是一个服务，它赋予我们直接访问 DOM 元素的能力。通过它的 `nativeElement` 属性和 `Renderer` 服务，我们可以设置元素的样式。

使用属性型指令

要使用这个新的 `HighlightDirective`，创建一个模板，把这个指令作为属性应用到一个段落 (`p`) 元素上。用 Angular 的话来说，`<p>` 元素就是这个属性型指令的 **宿主**。

我们把这个模板放到自己的 `app.component.html` 文件中，就像这样：

app/app.component.html

```
<h1>My First Attribute Directive</h1>
<p myHighlight>Highlight me!</p>
```

现在，引用 `AppComponent` 的模板：

app/app.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   moduleId: module.id,
5.   selector: 'my-app',
6.   templateUrl: 'app.component.html'
7. })
8.
9. export class AppComponent { }
```

接下来，添加了一个 `import` 语句来获得 'Highlight' 指令类，并把这个类添加到 `AppComponent` 组件的 `declarations` 数组中。这样，当 Angular 在模板中遇到 `myHighlight` 时，就能认出这是指令了。

app/app.module.ts

```
1. import { NgModule } from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3.
4. import { AppComponent } from './app.component';
5. import { HighlightDirective } from './highlight.directive';
6.
7. @NgModule({
8.   imports: [ BrowserModule ],
9.   declarations: [
10.     AppComponent,
11.     HighlightDirective
12.   ],
13.   bootstrap: [ AppComponent ]
14. })
15. export class AppModule { }
```

运行应用，就会看到我们的指令确实高亮了段落中的文本。

My First Angular 2 App

Highlight me!

你的指令没生效？

你记着设置 `@NgModule` 的 `declarations` 数组了吗？它很容易被忘掉。

打开浏览器调试工具的控制台，会看到像这样的错误信息：

```
EXCEPTION: Template parse errors:  
  Can't bind to 'myHighlight' since it isn't a known property  
  of 'p'.
```

Angular 检测到你正在尝试绑定到 **某些东西**，但它不认识。所以它在 `declarations` 元数据数组中查找。把 `HighlightDirective` 列在元数据的这个数组中，Angular 就会检查对应的导入语句，从而找到 `highlight.directive.ts`，并了解 `myHighlight` 的功能。

总结：Angular 在 `<p>` 元素上发现了一个 `myHighlight` 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，我们把 `<p>` 元素的背景设置为了黄色。

响应用户引发的事件

当前，`myHighlight` 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。

我们需要：

1. 检测用户的鼠标啥时候进入和离开这个元素。
2. 通过设置和清除高亮色来响应这些操作。

把 `host` 属性加入指令的元数据中，并给它一个配置对象，用来指定两个鼠标事件，并在它们被触发时，调用指令中的方法：

```
@HostListener('mouseenter') onMouseEnter() {  
  /* . . . */  
}  
  
@HostListener('mouseleave') onMouseLeave() {  
  /* . . . */  
}
```

`@HostListener` 装饰器引用的是我们这个属性型指令的宿主元素，在这个例子中就是 `<p>`。

可以通过直接操纵 DOM 元素的方式给宿主 DOM 元素挂上一个事件监听器，但是

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 **摘掉** 事件监听器，否则就会导致内存泄露。
3. 必须直接和 DOM API 打交道，但正如我们学过的那样，应该避免这样做。

现在，我们实现那两个鼠标事件处理器：

```
@HostListener('mouseenter') onMouseEnter() {  
  this.highlight('yellow');  
}  
  
@HostListener('mouseleave') onMouseLeave() {  
  this.highlight(null);  
}
```

```
private highlight(color: string) {
  this.renderer.setStyle(this.el.nativeElement,
  'backgroundColor', color);
}
```

注意，它们把处理逻辑委托给了一个辅助方法，这个方法会通过一个私有变量 `el` 来设置颜色。我们要修改构造函数，来把 `ElementRef.nativeElement` 存进这个私有变量。

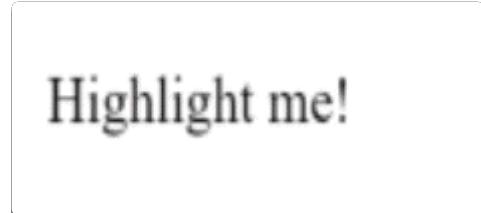
```
constructor(private el: ElementRef, private renderer: Renderer) { }
```

这里是更新过的指令：

app/highlight.directive.ts

```
1. import { Directive, ElementRef, HostListener, Input, Renderer } from
   '@angular/core';
2.
3. @Directive({
4.   selector: '[myHighlight]'
5. })
6.
7. export class HighlightDirective {
8.   constructor(private el: ElementRef, private renderer: Renderer) { }
9.
10.  @HostListener('mouseenter') onMouseEnter() {
11.    this.highlight('yellow');
12.  }
13.
14.  @HostListener('mouseleave') onMouseLeave() {
15.    this.highlight(null);
16.  }
17.
18.  private highlight(color: string) {
19.    this.renderer.setStyle(this.el.nativeElement,
20.      'backgroundColor', color);
21.  }
22.}
```

运行本应用，就可以确认：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!

通过绑定来传递值到指令中

现在的高亮颜色是在指令中硬编码进去的。这样没有弹性。我们应该通过绑定从外部设置这个颜色。就像这样：

```
<p [myHighlight]="color">Highlight me!</p>
```

我们将给指令类增加一个可绑定 **输入** 属性 `highlightColor`，当需要高亮文本的时候，就用它。

这里是该类的最终版：

app/highlight.directive.ts (class)

```
export class HighlightDirective {
  private _defaultColor = 'red';

  constructor(private el: ElementRef, private renderer: Renderer) { }

  @Input('myHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || this._defaultColor);
  }

  @HostListener('mouseleave') onMouseLeave() {
```

```

    this.highlight(null);
}

private highlight(color: string) {
  this.renderer.setStyle(this.el.nativeElement,
  'backgroundColor', color);
}
}

```

新的 `highlightColor` 属性被称为“输入”属性，这是因为数据流是从绑定表达式到这个指令的。注意，在定义这个属性的时候，我们调用了 `@Input()` 装饰器。

app/highlight.directive.ts (color)

```
@Input('myHighlight') highlightColor: string;
```

`@Input` 把元数据添加到了类上，这让 `highlightColor` 能被以 `myHighlight` 为别名进行绑定。必须添加这个 `input` 元数据，否则 Angular 会拒绝绑定。参见下面的 [附录](#) 来了解为何如此。

`@Input(别名)`

当前，代码通过将 `myHighlight` 传递到 `@Input` 装饰器，把 `myHighlight` 属性 **别名** 到属性名字上。

```
@Input('myHighlight') highlightColor: string;
```

代码绑定到 `myHighlight` 属性名，但是指令属性名为 `highlightColor`。这是一个断点。

你可以通过重命名属性名到 `myHighlight` 来移除这个区别，像这样：

```
@Input() myHighlight: string;
```

现在，通过输入型属性得到了高亮的颜色，然后修改 `onMouseEnter()` 来使用它代替硬编码的那个颜色名。我们还把红色定义为默认颜色，以便在用户忘了绑定颜色时作为备用。

```
@HostListener('mouseenter') onMouseEnter() {  
  this.highlight(this.highlightColor || this._defaultColor);  
}
```

更新 `AppComponent` 的模板，来让用户选择一个高亮颜色，并把选择结果绑定到指令上：

```
<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 [myHighlight]="color">Highlight me!</p>
```

模板的 `color` 属性在哪里？

你可能注意到，模板中的单选按钮的点击事件处理器设置了一个 `color` 属性，而且把 `color` 绑定到指令上。但是，你从未在这个宿主 `AppComponent` 中定义 `color` 属性，代码仍然工作正常。模板的 `color` 值去哪儿了？

在浏览器中调试就会发现，Angular 在 `AppComponent` 的运行期实例上添加了一个 `color` 属性。

这是一个 **很便利的** 行为，但它也是 **隐式的** 行为，这容易让人困惑。虽然这样也可行，但我们建议你还是要把 `color` 属性加到 `AppComponent` 中。

下面是指令操作演示的第二版。

My First Attribute Directive

Pick a highlight color

- Green
- Yellow
- Cyan

Highlight me!

绑定到第二个属性

本例的指令只有一个可定制属性，真实的引用通常需要更多。

要让模板开发者设置一个默认颜色，直到用户选择了一个高亮颜色才失效。给 `HighlightDirective` 添加第二个 **输入型** 属性 `defaultColor`：

```
@Input() set defaultColor(colorName: string){
  this._defaultColor = colorName || this._defaultColor;
}
```

`defaultColor` 属性是一个 `setter` 函数，它代替了硬编码的默认颜色“`red`”。不需要 `getter` 函数。

该如何绑定到它？别忘了已经把 `myHighlight` 属性名用作绑定目标了。

记住，**组件也是指令**。只要需要，就可以通过把它们依次串在模板中来为组件添加多个属性绑定。下面这个例子中就把 `a`、`b`、`c` 属性设置为了字符串字面量 `'a'`, `'b'`, `'c'`。

```
<my-component [a]="'a'" [b]="'b'" [c]="'c'"><my-component>
```

在属性型指令中也可以这样做。

```
<p [myHighlight]="color" [defaultColor]="'violet'">
  Highlight me too!
</p>
```

这里，我们像以前一样把用户选择的颜色绑定到了 `myHighlight` 上。我们还把字符串字面量 'violet' 绑定到了 `defaultColor` 上。

下面就是该指令最终版的操作演示。



总结

本章介绍了如何：

- 构建一个简单的 **属性型指令** 来为一个 HTML 元素添加行为，
- 在模板中使用那个指令，
- 响应 **事件**，以便基于事件改变行为，
- 以及 **使用 绑定** 来把值传给属性型指令。

最终的源码如下：

```
1. import { Component } from '@angular/core';
2.
```

```

3.  @Component({
4.    moduleId: module.id,
5.    selector: 'my-app',
6.    templateUrl: 'app.component.html'
7.  })
8.
9.  export class AppComponent { }

```

附录：Input 属性

本例中，`highlightColor` 属性是 `HighlightDirective` 指令的一个 `input` 属性。

以前也见过属性绑定，但我们从没有定义过它们。为什么现在就不行了？

Angular 在绑定的 **源** 和 **目标** 之间有一个巧妙但重要的区别。

在以前的所有绑定中，指令或组件的属性都是绑定 **源**。如果属性出现在了模板表达式等号 (=) 的 **右侧**，它就是一个 **源**。

如果它出现在了 **方括号** ([]) 中，并且出现在等号 (=) 的 **左侧**，它就是一个 **目标**……就像在绑定到 `HighlightDirective` 的 `myHighlight` 属性时所做的那样。

```
<p [myHighlight]="color">Highlight me!</p>
```

`[myHighlight]="color"` 中的 'color' 就是绑定 **源**。源属性不需要特别声明。

`[myHighlight]="color"` 中的 'myHighlight' 就是绑定 **目标**。必须把它定义为一个 `Input` 属性，否则，Angular 就会拒绝这次绑定，并给出一个明确的错误。

Angular 这样区别对待 **目标** 属性有充分的理由。作为目标的组件或指令需要保护。

假想一下，`HighlightDirective` 真是一个好东西。我们优雅的把它当作礼物送给全世界。

出乎意料的是，有些人（可能因为太天真）开始绑定到这个指令中的 **每一个** 属性。不仅仅只是我们预期为绑定目标的那一两个属性，而是 **每一个**。这可能会扰乱指令的工作方

式——我们既不想这样做也不想支持它们这样做。

于是，这种 **输入** 声明可以确保指令的消费者只能绑定到公开 API 中的属性，其它的都不行。

浏览器支持

浏览器支持与填充（ Polyfill ）指南

浏览器支持

Angular 支持大多数常用浏览器，包括下列版本：

Chrome	Firefox	Edge	IE	Safari	iOS	Android	IE mobile
最新版	最新版	14	11	10	10	Marshmallow (6.0)	11
		13	10	9	9	Lollipop (5.0, 5.1)	
		9	8	8	8	KitKat (4.4)	
			7	7	7	Jelly Bean (4.1, 4.2, 4.3)	

Angular 在持续集成过程中，对每一个提交都会使用 [SauceLabs](#) 和 [Browserstack](#) 在上述所有浏览器上执行单元测试。

填充库（ Polyfill ）

Angular 构建于 Web 平台的最新标准之上。要支持这么多浏览器是一个不小的挑战，因为它们不支持现代浏览器的所有特性。

你可以通过在宿主页面（`index.html`）中加载填充脚本（“polyfills”）来加以弥补，这些脚本实现了浏览器缺失的JavaScript特性。

```
<!-- Polyfill(s) for older browsers -->
<script src="node_modules/core-js/client/shim.min.js"></script>
```

要运行Angular应用，某些浏览器可能需要至少一个填充库。除此之外，如果要支持某些特定的特性，你可能还需要另一些填充库。

下表将帮你决定该加载哪些填充库，具体取决于目标浏览器和要用到的特性。

这些建议的填充库假设你运行的是完全由Angular书写的应用。你可能还会需要另一些的填充库来支持没有出现在此列表中的哪些特性。注意，这些填充库并没有神奇的魔力来把老旧、慢速的浏览器变成现代、快速的浏览器，它只是填充了API。

强制性填充库

下表是填充库对每个支持的浏览器都是需要的：

浏览器（桌面 & 移动）	需要的填充库
Chrome, Firefox, Edge, Safari 9+	None
Safari 7 & 8, IE10 & 11, Android 4.1+	ES6
IE9	ES6 classList

可选浏览器特性的填充库

有些Angular特性可能需要更多填充库。

比如，动画库依赖于标准的动画API，目前它只在Chrome和Firefox上可用。你可能需要一个填充库来在其它浏览器上使用动画功能。

下列特性可能需要更多填充库：

特性	填充库	浏览器（桌面

		& 移动)
动画	Web Animations	除 Chrome 和 Firefox 外的所有，但不支持 IE9
Date, currency, decimal 和 percent 管道	Intl API	除了 Chrome、Firefox、Edge、IE11 和 Safari 10 外的所有浏览器
在 SVG 元素上用 NgClass	classList	IE10, IE11
用 Http 发送和接受二进制数据	Typed Array Blob FormData	IE 9

建议的填充库

下表中是用来测试框架本身的填充库，它们是应用程序的优质起点。

填充库	授权方式	大小 *
ES6	MIT	27.4KB

classList	公共域	1KB
Intl	MIT / Unicode licence	13.5KB
Web Animations	Apache	14.8KB
Typed Array	MIT	4KB
Blob	MIT	1.3KB
FormData	MIT	0.4KB

- 这些指标测量的是最小化 (minify) 并且 gzip 过的代码，使用 [closure compiler](#) 计算出的结果。

组件样式

学习如何给组件应用 CSS 样式。

Angular 2 应用使用标准的 CSS 来设置样式。这意味着我们可以把关于 CSS 的那些知识和技能直接用于我们的 Angular 程序中，比如：样式表、选择器、规则，以及媒体查询等。

在此基础上，Angular 还能把 **组件样式** 紧紧的“捆绑”在我们的组件上，以实现一种比标准样式表更加模块化的设计。

在本章中，我们将学到如何加载和使用这些 **组件样式**。

目录

- [使用组件样式](#)
- [特殊的选择器](#)
- [把样式加载进组件](#)
- [控制视图的包装模式：仿真 \(Emulated\)、原生 \(Native\) 或无 \(None\)](#)
- [附录 1：审查生成的运行时组件样式](#)
- [附录 2：使用相对 URL 加载样式](#)

运行本章这些代码的 [在线例子](#)。

使用组件样式

对于我们写的每个 Angular 组件来说，除了定义 HTML 模板之外，我们还要用于模板的 CSS 样式、指定需要的选择器、规则和媒体查询。

它的实现方式之一，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。通常我们只给它一个字符串就行了，如同下例：

```
@Component({
  selector: 'hero-app',
  template: `
    <h1>Tour of Heroes</h1>
    <hero-app-main [hero]=hero></hero-app-main>`,
  styles: ['h1 { font-weight: normal; }']
})
export class HeroAppComponent {
  /* . . . */
}
```

组件样式在很多方面都不同于传统的全局性样式。

首先，我们放在组件样式中的选择器，只会应用在组件自身的模板中。上面这个例子中的 `h1` 选择器只会对 `HeroAppComponent` 模板中的 `<h1>` 标签生效，而对应用中其它地方的 `<h1>` 元素毫无影响。

这种模块化相对于 CSS 的传统工作方式是一个巨大的改进：

1. 只有在每个组件的情境中使用 CSS 类名和选择器，才是最有意义的。
2. 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。
3. 我们组件的样式 **不会** 因为别的地方修改了样式而被意外改变。
4. 我们可以让每个组件的 CSS 代码和它的 TypeScript 代码、HTML 代码放在一起，这将促成清爽整洁的项目结构。
5. 将来我们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 [Shadow DOM style scoping\(范围化样式 \)](#) 领域引入的特殊 **选择器**：

:host

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）。

```
:host {  
  display: block;  
  border: 1px solid black;  
}
```

这是我们能以宿主元素为目标的 **唯一** 方式。除此之外，我们将没办法指定它，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。

在下一个例子中，我们又一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。

```
:host(.active) {  
  border-width: 3px;  
}
```

:host-context

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。比如，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，而我们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，我们才会把 `background-color` 样式应用到组件 **内部** 的所有 `<h2>` 元素中。

```
:host-context(.theme-light) h2 {  
  background-color: #eef;  
}
```

/deep/

“组件样式”通常只会作用于组件自身的 HTML 上。

我们可以使用 `/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**。

在这个例子中，我们以所有的 `<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：

```
:host /deep/ h3 {  
  font-style: italic;  
}
```

`/deep/` 选择器还有一个别名 `>>>`。我们可以任意交替使用它们。

`/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。要了解更多，请参阅 [控制视图包装模式](#) 一节。

把样式加载进组件中

我们有几种方式来把样式加入组件：

- 内联在模板的 HTML 中
- 设置 `styles` 或 `styleUrls` 元数据

- 通过 CSS 文件导入

上述局限化规则对所有这些加载模式都适用。

元数据中的样式

我们可以给 `@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。

```
1.  @Component({
2.    selector: 'hero-app',
3.    template: `
4.      <h1>Tour of Heroes</h1>
5.      <hero-app-main [hero]=hero></hero-app-main>`,
6.      styles: ['h1 { font-weight: normal; }']
7.  })
8.  export class HeroAppComponent {
9.  /* . . . */
10. }
```

模板内联样式

我们也可以把它们放到 `<style>` 标签中来直接在 HTML 模板中嵌入样式。

```
1.  @Component({
2.    selector: 'hero-controls',
3.    template: `
4.      <style>
5.        button {
6.          background-color: white;
7.          border: 1px solid #777;
8.        }
9.      </style>
10.     <h3>Controls</h3>
11.     <button (click)="activate()">Activate</button>
12.   `
13. })
```

元数据中的样式表 URL

我们还可以通过给组件的 `@Component` 装饰器中添加一个 `styleUrls` 属性来从外部 CSS 文件中加载样式：

```
1.  @Component({
2.    selector: 'hero-details',
3.    template: `
4.      <h2>{{hero.name}}</h2>
5.      <hero-team [hero]=hero></hero-team>
6.      <ng-content></ng-content>
7.    `,
8.    styleUrls: ['app/hero-details.component.css']
9.  })
10. export class HeroDetailsComponent {
11.   /* . . . */
12. }
```

URL 是 **相对于应用程序根目录的**，它通常是应用的宿主页面 `index.html` 所在的地方。这个样式文件的 URL **不是** 相对于组件文件的。这就是为什么范例中的 URL 用 `app/` 开头。参见 [附录 2](#) 来了解如何指定相对于组件文件的 URL。

像 Webpack 这类模块打包器的用户可能会使用 `styles` 属性来在构建时从外部文件中加载样式。它们可能这样写：

```
styles: [require('my.component.css')]
```

注意，这时候我们是在设置 `styles` 属性，**而不是** `styleUrls` 属性！是模块打包器在加载 CSS 字符串，而不是 Angular。Angular 看到的只是打包器加载它们之后的 CSS 字符串。对 Angular 来说，这跟我们手写了 `styles` 数组没有任

何区别。要了解这种 CSS 加载方式的更多信息，请参阅相应模块打包器的文档。

模板中的 link 标签

我们也可以在组件的 HTML 模板中嵌入 `<link>` 标签。

像 `styleUrls` 标签一样，这个 link 标签的 `href` 指向的 URL 也是相对于应用的根目录的，而不是组件文件。

```
1.  @Component({
2.    selector: 'hero-team',
3.    template: `
4.      <link rel="stylesheet" href="app/hero-team.component.css">
5.      <h3>Team</h3>
6.      <ul>
7.        <li *ngFor="let member of hero.team">
8.          {{member}}
9.        </li>
10.       </ul>``,
11.  })
```

CSS @imports

我们还可以利用标准的 CSS `@import` 规则 来把其它 CSS 文件导入到我们的 CSS 文件中。

在这种情况下，URL 是相对于我们执行导入操作的 CSS 文件的。

app/hero-details.component.css (excerpt)

```
@import 'hero-details-box.css';
```

控制视图的包装模式：原生 (Native) , 仿真 (Emulated) 和无 (None)

像上面讨论过的一样，组件的 CSS 样式被包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 **视图包装模式**，我们可以分别控制 **每个组件** 的包装模式。可选的包装模式一共有三种：

- `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 代码来仿真 `Shadow DOM` 的行为，以达到把 CSS 样式局限在组件视图中的目的。参见 [附录 1](#) 了解详情。（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 不使用视图包装。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局限化规则、隔离和保护等规则。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` 属性来设置组件包装模式：

```
// 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，这就是为什么我们会把仿真 (`Emulated`) 模式作为默认选项，并建议将其用于大多数情况。

附录 1 : 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` 局限化规则。

当我们查看启用了“仿真”模式的 Angular 应用时，我们看到每个 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 _ngghost-pmm-6>
    <h3 _ngcontent-pmm-6>Team</h3>
  </hero-team>
</hero-detail>
```

我们看到了两种被生成的属性：

- 一个元素在原生包装方式下可能是 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，并且我们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，就像我们在 DOM 的 `<head>` 区所看到的：

```
[_ngost-pmm-5] {
  display: block;
  border: 1px solid black;
}

h3[_ngcontent-pmm-6] {
  background-color: white;
  border: 1px solid #777;
}
```

这些就是我们写的那些样式被处理后的结果，于是每个选择器都被增加了 `_ngost` 或 `_ngcontent` 属性选择器。在这些附加选择器的帮助下，我们实现了本指南中所描述的这些局限化规则。

小伙伴们会很愉快的使用 **仿真** 模式——直到有一天 Shadow DOM 获得全面支持。

附录 2：使用相对 URL 加载样式

把组件的代码 (ts/js)、HTML 和 CSS 分别放到同一个目录下的三个不同文件，是一个常用的实践：

```
quest-summary.component.ts  
quest-summary.component.html  
quest-summary.component.css
```

我们会通过设置元数据的 `templateUrl` 和 `styleUrls` 属性把模板和 CSS 文件包含进来。既然这些文件都与组件（代码）文件放在一起，那么通过名字，而不是到应用程序根目录的全路径来指定它，就会是一个漂亮的方式。

通过把组件元数据的 `moduleId` 属性设置为 `module.id`，我们可以更改 Angular 计算完整 URL 的方式

app/quest-summary.component.ts

```
1.  @Component({  
2.    moduleId: module.id,  
3.    selector: 'quest-summary',  
4.    templateUrl: 'quest-summary.component.html',  
5.    styleUrls: ['quest-summary.component.css']  
6.  })  
7.  export class QuestSummaryComponent { }
```

要学习更多关于 `moduleId` 的知识，请参见 [相对于组件的路径](#) 一章。

多级依赖注入器

Angular 的多级依赖注入系统支持与组件树并行的嵌套式注入器。

在 [依赖注入](#) 一章中，我们已经学过了 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）。我们可以在组件树中的任何级别上重新配置注入器，达到一些有趣和有用的效果。

在本章中，我们将浏览这些要点，并写点代码来验证它。

试试 [在线例子](#) .

注入器树

在 [依赖注入](#) 一章中，我们学过如何配置注入器，以及如何在我们需要时用它获取依赖。

我们其实有点简化过度了。实际上，没有 **那个（唯一的）注入器** 这回事，因为一个应用中可能有很多注入器。

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。

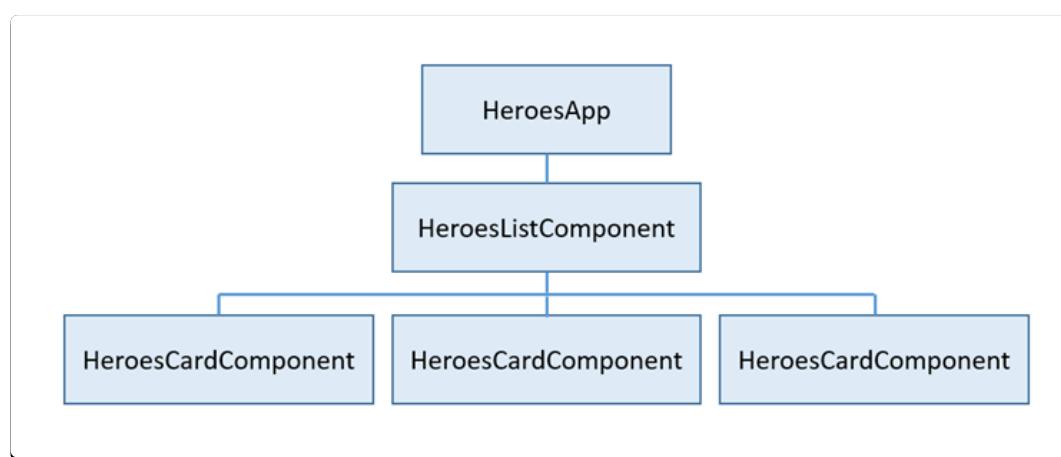
Angular 并没有像 **字面上理解的** 那样为每个组件都创建一个独立的注入器。不是每个组件都需要它自己的注入器，盲目创建大量没有明确用途的注入器是非常低效的。

但是，每个组件都 **有一个注入器** 是真实的（就算它与其它组件共享注入器，也是**有注入器**），并且确实可能会有大量不同的注入器实例工作在组件树的不同级别。

我们可以假装每个组件都有自己的注入器，因为这样有助于思考和理解。

考虑《英雄指南》应用的一个简单变种，它由三个不同的组件构成：`HeroesApp`、`HeroesListComponent` 和 `HeroesCardComponent`。`HeroesApp` 保存了 `HeroesListComponent` 的一个单例。新的变化是，`HeroesListComponent` 可以保存和管理 `HeroesCardComponent` 的多个实例。

下图展示了当 `HeroesCardComponent` 的三个实例同时展开时组件树的状态。



每个组件实例获得了它自己的注入器，并且每个级别上的注入器都是它上级节点的子注入器。

当一个底层的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——直到我们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。

其实还有第三种可能性。一个中层的组件可以声称它自己是“宿主”组件。向上查找提供商的过程会截止于这个“宿主”组件。我们先保留这个问题，等改天再讨论这个选项。

除非注入器能在各个不同层次上使用不同的提供商进行配置，否则没必要让注入器分裂成这么多。虽然我们并不是 **必须** 在每一层都重新配置提供商，但我们 **可以**这样做。

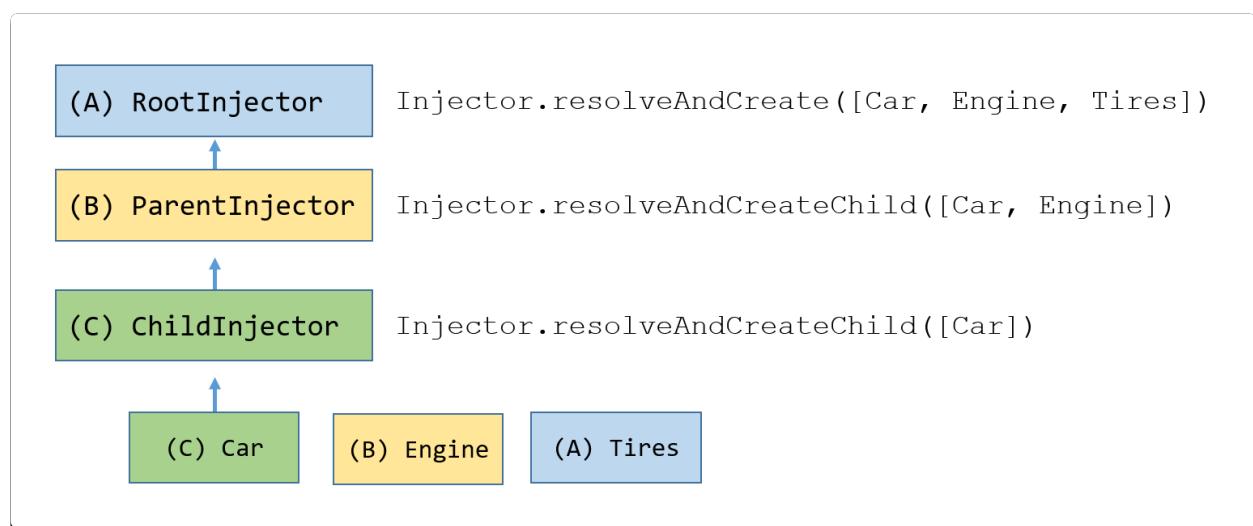
如果我们完全没有进行过重新配置，注入器的树看起来将是“平面”的。所有的申请都会被冒泡到根 NgModule 注入器进行处理，也就是我们在 `bootstrapModule` 方法中配置的那个。

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。

让我们回到“汽车 (Car)”类的例子。假设“根注入器”(记为 A) 配置过 `Car`、`Engine` 和 `Tires` 的提供商。然后创建了一个子组件 (B)，它为 `Car` 和 `Engine` 类定义了自己的提供商。这个子组件 (B) 又有另一个子组件 (C)，(C) 也为 `Car` 定义了自己的提供商。

幕后的情况是这样的：每个组件都设置了它自己的注入器，这些注入器都带着一个或多个为组件自身设计的提供商。

当我们在最底层的组件 (C) 中试图解析出一个 `Car` 的实例时，注入器 (C) 解析出的 `Car` 类的实例，该 `Car` 的实例带有一个 `Engine` 类的实例 (由注入器 (B) 解析出) 和一个 `Tires` 类的实例 (由跟注入器 (A) 解析出)。



组件注入器

在前一节中，我们讨论了注入器以及它们是如何被组织成一棵树的。Angular 会沿着注入器树往上逐级查找，直到发现了那个申请者要求注入的东西。但是，我们什么时候该在根注入器上提供提供商，什么时候又该在子注入器上提供它们呢？

考虑我们正在构建一个组件，用来显示一个超级英雄的列表，它需要在每个卡片中显示一位超级英雄的名字和超能力。还要有一个编辑按钮来打开编辑器，以修改这位英雄的名字和超能力。

关于编辑功能的一个重要方面是，我们得允许多个英雄同时进入编辑模式，每一个都要始终允许提交或取消所做的修改。

我们看看 `HeroesListComponent`，也就是这个例子中的根组件。

app/heroes-list.component.ts

```
1. import { Component } from '@angular/core';
2.
3. import { EditItem } from './edit-item';
4. import { HeroesService } from './heroes.service';
5. import { Hero } from './hero';
6.
7. @Component({
8.   selector: 'heroes-list',
9.   template: `
10.     <div>
11.       <ul>
12.         <li *ngFor="let editItem of heroes">
13.           <hero-card
14.             [hidden]="editItem.editing"
15.             [hero]="editItem.item">
16.           </hero-card>
17.           <button
18.             [hidden]="editItem.editing"
19.             (click)="editItem.editing = true">
20.             edit
21.           </button>
22.           <hero-editor
23.             (saved)="onSaved(editItem, $event)"
24.             (canceled)="onCanceled(editItem)"
25.             [hidden]="!editItem.editing"
26.             [hero]="editItem.item">
```

```

27.         </hero-editor>
28.     </li>
29.     </ul>
30.   </div>
31. }
32. export class HeroesListComponent {
33.   heroes: Array<EditItem<Hero>>;
34.   constructor(heroesService: HeroesService) {
35.     this.heroes = heroesService.getHeroes()
36.       .map(item => new EditItem(item));
37.   }
38.
39.   onSave (editItem: EditItem<Hero>, updatedHero: Hero) {
40.     editItem.item = updatedHero;
41.     editItem.editing = false;
42.   }
43.
44.   onCancel (editItem: EditItem<Hero>) {
45.     editItem.editing = false;
46.   }
47. }

```

注意，它导入了 `HeroService` 类，我们以前用到过这个类，所以直接跳过它的声明。唯一的不同是，这里我们用一种更正规的方式使用 `Hero` 模型，提前定义它。

app/hero.ts

```

export class Hero {
  name: string;
  power: string;
}

```

我们的 `HeroesListComponent` 组件定义了一个模板，来创建 `HeroCardComponent` 和 `HeroEditorComponent` 的列表，它们的每个条目都绑定到一个由 `HeroService` 返回的 `Hero` 实例。好吧，这么说也不完全对。它实际上绑定到了一个 `EditItem<Hero>` 实例，这是一个简单的泛型数据类型，它可以包裹任何类型的实例，并额外添加一个 `editing` 属性，用来标记被包裹的实例是否正在编辑状态。

app/edit-item.ts

```
export class EditItem<T> {
  editing: boolean;
  constructor (public item: T) {}
}
```

但 `HeroCardComponent` 是怎么实现的？我们来看看。

app/hero-card.component.ts

```
1. import { Component, Input } from '@angular/core';
2.
3. import { Hero } from './hero';
4.
5. @Component({
6.   selector: 'hero-card',
7.   template: `
8.     <div>
9.       <span>Name:</span>
10.      <span>{{hero.name}}</span>
11.    </div>
12.  })
13. export class HeroCardComponent {
14.   @Input() hero: Hero;
15. }
```

`HeroCardComponent` 基本是一个组件，它定义了一个模板来渲染英雄。没别的。

让我们开始更有趣的部分，来看看 `HeroEditorComponent`：

app/hero-editor.component.ts

```
1. import { Component, EventEmitter, Input, Output } from
2.   '@angular/core';
3. import { RestoreService } from './restore.service';
```

```
4. import { Hero } from './hero';
5.
6. @Component({
7.   selector: 'hero-editor',
8.   providers: [RestoreService],
9.   template: `
10.     <div>
11.       <span>Name:</span>
12.       <input [(ngModel)]="hero.name"/>
13.       <div>
14.         <button (click)="onSaved()">save</button>
15.         <button (click)="onCanceled()">cancel</button>
16.       </div>
17.     </div>`  

18.   })
19.  

20. export class HeroEditorComponent {
21.   @Output() canceled = new EventEmitter();
22.   @Output() saved = new EventEmitter();
23.  

24.   constructor(private restoreService: RestoreService<Hero>) {}
25.  

26.   @Input()
27.   set hero (hero: Hero) {
28.     this.restoreService.setItem(hero);
29.   }
30.  

31.   get hero () {
32.     return this.restoreService.getItem();
33.   }
34.  

35.   onSaved () {
36.     this.saved.next(this.restoreService.getItem());
37.   }
38.  

39.   onCanceled () {
40.     this.hero = this.restoreService.restoreItem();
41.     this.canceled.next(this.hero);
42.   }
43. }
```

现在，事情开始变得有趣了。`HeroEditorComponent` 定义了一个模板，它有一个输入框来修改英雄的名字，以及一个 `cancel` 按钮和一个 `save` 按钮。还记得吗？我们说过要能够灵活的取消编辑，并且回复原值吗？这意味着我们得维护这个待编辑 `Hero` 的两份实例。想得超前一点，这是一个把它抽象成一个通用服务的完美案例，因为将来我们的应用中可能会需要更多与此类似的操作逻辑。

该是 `RestoreService` 登场的时候了！

app/restore.service.ts

```
1.  export class RestoreService<T> {
2.    originalItem: T;
3.    currentItem: T;
4.
5.    setItem(item: T) {
6.      this.originalItem = item;
7.      this.currentItem = this.clone(item);
8.    }
9.
10.   getItem(): T {
11.     return this.currentItem;
12.   }
13.
14.   restoreItem(): T {
15.     this.currentItem = this.originalItem;
16.     return this.getItem();
17.   }
18.
19.   clone(item: T): T {
20.     // super poor clone implementation
21.     return JSON.parse(JSON.stringify(item));
22.   }
23. }
```

这个微型服务唯一所做的是定义一个 API 来设置一个任意类型的值，它可以被修改、获取或者恢复成初始值。这正是我们需要实现的功能。

我们的 `HeroEditComponent` 组件悄悄把这个服务用在它的 `hero` 属性上。它拦截了 `get` 和 `set` 方法，并把真正的工作委托给我们的 `RestoreService` 服务，这样可以确保

我们并不是在操作原始条目，而是一个副本。

这时，我们可能挠着头问，这跟组件注入器有什么关系？凑近我们的 `HeroEditComponent` 的元数据看看。注意看它的 `providers` 属性。

```
providers: [RestoreService],
```

它往 `HeroEditComponent` 的注入器中添加了一个 `RestoreService` 提供商。不能简化点，直接在我们的 root `NgModule` 后面包含这个提供商吗？

app/app.module.ts (bad-alternative)

```
// Don't do this!
@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [ HeroesService, RestoreService ],
  declarations: [ HeroesListComponent ],
  bootstrap: [
    HeroesListComponent,
    HeroCardComponent,
    HeroEditorComponent
  ]
})
```

虽然理论上可以，但我们的组件却无法像预期的那样工作。记住，每个注入器都会把它提供的服务处理成单例。可是，我们需要同时用多个 `HeroEditComponent` 实例来编辑多个英雄，这就意味着我们需要 `RestoreService` 的多个实例。更明确的说，每个 `HeroEditComponent` 的实例都得绑定到它自己的 `RestoreService` 实例。

通过在 `HeroEditComponent` 上配置 `RestoreService` 的提供商，我们可以精确的实现每个 `HeroEditComponent` 都有一个 `RestoreService` 的新实例。

这是否意味着在 Angular 2 中服务再也不是单例的了？是，也不是。对某一个具体的注入器来讲，仍然是每个服务类型只有一个实例。但我们可以知道，在应用程序组件树的各个不同级别上可以有多个注入器。任何一个注入器都可以有它自己的服务实例。

如果我们只在根组件上定义了一个 `RestoreService` 提供商，我们就确实只有该服务的一个实例了，它会在整个应用程序中被共享。

但很明显，这个场景下我们不希望这样。我们希望每个组件都有它自己的 `RestoreService` 实例。在组件级别上定义（或重定义）一个提供商，将会为该组件创建一个新的服务实例。我们已经为 `HeroEditComponent` 制造了一种“私有” `RestoreService` 单例，它的作用域被局限在了该组件的实例及其子组件中。

HTTP客户端

通过 HTTP 客户端与远程服务器对话。

HTTP 是浏览器和服务器之间通讯的主要协议。

`WebSocket` 协议是另一种重要的通讯技术，但本章不会涉及它。

现代浏览器支持两种基于 HTTP 的 API : `XMLHttpRequest (XHR)` 和 `JSONP`。少数浏览器还支持 `Fetch`。

Angular HTTP 库简化了 `XHR` 和 `JSONP` API 的编程，这就是本章所要讲的。

- 英雄指南范例的 HTTP 客户端
- 用 `http.get` 获取数据
- RxJS 库
- 启用 RxJS 操作符
- 处理响应对象
- 总是处理错误
- 把数据发送到服务器
- 使用承诺 (Promise) 来取代可观察对象 (Observable)
 - 跨域请求： Wikipedia 例子
 - 设置查询参数

- 限制搜索词输入频率
- 防止跨站请求伪造
- 附录：英雄指南的内存 Web API 服务

我们在 [在线例子](#) 中展示了这些主题。

演示

本章通过下面这些演示，描述了服务端通讯的用法。

- 英雄指南 HTTP 客户端
- 回到使用承诺
- 跨站请求：Wikipedia 例子
- 更多可观察对象的探索

这些演示由根组件 `AppComponent` 统一演示。

app/app.component.ts

```
1. import { Component }           from '@angular/core';
2.
3. // Add the RxJS Observable operators.
4. import './rxjs-operators';
5.
6. @Component({
7.   selector: 'my-app',
8.   template: `
9.     <hero-list></hero-list>
10.    <hero-list-promise></hero-list-promise>
11.    <my-wiki></my-wiki>
12.    <my-wiki-smart></my-wiki-smart>
13.  `
14. })
```

```
15.  export class AppComponent { }
```

这里唯一值得注意的是对 RxJS 操作符的导入，[后面](#) 有详细介绍。

```
// Add the RxJS Observable operators.  
import './rxjs-operators';
```

提供 HTTP 服务

首先，配置应用来使用服务器对话设施。

我们通过 Angular `Http` 客户端，使用熟悉的 HTTP 请求 / 回应协议与服务器通讯。

`Http` 客户端是 Angular HTTP 库所提供的服务大家庭中的一员。

当我们从 `@angular/http` 模块中导入服务时，SystemJS 知道该如何从 Angular HTTP 库中加载它们，这是因为我们已经在 `system.config` 文件中注册过这个模块名。

要想使用 `Http` 客户端，我们得先通过依赖注入系统把它注册成一个服务提供商。

了解关于提供商的更多知识，参见 [依赖注入](#) 一章。

在 `app.module.ts` 中通过导入其他模块来注册提供商到根 NgModule 。

app/app.module.ts (v1)

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { HttpModule, JsonpModule } from '@angular/http';

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

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    JsonpModule
  ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

我们从导入所需的符号开始，它们中的大多数我们都熟悉了，只有 `HttpModule` 和 `JsonpModule` 是新面孔。

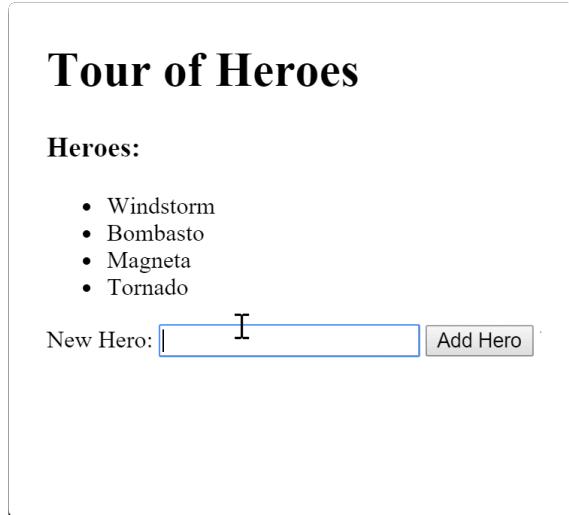
只要把它们传给我们这个根模块的 `imports` 数组，就可以把这些模块加入本应用。

我们需要 `HttpModule` 来发起 HTTP 调用。普通的 HTTP 调用并不需要用到 `JsonpModule`，不过稍后我们就会演示对 JSONP 的支持，所以现在就加载它，免得再回来改浪费时间。

《英雄指南》的 HTTP 客户端演示

我们的第一个演示是《英雄指南 (TOH)》教程的一个迷你版。这个版本从服务器获取一些英雄，把它们显示在列表中，还允许我们添加新的英雄并将其保存到服务器。借助 Angular `Http` 客户端，我们通过 `XMLHttpRequest (XHR)` 与服务器通讯。

它跑起来是这样的：



这个范例是一个单一组件 `HeroListComponent`，其模板如下：

`app/toh/hero-list.component.html (template)`

```

1.  <h1>Tour of Heroes ({{mode}})</h1>
2.  <h3>Heroes:</h3>
3.  <ul>
4.    <li *ngFor="let hero of heroes">{{hero.name}}</li>
5.  </ul>
6.
7.  <label>New hero name: <input #newHeroName /></label>
8.  <button (click)="addHero(newHeroName.value); newHeroName.value=''">Add
   Hero</button>
9.
10. <p class="error" *ngIf="errorMessage">{{errorMessage}}</p>
```

它使用 `ngFor` 来展现这个英雄列表。列表的下方是一个输入框和一个 **Add Hero** 按钮，在那里，我们可以输入新英雄的名字，并把它们加到数据库中。在 `(click)` 事件绑定中，我们使用 **模板引用变量** `newHeroName` 来访问这个输入框的值。当用户点击此按钮时，我们把这个值传给组件的 `addHero` 方法，然后清除它，以备输入新英雄的名字。

按钮的下方是一个错误信息区。

HeroListComponent 类

下面是这个组件类：

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

```

1.  export class HeroListComponent implements OnInit {
2.    errorMessage: string;
3.    heroes: Hero[];
4.    mode = 'observable';
5.
6.    constructor (private heroService: HeroService) {}
7.
8.    ngOnInit() { this.getHeroes(); }
9.
10.   getHeroes() {
11.     this.heroService.getHeroes()
12.       .subscribe(
13.         heroes => this.heroes = heroes,
14.         error => this.errorMessage = <any>error);
15.   }
16.
17.   addHero (name: string) {
18.     if (!name) { return; }
19.     this.heroService.addHero(name)
20.       .subscribe(
21.         hero => this.heroes.push(hero),
22.         error => this.errorMessage = <any>error);
23.   }
24. }
```

Angular 会把一个 `HeroService` 注入 到组件的构造函数中，该组件将调用此服务来获取和保存数据。

这个组件 不会直接和 Angular `Http` 客户端打交道！ 它既不知道也不关心我们如何获取数据，这些都被委托给了 `HeroService` 去做。

这是一条“黄金法则”： 总是把数据访问工作委托给一个支持服务类。

虽然 在运行期间，组件会在创建之后立刻请求这些英雄数据，但我们 不 在组件的构造函数中调用此服务的 `get` 方法。而是在 `ngOnInit` 生命周期钩子 中调用它，Angular

会在初始化该组件时调用 `ngOnInit` 方法。

这是 **最佳实践**。当组件的构造函数足够简单，并且所有真实的工作（尤其是调用远端服务器）都在一个独立的方法中处理时，组件会更加容易测试和调试。

服务的 `getHeroes()` 和 `addHero()` 方法返回一个英雄数据的可观察对象（`Observable`），这些数据是由 Angular `Http` 从服务器上获取的。

我们可以把可观察对象 `Observable` 看做一个由某些“源”发布的事件流。通过 **订阅** 到可观察对象 `Observable`，我们监听这个流中的事件。在这些订阅中，我们指定了当 Web 请求生成了一个成功事件（有效载荷是英雄数据）或失败事件（有效载荷是错误对象）时该如何采取行动。

关于组件的浅显讲解已经结束了，我们可以到 `HeroService` 的内部实现中看看。

通过 `http.get` 获取数据

在前面的很多例子中，我们通过在服务中返回一个模拟的英雄列表来伪造了与服务器的交互过程。就像这样：

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

import { Hero } from './hero';
import { HEROES } from './mock-heroes';

@Injectable()
export class HeroService {
  getHeroes(): Promise<Hero[]> {
    return Promise.resolve(HEROES);
  }
}
```

在本章中，我们会修改 `HeroService`，改用“Angular `Http` 客户端”服务来从服务器上获取英雄列表：

app/toh/hero.service.ts (revised)

```
1. import { Injectable }      from '@angular/core';
2. import { Http, Response } from '@angular/http';
3.
4. import { Hero }           from './hero';
5. import { Observable }     from 'rxjs/Observable';
6.
7. @Injectable()
8. export class HeroService {
9.   private heroesUrl = 'app/heroes'; // URL to web API
10.
11. constructor (private http: Http) {}
12.
13. getHeroes (): Observable<Hero[]> {
14.   return this.http.get(this.heroesUrl)
15.     .map(this.extractData)
16.     .catch(this.handleError);
17. }
18. private extractData(res: Response) {
19.   let body = res.json();
20.   return body.data || {};
21. }
22.
23. private handleError (error: Response | any) {
24.   // In a real world app, we might use a remote logging
25.   // infrastructure
26.   let errMsg: string;
27.   if (error instanceof Response) {
28.     const body = error.json() || '';
29.     const err = body.error || JSON.stringify(body);
30.     errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
31.   } else {
32.     errMsg = error.message ? error.message : error.toString();
33.   }
34.   console.error(errMsg);
35.   return Observable.throw(errMsg);
36. }
```

注意，这个“Angular `Http` 客户端”服务 被注入 到了 `HeroService` 的构造函数中。

```
constructor (private http: Http) {}
```

仔细看看我们是如何调用 `http.get` 的

app/toh/hero.service.ts (getHeroes)

```
getHeroes (): Observable<Hero[]> {
  return this.http.get(this.heroesUrl)
    .map(this.extractData)
    .catch(this.handleError);
}
```

我们把资源的 URL 传进 `get` 函数，它调用了服务器，而服务器应该返回英雄列表。

一旦我们按附录中所描述的那样准备好了 内存 Web API，它 将 返回英雄列表。但目前，我们只能（临时性的）使用一个 JSON 文件来代替这个“内存 Web API”。只要修改下服务器的 URL就行了：

```
private heroesUrl = 'app/heroes.json'; // URL to JSON file
```

返回值可能会让我们感到意外。对熟悉现代 JavaScript 中的异步调用方法的人来说，我们期待 `get` 方法返回一个 承诺 (promise)。我们期待链接调用 `then()` 方法，并从中取得英雄列表。取而代之，这里调用了一个 `map()` 方法。显然，这并不是承诺 (Promise)。

事实上，`http.get` 方法返回了一个 HTTP Response 类型的 **可观察对象** (`Observable<Response>`)，这个对象来自 RxJS 库，而 `map` 是 RxJS 的 **操作符**之一。

RxJS 库

RxJS("Reactive Extensions" 的缩写)是一个被 Angular 认可的第三方库，它实现了 **异步可观察对象 (asynchronous observable)** 模式。

本开发指南中的所有例子都安装了 RxJS 的 npm 包，而且都被 `system.js` 加载过了。这是因为可观察对象在 Angular 应用中使用非常广泛。

HTTP 客户端更需要它。经过一个关键步骤，我们才能让 RxJS 可观察对象可用。

启用 RxJS 操作符

RxJS 库实在是太大了。当构建一个产品级应用，并且把它发布到移动设备上的时候，大小就会成为一个问题。我们应该只包含那些我们确实需要的特性。

因此，Angular 在 `rxjs/Observable` 模块中导出了一个精简版的 `Observable` 类，这个版本缺少很多操作符，比如我们在上面的 `getHeroes` 方法中用过的 `map` 函数。

这让我们可以自由决定添加哪些操作符。

我们可以通过一条 `import` 语句把 **每个** RxJS 操作符都添加进来。虽然这是最简单的方式，但我们也得付出代价，主要是在启动时间和应用大小上，因为完整的库实在太大了。而我们其实只需要用到少量操作符。

因为本应用只使用了少许操作符，所以将一个一个的导入 `Observable` 的操作符和静态类方法比较合适，直到我们得到了一个精确符合我们需求的自定义 **Observable** 实现。我们将把这些 `import` 语句放进一个 `app/rxjs-operators.ts` 文件里。

app/rxjs-operators.ts

```
// import 'rxjs/Rx'; // adds ALL RXJS statics & operators to
observable
```

```
// See node_module/rxjs/Rxjs.js
// Import just the rxjs statics and operators needed for THIS app.

// Statics
import 'rxjs/add/observable/throw';

// Operators
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/toPromise';
```

如果忘了导入某个操作符，TypeScript 编译器就会警告说找不到它，那时候我们再来更新此文件。

在 `HeroService` 中，我们并不需要在这里导入的 **全部** 操作符——我们只用到了 `map`、`catch` 和 `throw`。我们稍后的 [Wiki 例子](#) 中，还会用到其它操作符。

最后，我们把 `rxjs-operator` **本身** 导入 `app.component.ts` 文件中：

app/app.component.ts (import rxjs)

```
// Add the RXJS Observable operators.
import './rxjs-operators';
```

现在，继续到下一节，返回到 `HeroService`。

处理 Response 响应对象

记住，`getHeroes()` 借助一个 `extractData` 辅助方法来把 `http.get` 的响应对象映射成了英雄列表：

app/toh/hero.service.ts (excerpt)

```
private extractData(res: Response) {
  let body = res.json();
  return body.data || {};
}
```

这个 `response` 对象并没有以一种我们能直接使用的格式来保存数据。要让它在应用程序中可用，我们就必须把这个响应数据解析成一个 JSON 对象。

解析成 JSON

响应数据是 JSON 字符串格式的。我们必须把这个字符串解析成 JavaScript 对象——只要调一下 `response.json()` 就可以了。

这不是 Angular 自己的设计。Angular HTTP 客户端遵循 ES2015 规范来处理 `Fetch` 函数返回的 `响应对象`。此规范中定义了一个 `json()` 函数，来把响应体解析成 JavaScript 对象。

我们不应该期待解码后的 JSON 直接就是一个英雄数组。调用的这个服务器总会把 JSON 结果包装进一个带 `data` 属性的对象中。我们必须解开它才能得到英雄数组。这是一个约定俗成的 Web API 行为规范，它是出于 [安全方面的考虑](#)。

不要对服务端 API 做任何假设。并非所有服务器都会返回一个带 `data` 属性的对象。

不要返回响应 (Response) 对象

`getHeroes()` 确实可以返回 HTTP 响应对象，但这不是最佳实践。数据服务的重点在于，对消费者隐藏与服务器交互的细节。调用 `HeroService` 的组件希望得到英雄数组。它并不关心我们如何得到它们。它也不在乎这些数据从哪里来。毫无疑问，它也不希望直接和一个响应对象打交道。

HTTP 的 GET 方法被推迟执行了

`http.get` 仍然没有发送请求！这是因为可观察对象是 **冷淡的**，也就是说，只有当某人 **订阅** 了这个可观察对象时，这个请求才会被发出。这个场景中的 **某人** 就是 `HeroListComponent`。

总是处理错误

一旦开始与 I/O 打交道，我们就必须准备好接受墨菲定律：如果一件倒霉事 **可能发生**，它就 **迟早会** 发生。我们可以在 `HeroService` 中捕获错误，并对它们做些处理。只有在用户可以理解并采取相应行动的时候，我们才把错误信息传回到组件，让组件展示给最终用户。

在这个简单的应用中，我们在服务和组件中都只提供了最原始的错误处理方式。

`catch` 操作符将错误对象传递给 `http` 的 `handleError` 方法。服务处理器 (`handleError`) 把响应对象记录到控制台中，把错误转换成对用户友好的消息，并且通过 `Observable.throw` 来把这个消息放进一个新的、用于表示“失败”的可观察对象。

app/toh/hero.service.ts (excerpt)

```
getHeroes (): Observable<Hero[]> {
  return this.http.get(this.heroesUrl)
    .map(this.extractData)
    .catch(this.handleError);
}

private handleError (error: Response | any) {
  // In a real world app, we might use a remote logging
  // infrastructure
  let errMsg: string;
```

```

if (error instanceof Response) {
  const body = error.json() || '';
  const err = body.error || JSON.stringify(body);
  errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
} else {
  errMsg = error.message ? error.message : error.toString();
}
console.error(errMsg);
return Observable.throw(errMsg);
}

```

HeroListComponent 错误处理

回到 `HeroListComponent`，这里我们调用了 `heroService.getHeroes()`。我们提供了 `subscribe` 函数的第二个参数来处理错误信息。它设置了一个 `errorMessage` 变量，被有条件的绑定到了 `HeroListComponent` 模板中。

app/toh/hero-list.component.ts (getHeroes)

```

getHeroes() {
  this.heroService.getHeroes()
    .subscribe(
      heroes => this.heroes = heroes,
      error => this.errorMessage = <any>error);
}

```

想看到它失败时的情况吗？在 `HeroService` 中把 API 的端点设置为一个无效值就行了。但别忘了恢复它。

往服务器发送数据

前面我们已经看到如何用一个 HTTP 服务从远端获取数据了。但我们还能再给力一点，让它可以创建新的英雄，并把它们保存到后端。

我们将为 `HeroListComponent` 创建一个简单的 `addHero()` 方法，它将接受新英雄的名字：

```
addHero (name: string): Observable<Hero> {
```

要实现它，我们得知道关于服务端 API 如何创建英雄的一些细节。

我们的 **数据服务器** 遵循典型的 REST 指导原则。它期待在和 `GET` 英雄列表的同一个端点上存在一个 `POST` 请求。它期待从请求体 (`body`) 中获得新英雄的数据，数据的结构和 `Hero` 对象相同，但是不带 `id` 属性。请求体应该看起来像这样：

```
{ "name": "Windstorm" }
```

服务器将生成 `id`，并且返回新英雄的完整 `JSON` 形式，包括这个生成的 `id`。该英雄的数据被塞进一个响应对象的 `data` 属性中。

现在，知道了这个 API 如何工作，我们就可以像这样实现 `addHero()` 了：

app/toh/hero.service.ts (additional imports)

```
import { Headers, RequestOptions } from '@angular/http';
```

app/toh/hero.service.ts (addHero)

```
addHero (name: string): Observable<Hero> {
  let headers = new Headers({ 'Content-Type': 'application/json' });
  let options = new RequestOptions({ headers: headers });

  return this.http.post(this.heroesUrl, { name }, options)
    .map(this.extractData)
```

```

    .catch(this.handleError);
}

```

请求头 (Headers)

我们通过 `Content-Type` 头告诉服务器，`body` 是 JSON 格式的。

接下来，使用 `headers` 对象来配置 `options` 对象。`options` 对象是 `RequestOptions` 的新实例，该类允许你在实例化请求时指定某些设置。这样，`Headers` 是 `RequestOptions` 中的一员。

在 `return` 声明中，`options` 是传给 `post` 方法的 **第三个** 参数，就像前面见过的那样。

JSON 结果

像 `getHeroes()` 中一样，我们可以使用 `extractData()` 辅助函数从响应中 **提取出数据**。

回到 `HeroListComponent`，我们看到 **该组件的** `addHero()` 方法中订阅了这个由 **服务中** 的 `addHero()` 方法返回的可观察对象。当有数据到来时，它就会把这个新的英雄对象追加 (push) 到 `heroes` 数组中，以展现给用户。

app/toh/hero-list.component.ts (addHero)

```

addHero (name: string) {
  if (!name) { return; }
  this.heroService.addHero(name)
    .subscribe(
      hero => this.heroes.push(hero),
      error => this.errorMessage = <any>error);
}

```

倒退为承诺 (Promise)

虽然 Angular 的 `http` 客户端 API 返回的是 `Observable<Response>` 类型的对象，但我们也把它转成 `Promise<Response>`。这很容易，并且在简单的场景中，一个基于

承诺 (Promise) 的版本看起来很像基于可观察对象 (Observable) 的版本。

可能“承诺”看起来更熟悉一些，但“可观察对象”有很多优越之处。

下面是使用承诺重写 `HeroService`，要特别注意那些不同的部分。

```
1.   getHeroes (): Promise<Hero[]> {
2.     return this.http.get(this.heroesUrl)
3.       .toPromise()
4.       .then(this.extractData)
5.       .catch(this.handleError);
6.   }
7.
8.   addHero (name: string): Promise<Hero> {
9.     let headers = new Headers({ 'Content-Type': 'application/json' });
10.    let options = new RequestOptions({ headers: headers });
11.
12.    return this.http.post(this.heroesUrl, { name }, options)
13.      .toPromise()
14.      .then(this.extractData)
15.      .catch(this.handleError);
16.  }
17.
18.  private extractData(res: Response) {
19.    let body = res.json();
20.    return body.data || {};
21.  }
22.
23.  private handleError (error: Response | any) {
24.    // In a real world app, we might use a remote logging infrastructure
25.    let errMsg: string;
26.    if (error instanceof Response) {
27.      const body = error.json() || '';
28.      const err = body.error || JSON.stringify(body);
29.      errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
30.    } else {
31.      errMsg = error.message ? error.message : error.toString();
32.    }
33.  }
34.
```

```

32.      }
33.      console.error(errMsg);
34.      return Promise.reject(errMsg);
35.  }

```

在本例中，你可以遵循承诺的 `then(this.extractData).catch(this.handleError)` 模式。

另外，你也可以调用 `toPromise(success, fail)`。可观察对象的 `map` 第一个参数为 **成功** 时的回调函数，它的第二个参数用 `.toPromise(this.extractData, this.handleError)` 来拦截失败。

我们的 `errorHandler` 也改用了一个失败的承诺，而不再是失败的可观察对象。

把诊断信息记录到控制台 也只是在承诺的处理链中多了一个 `then` 而已。

我们还得对调用方组件进行调整，让它期待一个 `Promise` 而非 `Observable`：

```

1.  getHeroes() {
2.    this.heroService.getHeroes()
3.      .then(
4.        heroes => this.heroes = heroes,
5.        error => this.errorMessage = <any>error);
6.  }
7.
8.  addHero (name: string) {
9.    if (!name) { return; }
10.   this.heroService.addHero(name)
11.     .then(
12.       hero => this.heroes.push(hero),
13.       error => this.errorMessage = <any>error);
14. }

```

唯一一个比较明显不同点是我们调用这个返回的承诺的 `then` 方法，而不再是 `subscribe`。我们给了这两个方法完全相同的调用参数。

细微却又关键的不同点是，这两个方法返回了非常不同的结果！

基于承诺的 `then` 返回了另一个承诺。我们可以链式调用多个 `then` 和 `catch` 方法，每次都返回一个新的承诺。

但 `subscribe` 方法返回一个 `Subscription` 对象。但 `Subscription` 不是另一个 `Observable`。它是可观察对象的末端。我们不能在它上面调用 `map` 函数或再次调用 `subscribe` 函数。`Subscription` 对象的设计目的是不同的，这从它的主方法 `unsubscribe` 就能看出来。

要理解订阅 `Subscription` 的实现和效果，请看 [Ben Lesh 关于可观察对象的演讲](#) 或者他在 [egghead.io](#) 的课程。

跨域请求：Wikipedia 范例

我们刚刚学习了用 Angular `Http` 服务发起 `XMLHttpRequests`。这是与服务器通讯时最常用的方法。但它不适合所有场景。

出于安全的考虑，网络浏览器会阻止调用与当前页面不“同源”的远端服务器的 `XHR`。所谓 **源** 就是 URI 的协议 (scheme)、主机名 (host) 和端口号 (port) 这几部分的组合。这被称为 [同源策略](#)。

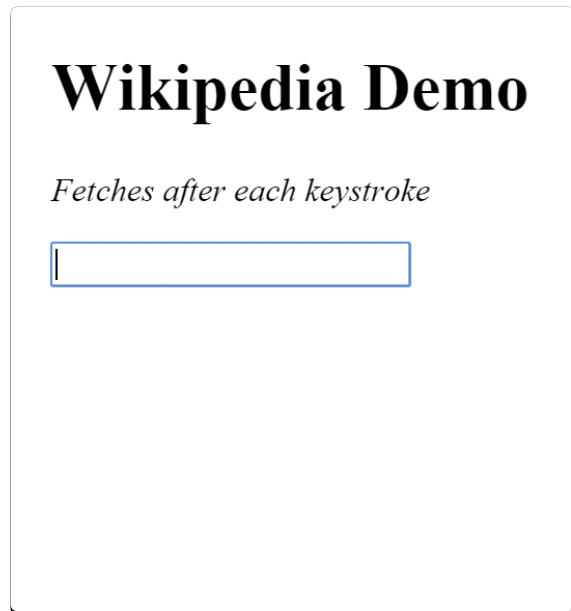
在现代浏览器中，如果服务器支持 `CORS` 协议，那么也可以向不同源的服务器发起 `XHR` 请求。如果服务器要请求用户凭证，我们就在 [请求头](#) 中启用它们。

有些服务器不支持 `CORS`，但支持一种老的、只读的（译注：即仅支持 GET）备选协议，这就是 `JSONP`。Wikipedia 就是一个这样的服务器。

这个 [StackOverflow 上的答案](#) 覆盖了关于 `JSONP` 的很多细节。

搜索 Wikipedia

我们来构建一个简单的搜索程序，当我们在文本框中输入时，它会从 Wikipedia 中获取并显示建议的词汇列表：



Wikipedia 提供了一个现代的 `CORS` API 和一个传统的 `JSONP` 搜索 API。在这个例子中，我们使用后者。Angular 的 `Jsonp` 服务不但通过 `JSONP` 扩展了 `Http` 服务，而且限制我们只能用 `GET` 请求。尝试调用所有其它 HTTP 方法都将抛出一个错误，因为 `JSONP` 是只读的。

像往常一样，我们把和 Angular 数据访问服务进行交互的代码全都封装在一个专门的服务中。我们称之为 `WikiService`。

```
app/wiki/wikipedia.service.ts
```

```
1. import { Injectable } from '@angular/core';
2. import { Jsonp, URLSearchParams } from '@angular/http';
3.
4. @Injectable()
5. export class WikipediaService {
6.   constructor(private jsonp: Jsonp) {}
7.
8.   search (term: string) {
9. }
```

```

10.     let wikiurl = 'http://en.wikipedia.org/w/api.php';
11.
12.     let params = new URLSearchParams();
13.     params.set('search', term); // the user's search value
14.     params.set('action', 'opensearch');
15.     params.set('format', 'json');
16.     params.set('callback', 'JSONP_CALLBACK');
17.
18.     // TODO: Add error handling
19.     return this.jsonp
20.         .get(wikiurl, { search: params })
21.         .map(response => <string[]> response.json()[1]);
22.     }
23.   }

```

这个构造函数期望 Angular 给它注入一个 `jsonp` 服务。前面我们已经把 `JsonpModule` 导入到了根模块中，所以这个服务已经可以使用了。

搜索参数

[Wikipedia 的 'opensearch' API](#) 期待在所请求的 URL 中带四个查询参数（键 / 值对格式）。这些键 (key) 分别是 `search`、`action`、`format` 和 `callback`。`search` 的值是用户提供的用于在 Wikipedia 中查找的关键字。另外三个参数是固定值，分别是 "`opensearch`"、"`json`" 和 "`JSONP_CALLBACK`"。

`JSONP` 技术需要我们通过查询参数传给服务器一个回调函数的名字：`callback=JSONP_CALLBACK`。服务器使用这个名字在它的响应体中构建一个 JavaScript 包装函数，Angular 最终会调用这个包装函数来提取出数据。这些都是 Angular 在背后默默完成的，你不会感受到它。

如果我们要找那些含有关键字“Angular”的文档，我们可以先手工构造出查询字符串，并像这样调用 `jsonp`：

```
let queryString =
`?
search=${term}&action=opensearch&format=json&callback=JSONP_CALLBACK`;

return this.jsonp
  .get(wikiUrl + queryString)
  .map(response => <string[]> response.json()[1]);
```

在更加参数化的例子中，我们会首选 Angular 的 `URLSearchParams` 辅助类来构建查询字符串，就像这样：

app/wiki/wikipedia.service.ts (search parameters)

```
let params = new URLSearchParams();
params.set('search', term); // the user's search value
params.set('action', 'opensearch');
params.set('format', 'json');
params.set('callback', 'JSONP_CALLBACK');
```

这次我们使用了 **两个** 参数来调用 `jsonp`： `wikiUrl` 和一个配置对象，配置对象的 `search` 属性是刚构建的这个 `params` 对象。

app/wiki/wikipedia.service.ts (call jsonp)

```
// TODO: Add error handling
return this.jsonp
  .get(wikiUrl, { search: params })
  .map(response => <string[]> response.json()[1]);
```

`Jsonp` 把 `params` 对象平面化为一个查询字符串，而这个查询字符串和以前我们直接放在请求中的那个是一样的。

WikiComponent 组件

现在，我们有了一个可用于查询 Wikipedia API 的服务，我们重新回到组件中，接收用户输入，并显示搜索结果。

app/wiki/wiki.component.html

```
1.  <h1>{{title}}</h1>
2.  <p><i>{{fetches}}</i></p>
3.
4.  <input #term (keyup)="search(term.value)"/>
5.
6.  <ul>
7.    <li *ngFor="let item of items | async">{{item}}</li>
8.  </ul>
```

app/wiki/wiki.component.ts

```
1.  import { Component }          from '@angular/core';
2.  import { Observable }         from 'rxjs/Observable';
3.
4.  import { WikipediaService }   from './wikipedia.service';
5.
6.  @Component({
7.    moduleId: module.id,
8.    selector: 'my-wiki',
9.    templateUrl: 'wiki.component.html',
10.   providers: [ WikipediaService ]
11. })
12. export class WikiComponent {
13.   title = 'Wikipedia Demo';
14.   fetches = 'Fetches after each keystroke';
15.   items: Observable<string[]>;
16.
17.   search (term: string) {
18.     this.items = this.wikipediaService.search(term);
19.   }
20.
21.   constructor (private wikipediaService: WikipediaService) { }
22. }
```

该组件有一个 `<input>` 元素，它是用来从用户获取搜索关键词的 **搜索框**。在每次 `keyup` 事件被触发时，它调用 `search(term)` 方法。

wiki/wiki.component.html

```
<input #term (keyup)="search(term.value)" />
```

`search(term)` 方法委托我们的 `WikipediaService` 服务来完成实际操作。该服务返回的是一个字符串数组的可观察对象 (`Observable<string[]>`)。该组件的内部订阅了这个可观察对象，就像我们曾在 `HeroListComponent` 中所做的那样，我们把这个可观察对象作为结果传给模板 (通过 `items` 属性)，模板中 `ngFor` 上的 `async(异步)` 管道会对这个订阅进行处理。

我们通常在只读组件中使用 `async` 管道，这种组件不需要与数据进行互动。

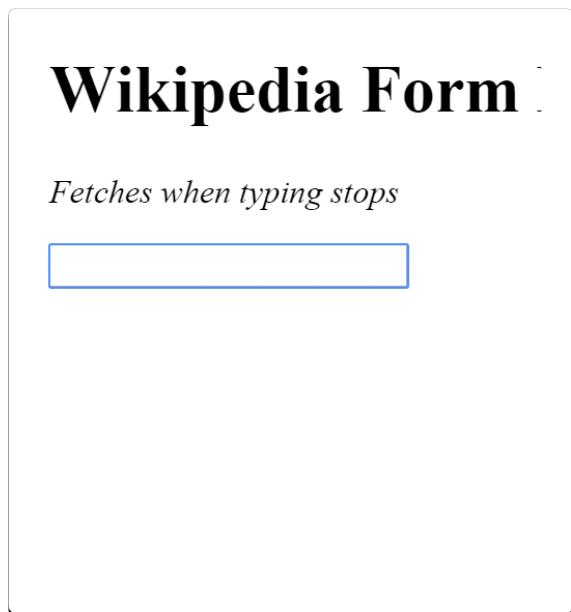
但我们不能在 `HeroListComponent` 中使用这个管道，这是因为“添加新英雄”特性会把一个新创建的英雄追加到英雄列表中。

奢侈的应用程序

我们这个 Wikipedia 搜索程序触发了过多的服务器调用 (每次按键发一次)。这样效率很低，而且在流量受限的移动设备上会显得过于昂贵。

1. 等用户停止输入

我们目前会在每次按键之后调用服务器。但合理的方式是只在用户 **停止输入** 之后才发起请求。这是它 **应该** 而且 **即将使用** 的工作方式，我们马上就重构它。



2. 当搜索关键字变化了才搜索

假设用户在输入框中输入了单词 `angular`，然后稍等片刻。应用程序就会发出一个对 `Angular` 的搜索请求。

然后，用户用退格键删除了最后三个字符 `lar`，并且毫不停顿的重新输入了 `lar`。搜索关键词仍然是“`angular`”。这时应用程序不应该发起另一个请求。

3. 对付乱序响应体

用户输入了 `angular`，暂停，清除搜索框，然后输入 `http`。应用程序发起了两个搜索请求，一个搜 `angular`，一个搜 `http`。

哪一个响应会先回来？我们是没法保证的。负载均衡器可能把这个请求分发给了响应时间不同的两台服务器。搜 `angular` 的结果可能晚于稍后搜 `http` 的结果。用户可能会困惑：为什么搜 `http` 时显示了关于 `angular` 的结果。

即使有多个尚未返回的请求，应用程序也应该按照原始请求的顺序展示对它们的响应。如果能让 `angular` 的结果始终在后面返回，就不会发生这样的混乱了。

Observable 的更多乐趣

借助一些漂亮的可观察能操作符，我们可以解决这些问题，并提升我们的应用程序。

我们本可以把这些改动合并进 `WikipediaService` 中，但是为了更好用户体验，转而拷贝 `WikiComponent`，把它变得更智能。下面是 `WikiSmartComponent`，它使用同样的模板：

```
app/wiki/wiki-smart.component.ts

1. import { Component }      from '@angular/core';
2. import { observable }     from 'rxjs/Observable';
3. import { Subject }        from 'rxjs/Subject';
4.
5. import { wikipediaService } from './wikipedia.service';
6.
7. @Component({
8.   moduleId: module.id,
9.   selector: 'my-wiki-smart',
10.  templateUrl: 'wiki.component.html',
11.  providers: [ wikipediaService ]
12. })
13. export class WikiSmartComponent {
14.   title = 'Smarter wikipedia Demo';
15.   fetches = 'Fetches when typing stops';
16.   items: Observable<string[]>;
17.
18.   private searchTermStream = new Subject<string>();
19.   search(term: string) { this.searchTermStream.next(term); }
20.
21.   constructor (private wikipediaService: wikipediaService) {
22.     this.items = this.searchTermStream
23.       .debounceTime(300)
24.       .distinctUntilChanged()
25.       .switchMap((term: string) =>
26.         this.wikipediaService.search(term));
27.   }
}
```

创建一个由搜索关键字组成的“流 (Stream) ”

模板仍然绑定搜索框的 `keyup` 事件，并在每次用户按键时，将搜索框的值传递到组件的 `search` 方法。

app/wiki/wiki.component.html (input)

```
<input #term (keyup)="search(term.value)" />
```

利用从 RxJS 库导入的 `Subject` , `WikiSmartComponent` 将搜索框的值变为一个 **搜索关键词流** 可观察对象 :

app/wiki/wiki-smart.component.ts

```
import { Subject } from 'rxjs/Subject';
```

组件创建 `searchTermStream` 为 `string` 类型的 `Subject` 。 `search` 方法通过 `subject` 的 `next` 方法 , 将每个新搜索框的值添加到数据流中。

app/wiki/wiki-smart.component.ts

```
private searchTermStream = new Subject<string>();
search(term: string) { this.searchTermStream.next(term); }
```

监听搜索关键字

以前 , 我们每次都把搜索关键字直接传给服务 , 并且把模板绑定到服务返回的结果。

而现在我们在监听 **关键字组成的流** , 并在把它传给 `WikipediaService` 之前操作这个流。

app/wiki/wiki-smart.component.ts

```
this.items = this.searchTermStream
  .debounceTime(300)
  .distinctUntilChanged()
  .switchMap((term: string) => this.wikipediaService.search(term));
```

我们先等待用户停止输入至少 300 毫秒 (`debounceTime`)。只有当搜索关键字变化的时候，才把它传给服务 (`distinctUntilChanged`)。

`WikipediaService` 服务为每个请求返回一个独立的可观察的字符串数组 (`Observable<string[]>`)。我们可以同时有多个 **发送中** 的请求，它们都在等服务器的回复，这意味着多个 **可观察的字符串数组** 有可能在任何时刻以任何顺序抵达。

`switchMap`(以前叫 `flatMapLatest`) 返回一个新的可观察对象，它组合了所有这些“可观察的字符串数组”，重新按照它们的原始请求顺序进行排列，然后把最近的一个搜索结果交付给调用者。

于是，最终显示的搜索结果列表和用户输入的搜索关键字在顺序上保持了一致。

在 [前面提过的 `rxjs-operators`](#) 文件中，我们把 `debounceTime`、
`distinctUntilChanged` 和 `switchMap` 操作符加到了 RxJS 的 `Observable` 类中。

预防跨站请求伪造攻击

在一个跨站请求伪造攻击（CSRF 或 XSRF）中，攻击者欺骗用户访问一个不同的网页，它带有恶意代码，秘密向你的应用程序服务器发送恶意请求。

客户端和服务器必须合作来抵挡这种攻击。Angular 的 `http` 客户端自动使用它默认的 `XSRFStrategy` 来完成客户端的任务。

`CookieXSRFStrategy` 支持常见的反 XSRF 技术，服务端发送一个随机生成的认证令牌到名为 `XSRF-TOKEN` 的 cookie 中。HTTP 客户端使用该令牌的值为所有请求添加一个 `X-XSRF-TOKEN` 页头。服务器接受这个 cookie 和页头，比较它们，只有在它们匹配的时候才处理请求。

参见 [安全章的关于 XSRF 讨论](#)，学习更多关于 XSRF 和 Angular 的 `XSRFStrategy` 的应对措施。

附录：《英雄指南》的内存 (in-memory) 服务器

如果我们只关心获取到的数据，我们可以告诉 Angular 从一个 `heroes.json` 文件中获取英雄列表，就像这样：

`app/heroes.json`

```
{
  "data": [
    { "id": 1, "name": "Windstorm" },
    { "id": 2, "name": "Bombasto" },
    { "id": 3, "name": "Magneta" },
    { "id": 4, "name": "Tornado" }
  ]
}
```

我们把英雄数组包装进一个带 `data` 属性的对象中，就像一个真正的数据服务器所应该做的那样。这样可以缓解由顶级 JSON 数组导致的 [安全风险](#)。

我们要像这样把端点设置为这个 JSON 文件：

`app/toh/hero.service.ts`

```
private heroesUrl = 'app/heroes.json'; // URL to JSON file
```

这在“**获取**”英雄数据的场景下确实能工作，但我们还想**保存**数据。我们不能把这些改动保存到 JSON 文件中，我们需要一个 Web API 服务器。在本章中，我们不想惹上配置和维护真实服务器的那些麻烦事。所以，我们转而使用一种**内存 Web API 仿真器**代替它。

内存 Web API 不是 Angular 内核的一部分。它是一个可选的服务，来自独立的 [angular-in-memory-web-api](#) 库。我们可以通过 npm(参见 `package.json`) 来

安装它， 并且通过 SystemJS(参见 [systemjs.config.js](#)) 把它注册进模块加载器。

内存 Web API 从一个带有 `createDb()` 方法的自定义类中获取数据，并且返回一个 map，它的主键 (key) 是一组名字，而值 (value) 是一组与之对应的对象数组。

这里是与范例中基于 JSON 的数据源完成相同功能的类：

app/hero-data.ts

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
export class HeroData implements InMemoryDbService {
  createDb() {
    let heroes = [
      { id: 1, name: 'Windstorm' },
      { id: 2, name: 'Bombasto' },
      { id: 3, name: 'Magneta' },
      { id: 4, name: 'Tornado' }
    ];
    return {heroes};
  }
}
```

确保 `HeroService` 的端点指向了这个 Web API：

app/toh/hero.service.ts

```
private heroesUrl = 'app/heroes'; // URL to web API
```

最后，把来自 HTTP 客户端的请求重定向到这个内存 Web API。

使用内存 Web API 服务模块很容易配置重定向，将 `InMemoryWebApiModule` 添加到 `AppModule.imports` 列表中，同时在 `HeroData` 类中调用 `forRoot` 配置方法。

app/app.module.ts

```
InMemoryWebApiModule.forRoot(HeroData)
```

工作原理

这次重定向非常容易配置，这是因为 Angular 的 `http` 服务把客户端 / 服务器通讯的工作委托给了一个叫做 `XHRBackend` 的辅助服务。

使用标准 Angular 提供商注册方法，`InMemoryWebApiModule` 替代默认的 `XHRBackend` 服务并使用它自己的内存存储服务。`forRoot` 方法来自模拟的英雄数据集的 **种子数据** 初始化了这个内存 Web API

`forRoot` 方法的名字告诉我们，应该只在设置根模块 `AppModule` 时调用 `InMemoryWebApiModule` 一次。不要再次调用它。

下面是修改过的（也是最终的）`app/app.module.ts` 版本，用于演示这些步骤。

app/app.module.ts (excerpt)

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { HttpModule, JsonpModule } from '@angular/http';

import { InMemoryWebApiModule }      from 'angular-in-memory-web-api';
import { HeroData }                 from './hero-data';

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

import { HeroListComponent }        from './toh/hero-list.component';
import { HeroListPromiseComponent } from './toh/hero-
list.component.promise';

import { wikiComponent }           from './wiki/wiki.component';
```

```
import { wikiSmartComponent } from './wiki/wiki-smart.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    JsonpModule,
    InMemoryWebApiModule.forRoot(HeroData)
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    HeroListPromiseComponent,
    wikiComponent,
    wikiSmartComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Import the `InMemoryWebApiModule` *after* the `HttpModule` to ensure that the `XHRBackend` provider of the `InMemoryWebApiModule` supersedes all others.

要想查看完整的源代码，请参见 [在线例子](#)。

生命周期钩子

Angular 调用指令和组件的生命周期钩子函数，包括它的创建、变更和销毁时。

组件生命周期

constructor

每个组件都有一个被 Angular 管理的生命周期。

ngOnChanges

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。

ngOnInit

Angular 提供了 **生命周期钩子**，把这些关键时刻暴露出来，赋予我们在它们发生时采取行动的能力。

ngDoCheck

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。

ngAfterContentInit

ngAfterContentChecked

ngAfterViewInit

ngAfterViewChecked

ngOnDestroy

目录

- 概述
- 每个钩子的目的和时机
- 接口是可选的（理论上）
- 其他 Angular 生命周期钩子
- 生命周期例子

- 全部钩子
 - Spy 刺探 OnInit 和 OnDestroy
 - OnChanges
 - DoCheck
 - AfterViewInit 和 AfterViewChecked
 - AfterContentInit 和 AfterContentChecked

试一试 [在线例子](#)。

组件生命周期钩子

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 在创建组建后立刻调用它：

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	目的和时机
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
ngOnInit	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。
ngDoCheck	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。
ngAfterContentInit	在第一轮 ngOnChanges 完成之后调用，只调用 一次 。
	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。
	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。

只适用于组件。

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

只适合组件。

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。

只适合组件。

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。

只适合组件。

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（理论上）？

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，它们也不是必须的。我们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 会去检测我们的指令和组件的类，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，我们还是强烈建议你在 TypeScript 指令类中添加接口，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，以便让我们这些开发者在使用时能做更多的控制。

生命周期练习

[在线例子](#) 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	描述
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	<p>指令也同样有生命周期钩子。我们新建了一个 <code>SpyDirective</code>，利用 <code>ngOnInit</code> 和 <code>ngOnDestroy</code> 钩子，在它所监视的每个元素被创建或销毁时输出日志。</p> <p>本例把 <code>SpyDirective</code> 应用到父组件里的 <code>ngFor</code> 英雄 重复器 (repeater) 的 <code><div></code> 里面。</p>
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 <code>changes</code> 对象作为参数去调用 <code>ngOnChanges</code> 钩子。展示了该如何理解和使用 <code>changes</code> 对象。
DoCheck	实现了一个 <code>ngDoCheck</code> 方法，通过它可以自定义变更检测逻辑。这里将会看到：Angular 会用什么频率调用这个钩子，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 <code>ngAfterViewInit</code> 和 <code>ngAfterViewChecked</code> 钩子。

AfterContent

展示了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，我们还把前一个例子中的 SpyDirective 用在 CounterComponent 上，来提供日志，可以同时观察到日志的创建和销毁过程。

接下来，我们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。我们之所以在 peek-a-boo 中这么做，只是为了观看 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。

如果我们点击 **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**，所以我们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时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，我们的英雄永远不会知道该指令的存在。

不开玩笑啦，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) 指令可以让我们在无法直接修改 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。但我们用一个指令就能监视它们了。

我们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。

```
// 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}`);
  }
}
```

我们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。

```
<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** 中的结构看起来是这样的：



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

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。

有经验的开发者认同组件的构建应该很便宜和安全。

Misko Hevery，Angular 项目的头，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我们决定显示它之前，我们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](#) 和 [HTTP](#) 章讲解了如何这样做。

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。如果我们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。

我们访问这些属性的第一次机会，实际上是 `ngOnChanges` 方法，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

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

在这个例子中，我们监听了 `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。

OnChangesComponent (ngOnChanges)

```
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。我们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`。

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

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

```
<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：

OnChanges

Power: sing, ski, fly
Hero.name: Windstorm windy

[Reset Log](#)

Windstorm windy can sing, ski, fly

-- Change Log --

```
hero: currentValue = {"name": "Windstorm"}, previousValue = {}
power: currentValue = "sing", previousValue = {}
power: currentValue = "sing,", previousValue = "sing"
power: currentValue = "sing, ", previousValue = "sing,"
power: currentValue = "sing, s", previousValue = "sing, "
power: currentValue = "sing, sk", previousValue = "sing, s"
power: currentValue = "sing, ski", previousValue = "sing, sk"
power: currentValue = "sing, ski,", previousValue = "sing, ski"
power: currentValue = "sing, ski, ", previousValue = "sing, ski,"
power: currentValue = "sing, ski, f", previousValue = "sing, ski,"
power: currentValue = "sing, ski, fl", previousValue = "sing, ski, f"
power: currentValue = "sing, ski, fly", previousValue = "sing, ski, fl"
```

[back to top](#)

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

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 `DoCheck` 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，我们就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: Hero.name:

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` 钩子可以监测到英雄的 `name` 什么时候发生了变化。但我们必须小心。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输入框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。

我们还看到，`ngOnChanges` 方法的调用方式与 [API 文档](#) 中是不一样的，这是因为 API 文档过时了。（译注：这是经过与官方开发组沟通得到的消息，由于代码快速迭代，因此 API 文档现在的更新不够及时，将来会进行一次系统的梳理和更正）

AfterView

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输入框中：

ChildComponent

```
@Component({
  selector: 'my-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>
<my-child-view></my-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，我们只能通过带 `@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。

如果我们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：

The screenshot shows a browser window with a yellow background. At the top, it says "AfterView". Below that, there's some code-like text: "-- child view begins --", a blue-outlined box containing "Magnetam", and "-- child view ends --". Underneath, it says "-- AfterView Logs --". There is a "Reset" button with a hand cursor hovering over it. The log area contains many repeated entries of "AfterViewChecked":
AfterView constructor: no child view
AfterViewInit: Magneta child view
AfterViewChecked: Magneta child view
AfterViewChecked (no change): Magneta child view (2x)
AfterViewChecked: Magnetam child view
AfterViewChecked (no change): Magnetam child view (2x)
AfterViewChecked: Magnetamm child view
AfterViewChecked (no change): Magnetamm child view (2x)
AfterViewChecked: Magnetammm child view
AfterViewChecked (no change): Magnetammm child view (2x)
AfterViewChecked: Magnetammmm child view
AfterViewChecked (no change): Magnetammmm child view (4x)
AfterViewChecked: Magnetammmmm child view
AfterViewChecked (no change): Magnetammmmm child view (2x)
AfterViewChecked: Magnetammmmm child view
AfterViewChecked (no change): Magnetammmmm child view (2x)
AfterViewChecked: Magnetammmmm child view

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

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。

Angular 1 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，我们不再通过模板来把子视图包含进来，而是改从 `AfterContentComponent` 的父组件中导入它。下面是父组件的模板。

AfterContentParentComponent (template excerpt)

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

注意，`<my-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——除非我们想把这些内容投影进这个组件中。

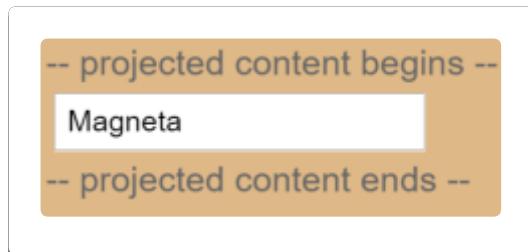
现在来看下 `<after-content>` 组件的模板：

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 `<my-child>` 标签。



下列迹象表明存在着 **内容投影**：(a) 在组件的元素标签中有 `HTML`；(b) 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，这里我们只能通过带有 `@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。所以我们仍然有机会去修改那个视图。

NPM包

推荐的 npm 包以及如何指定所依赖的包

Angular 应用程序以及 Angular 本身都依赖于很多第三方包 (包括 Angular 自己) 提供的特性和功能。这些包由 Node 包管理器 ([npm](#)) 负责安装和维护。

Node.js 和 npm 是做 Angular 2 开发的基础。

如果你的电脑上还没有装过，请 [立即获取它](#)！

通过在终端 / 控制台窗口中运行 `node -v` 和 `npm -v` 命令，来 **验证下你是否正在使用 node v5.x.x 和 npm 3.x.x**。过老的版本有可能出现问题。

我们建议使用来管理 node 和 npm 的多个版本。如果你机器上已经有某些项目运行了 node 和 npm 的其它版本，你就会需要 [nvm](#) 了。

我们在“快速起步”一章中 `package.json` 文件的 `dependencies` 和 `devDependencies` 区中指定了一组适用于新手的综合依赖包。

package.json (dependencies)

```
{  
  "dependencies": {  
    "@angular/common": "~2.1.1",  
    "@angular/compiler": "~2.1.1",  
    "@angular/core": "~2.1.1",  
    "@angular/forms": "~2.1.1",  
    "@angular/http": "~2.1.1",  
    "@angular/platform-browser": "~2.1.1",  
    "@angular/platform-browser-dynamic": "~2.1.1",  
    "@angular/router": "~3.1.1",  
    "@angular/router-deprecated": "~2.1.1",  
    "@angular/upgrade": "~2.1.1",  
    "core-js": "2.4.1",  
    "reflect-metadata": "0.1.3",  
    "rxjs": "5.0.1",  
    "zone.js": "0.7.4"  
  }  
}
```

```
    "@angular/http": "~2.1.1",
    "@angular/platform-browser": "~2.1.1",
    "@angular/platform-browser-dynamic": "~2.1.1",
    "@angular/router": "~3.1.1",
    "@angular/upgrade": "~2.1.1",
    "angular-in-memory-web-api": "~0.1.13",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.8",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.39",
    "zone.js": "^0.6.25"
  },
  "devDependencies": {
    "@types/core-js": "^0.9.34",
    "@types/node": "^6.0.45",
    "concurrently": "^3.0.0",
    "lite-server": "^2.2.2",
    "typescript": "^2.0.3"
  }
}
```

你当然可以使用其它包，不过我们建议你先使用 **这一组**，因为：(a) 我们知道它们可以很好的协同工作；(b) 它们包含了我们在个系列文档中构建和运行范例应用时所需的一切。

注意：烹饪宝典或开发指南中的页面可能需要其它库，比如 **jQuery**。

它们远远超过了我们将在“快速起步”中所需要用到的。实际上，它比我们在大多数应用中需要的还多。安装的包比我们实际需要的包多，其实并没有什么坏处。我们最终只会往客户端发送程序中实际用到的那些包。

本页面会解释每一个包是干什么的，以后你就可以根据自己的喜好和经验，随意替换它们了。

dependencies 和 devDependencies

package.json 包含两组包：[dependencies](#) 和 [devDependencies](#)。

dependencies 下的这些包是 **运行** 本应用的基础，而 **devDependencies** 下的只在 **开发** 此应用时才用得到。通过为 `install` 命令添加 `--production` 参数，你在产品环境下安装时排除 **devDependencies** 下的包，就像这样：

```
npm install my-application --production
```

dependencies

应用程序的 `package.json` 文件中，`dependencies` 区下有三类包：

- **特性** - 特性包为应用程序提供了框架和工具方面的能力。
- **填充 (Polyfills)** - 填充包弥合了不同浏览器上的 JavaScript 实现方面的差异。
- **其它** - 其它库对本应用提供支持，比如 `bootstrap` 包提供了 HTML 中的小部件和样式。

特性包

@angular/core - 框架中关键的运行期部件，每一个应用都需要它。包括所有的元数据装饰器：`Component`、`Directive`，依赖注入系统，以及组件生命周期钩子。

@angular/common - 常用的那些由 Angular 开发组提供的服务、管道和指令。

@angular/compiler - Angular 的 **模板编译器**。它会理解模板，并且把模板转化成代码，以供应用程序运行和渲染。开发人员通常不会直接跟这个编译器打交道，而是通过 `platform-browser-dynamic` 或离线模板编译器间接使用它。

@angular/platform-browser - 与 DOM 和浏览器相关的每样东西，特别是帮助往 DOM 中渲染的那部分。这个包还包含 `bootstrapStatic` 方法，用来引导那些在产品构建时需要离线预编译模板的应用程序。

@angular/platform-browser-dynamic - 为应用程序提供一些 提供商 和 bootstrap 方法，以便在客户端编译模板。不要用于离线编译。我们使用这个包在开发期间引导应用，以及引导 plunker 中的范例。

@angular/http - Angular 的 HTTP 客户端。

@angular/router - 路由器。

@angular/upgrade - 一组用于升级 Angular 1 应用的工具。

system.js - 是一个动态的模块加载器，与 ES2015 模块 规范兼容。还有很多其它选择，比如广受欢迎的 [webpack](#)。SystemJS 被用在了我们的文档范例中。因为它能工作。

今后，应用程序很可能还会需要更多的包，比如 HTML 控件、主题、数据访问，以及其他多种工具。

填充 (Polyfill) 包

在应用程序的运行环境中，Angular 需要某些 填充库。我们通过特定的 npm 包来安装这些填充库，Angular 本身把它列在了 `package.json` 中的 **peerDependencies** 区。

但我们必须把它列在我们 `package.json` 文件的 `dependencies` 区。

查看下面的“[为什么用 peerDependencies?](#)”，以了解这项需求的背景。

core-js - 为全局上下文 (`window`) 打的补丁，提供了 ES2015(ES6) 的很多基础特性。我们也可以把它换成提供了相同内核 API 的其它填充库。一旦所有的“主流浏览器”都实现了这些 API，这个依赖就可以去掉了。

reflect-metadata - 一个由 Angular 和 **TypeScript** 编译器共享的依赖包。开发人员需要能单独更新 TypeScript 包，而不用升级 Angular。这就是为什么把它放在本应用程序的依赖中，而不是 Angular 的依赖中。

rxjs - 一个为 [可观察对象 \(Observable\) 规范](#) 提供的填充库，该规范已经提交给了 [TC39 委员会](#)，以决定是否要在 JavaScript 语言中进行标准化。开发人员应该能在兼容的版本中选择一个喜欢的 **rxjs** 版本，而不用等 Angular 升级。

zone.js - 一个为 [Zone 规范](#) 提供的填充库，该规范已经提交给了 [TC39 委员会](#)，以决定是否要在 JavaScript 语言中进行标准化。开发人员应该能在兼容的版本中选择一个喜欢的 **zone.js** 版本，而不用等 Angular 升级。

其它辅助库

angular-in-memory-web-api - 一个 Angular 的支持库，它能模拟一个远端服务器的 Web API，而不需要依赖一个真实的服务器或发起真实的 HTTP 调用。对演示、文档范例和开发的早期阶段（那时候我们可能还没有服务器呢）非常有用。请到 [Http 客户端](#) 一章中了解更多知识。

bootstrap - [bootstrap](#) 是一个广受欢迎的 HTML 和 CSS 框架，可用来设计响应式网络应用。有些文档中的范例使用了 **bootstrap** 来强化它们的外观。

devDependencies

列在 `package.json` 文件中 **devDependencies** 区的包会帮助我们开发该应用程序。我们不用把它们部署到产品环境的应用程序中——虽然这样做也没什么坏处。

concurrently - 一个用来在 OS/X、Windows 和 Linux 操作系统上同时运行多个 `npm` 命令的工具

lite-server - 一个轻量级、静态的服务器，由 [John Papa](#) 开发和维护。对使用到路由的 Angular 程序提供了很好的支持。

typescript - TypeScript 语言的服务器，包含了 TypeScript 编译器 `tsc`。

@types/* - “TypeScript 定义”文件管理器。要了解更多，请参见 [TypeScript 配置](#) 页。

为什么使用 peerDependencies ?

在“快速起步”的 `package.json` 文件中，并没有 **peerDependencies** 区。但是 Angular 本身在 **它自己的** `package.json` 中有，它对我们的应用程序有重要的影响。

它解释了为什么我们要在“快速起步”的 `package.json` 文件中加载这些 **填充库 (polyfill)** 依赖包，以及为什么我们在自己的应用中会需要它们。

然后是对 **平级依赖 (peer dependencies)** 的简短解释。

每个包都依赖其它的包，比如我们的应用程序就依赖于 Angular 包。

两个包，“A”和“B”，可能依赖共同的第三个包“C”。 “A”和“B”可能都在它们的 **dependencies** 中列出了“C”。

如果“A”和“B”依赖于“C”的不同版本（“C1”和“C2”）。npm 包管理系统也能支持！它会把“C1”安装到“A”的 `node_modules` 目录下给“A”用，把“C2”安装到“B”的 `node_modules` 目录下给“B”用。现在，“A”和“B”都有了它们自己的一份“C”的副本，它们运行起来也互不干扰。

但是有一个问题。包“A”可能只需要“C1”出现就行，而实际上并不会直接调用它。“A”可能只有当 **每个人都使用 “C1” 时** 才能正常工作。如果程序中的任何一个部分依赖了“C2”，它就会失败。

要想解决这个问题，“A”就需要把“C1”定义为它的 **平级依赖**。

在 `dependencies` 和 `peerDependencies` 之间的区别大致是这样的：

dependency 说：“我需要这东西 **对我** 直接可用。”

peerDependency 说：“如果你想使用我，你得先确保这东西 **对你** 可用”

Angular 就存在这个问题。因此，Angular 的 `package.json` 中指定了一系列 **平级依赖** 包，把每个第三方包都固定在一个特定的版本上。

我们必须自己安装 Angular 的 peerDependencies。

当 npm 安装那些在 **我们的** `dependencies` 区指定的包时，它也会同时安装上在 **那些包** 的 `dependencies` 区所指定的那些包。这个安装过程是递归的。

但是在 npm 的第三版中，**它不会** 安装列在 `peerDependencies` 区的那些包。

这意味着，当我们的应用程序安装 Angular 时， npm 将不会自动安装列在 Angular 的 **peerDependencies** 区的那些包

幸运的是，npm 会在下列情况下给我们警告：(a) 当任何 **平级依赖** 缺失时或 (b) 当应用程序或它的任何其它依赖安装了与 **平级依赖** 不同版本的包时。

这些警告可以避免因为版本不匹配而导致的意外错误。它们让我们可以控制包和版本的解析过程。

我们的责任是，把所有 **平级依赖** 包都 **列在我们自己的 devDependencies 中**。

PEERDEPENDENCIES 的未来

Angular 的填充库依赖只是一个给开发人员的建议或提示，以便它们知道 Angular 期望用什么。它们不应该像现在一样是硬需求，但目前我们也不知道该如何把它们设置为可选的。

不过，有一个 npm 的新特性申请，叫做“可选的 peerDependencies”，它将会允许我们更好的对这种关系建模。一旦它被实现了，Angular 将把所有填充库从 **peerDependencies** 区切换到 **optionalPeerDependencies** 区。

管道

管道可以在模板中转换显示的内容。

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，我们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。我们很快就会发现，在很多不同的应用中，都在重复做出某些相同的变换。我们几乎会把它们看做某种 CSS 样式，事实上，我们也确实更喜欢在 HTML 模板中应用它们——就像 CSS 样式一样。

欢迎来到 Angular “管道 (Pipe) ”的世界！我们可以把这种简单的“值 - 显示”转换器声明在 HTML 中。试试 [在线例子](#)。

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。我们将把组件的 `birthday` 属性转换成对人类更友好的日期格式，来说明这一点：

app/herobirthday1.component.ts

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

@Component({
```

```
selector: '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。

```
<p>The hero's birthday is {{ birthday | date }}</p>
```

在这个插值表达式中，我们让组件的 `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。

`Date` 和 `Currency` 管道需要 **ECMAScript 国际化 (I18n) API**，但 Safari 和其它老式浏览器不支持它，该问题可以用垫片 (Polyfill) 解决。

```
<script src="https://cdn.polyfill.io/v2/polyfill.min.js?
features=Intl~locale.en"></script>
```

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。 我们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'`）。 如果我们的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`）。

我们将通过修改生日模板来给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。

```
<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](#)，比如字符串字面量或组件的属性。 换句话说，借助属性绑定，我们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。

我们来写第二个组件，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：

app/hero-birthday2.component.ts (template)

```
template: `
  <p>The hero's birthday is {{ birthday | date:format }}</p>
  <button (click)="toggleFormat()">Toggle Format</button>
`
```

我们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。

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; }
}
```

当我们点击按钮的时候，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

我们可以把管道链在一起，以组合出一些潜在的有用功能。下面这个例子中，我们把 `birthday` 链到 `DatePipe` 管道，然后又链到 `UpperCasePipe`，这样我们就可以把生日显示成大写形式了。比如下面的代码就会把生日显示成 APR 15, 1988：

```
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。

```
The chained hero's birthday is
{{ birthday | date:'fullDate' | uppercase}}
```

自定义管道

我们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：

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，比如我们这个管道中的 `exponent` (放大指数)。

- 我们通过 `@Pipe` 装饰器告诉 Angular : 这是一个管道。该装饰器是从 Angular 的 `core` 库中引入的。
- 这个 `@Pipe` 装饰器允许我们定义管道的名字，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，我们这个管道的名字是 `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。严格来说，它是可选的。Angular 不会管它，而是直接查找并执行 `transform` 方法。

现在，我们需要一个组件来演示这个管道。

app/power-booster.component.ts

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

@Component({
  selector: 'power-booster',
  template: `
    <h2>Power Booster</h2>
    <p>Super power boost: {{2 | exponentialStrength: 10}}</p>
  `
})
export class PowerBoosterComponent { }
```

Power Booster

Super power boost: 1024

要注意的有两点：

1. 我们使用自定义管道的方式和内置管道完全相同。
2. 我们必须在 `AppModule` 的 `declarations` 数组中包含这个管道。

别忘了 DECLARATIONS 数组！

如果我们忘了列出这个自定义管道，Angular 就会报告一个错误。在前一个例子中我们没有把 `DatePipe` 列进去，这是因为 Angular 所有的内置管道都已经预注册过了。但自定义管道必须手工注册。

如果我们试一下这个 [在线例子](#)，就可以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（加分项）

仅仅升级模板来测试这个自定义管道其实没多大意思。我们干脆把这个例子升级为“能力倍增计算器”，它可以把该管道和使用 `ngModel` 的双向数据绑定组合起来。

/app/power-boost-calculator.component.ts

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

@Component({
  selector: '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:

Boost factor:

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。

当我们使用管道时，Angular 选用了一种简单、快速的变更检测算法。

无管道

我们下一个例子中的组件使用默认的、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：

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。

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(); }
}

```

我们可以添加新的英雄，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。如果我们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

我们来往 `*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。

app/flying-heroes.component.html (flyers)

```

<div *ngFor="let hero of (heroes | flyingHeroes)">
  {{hero.name}}
</div>

```

下面是 `FlyingHeroesPipe` 的实现，它遵循了我们以前见过的那些写自定义管道的模式。

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);
  }
}
```

当运行例子时，我们看到一种奇怪的行为（试试[在线例子](#)）。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。

虽然我们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。

来看看我们是如何添加新英雄的：

```
this.heroes.push(hero);
```

当我们往 `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。**

我们可以修复它。让我们创建一个新数组，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。

如果我们**修改了**这个数组，没有管道被执行，也没有显示被更新。如果我们**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，来帮我们体验这些效果。

Flying Heroes

New flying hero: can fly

Mutate array Reset

Heroes who fly (piped)

Windstorm
Tornado

All Heroes (no pipe)

Windstorm
Bombasto
Magneto
Tornado

直接替换这个数组是通知 Angular 更新显示的一种高效方式。 我们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，我们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——扭曲我们的组件设计来适应管道。 我们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。

也行我们可以考虑用另一种管道来过滤会飞的英雄了，一个 **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。 我们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`，我们可以制作一个非纯管道。 我们可以像这样让 `FlyingHeroesPipe` 变成非纯的：

```
@Pipe({  
  name: 'flyingHeroesImpure',  
})
```

```
    pure: false  
})
```

在继续往下走之前，我们先理解一下 **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。如果我们更改了输入日期 (`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，如果我们要和变更检测策略打交道，就会更喜欢用纯管道。如果不能，我们就 **可以** 转回到非纯管道。

或者我们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，这一点我们等后面会再提起。

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，我们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。

非纯版本的 FlyingHeroesPipe

我们把 `FlyingHeroesPipe` 换成了 `FlyingHeroesImpurePipe`。下面是完整的实现：

```
@Pipe({
  name: 'flyingHeroesImpure',
  pure: false
})
export class FlyingHeroesImpurePipe extends FlyingHeroesPipe {}
```

我们把它从 `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。

```
return allHeroes.filter(hero => hero.canFly);
```

我们可以从 `FlyingHeroesComponent` 派生出一个 `FlyingHeroesImpureComponent`。

`app/flying-heroes-impure.component.html (FlyingHeroesImpureComponent)`

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

唯一重大改动就是管道。 我们可以在 [在线例子](#) 中确认，当我们输入新的英雄甚至修改 `heroes` 数组时，这个会飞的英雄的显示也跟着更新了。

非纯管道 `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。

而且它是有状态的。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。

在下面例子中，我们使用该 `async` 管道把一个消息字符串 (`message$`) 的 `Observable` 绑定到视图中。

app/hero-async-message.component.ts

```

1. import { Component } from '@angular/core';
2. import { observable } from 'rxjs/Rx';
3.
4. @Component({
5.   selector: 'hero-message',
6.   template: `
7.     <h2>Async Hero Message and AsyncPipe</h2>
8.     <p>Message: {{ message$ | async }}</p>
9.     <button (click)="resend()">Resend</button>`,
10. })
11. export class HeroAsyncMessageComponent {
12.   message$: Observable<string>;
13.
14.   private messages = [
15.     'You are my hero!',
16.     'You are the best hero!',
17.     'Will you be my hero?'
18.   ];
19.
20.   constructor() { this.resend(); }
21.
22.   resend() {
23.     this.message$ = observable.interval(500)
24.       .map(i => this.messages[i])
25.       .take(this.messages.length);
26.   }
27. }
```

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

我们来写更多的非纯管道：一个向服务器发起 HTTP 请求的管道。这通常是一个可怕的主意。不管我们怎么做，估计它都会是一个可怕的主意。但即便如此，为了讲解这个技术点，我们还是写一个吧。时刻记住，非纯管道可能每隔几微妙就会被调用一次。如果我们不小心点，这个管道就会发起一大堆请求“攻击”服务器。

我们确实得小心点。所以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。它会缓存这个请求的 URL，并等待一个结果，当结果发回来时，就缓存它。以后每当 Angular 调用此管道时，它都会返回这个缓存的结果（当请求尚未返回时，此结果是空），只在必要时才会去联系服务器。

下面就是使用 Angular http 获取 heroes.json 文件的代码：

app/fetch-json.pipe.ts

```
1. import { Pipe, PipeTransform } from '@angular/core';
2. import { Http } from '@angular/http';
3.
4. @Pipe({
5.   name: 'fetch',
6.   pure: false
7. })
8. export class FetchJsonPipe implements PipeTransform {
9.   private fetchedJson: any = null;
10.  private prevUrl = '';
11.
12.  constructor(private _http: Http) { }
13.
14.  transform(url: string): any {
15.    if (url !== this.prevUrl) {
16.      this.prevUrl = url;
17.      this.fetchedJson = null;
18.      this._http.get(url)
19.        .map( result => result.json() )
20.        .subscribe( result => this.fetchedJson = result );
21.    }
22.
23.    return this.fetchedJson;
24.  }
25.}
```

接下来我们用一个测试台组件演示一下它，该组件的模板中定义了两个使用到此管道的绑定。

app/hero-list.component.ts (template)

```
1. template: ` 
2.   <h2>Heroes from JSON File</h2>
3.
4.   <div *ngFor="let hero of ('heroes.json' | fetch) ">
5.     {{hero.name}}
6.   </div>
7.
8.   <p>Heroes as JSON:
9.     {{'heroes.json' | fetch | json}}
10.  </p>
11. `
```

尽管这里有两个绑定，并且我们知道它们频繁的调用了管道，但在浏览器的开发者工具中的“ Network ”页，我们可以确认只发起了一次到该文件的请求。

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" }]

JsonPipe

第二个绑定除了用到 `FetchPipe` 之外还链接了更多管道。我们把获取数据的结果同时显示在第一个绑定和第二个绑定中。第二个绑定中，我们通过链接到一个内置管道

JsonPipe 把它转成了 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。

下面是组件完整的实现代码：

app/hero-list.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'hero-list',
5.   template: `
6.     <h2>Heroes from JSON File</h2>
7.
8.     <div *ngFor="let hero of ('heroes.json' | fetch) ">
9.       {{hero.name}}
10.    </div>
11.
12.    <p>Heroes as JSON:
13.      {{'heroes.json' | fetch | json}}
14.    </p>
15.  `
16. })
17. export class HeroListComponent { }
```

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。

我们在本章前面见过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。 不久前我们刚看过的 FlyingHeroesImpurePipe，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。我们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。

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 Angular 1 的开发人员应该知道 `filter` 和 `orderBy` 过滤器，但在 Angular 2 中它们没有等价物。

这并不是疏忽。Angular 2 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。我们前面学过，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，即使是一中等规模的列表都可能严重降低用户体验。在 Angular 1 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 Angular 1 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。我们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>
```

我们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。

我们中的一些人可能不想做那么激进的最小化。但那不过是 **我们的** 选择而已。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。

路由与导航

揭示如何通过 Angular 路由进行基本的屏幕导航。

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。

本章覆盖了该路由器的主要特性。我们通过一个小型应用的成长演进来讲解它。参见 [在线例子](#)。

请点击右上角的蓝色 'X' 按钮以弹出预览窗口，这样可以看到浏览器地址栏的变化情况。



概览

浏览器是一个熟悉的应用导航操作模型。如果在地址栏中输入一个 URL，浏览器就会导航到相应的页面。如果你在页面中点击链接，浏览器就会导航到一个新的页面。如果你点击浏览器上的前进和后退按钮，浏览器就会根据你看过的页面历史向前或向后进行导航。

Angular 的 **路由器**（以下简称“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。我们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，我们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。

在本章中，我们将会学到关于路由器的更多细节知识：

- 设置 [页面的基地址 \(base href \)](#)
- 从 [路由器库](#) 中导入
- [配置路由器](#)
- 推动路由器导航的 [链接参数数组](#)，
- 在用户点击绑定到数据的 [RouterLink](#) 时进行导航
- 在 [程序的控制下](#) 进行导航
- 从 [当前路由获取信息](#)
- 为路由组件添加转场 [动画](#)
- 相对当前 URL 进行导航
- 利用 [`router-link-active` 指令] 切换 CSS 类 (#router-link-active)
- 用 [路由参数](#) 把重要信息嵌入 URL
- 在 [可选路由参数](#) 中提供非关键信息
- 重构路由到 [路由器模块](#)
- 在“特性分区”下添加 [子路由](#)
- 不借助组件 [对子路由进行分组](#)
- 从一个路由 [重定向](#) 到另一个路由
- 借助 [守卫函数](#) 确认或取消导航
 - 用 [CanActivate](#) 阻止导航进某路由
 - 用 [CanActivateChild](#) 阻止导航进某子路由

- 用 `CanDeactivate` 阻止离开当前路由的导航
 - 用 `Resolve` 在路由激活之前预先获取数据
 - 用 `CanLoad` 阻止异步路由
- 用 `查询参数` 提供跨路由的可选信息
 - 使用 `fragment` 跳转到其它元素
 - 异步 加载特性分区
 - 在导航时 预加载特性分区
 - 使用 `自定义策略` 来只预加载指定分区
 - 选择 "HTML5" 或 "hash" URL 风格

接下来，我们分为几个里程碑阶段，把一个简单的、只有占位 (placeholder) 视图的双页范例，升级成一个模块化的、带有子路由的多视图设计。

不过，还是先来做一个路由器基础知识的概览。

基础知识

我们从路由器的少量核心概念开始，然后在通过一系列的例子来了解它们的各种细节。

<base href>

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（就像我们的范例应用一样），那就把 `href` 的值设置为下面这样：

```
index.html (base-ref)
```

```
<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 2 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，我们可以从它导入所需的一切。

app/app.module.ts (import)

```
import { RouterModule } from '@angular/router';
```

我们将会在 [后面](#) 详细讲解其它选项。

配置

该应用将有一个 `router`（路由器）。当浏览器的地址变化时，该路由器会查找相应的 `Route`（路由定义，简称路由），并据此确定所要显示的组件。

需要先配置路由器，才会有路由信息。首选方案是用带有“路由数组”的 `provideRouter` 工厂函数（`[provideRouter(routes)]`）来启动此应用。

在下面的例子中，我们用四个路由定义配置了本应用的路由器。这个配置的 `RouterModule` 被添加到 `AppModule` 的 `imports` 数组中。

app/app.module.ts (excerpt)

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot([
      { path: 'hero/:id', component: HeroDetailComponent },
      { path: 'crisis-center', component: CrisisListComponent },
    ])
  ],
  declarations: [ AppComponent, HeroDetailComponent, CrisisListComponent ]
})
```

```
{  
  path: 'heroes',  
  component: HeroListComponent,  
  data: {  
    title: 'Heroes List'  
  },  
  { path: '', component: HomeComponent },  
  { path: '**', component: PageNotFoundComponent }  
}  
],  
declarations: [  
  AppComponent,  
  HeroListComponent,  
  HeroDetailComponent,  
  CrisisListComponent,  
  PageNotFoundComponent  
],  
bootstrap: [ AppComponent ]  

```

`RouterConfig` 是一个 **路由** 数组，它会决定如何导航。每个 **Route** 会把一个 URL 的 `path` 映射到一个组件。

`path` 中 **不能用斜线** `/` 开头。路由器会为我们解析和生成 URL，以便在多个视图间导航时，可以自由使用相对路径和绝对路径。

第一个路由中的 `: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，我们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、面包屑以及其它只读数据。本章稍后的部分，我们将使用 `resolve 守卫` 来获取更多数据。

第四个路由中的 `empty path` 匹配各级路由的默认路径。它还支持在不扩展 URL 路径的前提下添加路由。

第四个路由中的 `**` 代表该路由是一个 **通配符** 路径。如果当前 URL 无法匹配上我们配置过的任何一个路由中的路径，路由器就会匹配上这一个。当需要显示 404 页面或者重定向到其它路由时，该特性非常有用。

这些路由的定义顺序 是故意如此设计的。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，这是因为它是最通用的路由，应该 **只在** 前面找不到其它能匹配的路由时才匹配它。

路由插座

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` 中显示 `HeroListComponent` 组件。

```
<!-- Routed views go here -->
<router-outlet></router-outlet>
```

路由器链接

现在，我们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。

我们往 A 标签上添加了 `RouterLink` 指令。由于我们知道链接中不包含任何动态信息，因此我们使用一次性绑定的方式把它绑定到我们路由中的 `path` 值。

如果 `RouterLink` 需要动态信息，我们就可以把它绑定到一个能返回路由链接数组（**链接参数数组**）的模板表达式上。路由器最终会把此数组解析成一个 URL 和一个组件视图。

我们还往每个 A 标签上添加了一个 `RouterLinkActive` 指令，用于在相关的 `RouterLink` 被激活时为所在元素添加或移除 CSS 类。该指令可以直接添加到该元素上，也可以添加到其父元素上。

我们会在下面的 `AppComponent` 模板中看到类似这样的绑定：

```
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>
`
```

我们用 `RouterLink` 指令添加了两个带 `RouterLink` 和 `RouterLinkActive` 指令的 A 标签。每个 `RouterLink` 都绑定到了一个包含路由路径的字符串上。`'/crisis-center'` 和 `'/heroes'` 都是我们前面配置过的 `Routes` 中的路径。

在本章的 [后面](#)，我们还将学到如何写链接表达式，以及了解它们为什么是数组。

利用 `RouterLinkActive` 指令，我们把 `active` 作为当路由被激活时为 `RouterLink` 切换的 CSS 类。必要时，还可以为 `RouterLink` 添加多个类。

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。我们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。

路由器状态为我们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。

总结一下

为应用提供了一个配置过的路由器。 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 (路由器模块)	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 (路由插座)	该指令 (<code><router-outlet></code>) 用来标记出路由器该在哪里显示视图。
RouterLink (路由链接)	该指令用来把一个可点击的 HTML 元素绑定到路由。点击带有绑定到字符串 或 链接参数数组 的

routerLink 指令的 A 标签就会触发一次导航。

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。我们可以把一个 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。

我们已经对路由器及其能力有了肤浅的了解。

下面的详情区描述了一个带路由的范例应用，它经过一系列里程碑一步步演进。我们强烈建议你花点时间阅读并理解这个过程。

范例应用

从一个里程碑前进到另一个里程碑，我们总是有一个应用程序在心中。

虽然我们会渐进式的前进到最终的范例应用，但本章并不是一个教程。我们讨论路由和应用设计有关的代码和设计决策，并在这期间，处理遇到的所有问题。

完整代码可以在 [在线例子](#) 中找到。

我们的客户是“英雄管理局”。英雄们需要找工作，而“英雄管理局”为它们寻找待解决的危机。

本应用分为三个主要的特性区：

1. 一个 **危机中心** 区，它维护一个危机列表，用来分派给英雄。
2. 一个 **英雄** 区，它维护由英雄管理局雇佣的英雄列表。
3. 一个 **管理** 区，用来维护该中心危机列表和雇佣英雄的列表。

运行 [在线例子](#)。它打开了 **危机中心**，一会儿我们就会回到那里。

点击 **英雄** 链接，我们就会展现出一个英雄列表。

The screenshot shows a mobile-style interface with a header containing two tabs: "Crisis Center" and "Heroes". The "Heroes" tab is active and highlighted in blue. Below the tabs is the title "HEROES". A list of heroes is displayed in a table format with 16 rows. The first five rows have dark blue backgrounds, while the last eleven rows have light gray backgrounds. Each row contains a small dark blue box with a number from 11 to 16 followed by the hero's name. The names listed are Mr. Nice, Narco, Bombasto, Celeritas, Magneta, and RubberMan.

	英雄名称
11	Mr. Nice
12	Narco
13	Bombasto
14	Celeritas
15	Magneta
16	RubberMan
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	

选择其中之一，该应用就会把我们带到此英雄的编辑页面。

The screenshot shows the same mobile-style interface. The "Heroes" tab is still active. The title "HEROES" is present. Below it, the name "Magneta" is displayed in a large, bold font. Underneath the name, the text "Id: 15" is shown. Below that is a text input field containing the name "Magneta". At the bottom left is a "Back" button.

修改会立即见效。我们再点击“后退”按钮，该应用又把我们带回了英雄列表页。

另外我们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，我们就会前往 **危机中心** 页，那里列出了待处理的危机。

The screenshot shows a web application interface titled "CRISIS CENTER". At the top, there are two tabs: "Crisis Center" (which is active) and "Heroes". Below the tabs is a list of four crisis items, each with a numbered icon and a title:

- 1 Dragon Burning Cities
- 2 Sky Rains Great White Sharks
- 3 Giant Asteroid Heading For Earth
- 4 Procrastinators Meeting Delayed Again

选择其中之一，该应用就会把我们带到此危机的编辑页面。

The screenshot shows the "Giant Asteroid Heading For Earth" crisis detail page. It includes the following elements:

- Top navigation tabs: "Crisis Center" (active) and "Heroes".
- Title: "CRISIS CENTER".
- Section title: "Giant Asteroid Heading For Earth".
- Form fields:
 - Id: 3
 - Name:
- Buttons: "Save" and "Cancel".

这和 **英雄详情** 页略有不同。 **英雄详情** 会立即保存我们所做的更改。而 **危机详情** 页中，我们的更改都是临时的——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。

假如我们点击一个危机、做了修改，但是 **没有点击任何按钮**，可能点击了浏览器的后退按钮，也可能点击了“英雄”链接。

无论哪种情况，我们都应该弹出一个对话框。

The screenshot shows a confirmation dialog box with the following content:

The page at 127.0.0.1:8080 says: ×

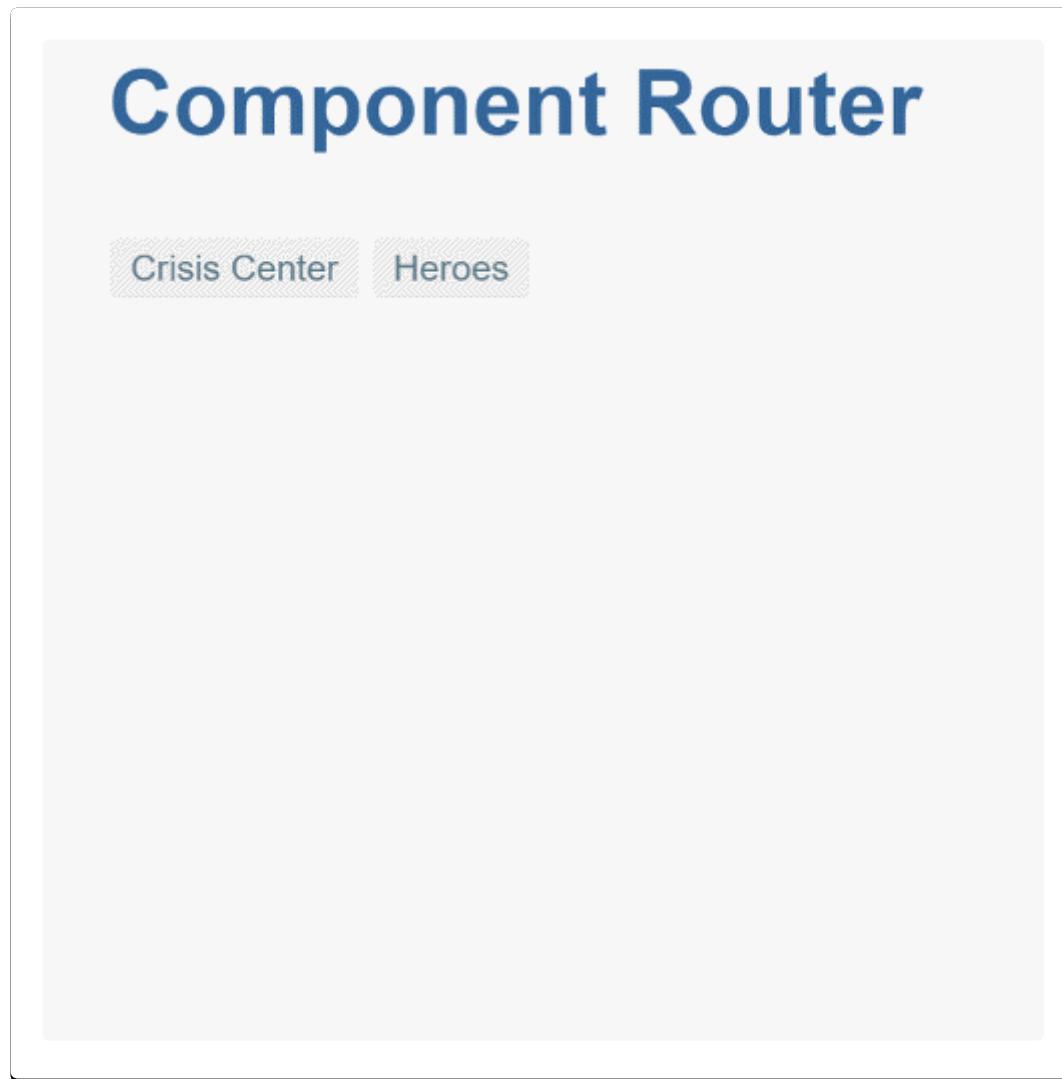
Discard changes?

OK Cancel

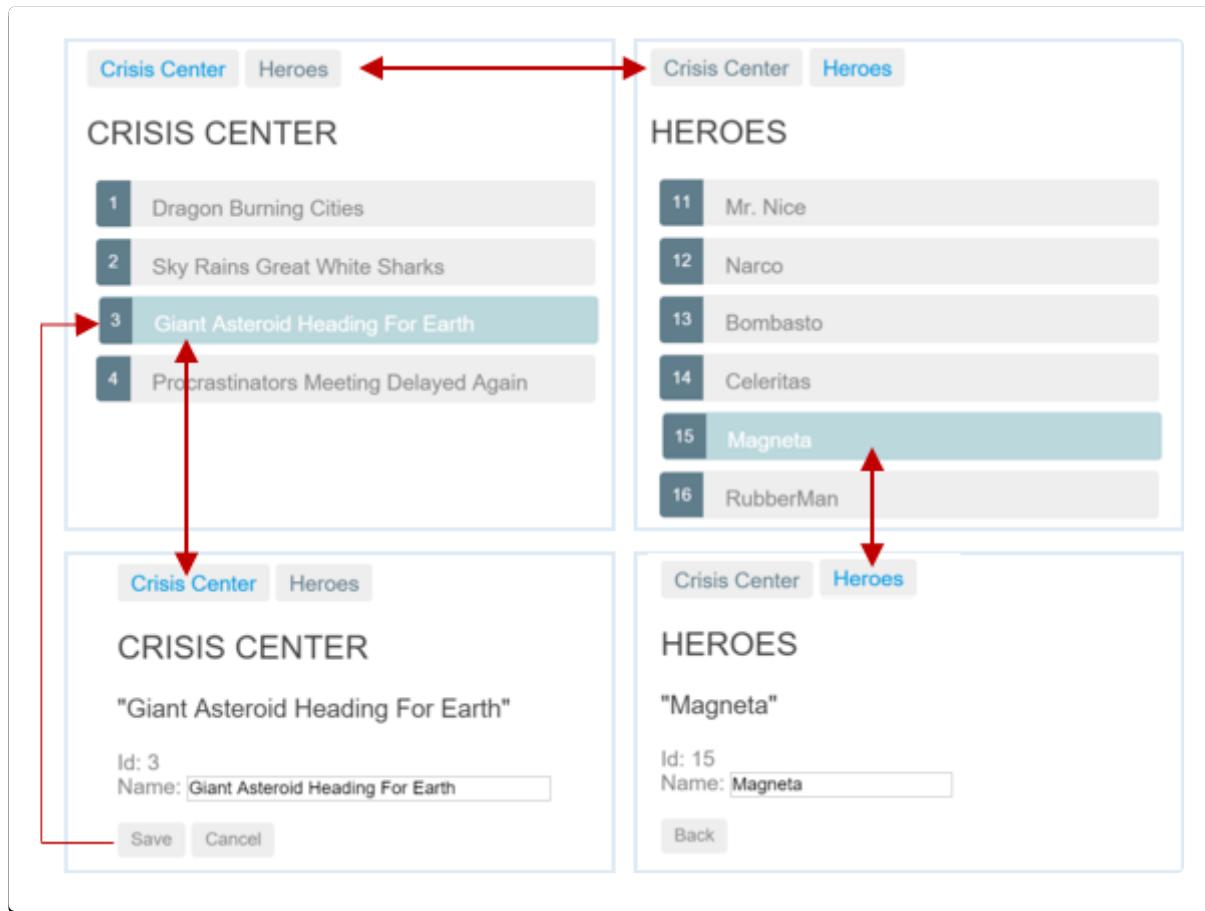
我们可以回答“确定”以放弃这些更改，或者回答“取消”来继续编辑。

路由器支持 `CanDeactivate` 守卫函数，它让我们有机会进行清理工作，或在离开当前视图之前征求用户的许可。

这里，我们看到的是一次完整的用户会话，它涉及到了所有这些我们要讲的特性。



下面是该应用中所有可选路由导航图：

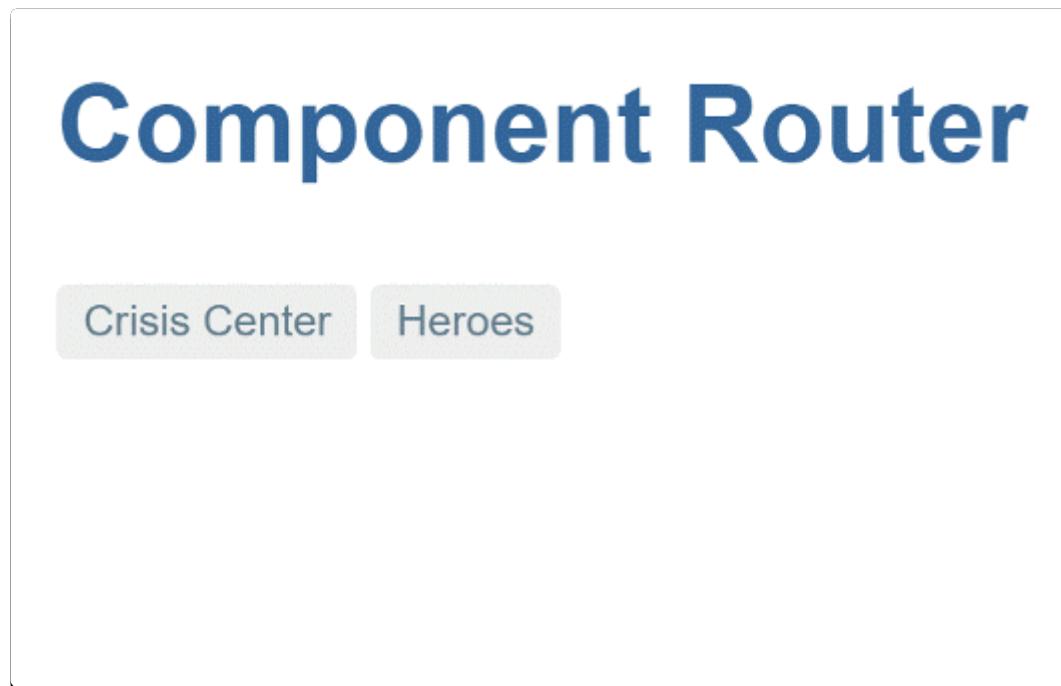


该应用展示了本章中涉及到的所有路由器特性：

- 把应用的特性组织成模块
- 导航到一个组件（英雄链接到“英雄列表”）
- 包含一个路由参数（当路由到“英雄详情”时传入该英雄的 id ）
- 子路由（危机中心有一组自己的路由）
- CanActivate 守卫（检查路由访问权）
- CanActivateChild 守卫（检查子路由访问权）
- CanDeactivate 守卫（询问是否丢弃未保存的修改）
- Resolve 守卫（预先获取路由数据）
- 惰性加载特性路由
- CanLoad 守卫（在加载特性模块之前进行检查）

里程碑 #1：从路由器开始

我们从该应用的一个简化版开始，它在两个空视图之间导航。



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，我们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`。虽然我们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。

我们必须往 `index.html` 中 **添加一个 `<base href>` 元素** 标签来让 `pushState` 路由正常工作。浏览器也需要这个 `base` 的 `href` 值，以便在下载和链接 css 文件、脚本和图片时，为那些 **相对 URL** 加前缀。

在 `<head>` 标签中的最前面添加 `base` 元素。如果 `app` 文件夹是应用的根目录（就像我们这个一样），那么就像下面展示的这样在 `index.html` 中设置 `href` 的值：

```
index.html (base-href)
<base href="/">
```

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。

在线例子说明

当我们运行在线例子时，我们不得不耍一点小花招，因为在线例子的宿主服务会动态设置应用的基地址。这就是为什么我们要把 `<base href...>` 替换成了一个脚本，用来动态写入 `<base>` 标签来适应这种情况。

```
<script>document.write('<base href="' + document.location +
'" />');</script>
```

我们只应该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。

为路由器配置一些路由

我们先从路由库中导入一些符号。

路由器在它自己的 `@angular/router` 包中。它不是 Angular 2 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，我们可以教它如何进行导航。

定义一些路由

路由器必须用“路由定义”的列表进行配置。

我们的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，我们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`，路由器就会创建或获取一个 `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，创建或者取回一个 `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。我们将路由数组传递到 `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。

app/app.module.ts (excerpt)

```
import { NgModule }           from '@angular/core';
import { BrowserModule }     from '@angular/platform-browser';
import { FormsModule }        from '@angular/forms';
import { RouterModule }       from '@angular/router';

import { AppComponent }       from './app.component';
import { CrisisListComponent } from './crisis-list.component';
import { HeroListComponent }   from './hero-list.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot([
      { path: 'crisis-center', component: CrisisListComponent },
      { path: 'heroes', component: HeroListComponent }
    ])
  ],
  declarations: [ AppComponent ]
})
```

```

        ])
],
declarations: [
  AppComponent,
  HeroListComponent,
  CrisisListComponent
],
bootstrap: [ AppComponent ]
})
export class AppModule {
}

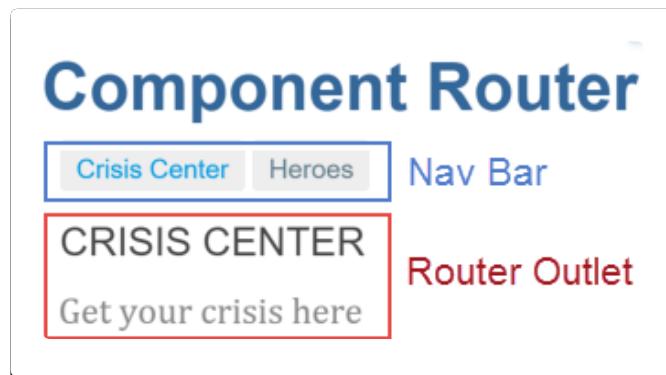
```

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，我们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、专门为特征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。

壳组件 `AppComponent`

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 **路由器插座**，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：

```
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>
```

RouterOutlet

`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。

一个模板中只能有一个 **未命名的** `<router-outlet>` 。 但路由器可以支持多个 **命
名的** 插座（ outlet ），将来我们会涉及到这部分特性。

RouterLink binding

在插座上方的 A 标签中，有一个绑定 `RouterLink` 指令的 **属性绑定**，就像这样：

`routerLink="..."` 。我们从路由库中导入了 `RouterLink` 。

例子中的每个链接都有一个字符串型的路径，也就是我们以前配置过的路由路径，但还没有指定路由参数。

我们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。我们还可以把 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。现在它已经可用于我们自己的模板中。

app.component.ts 目前看起来是这样的：

app/app.component.ts (excerpt)

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

@Component({
  selector: 'my-app',
  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 { }
```

“起步阶段”总结

我们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。

我们已经学会了如何：

- 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“HTML 5”模式的浏览器 URL。

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看起来是这样的：

```
router-sample
  app
    app.component.ts
    app.module.ts
    crisis-list.component.ts
    hero-list.component.ts
    main.ts
  node_modules ...
  index.html
  package.json
  styles.css
```

```
└─ tsconfig.json
```

下面是当前里程碑中讨论过的文件列表：

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
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 #2：路由模块

在原始的路由配置中，我们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，我们使用更多 **路由器** 特征，比如守卫、解析器和子路由等，我们很自然想要重构路由。建议将路由移到单独的文件，使用专门目的的服务，叫做 **路由模块**。

路由模块

- 分离路由与特征模块
- 测试特征模块时，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

- 与特征的 声明 无关

将路由重构为模块

在 `/app` 目录新建一个文件，名叫 `app-routing.module.ts`。路由模块将导入 `RouterModule` 令牌，并配置路由。我们遵循文件名约定，并命名 Angular 模块为 `AppRoutingModule`。

和 `app.module.ts` 中一样，导入 `CrisisListComponent` 和 `HeroListComponent` 组件。然后在路由模块中导入 `Router` 和路由配置 (`RouterModule.forRoot`)。

同时导出 `AppRoutingModule`，这样我们可以将它添加到 `AppModule` 的 `imports` 中。

最后，重新导出 `RouterModule`，这样，特征模块在使用 **路由模块** 时，将获得 **路由指令**。

下面是首个 **路由模块**：

app/app-routing.module.ts (excerpt)

```
import { NgModule }      from '@angular/core';
import { RouterModule } from '@angular/router';

import { CrisisListComponent } from './crisis-list.component';
import { HeroListComponent } from './hero-list.component';

@NgModule({
  imports: [
    RouterModule.forRoot([
      { path: 'crisis-center', component: CrisisListComponent },
      { path: 'heroes', component: HeroListComponent }
    ])
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

接下来，我们将更新 `app.module.ts` 文件，从 `app-routing.module.ts` 中导入 `AppRoutingModule` 令牌，并用它替换 `RouterModule.forRoot`。

app/app.module.ts (excerpt)

```
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';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    CrisisListComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

应用继续正常运行，我们可以把路由模块作为为每个特征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 在根模块或者特征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`）。

我们建议你选择其中一种模式，并坚持模式的一致性。

大多数开发者应该采用路由模块，以保持一致性。

它在配置复杂时，能确保代码干净。它让测试特征模块更加容易。它的存在突出了模块时被路由的事实。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

我们刚刚学习了如何用 `RouterLink` 指令进行导航。

现在，我们将学习一些新的技巧，比如该如何：

- 用模块把应用和路由组织为一些 **特性区**
- 命令式地从一个组件导航到另一个组件
- 通过路由传递必要信息和可选信息

作为演示，我们将构建出 **英雄** 特性区。

英雄“特性区”

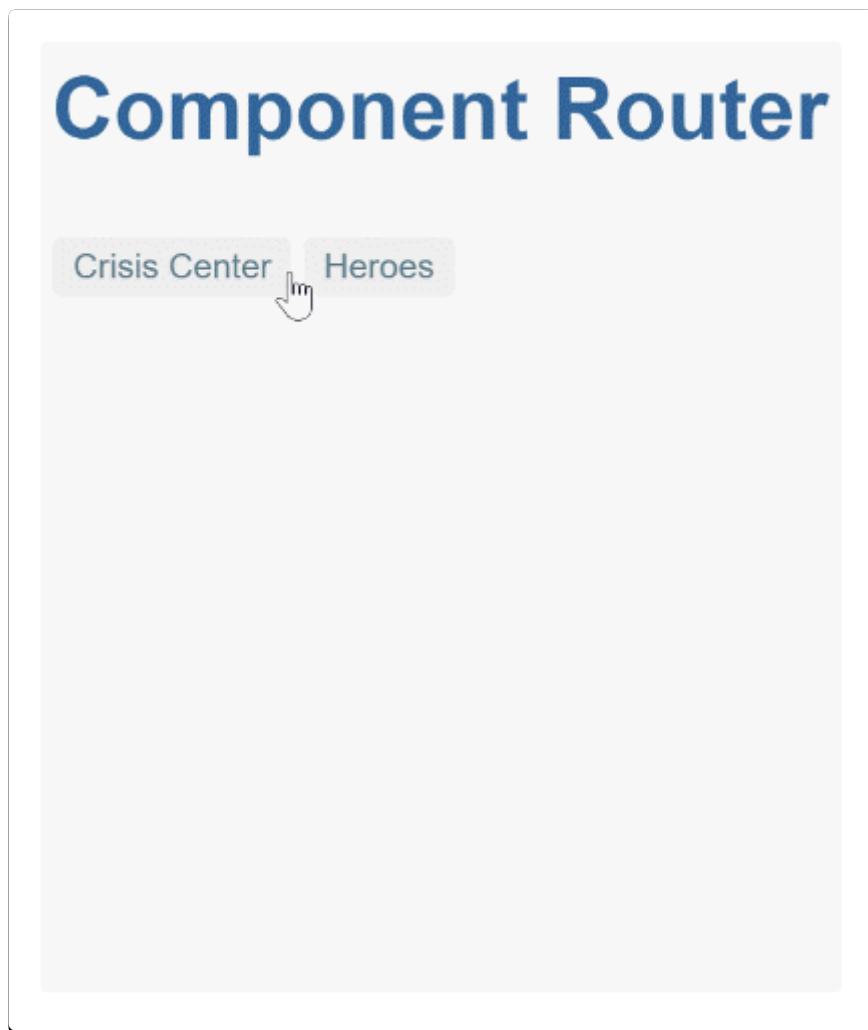
典型的应用中有多个 **特性区**，每个区都是一个“功能岛”，它们有自己的工作流、实现一个特定的业务目标。

我们可以继续把文件全添加到 `app/` 目录中。但那么做不太现实，并且最终将无法维护。因此，把每个特性区都放进自己的目录中会更好一些。

第一步是 **创建一个独立的 `app/heroes/` 文件夹**，并在其中添加属于 **英雄管理** 特性区的文件。

我们没有在这里引进新东西。这个例子从代码上和功能上来看，几乎就是“[教程：英雄指南](#)”的一份拷贝。

下面是该版本应用的用户体验演示：



添加“英雄”功能

我们要把应用拆成不同的 **特性模块**，然后把它们导入我们的主模块中，以便使用它们。

首先，我们要在 heroes 目录下创建一个 `heroes.module.ts` 文件。

我们删除了位于 `app/` 目录下的占位文件 `hero-list.component.ts`。

然后在 `app/heroes/` 目录下创建了一个 `hero-list.component.ts` 文件，并从上面的教程中把 `heroes.component.ts` 最终版的内容拷贝进来。再把 `hero-detail.component.ts` 和 `hero.service.ts` 文件拷贝到 `heroes/` 目录下。

我们在 `Heroes` 模块的 `providers` 数组中提供了 `HeroService`，以便它可用于模块中的所有组件。

我们的 `Heroes` 模块准备好路由了。

app/heroes/heroes.module.ts (excerpt)

```
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，我们就有了四个 **英雄管理** 特性区的文件：

```
app/heroes
  └── hero-detail.component.ts
  └── hero-list.component.ts
  └── hero.service.ts
  └── heroes.module.ts
```

现在到时间做一些“外科手术”了。我们利用应用程序的路由器，把这些文件和应用的其它部分联合起来。

英雄 特性区的路由需求

新的“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，我们导航到它，它会自行获取英雄列表并显示它们，该组件不需要任何外部信息。

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。

在这个例子中，当用户从列表中选择了一个英雄时，我们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

我们推荐的方式是为每个特性区创建它自己的路由配置文件。

在 `heroes` 目录下创建一个新的 `heroes-routing.module.ts` 文件，就像这样：

app/heroes/heroes-routing.module.ts (excerpt)

```
import { NgModule }      from '@angular/core';
import { RouterModule } from '@angular/router';

import { HeroListComponent }    from './hero-list.component';
import { HeroDetailComponent } from './hero-detail.component';

@NgModule({
  imports: [
    RouterModule.forChild([
      { path: 'heroes', component: HeroListComponent },
      { path: 'hero/:id', component: HeroDetailComponent }
    ])
  ],
  exports: [
    RouterModule
  ]
})
export class HeroRoutingModule { }
```

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。

我们使用与 `app.routes.ts` 中一样的技巧。

从它们所在的新 `app/heroes/` 目录导入列表和详情组件，定义两个英雄路由并导出到 `HeroesRoutes` 数组。

现在，我们的 `Heroes` 模块有路由了，我们得用 **路由器** 注册它们。我们像在 `app.routing.ts` 中那样导入 `RouterModule`，但这里稍微有一点不同。在设置 `app.routing.ts` 时，我们使用了静态的 `forRoot` 方法来注册我们的路由和全应用级服务提供商。在特性模块中，我们要改用 `Router.forChild` 静态方法。

`RouterModule.forRoot` 只能由 `AppModule` 提供。但我们位于特性模块中，所以使用 `RouterModule.forChild` 来单独注册更多路由。

我们在 `Heroes` 模块中从 `heroes-routing.module.ts` 中导入 `HeroRoutingModule`，并注册其路由。

app/heroes/heroes.module.ts (heroes routing)

```
import { HeroRoutingModule } from './heroes-routing.module';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    HeroRoutingModule
  ],
  declarations: [
    HeroListComponent,
    HeroDetailComponent
  ]
})
```

```
    HeroListComponent,  
    HeroDetailComponent  
],  
providers: [  
    HeroService  
]  
})
```

带参数的路由定义

HeroDetailComponent 的路由有点特殊。

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，我们期待路由器把英雄的 `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”，我们会期望这个英雄的 `id`（15）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：必选还是可选？

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。

当我们把一个 可选 值传给 `HeroDetailComponent` 时， 可选路由参数 可能是一个更好的选择。

命令式地导航到英雄详情

这次我们不打算通过点击链接来导航到详情组件，因此也不用再把带 `RouterLink` 的新的 A 标签加到壳组件中。

取而代之，当用户在列表中 点击 一个英雄时，我们将 命令 路由器导航到所选英雄的详情视图。

我们将调整 `HeroListComponent` 来实现这些任务。先从构造函数开始改：它通过依赖注入系统获得路由器服务和 `HeroService` 服务。

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

```
constructor(  
  private router: Router,  
  private service: HeroService) { }
```

还要对模板进行一点修改：

```
template: `  
  <h2>HEROES</h2>  
  <ul class="items">  
    <li *ngFor="let hero of heroes"  
        (click)="onSelect(hero)">  
      <span class="badge">{{hero.id}}</span> {{hero.name}}  
    </li>  
  </ul>
```

模板像 以前 一样定义了一个 `*ngFor` 重复器。还有一个 `(click)` 事件绑定，绑定到了组件的 `onSelect` 方法，就像这样：

app/heroes/hero-list.component.ts (select)

```
onSelect(hero: Hero) {  
  this.router.navigate(['/hero', hero.id]);  
}
```

它用一个 **链接参数数组** 调用路由器的 `navigate` 方法。如果我们想把它用在 HTML 中，那么也可以把相同的语法用在 `RouterLink` 中。

在列表视图中设置路由参数

我们将导航到 `HeroDetailComponent` 组件。在那里，我们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**。

app/heroes/hero-list.component.ts (link-parameters-array)

```
['/hero', hero.id] // { 15 }
```

路由器从该数组中组合出一个合适的两段式目标 URL：

```
localhost:3000/hero/15
```

在详情视图中获得路由参数

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。

ActivatedRoute : 一站式获得路由信息

每个路由都包含路径、数据参数、URL 片段等很多信息。所有这些信息都可以通过有路由器提供的一个叫 [ActivatedRoute](#) 的服务提供商来获取。

`ActivatedRoute` 包含你需要从当前路由组件中获得的全部信息，正如你可以从 `RouterState` 中获得关于其它激活路由的信息。

url : 该路由路径的 [Observable](#) 对象。它的值是一个由路径中各个部件组成的字符串数组。

data : 该路由提供的 `data` 对象的一个 [Observable](#) 对象。还包含从 `resolve` 守卫 中解析出来的值。

params : 包含该路由的必选参数和 可选参数 的 [Observable](#) 对象。

queryParams : 一个包含对所有路由都有效的 [查询参数](#) 的 [Observable](#) 对象。

fragment : 一个包含对所有路由都有效的 [片段](#) 值的 [Observable](#) 对象。

outlet : `RouterOutlet` 的名字，用于指示渲染该路由的位置。对于未命名的 `RouterOutlet`，这个名字是 **primary**。

routeConfig : 与该路由的原始路径对应的配置信息。

parent : 当使用 [子路由](#) 时，它是一个包含父路由信息的 [ActivatedRoute](#) 对象。

firstChild : 包含子路由列表中的第一个 [ActivatedRoute](#) 对象。

children : 包含当前路由下激活的全部 [子路由](#)。

我们要从路由器 (`router`) 包中导入 `Router`、`ActivatedRoute` 和 `Params` 类。

app/heroes/hero-detail.component.ts (activated route)

```
import { Router, ActivatedRoute, Params } from '@angular/router';
```

通常，我们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。

app/heroes/hero-detail.component.ts (constructor)

```
constructor(
  private route: ActivatedRoute,
  private router: Router,
  private service: HeroService) {}
```

然后，在 `ngOnInit` 方法中，我们用 `ActivatedRoute` 服务来接收本路由的参数。由于这些参数是作为 `Observable`（可观察对象）提供的，所以我们 **订阅**（`subscribe`）它们，通过名字引用 `id` 参数，并告诉 `HeroService` 获取指定 `id` 的英雄。我们还要保存这个 `Subscription`（订阅的返回值）的引用，供后面做清理工作。

app/heroes/hero-detail.component.ts (ngOnInit)

```
ngOnInit() {
  this.route.params.forEach((params: Params) => {
    let id = +params['id']; // (+) converts string 'id' to a number
    this.service.getHero(id).then(hero => this.hero = hero);
  });
}
```

在创建了 `HeroDetailComponent` 之后，Angular 很快就会调用 `ngOnInit` 方法。

我们要把数据访问逻辑放进 `ngOnInit` 方法中而不是构造函数中，以提高该组件的可测试性。在后面的 `OnInit` 附录中，我们会再详细讲解这一点。

要学习关于 `ngOnInit` 和 `ngOnDestroy` 方法的更多知识，参见 [生命周期钩子](#) 一章。

OBSERVABLE 参数与组件复用

在这个例子中，我们订阅了路由参数的 `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。

我们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。

但是 `ngOnInit` 对每个实例只调用一次。我们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。

快照：不需要可观察（NO-OBSERVABLE）时的替代方案

本应用不需要复用 `HeroDetailComponent`。我们总会先返回英雄列表，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。这意味着我们每次都会得到一个全新的 `HeroDetailComponent` 实例。

假如我们很确定 `HeroDetailComponent` 组件 永远、永远 不会被复用，每次导航到英雄详情时都会重新创建该组件。

路由器提供了一个备选方案：**快照（snapshot）**，它会给我们路由参数的初始值。这样我们就不用订阅，也不用被迫在 `ngDestroy` 中反订阅了。这样会更容易书写和阅读：

app/heroes/hero-detail.component.ts (ngOnInit snapshot)

```
ngOnInit() {
  // (+) converts string 'id' to a number
  let id = +this.route.snapshot.params['id'];
  this.service.getHero(id).then(hero => this.hero = hero);
}
```

记住：，用这种技巧，我们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 `params` 可观察对象的方式。我们在这里选择使用 `params` 可观察对象策略，以防万一。

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**，我们也可以把它绑定到 `[routerLink]` 指令上。它保存着 **到 `HeroListComponent` 组件的路径**：

app/heroes/hero-detail.component.ts (excerpt)

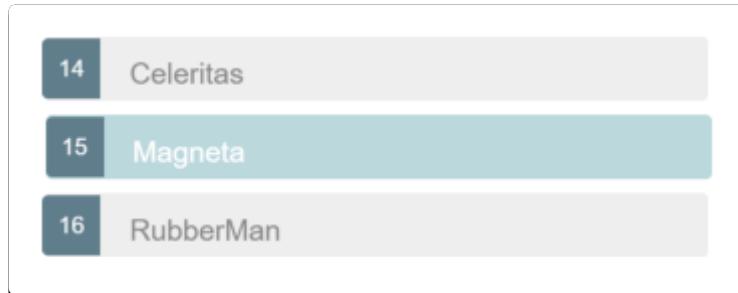
```
gotoHeroes() { this.router.navigate(['/heroes']); }
```

路由参数

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

有时我们希望往路由请求中添加 **可选的** 信息。例如，`HeroListComponent` 虽然不需要借助此信息显示英雄列表，但是如果从 `HeroDetailComponent` 返回时，它能自动选中刚刚查看过的英雄就好了。



如果我们能在从 `HeroDetailComponent` 返回时在 URL 中带上英雄 Magneta 的 `id`，不就可以了吗？接下来我们就尝试实现这个场景。

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。即使我们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。我们在定义完必要参数之后，通过一个 **对象** 来定义 **可选参数**。

路由参数：用必要的还是可选的？

并没有一劳永逸的规则，通常：

下列情况下优先使用必要参数

- 该值是必须的。
- 该值在区分此路由与其它路由时是必要的。

下列情况下优先使用可选参数

- 该值是可选的、复杂的，和 / 或多变量的。

路由参数

要导航到 `HeroDetailComponent`，我们需要在 **路由参数** 中指定要编辑英雄的必要参数 `id`，把这个 `id` 作为 **链接参数数组** 的第二个条目。

```
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，这是因为我们把它用一个 `:id` 占位符当做路由参数定义在了路由的 `path` 中：

```
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`。

```
app/heroes/hero-detail.component.ts (gotoHeroes)
```

```
gotoHeroes() { this.router.navigate(['/heroes']); }
```

该数组缺少一个路由参数，这是因为我们那时没有理由往 `HeroListComponent` 发送信息。

但现在有了。我们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。

我们通过一个包含 **可选** `id` 参数的对象来做到这一点。为了演示，我们还定义了一个没用的参数（`foo`），`HeroListComponent` 应该忽略它。下面是修改过的导航语句：

app/heroes/hero-detail.component.ts (go to heroes)

```
gotoHeroes() {
  let heroId = this.hero ? this.hero.id : null;
  // Pass along the hero id if available
  // so that the HeroList component can select that hero.
  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。

当在 plunker 中运行时，请点击右上角的蓝色‘X’按钮来弹出预览窗口，否则你看不到地址栏的变化。



[Launch the preview in a separate window](#)

看起来应该是这样，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——我们以前可能从未见过。

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。

这种语法对我们来说可能有点奇怪，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 高亮了选中的行，因为它演示的是应用的最终状态，因此包含了我们 **即将** 示范的步骤。此刻，我们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。我们这就改变这一点。

以前，当从 `HeroListComponent` 导航到 `HeroDetailComponent` 时，我们通过 `ActivatedRoute` 服务订阅了路由参数这个 `Observable`，并让它能用在 `HeroDetailComponent` 中。我们把该服务注入到了 `HeroDetailComponent` 的构造函数中。

这次，我们要进行反向导航，从 `HeroDetailComponent` 到 `HeroListComponent`。

首先，我们扩展该路由的导入语句，以包含进 `ActivatedRoute` 服务的类；

app/heroes/hero-list.component.ts (import)

```
import { Router, ActivatedRoute, Params } from '@angular/router';
```

然后，使用 `ActivatedRoute` 来访问 `params` 这个 `Observable`，以便我们订阅它，并把其中的 `id` 参数提取到 `selectedId` 中：

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

```

private selectedId: number;

constructor(
  private service: HeroService,
  private route: ActivatedRoute,
  private router: Router
) {}

ngOnInit() {
  this.route.params.forEach((params: Params) => {
    this.selectedId = +params['id'];
    this.service.getHeroes()
      .then(heroes => this.heroes = heroes);
  });
}

```

所有的路由参数或查询参数都是字符串。`params['id']` 表达式前面的加号（`+`）是一个 JavaScript 的小技巧，用来把字符串转换成整数。

我们添加了一个 `isSelected` 方法，当英雄的 id 和选中的 id 匹配时，它返回真值。

app/heroes/hero-list.component.ts (isSelected)

```
isSelected(hero: Hero) { return hero.id === this.selectedId; }
```

最后，我们用 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 `` 标记中找到它，就像这样：

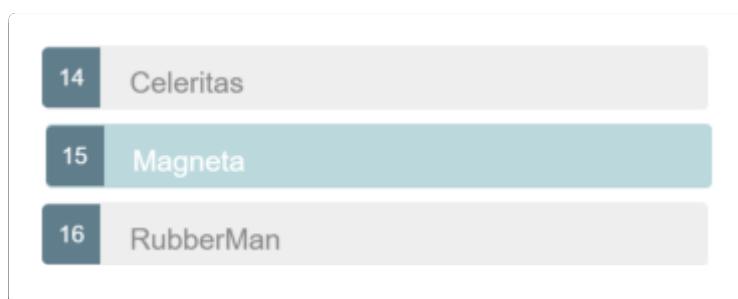
app/heroes/hero-list.component.ts (template)

```

template: `

<h2>HEROES</h2>
<ul class="items">
  <li *ngFor="let hero of heroes"
    [class.selected]="isSelected(hero)"
    (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
`
```

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

我们的“英雄”这个特性模块就要完成了，但这个特性还没有平滑的专场效果。我们知道 Angular 支持 [动画](#)，想要得到这项优点，就要为 [英雄详情](#) 组件添加一些动画。

首先，我们从导入动画函数开始，它们用于构建动画触发器，以控制状态和管理状态之间的转场。我们使用这些函数来把转场效果添加到路由组件中，这样当应用视图的多个状态之间发生转移时，就会触发动画。我们还要导入 `HostBinding` 装饰器来绑定到路由组件。

app/heroes/hero-detail.component.ts (animation imports)

```

import { Component, OnInit, HostBinding,
  trigger, transition, animate,
  style, state } from '@angular/core';
```

接下来，我们将对名叫 `@routeAnimation` 的路由动画使用 **宿主绑定（HostBinding）**。选择绑定名时没什么特别的要求，但是由于我们是在控制路由的动画，所以把它叫做 `routeAnimation`。该绑定值被设置为 `true`，因为我们只关心 `:enter` 和 `:leave` 状态，这些动画状态代表 **进场和离开**。

我们还将为样式添加一些显示和位置绑定。

app/heroes/hero-detail.component.ts (route animation binding)

```
export class HeroDetailComponent implements OnInit {
  @HostBinding('@routeAnimation') get routeAnimation() {
    return true;
  }

  @HostBinding('style.display') get display() {
    return 'block';
  }

  @HostBinding('style.position') get position() {
    return 'absolute';
  }

  hero: Hero;

  constructor(
    private route: ActivatedRoute,
    private router: Router,
    private service: HeroService) {}

  ngOnInit() {
    this.route.params.forEach((params: Params) => {
      let id = +params['id']; // (+) converts string 'id' to a
      // number
      this.service.getHero(id).then(hero => this.hero = hero);
    });
  }

  gotoHeroes() {
    let heroId = this.hero ? this.hero.id : null;
    // Pass along the hero id if available
    // so that the HeroList component can select that hero.
    this.router.navigate(['/heroes', { id: heroId, foo: 'foo' }]);
  }
}
```

```
    }  
}
```

现在，我们可以构建动画触发器了，我们称之为 **routeAnimation** 来匹配我们以前定义的绑定。我们的使用 **通配符状态**，它们匹配我们这个路由组件的任意动画状态，后面是两个 **转场动画**。一个转场动画（`:enter`）在组件进入应用视图时触发，另一个（`:leave`）在离开时触发。

如果需要，我们还可以为其它路由组件添加不同的转场动画。在这个里程碑中我们只为 `HeroDetailComponent` 添加动画。

在整个应用程序中，我们并不想在每个组件中都使用路由动画。我们认为基于 **路由路径** 进行路由动画会更好一些，本章将来的更新中会涉及到这个主题。

我们的路由动画看起来像这样：

app/heroes/hero-detail.component.ts (route animation)

```
@Component({  
  template: `  
    <h2>HEROES</h2>  
    <div *ngIf="hero">  
      <h3>"{{hero.name}}"</h3>  
      <div>  
        <label>Id: </label>{{hero.id}}</div>  
      <div>  
        <label>Name: </label>  
        <input [(ngModel)]="hero.name" placeholder="name"/>  
      </div>  
      <p>  
        <button (click)="gotoHeroes()">Back</button>  
      </p>  
    </div>  
  `,  
  animations: [
```

```

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%)'
    }))
  ])
])
])

export class HeroDetailComponent implements OnInit {

```

简单的说，当进入路由时，`HerodetailComponent` 组件会从左侧弹入，离开时会向下方滑出。这里我们还可以添加更复杂的动画，不过我们现在就先不这么做了。

把 hero 模块导入到 AppModule 中

英雄特性已经完工，但是应用还不知道我们的英雄模块。我们得把它导入我们在 `app.module.ts` 中导入的 `AppModule` 中。

像这样修改 `app.module.ts`：

app/app.module.ts (heroes module import)

```

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';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HeroesModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    CrisisListComponent
  ],
  bootstrap: [ AppComponent ]
})
```

我们导入了 `HeroesModule`，并把它加入了 `AppModule` 的 `imports` 中。

我们从 `AppModule` 的 `declarations` 中移除了 `HeroListComponent`，因为它现在改有 `HeroesModule` 提供了。这一步很重要，因为一个组件只能有一个所有者。现在这种情况下，`Heroes` 模块应该是 `Heroes` 组件的所有者，并让它可用于 `AppModule` 中。

由特性模块提供的路由将会被路由器和它们导入的模块提供的路由组合在一起。这让我们可以继续定义特性路由，而不用修改主路由配置。

这种调整的结果是：`app.module.ts` 不再具有任何有关英雄特性的特殊知识，关于它的组件或路由的细节。这个“英雄特性区”可以演化出更多的组件和不同的路由。这是为每个特性区创建一个独立模块带来的核心优势。

由于 `Heroes` 路由被定义在了我们的特性模块中，我们也可以从 `app-routing.module.ts` 中移除当初的 `heroes` 路由了。

app/app-routing.module.ts (v2)

```
import { NgModule }      from '@angular/core';
import { RouterModule }  from '@angular/router';

import { CrisisListComponent } from './crisis-list.component';

@NgModule({
  imports: [
    RouterModule.forRoot([
      { path: 'crisis-center', component: CrisisListComponent },
    ])
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

“英雄”应用总结

我们已经抵达了路由教程的第二个里程碑。

我们学到了如何：

- 把应用组织成一些 **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。
- 在 `AppModule` 中导入特征区的 `NgModule`。
- 把动画应用到路由组件上

做完这些修改之后，目录结构看起来就像这样：

```
router-sample
└─ 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
        └─ node_modules ...
        └─ index.html
        └─ package.json
        └─ styles.css
        └─ tsconfig.json
```

英雄应用的源码

这里是当前版本的范例程序相关文件。

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
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

此刻，**危机中心** 还只是一个假视图，该让它有用点了！

新的**危机中心** 从**英雄** 模块的一个拷贝开始。我们创建新的 `app/crisis-center` 目录，把**英雄**区的文件拷贝过去，并把所有的“hero”修改“crisis”。

`Crisis` 有一个 `id` 和一个 `name`，就像 `Hero` 一样。新的 `CrisisListComponent` 显示危机列表。如果用户选择了一个危机，该应用就会导航到 `CrisisDetailComponent`，用于显示和编辑危机的名字。

真棒！另一个特性模块诞生了！

除非我们能学到点新东西，否则这种练习就没啥亮点。不过，我们已经有了一些新主意和新技巧：

- 我们希望把这些路由地址组织成一棵子路由树，它应该能反映本特性区中组件树的结构。
- 该应用应该默认导航到 **危机中心** 路由。
- 当还有未处理的更改时，路由器应该阻止从这个详情视图导航出去。
- 用户应该能取消不想要的修改。
- 如果用户尚未登录，路由器就应该阻止它访问某些特性。

我们要在**危机中心** 中处理好所有这些问题。那就先从对**子路由** 的介绍开始吧。

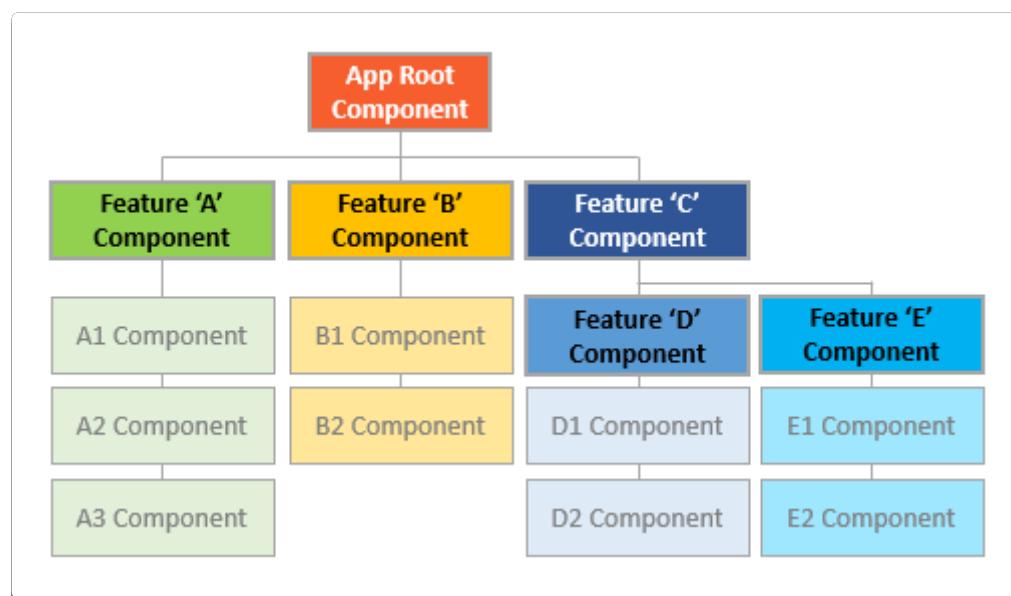
我们将把 **英雄** 特性区保留在不够完美的状态，以便与 **危机中心** 进行对比，我们相信后者是一个更好的设计。

带子路由的“危机中心”

我们将按照下列模式组织 **危机中心** 的目录结构。

- 每个特性区位于它自己的定义了模块的目录中
- 每个特性区有它自己的根组件
- 每个特性区的根组件有它自己的 `<router-outlet>` 及其子路由
- 特性区的路由很少交叉（如果还有的话）

如果我们有更多特性区，它们的组件树看起来是这样的：



子路由组件

往 `crisis-center` 目录下添加下列 `crisis-center.component.ts` 文件：

```
app/crisis-center/crisis-center.component.ts (minus imports)
```

```
@Component({
  template: `
    <h2>CRISIS CENTER</h2>
    <router-outlet></router-outlet>
  `
})
export class CrisisCenterComponent { }
```

CrisisCenterComponent 和壳组件 AppComponent 很像。

- 它是 **危机中心** 特性区的根组件，同样， AppComponent 是整个应用的根组件。
- 它是危机管理区的壳组件，同样， AppComponent 也是用来管理高层工作流的壳组件。
- 它太简单了，甚至比 AppComponent 的模板还要简单。它没有内容、没有链接，只有一个用来存放 **危机中心** 子视图的 <router-outlet> 指令。

但与 AppComponent (以及大多数其它组件) 不同的是，它 **缺少一个选择器 (selector)**。它不需要。我们不会把该组件嵌入到父模板中，只会通过路由器从外部 **导航** 到它。

我们可以为它指定一个选择器。这么做没有什么坏处。这里的观点是，我们 **不需要** 选择器，因为我们只能 * **导航** 到它。

子路由配置

CrisisCenterComponent 是一个像 AppComponent 一样的 **路由组件**。它有自己的 RouterOutlet 和自己的子路由。

把下面的 crisis-center-home.component.ts 文件添加到 crisis-center 目录中。

app/crisis-center/crisis-center-home.component.ts (minus imports)

```
@Component({
  template: `
    <p>Welcome to the Crisis Center</p>
  `
})
export class CrisisCenter HomeComponent { }
```

像 `heroes-routing.module.ts` 文件一样，我们也创建一个 `crisis-center-routing.module.ts`。但这次，我们要把 **子路由** 定义在父路由 `crisis-center` 中。

app/crisis-center/crisis-center-routing.module.ts (Routes)

```
@NgModule({
  imports: [
    RouterModule.forChild([
      {
        path: 'crisis-center',
        component: CrisisCenterComponent,
        children: [
          {
            path: '',
            component: CrisisListComponent,
            children: [
              {
                path: ':id',
                component: CrisisDetailComponent
              },
              {
                path: '',
                component: CrisisCenter HomeComponent
              }
            ]
          }
        ]
      }
    ],
    exports: [
      RouterModule
    ]
})
```

```
})  
export class CrisisCenterRoutingModule { }
```

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：`CrisisCenter HomeComponent` 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**，因此当我们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，每当我们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，我们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**）。

例如，`CrisisCenterComponent` 的路径是 `/crisis-center`。路由器就会把这些子路由的路径中添加上到父路由 `CrisisCenterComponent` 的路径（`/crisis-center`）。

- 要导航到 `CrisisCenter HomeComponent`，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'`)。

本例子中的绝对 URL，包含源站部分，就是：

localhost:3000/crisis-center/2

这里是完整的 `crisis-center.routing.ts` 及其导入语句。

app/crisis-center/crisis-center-routing.module.ts (excerpt)

```
import { NgModule }      from '@angular/core';
import { RouterModule }  from '@angular/router';

import { CrisisCenter HomeComponent } from './crisis-center-
home.component';
import { CrisisListComponent }       from './crisis-list.component';
import { CrisisCenterComponent }    from './crisis-
center.component';
import { CrisisDetailComponent }    from './crisis-
detail.component';

@NgModule({
  imports: [
    RouterModule.forChild([
      {
        path: 'crisis-center',
        component: CrisisCenterComponent,
        children: [
          {
            path: '',
            component: CrisisListComponent,
            children: [
              {
                path: ':id',
                component: CrisisDetailComponent
              },
              {
                path: '',
                component: CrisisCenter HomeComponent
              }
            ]
          }
        ]
      }
    ])
  ]
})
```

```
  ],
  exports: [
    RouterModule
  ]
})
export class CrisisCenterRoutingModule { }
```

把危机中心模块导入 ` AppModule` 的路由中

像 `Heroes` 模块中一样，我们必须把 `危机中心` 模块导入 `AppModule` 中：

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 { 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
  ],
  providers: [
    DialogService
  ],
})
```

```
bootstrap: [ AppComponent ]  
}  
  
export class AppModule {  
}
```

我们还从 `app.routing.ts` 中移除了危机中心的初始路由。我们的路由现在是由 `HeroesModule` 和 `CrisisCenter` 特性模块提供的。我们将保持 `app.routing.ts` 文件中只有通用路由，本章稍后会讲解它。

app/app-routing.module.ts (v3)

```
import { NgModule }      from '@angular/core';  
import { RouterModule } from '@angular/router';  
  
@NgModule({  
  imports: [  
    RouterModule.forRoot([  
  
    ])  
  ],  
  exports: [  
    RouterModule  
  ]  
})  
export class AppRoutingModule {}
```

重定向路由

当应用启动时，浏览器地址栏的初始 URL 是这样的：

```
localhost:3000
```

它无法匹配上我们配置过的任何路由，这意味着在应用启动的时候它将不会显示任何组件。用户必须点击一个导航链接来触发导航并显示点什么。

当用户点击了“Crisis Center”链接或者在地址栏粘贴 `localhost:3000/crisis-center/` 时，我们更希望该应用能直接显示危机列表。这就是默认路由。

首选的解决方案是添加一个 `redirect` 路由，它会把初始的相对 URL（`"/"`）悄悄翻译成默认路径（`/crisis-center`）。

```
{  
  path: '',  
  redirectTo: '/crisis-center',  
  pathMatch: 'full'  
},
```

“重定向（`redirect`）路由”需要一个 `pathMatch` 属性来告诉路由器如何把 URL 和路由中的路径进行匹配。本应用中，路由器应该只有在 **完整的 URL 匹配** `"` 时才选择指向 `CrisisListComponent` 的路由，因此，我们把 `pathMatch` 的值设置为 `'full'`。

从原理上说，`pathMatch = 'full'` 导致路由器尝试用 URL 中 **剩下的**、未匹配过的片段去匹配 `"`。在这个例子中，重定向发生在路由配置树的顶级，所以 **剩下的 URL 和完整的 URL 是完全一样的**。

`pathMatch` 的另一个可能的值是 `'prefix'`，这会告诉路由器去匹配 **剩下的 URL 是否以这个“重定向路由”的前缀路径开头**。

显然，在这里我们并不想这样。如果 `pathMatch` 的值是 `'prefix'`，每个 URL 都会匹配 `"`。因为这个“重定向路由”会首先匹配上并把我们引向 `CrisisListComponent`，所以我们就永远无法导航到 `/crisis-center/1` 了。

只有当完整的（剩余的）URL 是 `"` **时，我们才应该重定向到** `CrisisListComponent`。

要学习更多，请参见 Victor Savkin 的博客中 [关于重定向的帖子](#)。

我们将在未来的更新中深入讨论重定向问题。

修改过的路由定义看起来是这样的：

```
app/crisis-center/crisis-center-routing.module.ts (routes v2)

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

import { CrisisCenter HomeComponent } from './crisis-center-
home.component';
import { CrisisListComponent }      from './crisis-list.component';
import { CrisisCenterComponent }   from './crisis-
center.component';
import { CrisisDetailComponent }  from './crisis-
detail.component';

@NgModule({
  imports: [
    RouterModule.forChild([
      {
        path: '',
        redirectTo: '/crisis-center',
        pathMatch: 'full'
      },
      {
        path: 'crisis-center',
        component: CrisisCenterComponent,
        children: [
          {
            path: '',
            component: CrisisListComponent,
            children: [
              {
                path: ':id',
                component: CrisisDetailComponent,
              },
              {
                path: '',
                component: CrisisCenter HomeComponent
              }
            ]
          }
        ]
      }
    ])
  ]
})
```

```
        }
    ])
],
exports: [
    RouterModule
]
})
export class CrisisCenterRoutingModule { }
```

相对导航

当构建 **危机中心** 特性时，我们将使用以 **斜线** 开头的 **绝对路径** 来导航到 **危机详情** 路由。这次导航会从路由配置的顶部开始查找路由，以匹配路径。

我们可以在 **危机中心** 特性区继续使用绝对路径进行导航，但这会让我们的链接过于死板。如果更改了父路由路径 `/crisis-center`，我们就不得不到处更改链接参数数组。

通过让路由器使用 **相对** 导航的方式，我们可以让链接更富有弹性。

- 不再需要到路由的完整路径。
- 当修改父路由路径时在我们的特性区中导航时不需要做任何改动。
- 链接参数数组** 只包含到当前 URL 的相对导航信息。

链接参数数组 通过目录式语法支持相对导航。

`./` 或“无前导斜线”时表示相当于当前级别。

`../` 表示在路由路径中往上走一级。

相对导航的语法可以和 **路径** 组合在一起，如果我们要从一个路由路径导航到一个兄弟路由路径，可以使用 `../path` 来简便的导航到上一级然后再进入兄弟路由路径。

要使用 `Router` 进行相对导航，可以使用 `ActivatedRoute` 来告诉路由器我们正在 `RouterState` 中的什么地方，`RouterState` 是激活路由组成的树。要做到这一点，我们可以为 `router.navigate` 方法中 **链接参数数组** 后的对象型参数指定 `relativeTo` 属性。只要把这个 `relativeTo` 属性设置为我们的 `ActivatedRoute`，路由器就会把我们的导航信息和当前 URL 合并在一起。

当使用路由器的 `navigateByUrl` 方法时，导航 **总是** 绝对的。

用相对方式导航到危机详情

我们来把 **危机列表** 中的 `onSelect` 方法改成相对导航，以便我们不必从路由配置的顶部开始。我们已经把相对导航所需的 `ActivatedRoute` 注入到了构造函数中。

app/crisis-center/crisis-list.component.ts (constructor)

```
constructor(
  private service: CrisisService,
  private route: ActivatedRoute,
  private router: Router) {}
```

当我们访问 **危机中心** 时，当前路径是 `/crisis-center`，所以我们只要把 **危机** 的 `id` 添加到现有路径中就可以了。当路由器导航时，它使用当前路径 `/crisis-center` 并追加上此 `id`。如果 `id` 为 `1`，结果路径就是 `/crisis-center/1`。

app/crisis-center/crisis-list.component.ts (relative navigation)

```
onSelect(crisis: Crisis) {
  this.selectedId = crisis.id;

  // Navigate with relative link
  this.router.navigate([crisis.id], { relativeTo: this.route });
}
```

我们还要修改 **危机详情** 组件以便导航回 **危机中心** 列表。我们得回到路径的上一级，所以我们使用 `../` 语法。如果当前 `id` 是 `1`，那么结果路径就会从 `/crisis-center/1` 变成 `/crisis-center`。

app/crisis-center/crisis-detail.component.ts (relative navigation)

```
gotoCrises() {
  let crisisId = this.crisis ? this.crisis.id : null;
  // Pass along the crisis id if available
  // so that the CrisisListComponent can select that crisis.
  // Add a totally useless `foo` parameter for kicks.
  // Relative navigation back to the crises
  this.router.navigate(['..'], { id: crisisId, foo: 'foo' }), {
    relativeTo: this.route
  }
}
```

如果我们正在使用 `RouterLink` 进行导航，而不是 `Router` 服务，仍然可以使用 **相同的** 链接参数数组，不过我们不用提供带 `relativeTo` 属性的对象。在 `RouterLink` 指令中 `ActivatedRoute` 是默认的。

app/crisis-center/crisis-list.component.ts (relative routerLink)

```
<a [routerLink]=[crisis.id]>{{ crisis.name }}</a>
```

路由守卫

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，我们可能得先获取某些数据。
- 在离开组件前，我们可能要先保存修改。
- 我们可能要询问用户：你是否要放弃本次更改，而不用保存它们？

我们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`。

路由器支持多种守卫：

1. 用 `CanActivate` 来处理导航 **到** 某路由的情况。
2. 用 `CanActivateChild` 处理导航 **到** 子路由的情况。
3. 用 `CanDeactivate` 来处理从当前路由 **离开** 的情况。
4. 用 `Resolve` 在路由激活 **之前** 获取路由数据。
5. 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，我们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` 守护条件。然后它会按照从上到下的顺序检查 `CanActivate` 守卫。如果 **任何** 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。

我们来看一些例子。

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。我们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

我们准备扩展“危机中心”，添加一些新的 **管理类** 特性。这些特性还没有定义过，所以我们先只添加一个名叫 `AdminModule` 的占位模块。我们会遵循与创建 `admin` 目录中的特性模块文件、路由文件和支持组件时相同的约定。

管理特性区的文件是这样的：

```
app/admin
  admin-dashboard.component.ts
  admin.component.ts
  admin.module.ts
  admin-routing.module.ts
  manage-crises.component.ts
  manage-heroes.component.ts
```

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。

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   template:
5.     <p>Dashboard</p>
6.
7. })
8. export class AdminDashboardComponent { }
```

由于 `AdminModule` 中管理仪表盘的 `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。但我们只有在访问 `Dashboard` 路由时才希望该链接被激活。所以我们往 `Dashboard` 这个 routerLink 上添加了另一个绑定 `[routerLinkActiveOptions]="{ exact: true }"`，这样就只有当我们导航到 `/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。

我们的初始管理路由配置如下：

app/admin/admin-routing.module.ts (admin routing)

```
@NgModule({
  imports: [
    RouterModule.forChild([
      {
        path: 'admin',
        component: AdminComponent,
        children: [
          {
            path: '',
            children: [
              { path: 'crises', component: ManageCrisesComponent },
              { path: 'heroes', component: ManageHeroesComponent },
              { path: '', component: AdminDashboardComponent }
            ]
          }
        ]
      }
    ])
  ]
})
```

```

        }
    ])
],
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，我们有一个带 `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。

虽然我们希望对 `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。这同时也允许我们 **守卫子路由**。

接下来，我们把 `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。

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 { 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  
],  
providers: [  
  DialogService  
],  
bootstrap: [ AppComponent ]  
}  
  
export class AppModule {  
}
```

然后我们往壳组件 `AppComponent` 中添加一个链接，让用户能点击它，以访问该特性。

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>  
  </nav>  
  <router-outlet></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。

我们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。

我们换种方式：写一个 `CanActivate` 守卫，当匿名用户尝试访问管理组件时，把它 / 她重定向到登录页。

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），所以我们在应用的根目录下创建一个 `auth-guard.ts` 文件。

此刻，我们的兴趣在于看看守卫是如何工作的，所以我们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：

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` 属性来引用 `AuthGuard`：

app/admin/admin-routing.module.ts (guarded admin route)

```
import { AuthGuard }           from '../auth-guard.service';

@NgModule({
  imports: [
    RouterModule.forChild([
      {
        path: 'admin',
        component: AdminComponent,
        canActivate: [AuthGuard],
        children: [
          {
            path: '',
            component: AdminDashboardComponent
          }
        ]
      }
    ])
  ],
  providers: [
    AuthGuard
  ]
})
```

```

        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ],
      }
    ]
  }
],
exports: [
  RouterModule
]
)
export class AdminRoutingModule {}

```

我们的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

我们先让 `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：

app/auth.service.ts (excerpt)

```

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

import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/delay';

@Injectable()
export class AuthService {
  isLoggedIn: boolean = false;

  // store the URL so we can redirect after logging in
  redirectUrl: string;

```

```
login(): Observable<boolean> {
    return observable.of(true).delay(1000).do(val => this.isLoggedIn
= true);
}

logout(): void {
    this.isLoggedIn = false;
}
}
```

虽然它不会真的进行登录，但足够让我们进行这个讨论了。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。`redirectTo` 属性将会保存在 URL 中，以便认证完之后导航到它。

我们这就修改 `AuthGuard` 来调用它。

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);
  }
}

private checkLogin(url: string): boolean {
  if (!this.authService.isLoggedIn) {
    this.router.navigate(['/login']);
    return false;
  }

  return true;
}
```

```

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，我们把 `AuthService` 和 `Router` 服务 **注入到** 构造函数中。我们还没有提供 `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。它们要通过我们的守卫进行检查。

如果用户还没有登录，我们会用 `RouterStateSnapshot.url` 保存用户来自的 URL 并让路由器导航到登录页（我们尚未创建该页）。这间接导致路由器自动中止了这次导航，我们返回 `false` 并不是必须的，但这样可以更清楚的表达意图。

添加 LOGINCOMPONENT

我们需要一个 `LoginComponent` 来让用户登录进这个应用。在登录之后，我们跳转到前面保存的 URL，如果没有，就跳转到默认 URL。该组件没有什么新内容，我们把它放进路由配置的方式也没什么新意。

我们将在 `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，我们导入 `LoginComponent` 并把它加入根模块的 `declarations` 中。同时在 `AppModule` 中导入并添加 `LoginRoutingModule`。

```

1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';

```

```
4.  
5. import { AppComponent }      from './app.component';  
6. import { AppRoutingModule }   from './app-routing.module';  
7.  
8. import { HeroesModule }     from './heroes/heroes.module';  
9. import { LoginRoutingModule } from './login-routing.module';  
10. import { LoginComponent }    from './login.component';  
11.  
12. import { DialogService }    from './dialog.service';  
13.  
14. @NgModule({  
15.   imports: [  
16.     BrowserModule,  
17.     FormsModule,  
18.     HeroesModule,  
19.     LoginRoutingModule,  
20.     AppRoutingModule  
21.   ],  
22.   declarations: [  
23.     AppComponent,  
24.     LoginComponent  
25.   ],  
26.   providers: [  
27.     DialogService  
28.   ],  
29.   bootstrap: [ AppComponent ]  
30. })  
31. export class AppModule {  
32. }
```

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: 守卫子路由

就像我们可以通过 `CanActivate` 来守卫路由一样，我们也能通过 `CanActivateChild` 守卫来保护子路由。`CanActivateChild` 守卫的工作方式和 `CanActivate` 守卫很相似，不同之处在在于它会在每个子路由被激活 **之前** 运行。我们保护了管理特性模块不受未授权访问，也同样可以在特性模块中保护子路由。

让我们扩展一下 `AuthGuard`，让它能在 `admin` 路由之间导航时提供保护。首先，打开 `auth-guard.service.ts` 并从 `router` 包中导入 `CanActivateChild` 接口。

然后，我们实现 `canActivateChild` 方法，它接收与 `canActivate` 方法相同的参数：

`ActivatedRouteSnapshot` 和 `RouterStateSnapshot`。`canActivateChild` 和其它守卫的行为一样，都返回 `Observable<boolean>` 或 `Promise<boolean>` 以支持异步检查，或返回 `boolean` 来支持同步检查。这里我们直接返回 `boolean`：

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);
  }
}
```

}

我们往“无组件”的管理路由中添加同一个 `AuthGuard` 以同时保护所有子路由，而不是挨个添加它们。

app/admin/admin-routing.module.ts (excerpt)

```
@NgModule({
  imports: [
    RouterModule.forChild([
      {
        path: 'admin',
        component: AdminComponent,
        canActivate: [AuthGuard],
        children: [
          {
            path: '',
            canActivateChild: [AuthGuard],
            children: [
              { path: 'crises', component: ManageCrisesComponent },
              { path: 'heroes', component: ManageHeroesComponent },
              { path: '', component: AdminDashboardComponent }
            ]
          }
        ]
      }
    ])
  ],
  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，我们得先把用户的改动积累起来。 我们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？ 我们不能马上离开，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。

我们应该暂停，并让用户决定该怎么做。 如果用户选择了取消，我们就留下来，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，我们还可以继续推迟导航。 如果我们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），我们就会丢失该错误的上下文环境。

在等待服务器的答复时，我们没法阻塞它——这在浏览器中是不可能的。 我们只能用异步的方式在等待服务器答复之前先停止导航。

我们需要 `CanDeactivate` 守卫。

取消与保存

我们的范例应用不会与服务器通讯。 幸运的是，我们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，我们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。

app/crisis-center/crisis-detail.component.ts (excerpt)

```
export class CrisisDetailComponent implements OnInit {
  @HostBinding('@routeAnimation') get routeAnimation() {
    return true;
  }

  @HostBinding('style.display') get display() {
    return 'block';
  }
}
```

```
}

@HostBinding('style.position') get position() {
  return 'absolute';
}

crisis: Crisis;
editName: string;

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？

我们两个都不采取。 我们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**。

我们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）。

我们创建了一个 `Guard`，它将检查这个组件中 `canDeactivate` 函数的工作现场，在这里，它就是 `CrisisDetailComponent`。我们并不需要知道 `CrisisDetailComponent` 确认退出激活状态的详情。这让我们的守卫可以被复用，这是一次轻而易举的胜利。

app/can-deactivate-guard.service.ts

```

1. import { Injectable }      from '@angular/core';
2. import { CanDeactivate }   from '@angular/router';
3. import { Observable }      from 'rxjs/Observable';
4.
5. export interface CanComponentDeactivate {
6.   canDeactivate: () => Observable<boolean> | Promise<boolean> |
7.   boolean;
8.
9. @Injectable()
10. export class CanDeactivateGuard implements
11.   CanDeactivate<CanComponentDeactivate> {
12.   canDeactivate(component: CanComponentDeactivate) {
13.     return component.canDeactivate ? component.canDeactivate() : true;
14.   }

```

另外，我们也可以为 `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，并且需要使用该组件属性、或者需要路由器确认是否允许从该组件导航出去时，这个守卫就非常有用。

app/can-deactivate-guard.service.ts (component-specific)

```

import { Injectable }          from '@angular/core';
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
  ): Promise<boolean> | boolean {
    // Get the Crisis Center ID
    console.log(route.params['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
    // promise which resolves to true or false when the user decides
    return component.dialogService.confirm('Discard changes?');
  }
}

```

看看 CrisisDetailComponent 组件，我们已经实现了对未保存的更改进行确认的工作流。

app/crisis-center/crisis-detail.component.ts (excerpt)

```

canDeactivate(): Promise<boolean> | boolean {
  // Allow synchronous navigation (`true`) if no crisis or the crisis
  crisis is unchanged
  if (!this.crisis || this.crisis.name === this.editName) {
    return true;
  }
  // otherwise ask the user with the dialog service and return its
  // promise 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）。

我们往 `crisis-center.routing.ts` 的危机详情路由中用 `canDeactivate` 数组添加一个 `Guard`（守卫）。

app/crisis-center/crisis-center-routing.module.ts (can deactivate guard)

```
import { NgModule }      from '@angular/core';
import { RouterModule }  from '@angular/router';

import { CrisisCenter HomeComponent } 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';

@NgModule({
  imports: [
    RouterModule.forChild([
      {
        path: '',
        redirectTo: '/crisis-center',
        pathMatch: 'full'
      },
      {
        path: 'crisis-center',
        component: CrisisCenterComponent,
        children: [
          {
            path: '',
            component: CrisisListComponent,
            children: [
              {
                path: ':id',
                component: CrisisDetailComponent
              }
            ]
          }
        ]
      }
    ])
  ],
  exports: [RouterModule]
})
export class CrisisCenterRoutingModule { }
```

```

        component: CrisisDetailComponent,
        canDeactivate: [CanDeactivateGuard]
    },
    {
        path: '',
        component: CrisisCenter HomeComponent
    }
]
}
]
})
],
exports: [
    RouterModule
]
})
export class CrisisCenterRoutingModule { }

```

我们还要把这个 `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 } from '@angular/router';
3.
4. import { CanDeactivateGuard } from './can-deactivate-guard.service';
5.
6. @NgModule({
7.   imports: [
8.     RouterModule.forRoot([
9.
10.   ])
11. ],
12. exports: [
13.   RouterModule
14. ],
15. providers: [
16.   CanDeactivateGuard
17. ]
18. })

```

```
19.  export class AppRoutingModule {}
```

现在，我们已经给了用户一个能保护未保存更改的安全守卫。

解析：提前获取组件数据

在 `Hero Detail` 和 `Crisis Detail` 中，它们等待路由读取对应的英雄和危机。

这种方式没有问题，但是它们还有进步的空间。如果我们在使用真实 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。

可以预先从服务器读取数据，这样在路由器被激活时，数据已经返回。同时，我们还需要处理数据返回失败和其它出错情况。这样，在 `Crisis Center` 中，对处理导航到一个无返回数据的 `:id` 有帮助。我们可以将用户发回只列出有效危机的 `Crisis List`。我们需要延迟渲染路由组件，来等待所有必要的数据都成功获取或做一些其他操作。

我们需要 `Resolve` 守卫。

导航前预先加载路由信息

我们需要更新 `Crisis Detail` 路由，让它先解析必要的危机，再加载路由。或者当用户导航到一个无效的危机 `:id` 时，将它们导航回危机列表。

和 `CanActivate` 和 `CanDeactivate` 守卫一样，服务可以实现 `Resolve` 守卫接口来同步或异步解析路由数据。`Crisis Detail` 组件使用 `ngOnInit` 来获取 `Crisis` 信息。如果 `Crisis` 找不到，用户会被导航出去。在路由被激活之前就处理这些情况会更加有效。

现在创建一个 `CrisisDetailResolve` 服务，用它来处理 `Crisis` 数据读取和在 `Crisis` 不存在时将用户导航出去。然后可以确保当激活 `CrisisDetailComponent` 时，关联的 `Crisis` 已经为显示准备妥当。

下面在 `Crisis Center` 特征区新建的 `crisis-detail-resolve.service.ts` 文件：

```
app/crisis-center/crisis-detail-resolve.service.ts
```

```

1. import { Injectable }           from '@angular/core';
2. import { Router, Resolve,
3.          ActivatedRouteSnapshot } from '@angular/router';
4.
5. import { Crisis, CrisisService } from './crisis.service';
6.
7. @Injectable()
8. export class CrisisDetailResolve implements Resolve<Crisis> {
9.   constructor(private cs: CrisisService, private router: Router) {}
10.
11. resolve(route: ActivatedRouteSnapshot): Promise<Crisis>|boolean {
12.   let id = +route.params['id'];
13.
14.   return this.cs.getCrisis(id).then(crisis => {
15.     if (crisis) {
16.       return crisis;
17.     } else { // id not found
18.       this.router.navigate(['/crisis-center']);
19.       return false;
20.     }
21.   });
22. }
23. }

```

接下来，将 `CrisisDetailComponent` 中 `ngOnInit` 生命周期钩子里面相关的部分移动到 `CrisisDetailResolve` 守卫里面，然后导入 `Crisis` 模型，`CrisisService` 服务和 `Router`。为了特殊指定什么样的数据需要解析，我们在 `Resolve` 接口的实现上指定了 `Crisis` 类型。这样告诉我们解析的结果将于 `Crisis` 模型对应。然后注入 `CrisisService` 和 `Router`，并实现支持 `Promise`、`Observable` 和异步返回值 `resolve` 方法。

我们使用 `CrisisService.getCrisis` 方法来获取一个 **承诺对象**，用于防止路由在成功获取数据之前被加载。如果没有找到对应 `Crisis`，便将用户导航回 `CrisisList`，取消之前导航到危机详情的路由。

解析守卫现在准备好了，将它导入到 `crisis-center-routing.module.ts` 中，然后在路由配置中设置 `resolve` 对象。

接下来，将 `CrisisDetailResolve` 服务添加到危机中心路由模块的 `providers` 数组中，这样 `Router` 在路由过程中可以使用它。

app/crisis-center/crisis-center-routing.module.ts (resolve)

```
import { CrisisDetailResolve } from './crisis-detail-
resolve.service';

@NgModule({
  imports: [
    RouterModule.forChild([
      {
        path: '',
        redirectTo: '/crisis-center',
        pathMatch: 'full'
      },
      {
        path: 'crisis-center',
        component: CrisisCenterComponent,
        children: [
          {
            path: '',
            component: CrisisListComponent,
            children: [
              {
                path: ':id',
                component: CrisisDetailComponent,
                canDeactivate: [CanDeactivateGuard],
                resolve: {
                  crisis: CrisisDetailResolve
                }
              },
              {
                path: '',
                component: CrisisCenter HomeComponent
              }
            ]
          }
        ]
      }
    ])
  ],
  providers: [
    CrisisDetailResolve
  ]
})
```

```

exports: [
  RouterModule
],
providers: [
  CrisisDetailResolve
]
})
export class CrisisCenterRoutingModule { }

```

因为添加了 `Resolve` 守卫并用它在加载路由之前读取数据，所以在加载 `CrisisDetailComponent` 后，不再需要读取数据。因此，可以更新 `CrisisDetailComponent` 组件，让它使用 `Resolve` 守卫通过 `crisis` 属性提供的 `ActivatedRoute.data`。一旦激活 `CrisisDetailComponent`，我们可以使用解析过的 `Crisis` 信息赋值本地属性 `crisis` 和 `editName`。我们也不再需要通过订阅和反订阅 `ActivatedRoute` 的参数来获取 `Crisis`，因为它已经在路由组件被激活时被同步提供。

app/crisis-center/crisis-detail.component.ts (ngOnInit v2)

```

ngOnInit() {
  this.route.data.forEach((data: { crisis: Crisis }) => {
    this.editName = data.crisis.name;
    this.crisis = data.crisis;
  });
}

```

两个关键点

1. 路由器接口是可选的。我们不必从基类中继承它，只需要实现这个接口方法或不实现。
2. 我们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。我们只要写出这个类，等路由器从那里取出它就可以了。

本里程碑中与 **危机中心** 有关的代码如下：

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

```

2.
3. @Component({
4.   selector: 'my-app',
5.   template: `
6.     <h1 class="title">Angular Router</h1>
7.     <nav>
8.       <a routerLink="/crisis-center" routerLinkActive="active">Crisis
9.         Center</a>
10.        <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
11.        <a routerLink="/admin" routerLinkActive="active">Admin</a>
12.        <a routerLink="/login" routerLinkActive="active">Login</a>
13.      </nav>
14.      <router-outlet></router-outlet>
15.    `})
16. export class AppComponent {
17. }

```

```

1. import { Injectable }           from '@angular/core';
2. import {
3.   CanActivate, Router,
4.   ActivatedRouteSnapshot,
5.   RouterStateSnapshot,
6.   CanActivateChild
7. }                               from '@angular/router';
8. import { AuthService }         from './auth.service';
9.
10. @Injectable()
11. export class AuthGuard implements CanActivate, CanActivateChild {
12.   constructor(private authService: AuthService, private router: Router) {}
13.
14.   canActivate(route: ActivatedRouteSnapshot, state:
15.     RouterStateSnapshot): boolean {
16.     let url: string = state.url;
17.
18.     return this.checkLogin(url);
19.   }

```

```

20.      canActivateChild(route: ActivatedRouteSnapshot, state:
21.        RouterStateSnapshot): boolean {
22.          return this.canActivate(route, state);
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，我们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？要达到这个目的，该“查询参数”大显身手了。

[片段](#) 可以引用页面中带有特定 `id` 属性的元素。

接下来，我们将更新 `AuthGuard` 来提供 `session_id` 查询参数，在导航到其它路由后，它还会存在。

我们还将随意提供一个锚点片段，它用来跳转到页面中指定的位置。

我们还将为 `router.navigate` 方法传入一个 `NavigationExtras` 对象，用来导航到 `/login` 路由。

app/auth-guard.service.ts (v3)

```

import { Injectable }           from '@angular/core';
import {
  CanActivate, Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,

```

```
CanActivatechild,
NavigationExtras
}
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;
  }
}
```

还可以在导航之间 **保留** 查询参数和片段，而无需再次在导航中提供。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。

app/login.component.ts (preserve)

```
// Set our navigation extras object
// that passes on our global query params and fragment
let navigationExtras: NavigationExtras = {
  preserveQueryParams: true,
  preserveFragment: true
};

// Redirect the user
this.router.navigate([redirect], navigationExtras);
```

由于要在登录后导航到 **危机管理** 特征区的路由，所以我们还得更新它，来处理这些全局查询参数和片段。

app/admin/admin-dashboard.component.ts (v2)

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';

@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
    .queryParams
    .map(params => params['session_id'] || 'None');

  // Capture the fragment if available
  this.token = this.route
    .fragment
    .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。在更新过的 **英雄管理** 组件中，我们将直接把 `Observable` 传给模板，借助 `AsyncPipe` 在组件被销毁时自动 **取消** 对 `Observable` 的订阅。

当在 plunker 中运行时，可以点击右上角的蓝色 'X' 按钮来弹出预览窗口。



按照下列步骤试验下：点击 **Crisis Admin** 按钮，它会带着我们提供的“**查询参数**”和“**片段**”跳转到登录页。点击登录按钮，我们就会被带到 **Crisis Admin** 页，仍然带着上一步提供的“**查询参数**”和“**片段**”。我们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 5：异步路由

完成上面的里程碑后，我们的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，我们将达到一个顶点，应用 将会需要过多的时间来加载。长远来看，这不是一个办法。

如何才能解决这个问题呢？我们引进了异步路由到应用程序中，并获得 懒惰 加载特征区域的能力。这样给我们带来了下列好处：

- 可以继续构建特征区，但不再增加初始包大小。
- 只有在用户请求时才加载特征区。
- 为那些只访问应用程序某些区域的用户加快加载速度。

我们接下来在当前的项目中添加这些特征。现在已经有一系列模块将应用组织为四大块：`AppModule`，`HeroesModule`，`AdminModule` 和 `CrisisCenterModule`。

`AdminModule` 在我们的应用中只被小部分用户访问，所以我们利用异步路由来实现只有在请求时才加载 `Admin` 特性区域。

惰性加载路由配置

首先，把 `admin` 路由添加到 `app-routing.module.ts` 文件中。我们想要异步加载 `Admin` 模块，就得在路由配置中使用 `loadChildren` 属性，而以前我们已经用 `children` 属性包含了这些子路由。

接下来，在 `admin.routing.ts` 中，把 `admin` 的 **path** 更改为空路径。路由器支持 空路径 路由，它可以在不必把别的路径添加到 URL 中的情况下，将多个路由组合到一起。用户还是可以访问 `/crisis-center`，`CrisisCenterComponent` 组件还是包含了子级路由的 **路由组件**。

```
1.  @NgModule({
2.    imports: [
3.      RouterModule.forRoot([
4.        {
```

```

4.      {
5.        path: 'admin',
6.        loadChildren: 'app/admin/admin.module#AdminModule',
7.      }
8.    ],
9.  ],
10. exports: [
11.   RouterModule
12. ],
13. providers: [
14.   CanDeactivateGuard
15. ]
16. })
17. export class AppRoutingModule {}

```

路由器用 `loadChildren` 属性来映射我们希望惰性加载的捆文件，这里是 `AdminModule`。

仔细看 `loadChildren` 字符串，就会发现它直接映射到了我们以前在管理特性区构建的 `admin.module.ts` 文件。在文件路径后面，我们使用 `#` 来标记出文件路径的末尾，并告诉路由器 `AdminModule` 的名字。打开 `admin.module.ts` 文件，我们就会看到它正是我们所导出的模块类的名字。

app/admin/admin.module.ts (export)

```
export class AdminModule {}
```

路由器用 `loadChildren` 属性来映射我们希望惰性加载的捆文件，这里是 `AdminModule`。路由器将接收我们的 `loadChildren` 字符串，并把它动态加载进 `AdminModule`，它的路由被 **动态** 合并到我们的配置中，然后加载所请求的路由。但只有在首次加载该路由时才会这样做，后续的请求都会立即完成。

Angular 提供一个内置模块加载器，支持 `SystemJS` 来异步加载模块。如果我们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。

我们构建了特性区，更新了路由配置来实现惰性加载，现在该做最后一步：将 AdminModule 分离到一个彻底独立的模块。因为现在按需加载 AdminModule，所以在 app.module.ts 中，从 imports 数组中删除它。

app/app.module.ts (async admin module)

```
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 { CrisisCenterModule } from './crisis-center/crisis-
center.module';
import { LoginRoutingModule } from './login-routing.module';
import { LoginComponent }     from './login.component';

import { DialogService }      from './dialog.service';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HeroesModule,
    CrisisCenterModule,
    LoginRoutingModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    LoginComponent
  ],
  providers: [
    DialogService
  ],
  bootstrap: [ AppComponent ]
})
```

```
export class AppModule {  
}
```

CanLoad 守卫：保护特性模块的加载

我们已经使用 `CanActivate` 保护 `AdminModule` 了，它会阻止对管理特性区的匿名访问。我们在请求时可以异步加载管理类路由，检查用户的访问权，如果用户未登录，则跳转到登陆页面。但更理想的是，我们只在用户已经登录的情况下加载 `AdminModule`，并且直到加载完才放行到它的路由。

`CanLoad` 守卫适用于这个场景。

我们可以用 `CanLoad` 守卫来保证只在用户已经登录并尝试访问管理特性区时才加载一次 `AdminModule`。我们这就升级 `AuthGuard` 来支持 `CanLoad` 守卫。我们先导入 `CanLoad` 接口，它被调用时守卫会提供一个 `Route` 参数，其中包含所请求的路径。

我们还要把此接口加入到服务中，并实现它。由于我们的 `AuthGuard` 已经能检查用户的登录状态了，所以把 `canLoad` 方法的权限检查工作直接转给它。`canLoad` 方法中的 `Route` 参数提供了一个路径，它来自我们的路由配置。

app/auth-guard.service.ts (can load guard)

```
import { Injectable }           from '@angular/core';  
import {  
  CanActivate, Router,  
  ActivatedRouteSnapshot,  
  RouterStateSnapshot,  
  CanActivateChild,  
  NavigationExtras,  
  CanLoad, Route  
}                           from '@angular/router';  
import { AuthService }         from './auth.service';  
  
@Injectable()  
export class AuthGuard implements CanActivate, CanActivateChild,  
CanLoad {  
  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);  
}  
  
canLoad(route: Route): boolean {  
  let url = `/${route.path}`;  
  
  return this.checkLogin(url);  
}  
  
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;  
}  
}
```

接下来，我们就把 `AuthGuard` 导入到 `app.routing.ts` 中，并把 `AuthGuard` 添加到 `admin` 路由的 `canLoad` 数组中。现在 `admin` 特性区就只有当获得访问授权时才会被加载了。

app/app-routing.module.ts (can load guard)

```
import { AuthGuard } from './auth-guard.service';
@NgModule({
  imports: [
    RouterModule.forRoot([
      {
        path: 'admin',
        loadChildren: 'app/admin/admin.module#AdminModule',
        canLoad: [AuthGuard]
      }
    ]),
    exports: [
      RouterModule
    ],
    providers: [
      CanDeactivateGuard
    ]
  )
})
export class AppRoutingModule {}
```

预加载：在后台加载特征区域

我们已经学会了如何按需加载模块，我们还可以 **预先** 加载特征区域。 **路由器** 支持在导航到特征区域 URL 之前，异步 **预加载** 它们。 预加载允许我们在快速加载初始路由的同时，在后台加载其他特征模块。当导航到这些区域时，它们已经被加载了，就像它们被包含在初始包中一样。

每次导航 **成功** 发生时，**路由器** 将查看惰性加载的特征区域的配置，并根据提供的策略作出反应。

路由器 默认支持两种预加载策略：

- 完全不预加载，这是默认值。惰性加载特征区域仍然按需加载。

- 预加载所有惰性加载的特征区域。

路由器还支持 **自定义预加载策略**，用来精细控制预加载。

我们将更新 **CrisisCenterModule**，让它默认惰性加载并使用 **PreloadAllModules** 策略来尽快加载 **所有** 惰性加载模块。

PreloadAllModules 策略不会加载被 **CanLoad** 守卫保护的特征区域，这是因为 Angular 是这样设计的。 **CanLoad** 守卫阻挡加载特征模块资源，直到授权为止。如果你希望预加载一个模块并保护未授权访问，使用 **CanActivate** 守卫。

使用与加载异步 **AdminModule** 一样流程，我们将更新路由配置，来惰性加载 **CrisisCenterModule**。在 **crisis-center-routing.module.ts** 中，将 **crisis-center** 的路径修改为 **空路径**。

接下来，将 **redirect** 和 **crisis-center** 路由移动到 **AppRoutingModule** 路由，并使用 **loadChildren** 字符串来加载 **CrisisCenterModule**。**redirect** 也被修改为初始加载 **/heroes** 路由。

然后将 **CrisisCenterModule** 移到 **AppModule** 的 **imports** 中。

下面是打开预加载之前的模块修改版：

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { AppComponent }   from './app.component';
6. import { AppRoutingModule } from './app-routing.module';
7.
8. import { HeroesModule }  from './heroes/heroes.module';
9. import { LoginRoutingModule } from './login-routing.module';
10. import { LoginComponent } from './login.component';
11.
```

```

12. import { DialogService }      from './dialog.service';
13.
14. @NgModule({
15.   imports: [
16.     BrowserModule,
17.     FormsModule,
18.     HeroesModule,
19.     LoginRoutingModule,
20.     AppRoutingModule
21.   ],
22.   declarations: [
23.     AppComponent,
24.     LoginComponent
25.   ],
26.   providers: [
27.     DialogService
28.   ],
29.   bootstrap: [ AppComponent ]
30. })
31. export class AppModule {
32. }

```

`RouterModule.forRoot` 方法的第二个参数接受一个附加配置选项对象。我们从路由器包导入 `PreloadAllModules` 令牌，并将这个配置选项的 `preloadingStrategy` 属性设置为 `PreloadAllModules` 令牌。这样，内置的 路由器 立刻预加载 所有 使用 `loadChildren` 的 未受保护 的特征区域。

app/app-routing.module.ts (preload all)

```

import { NgModule }      from '@angular/core';
import {
  RouterModule,
  PreloadAllModules
} from '@angular/router';

import { CanDeactivateGuard } from './can-deactivate-guard.service';
import { AuthGuard }          from './auth-guard.service';

@NgModule({
  imports: [

```

```
RouterModule.forRoot([
  {
    path: 'admin',
    loadChildren: 'app/admin/admin.module#AdminModule',
    canLoad: [AuthGuard]
  },
  {
    path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  {
    path: 'crisis-center',
    loadChildren: 'app/crisis-center/crisis-
center.module#CrisisCenterModule'
  },
],
{ preloadingStrategy: PreloadAllModules }
)
],
exports: [
  RouterModule
],
providers: [
  CanDeactivateGuard
]
})
export class AppRoutingModule {}
```

现在，访问 `http://localhost:3000` 时，`/heroes` 路由将在前台加载，同时，**CrisisCenterModule** 和其他异步特征模块将在后台被 **主动** 加载，等待我们导航到它们。

自定义预加载策略

在一些情况下，预加载所有模块很合适。但是在另外一些用例中，我们需要选择主动加载哪些模块。在移动设备上或者网速很低的情况下加载应用显得尤其明显。根据用户指标或者逐步收集的数据，我们可能只想预加载某些特征模块。路由器通过 **自定义** 预加载策略为我们提供了更多控制。

使用将 **PreloadAllModules** 模块策略提供给 **RouterModule.forRoot** 配置对象一样方式，我们可以定义自己的策略。

因为想利用这点，我们将添加自定义策略，**只** 预加载我们选择的模块。为了启用预加载，我们使用 **Route Data**，正如我们学过的，它是储存的路由数据和 **解析数据** 对象。

我们将自定义 **preload** 布尔值添加到 **crisis-center** 路由数据，自定义策略将使用它。然后在自定义策略服务中将 **route.path** 添加到 **preloadedModules** 数组。我们还将在控制台中为预加载模块输出一条消息。

app/app-routing.module.ts (route data preload)

```
{
  path: 'crisis-center',
  loadChildren: 'app/crisis-center/crisis-
center.module#CrisisCenterModule',
  data: {
    preload: true
  }
}
```

为了创建自定义策略，我们将需要实现抽象类 **PreloadingStrategy** 和 **preload** 方法。在异步加载特征模块和决定是否预加载它们时，路由器调用 **preload** 方法。**preload** 方法有两个参数，第一个参数 **Route** 提供路由配置，第二个参数是预加载特征模块的函数。

我们将命名自己的策略为 **PreloadSelectedModules**，因为我们**只** 想加载符合特定条件的模块。自定义策略在 **Route Data** 中查询 **preload** 布尔值，如果它为 **true**，就调用内置 **Router** 提供的 **load** 函数预主动加载这些特征模块。

app/selective-preload-strategy.ts (preload selected modules)

```
import 'rxjs/add/observable/of';
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class PreloadSelectedModules implements PreloadingStrategy {
```

```

preloadedModules: string[] = [];

preload(route: Route, load: Function): Observable<any> {
  if (route.data && route.data['preload']) {
    // add the route path to our preloaded module array
    this.preloadedModules.push(route.path);

    // log the route path to the console
    console.log('Preloaded: ' + route.path);

    return load();
  } else {
    return observable.of(null);
  }
}
}

```

要使用我们的自定义预加载策略，将它导入到 `app-routing.module.ts` 并替换 `PreloadAllModules` 策略。我们还将 `PreloadSelectedModules` 策略添加到 `AppRoutingModule` 的 `providers` 数组中。这样，路由器的预加载器可以注入我们的自定义策略。

要确认 `CrisisCenterModule` 是否被预加载，我们将在 `Admin` 管理控制台显示 `preloadedModules`。我们已经知道如何使用 `ngFor` 循环，所以在这里跳过了一些细节。因为 `PreloadSelectedModules` 只是一个服务，我们可以将其注入到 `AdminDashboardComponent` 并连接到列表中：

app/admin/admin-dashboard.component.ts (preloaded modules)

```

import { Component, OnInit }      from '@angular/core';
import { ActivatedRoute }        from '@angular/router';
import { Observable }           from 'rxjs/Observable';
import { PreloadSelectedModules } from '../selective-preload-
strategy';

import 'rxjs/add/operator/map';

@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: PreloadSelectedModules
  ) {
    this.modules = preloadStrategy.preloadedModules;
  }

  ngOnInit() {
    // Capture the session ID if available
    this.sessionId = this.route
      .queryParams
      .map(params => params['session_id'] || 'None');

    // Capture the fragment if available
    this.token = this.route
      .fragment
      .map(fragment => fragment || 'None');
  }
}
```

一旦应用加载到初始路由，**CrisisCenterModule** 被主动加载了。要验证它，登录到 Admin 特征区域，注意 `crisis-center` 被列到 Preloaded Modules 并输出到控制台。我们可以继续添加特征区域，以便有选择的加载它们。

总结

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 [在线例子](#)，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，它详尽阐述了我们曾匆匆带过的一些知识点。

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

我们已经数次提及 [链接参数数组](#)，也用过好几次了。

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

我们可以把 `RouterLink` 指令绑定到一个数组，就像这样：

```
<a [routerLink]="/heroes">Heroes</a>
```

在指定路由参数时，我们写过一个双元素的数组，就像这样：

```
this.router.navigate(['/hero', hero.id]);
```

我们可以在对象中提供可选的路由参数，就像这样：

```
<a [routerLink]="/crisis-center", { foo: 'foo' }>Crisis Center</a>
```

这三个例子覆盖了我们在单级路由的应用中所需的一切。在添加一个像 **危机中心** 一样的子路由时，我们创建新链接数组组合。

回忆一下，我们曾为 **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 。

```
<a [routerLink]="/crisis-center">Crisis Center</a>
```

让我们把它分解出来：

- 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，因此我们得选取一个。
- 我们决定跳转到 CrisisListComponent，它的路由路径是 '/'，但我们不用显式的添加它。
- 哇！ ['/crisis-center'] 。

在下一步，我们会用到它。这次，我们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：

- 数组中的第一个条目用来标记出父路由 ('/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。

- 我们把 **巨龙危机** 的 `id` 添加为该数组中的第二个条目 (`1`)。

看起来是这样的：

```
<a [routerLink]=["'/crisis-center', 1]">Dragon Crisis</a>
```

如果想，我们还能单独使用 **危机中心** 的路由来重定义 `AppComponent` 的模板。

```
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：我们可以用一级、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：为什么要使用 `ngOnInit` 方法

我们在很多组件类中实现了 `ngOnInit` 方法。比如在 `HeroDetailComponent` 中就这么用过。也可以把 `ngOnInit` 中的逻辑放在构造函数中，但为了一个理由而没那么做，这个理由就是 **可测试性**。

有显著副作用的构造函数很难测试，因为它在创建测试实例时就开始做事了。这种情况下，它可能已经向远程服务器发起了请求，但有些事情在测试时没法做。可能在测试环境下无法访问服务器。

更好地实践是限制一下构造函数能做什么。通常它会把参数保存到局部变量中，以及执行简单的实例配置工作。

然而我们还是需要该类的实例在创建完之后尽快从 `HeroService` 中获取英雄数据。如果不能放在构造函数中又该怎么办？

Angular 会负责检测组件是否具有特定的生命周期方法，比如 `ngOnInit` 和 `ngOnDestroy`，并在合适的时机调用它们。

Angular 会在我们导航到 `HeroDetailComponent` 时调用 `ngOnInit`，我们将从 `ActivatedRoute` 的路由参数中取得 `id`，并向服务器请求具有这个 `id` 的英雄。

在获得了已注入的 `HeroService` 实例并（可能）做好模拟（Mock）之后，我们可以随时在测试中调用 `ngOnInit` 方法。

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。我们可以在启动过程中改写（`override`）它，来切换到 `HashLocationStrategy` 风格——如果我们更喜欢这种。

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？

我们必须选择一种策略，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，所以我们 **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。

```
<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。

你可以到 API 指南中学习关于 `APP_BASE_HREF` 的更多知识。

HashLocationStrategy

我们可以在根模块的 `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。

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';
```

```
const routes: Routes = [  
];  
  
@NgModule({  
  imports: [  
    BrowserModule,  
    FormsModule,  
    RouterModule.forRoot(routes, { useHash: true }) // .../#/crisis-  
    center/  
  ],  
  declarations: [  
    AppComponent  
  ],  
  providers: [  
  ],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule {  
}
```

安全

开发“内容安全”的 Angular 应用。

Web 应用程序的安全涉及到很多方面。针对常见的漏洞和攻击，比如跨站脚本攻击，Angular 提供了一些内置的保护措施。本章将讨论这些内置保护措施，但不会涉及应用级安全，比如用户认证（**这个用户是谁？**）和授权（**这个用户能做什么？**）。

要了解更多攻防信息，参见 [开放式 Web 应用程序安全项目 \(OWASP\)](#)。

目录

- [举报漏洞](#)。
- [最佳实践](#)。
- [防范跨站脚本 \(XSS\) 攻击](#)。
- [信任安全值](#)。
- [HTTP 级别的漏洞](#)。
- [审计 Angular 应用程序](#)。

运行 [在线例子](#) 来试用本页的代码。

举报漏洞

给我们 (security@angular.io) 发邮件，报告 Angular 本身的漏洞。

要了解关于“谷歌如何处理安全问题”的更多信息，参见 [谷歌的安全哲学](#)。

最佳实践

- **及时把 Angular 包更新到最新版本。** 我们会频繁的更新 Angular 库，这些更新可能会修复之前版本中发现的安全漏洞。查看 Angular 的 [更新记录](#)，了解与安全有关的更新。
- **不要修改你的 Angular 副本。** 私有的、定制版的 Angular 往往跟不上最新版本，这可能导致你忽略重要的安全修复与增强。反之，应该在社区共享你对 Angular 所做的改进并创建 Pull Request。
- **避免使用本文档中带“安全风险”标记的 Angular API。**

防范跨站脚本 (XSS) 攻击

跨站脚本 (XSS) 允许攻击者将恶意代码注入到页面中。这些代码可以偷取用户数据（特别是它们的登录数据），还可以冒充用户执行操作。它是 Web 上最常见的攻击方式之一。

为了防范 XSS 攻击，我们必须阻止恶意代码进入 DOM。比如，如果某个攻击者能骗我们把 `<script>` 标签插入到 DOM，就可以在我们的网站上运行任何代码。除了 `<script>`，攻击者还可以使用很多 DOM 元素和属性来执行代码，比如 ``、``。如果攻击者所控制的数据混进了 DOM，就会导致安全漏洞。

Angular 的“跨站脚本安全模型”

为了系统性的防范 XSS 问题，Angular 默认把所有值都当做不可信任的。当值从模板中以属性 (Property)、DOM 元素属性 (Attribte)、CSS 类绑定或插值表达式等途径插入到 DOM 中的时候，Angular 将对这些值进行无害化处理 (Sanitize)，对不可信的值进行编码。

Angular 的模板同样是可执行的：模板中的 HTML 、 Attribute 和绑定表达式（还没有绑定到值的时候）会被当做可信任的。这意味着应用必须防止把可能被攻击者控制的值直接编入模板的源码中。永远不要根据用户的输入和原始模板动态生成模板源码！使用离线模板编译器 是防范这类“模板注入”漏洞的有效途径。

无害化处理与安全环境

无害化处理会审查不可信的值，并将它们转换成可以安全插入到 DOM 的形式。多数情况下，这些值并不会在处理过程中发生任何变化。无害化处理的方式取决于所在的环境：一个在 CSS 里面无害的值，可能在 URL 里很危险。

Angular 定义了四个安全环境 - HTML , 样式 , URL , 和资源 URL :

- **HTML** : 值需要被解释为 HTML 时使用，比如当绑定到 `innerHTML` 时。
- **样式** : 值需要作为 CSS 绑定到 `style` 属性时使用。
- **URL** : 值需要被用作 URL 属性时使用，比如 `<a href>` 。
- **资源 URL** : 值需要被当做代码而加载并执行时使用，比如 `<script src>` 中的 URL 。

Angular 会对前三项中种不可信的值进行无害化处理。但 Angular 无法对第四种资源 URL 进行无害化，因为它们可能包含任何代码。在开发模式下，如果 Angular 在进行无害化处理时需要被迫改变一个值，它就会在控制台上输出一个警告。

无害化示例

下面的例子绑定了 `htmlSnippet` 的值，一次把它放进插值表达式里，另一次把它绑定到元素的 `innerHTML` 属性上。

```
app/inner-html-binding.component.html

1.  <h3>Binding innerHTML</h3>
2.  <p>Bound value:</p>
3.  <p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>
4.  <p>Result of binding to innerHTML:</p>
5.  <p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>
```

插值表达式的内容总会被编码 - 其中的 HTML 不会被解释 , 所以浏览器会在元素的文本内容中显示尖括号。

如果希望这段 HTML 被正常解释 , 就必须绑定到一个 HTML 属性上 , 比如 `innerHTML` 。但是如果把一个可能被攻击者控制的值绑定到 `innerHTML` 就会导致 XSS 漏洞。比如 , 包含在 `<script>` 标签的代码就会被执行 :

app/inner-html-binding.component.ts (inner-html-controller)

```
export class InnerHtmlBindingComponent {
  // For example, a user/attacker-controlled value from a URL.
  htmlSnippet = 'Template <script>alert("Owned")</script>
<b>Syntax</b>';
}
```

Angular 认为这些值是不安全的 , 并自动进行无害化处理。它会移除 `<script>` 标签 , 但保留安全的内容 , 比如该片段中的文本内容或 `` 元素。

Binding innerHTML

Bound value:

Template `<script>alert("Owned")</script> Syntax`

Result of binding to innerHTML:

Template `alert("Owned") Syntax`

避免直接使用 DOM API

浏览器内置的 DOM API 不会自动针对安全漏洞进行防护。比如 , `document` (它可以通过 `ElementRef` 访问) 以及其它第三方 API 都可能包含不安全的方法。要避免直接与 DOM 交互 , 只要可能 , 就尽量使用 Angular 模板。

内容安全策略

内容安全策略 (CSP) 是用来防范 XSS 的纵深防御技术。要打开 CSP，请配置你的 Web 服务器，让它返回合适的 HTTP 头 Content_Security_Policy。

使用离线模板编译器

离线模板编译器阻止了一整套被称为“模板注入”的漏洞，并能显著增强应用程序的性能。尽量在产品发布时使用离线模板编译器，而不要动态生成模板（比如在代码中拼接字符串生成模板）。由于 Angular 会信任模板本身的代码，所以，动态生成的模板——特别是包含用户数据的模板——会绕过 Angular 自带的保护机制。要了解如何用安全的方式动态创建表单，请参见 [动态表单烹饪宝典](#) 一章。

服务器端 XSS 保护

服务器端构造的 HTML 很容易受到注入攻击。当需要在服务器端生成 HTML 时（比如 Angular 应用的初始页面），务必使用一个能够自动进行无害化处理以防范 XSS 漏洞的后端模板语言。不要在服务器端使用模板语言生成 Angular 模板，这样会带来很高的“模板注入”风险。

信任安全的值

有时候，应用程序确实需要包含可执行的代码，比如使用 URL 显示 <iframe>，或者构造出有潜在危险的 URL。为了防止在这种情况下被自动无害化，你可以告诉 Angular：我已经审查了这个值，检查了它是怎么生成的，并确信它总是安全的。但是 **千万要小心**！如果你信任了一个可能是恶意的值，就会在应用中引入一个安全漏洞。如果你有疑问，请找一个安全专家复查下。

注入 DomSanitizer 服务，然后调用下面的方法之一，你就可以把一个值标记为可信任的。

- `bypassSecurityTrustHtml`
- `bypassSecurityTrustScript`
- `bypassSecurityTrustStyle`
- `bypassSecurityTrustUrl`
- `bypassSecurityTrustResourceUrl`

记住，一个值是否安全取决于它所在的环境，所以你要为这个值按预定的用法选择正确的环境。假设下面的模板需要把 `javascript.alert(...)` 方法绑定到 URL。

app/bypass-security.component.html (dangerous-url)

```
<h4>An untrusted URL:</h4>
<p><a class="e2e-dangerous-url" [href]="dangerousUrl">click me</a>
</p>
<h4>A trusted URL:</h4>
<p><a class="e2e-trusted-url" [href]="trustedUrl">click me</a></p>
```

通常，Angular 会自动无害化这个 URL 并禁止危险的代码。为了防止这种行为，我们可以调用 `bypassSecurityTrustUrl` 把这个 URL 值标记为一个可信任的 URL：

app/bypass-security.component.ts (trust-url)

```
constructor(private sanitizer: DomSanitizer) {
  // javascript: URLs are dangerous if attacker controlled.
  // Angular sanitizes them in data binding, but you can
  // explicitly tell Angular to trust this value:
  this.dangerousUrl = 'javascript:alert("Hi there")';
  this.trustedUrl =
  sanitizer.bypassSecurityTrustUrl(this.dangerousUrl);
```

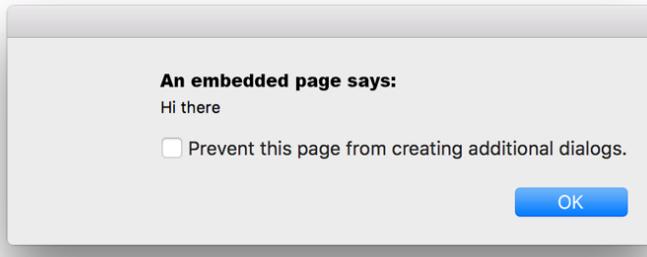
Bypass Security Component

An untrusted URL:

[Click me](#)

A trusted URL:

[Click me](#)



如果需要把用户输入转换为一个可信任的值，我们可以很方便的在控制器方法中处理。

下面的模板允许用户输入一个 YouTube 视频的 ID，然后把相应的视频加载到

`<iframe>` 中。`<iframe src>` 是一个“资源 URL”的安全环境，因为不可信的源码可能作

为文件下载到本地，被毫无防备的用户执行。所以我们要调用一个控制器方法来构造一个新的、可信任的视频 URL，然后把它绑定到 `<iframe src>`。

app/bypass-security.component.html (iframe-videooid)

```

<h4>Resource URL:</h4>
<p><label>Showing: <input
  (input)="updateVideoUrl($event.target.value)"></label></p>
<p>Trusted:</p>
<iframe class="e2e-iframe-trusted-src" width="640" height="390"
[src]="videoUrl"></iframe>
<p>Untrusted:</p>
<iframe class="e2e-iframe-untrusted-src" width="640" height="390"
[src]="dangerousVideoUrl"></iframe>

```

app/bypass-security.component.ts (trust-video-url)

```

updateVideoUrl(id: string) {
  // Appending an ID to a YouTube URL is safe.
  // Always make sure to construct SafeValue objects as
  // close as possible to the input data so
  // that it's easier to check if the value is safe.
  this.dangerousVideoUrl = 'https://www.youtube.com/embed/' + id;
  this.videoUrl =
    this.sanitizer.bypassSecurityTrustResourceUrl(this.dangerousVideoUrl);

}

```

HTTP 级别的漏洞

Angular 内置了一些支持来防范两个常见的 HTTP 漏洞：跨站请求伪造（XSRF）和跨站脚本包含（XSSI）。这两个漏洞主要在服务器端防范，但是 Angular 也自带了一些辅助特性，可以让客户端的集成变得更容易。

跨站请求伪造 (XSRF)

在跨站请求伪造 (XSRF 或 CSFR) 中，攻击者欺骗用户，让他们访问一个假冒页面 (例如 `evil.com`)，该页面带有恶意代码，秘密的向你的应用程序服务器发送恶意请求 (e.g. `example-bank.com`)。

假设用户已经在 `example-bank.com` 登录。用户打开一个邮件，点击里面的链接，在新页面中打开 `evil.com` 。

该 `evil.com` 页面立刻发送恶意请求到 `example-bank.com` 。这个请求可能是从用户账户转账到攻击者的账户。与该请求一起，浏览器自动发出 `example-bank.com` 的 cookie。

如果 `example-bank.com` 服务器缺乏 XSRF 保护，就无法辨识请求是从应用程序发来的合法请求还是从 `evil.com` 来的假请求。

为了防止这种情况，你必须确保每个用户的请求都是从你自己的应用中发出的，而不是从另一个网站发出的。客户端和服务器必须合作来抵挡这种攻击。

常见的反 XSRF 技术是服务器随机生成一个用户认证令牌到 cookie 中。客户端代码获取这个 cookie，并用它为接下来所有的请求添加自定义请求页头。服务器比较收到的 cookie 值与请求页头的值，如果它们不匹配，便拒绝请求。

这个技术之所以有效，是因为所有浏览器都实现了 **同源策略**。只有设置 cookie 的网站的代码可以访问该站的 cookie，并为该站的请求设置自定义页头。这就是说，只有你的应用程序可以获取这个 cookie 令牌和设置自定义页头。`evil.com` 的恶意代码不能。

Angular 的 `http` 客户端在其 `XSRFStrategy` 中具有对这项技术的内置支持。默认的 `CookieXSRFStrategy` 会被自动开启 在发送每个请求之前，`CookieXSRFStrategy` 查询名为 `XSRF-TOKEN` 的 cookie，并设置一个名为 `X-XSRF-TOKEN` 的 HTTP 请求头，并把该 cookie 的值赋给它。

服务器必须要完成自己的任务，设置初始 `XSRF-TOKEN` cookie，并确认接下来的每个请求包含了配对的 `XSRF-TOKEN` cookie 和 `X-XSRF-TOKEN` 页头。

CSRF 令牌对每个用户和 session 应该是唯一的，它包含一大串由安全的随机数字生成器生成的随机值，并且在一两天之内过期。

你服务器可能使用不同的 cookie 或者页头名字。Angular 应用可以通过自己的 `CookieXSRFStrategy` 值来自定义 cookie 和页头名字。

```
{ provide: XSRFStrategy, useValue: new  
CookieXSRFStrategy('myCookieName', 'My-Header-Name') }
```

或者你可以实现和提供完整的自定义 `XSRFStrategy`。

```
{ provide: XSRFStrategy, useClass: MyXSRFStrategy }
```

到开放式 Web 应用程序安全项目 (OWASP) 的 [这里](#) 和 [这里](#) 学习更多关于跨站请求伪造 (XSRF) 的知识。这个 [斯坦福大学论文](#) 有详尽的细节。

参见 Dave Smith 在 [AngularConnect 2016](#) 关于 XSRF 的演讲。

跨站脚本包含 (XSSI)

跨站脚本包含，也被称为 Json 漏洞，它可以允许一个攻击者的网站从 JSON API 读取数据。这种攻击发生在老的浏览器上，它重写原生 JavaScript 对象的构造函数，然后使用 `<script>` 标签包含一个 API 的 URL。

只有在返回的 JSON 能像 JavaScript 一样可以被执行时，这种攻击才会生效。所以服务端会约定给所有 JSON 响应体加上前缀 `")}]',\n"`，来把它们标记为不可执行的，以防范这种攻击，

Angular 的 `Http` 库会识别这种约定，并在进一步解析之前，自动把字符串 `")}]',\n"` 从所有响应中去掉。

要学习更多这方面的知识，请参见 [谷歌 Web 安全博客文章](#) 的 XSSI 小节。

审计 Angular 应用程序

Angular 应用应该遵循和常规 Web 应用一样的安全原则并按照这些原则进行审计。

Angular 中某些应该在安全评审中被审计的 API（比如 `bypassSecurityTrust` API）都在文档中被明确标记为安全性敏感的。

结构型指令

Angular 有一个强力的模板引擎，它能让你轻松维护元素的 DOM 树结构。

单页面应用的基本特性之一，就是它要操纵 DOM 树。不同于以前那种用户每次浏览都重新从服务器取得整个页面的方式，单页面应用中，DOM 中的各个区域会根据应用程序的状态而出现或消失。在本章中，我们将看看 Angular 如何操纵 DOM 树，以及我们该如何在自己的指令中这么做。

在本章中，我们将：

- 学习什么是结构型 (structural) 指令
- 研究 `ngIf`
- `<template>` 元素揭秘
- 理解 `*ngFor` 中的星号 (*)
- 写我们自己的结构型指令

试试 [在线例子](#)。

什么是结构型指令？

Angular 指令可分为三种：

1. 组件
2. 属性型指令
3. 结构型指令

组件 其实就是一个带模板的指令。它是这三种指令中最常用的，我们会写大量的组件来构建应用程序。

属性型 指令会修改元素的外观或行为。比如，内置指令 **NgStyle** 就能同时修改元素的好几个样式。通过绑定到组件的属性，我们可以把文本渲染成加粗、斜体、灰绿色这种肉麻的效果。

结构型 指令通过添加和删除 DOM 元素来改变 DOM 的布局。我们会在其它章节看到三个内置的结构型指令：**ngIf**、**ngSwitch** 以及 **ngFor**。

```
<div *ngIf="hero">{{hero}}</div>
<div *ngFor="let hero of heroes">{{hero}}</div>
<div [ngSwitch]="status">
  <template [ngSwitchCase]="'in-mission'">In Mission</template>
  <template [ngSwitchCase]="'ready'">Ready</template>
  <template ngSwitchDefault>Unknown</template>
</div>
```

NgIf 案例分析

我们重点看下 **ngIf**。它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。

```
<p *ngIf="condition">
  condition is true and ngIf is true.
</p>
<p *ngIf="!condition">
  condition is false and ngIf is false.
</p>
```

ngIf 指令并不会隐藏元素。使用浏览器的开发者工具就会看到：当 `condition` 为真的时候，只剩下了 DOM 顶部的段落，而底部无用的段落完全从 DOM 中消失了！在它的位置上是空白的 `<script>` 标签

```
<p _ngcontent-tnn-1>
  condition is true and ngIf is true.
</p>
<script></script>
```

为什么 移除 而不是 隐藏 ?

其实也可以通过把 CSS 样式 `display` 设置为 `none` 来隐藏掉那个不想要的段落。该元素仍然留在 DOM 中，只是看不到了。但我们却通过 `ngIf` 移除了它。

不同之处在于：当我们隐藏掉一个元素时，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，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。

而 `ngIf` 不同。把 `ngIf` 设置为假 **将会** 影响到组件的资源消耗。Angular 会从 DOM 中移除该元素，停止相关组件的变更检测，把它从 DOM 事件中摘掉（事件是组件造成的附加项），并销毁组件。组件会被垃圾回收（希望如此）并释放内存。

组件通常还有子组件，子组件还有自己的子组件。当 `ngIf` 销毁这个祖先组件时，它们全都会被销毁。这种清理工作通常会是好事。

当然，它也并不 **总是** 好事。如果我们很快就会再次需要这个组件，它就变成坏事了。

重建组件的状态可能是昂贵的。当 `ngIf` 重新变为 `true` 的时候，Angular 会重新创建该组件及其子树。Angular 会重新运行每个组件的初始化逻辑。那可能会很昂贵……比如当组件需要重新获取刚刚还在内存中的数据时。

设计思路：要最小化初始化的成本，并考虑把状态缓存在一个伴生的服务中。

虽然每种方法都有各自的优点和缺点，但使用 `ngIf` 来移除不需要的组件通常都会比隐藏它们更好一些。

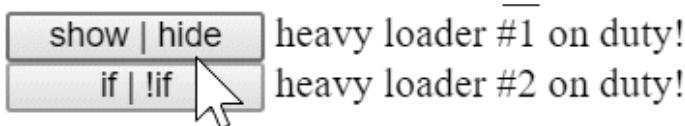
同样的考量也适用于每一个结构型指令，无论是内置的还是自定义的。我们应该提醒自己以及我们指令的使用者，来仔细考虑添加元素、移除元素以及创建和销毁组件的后果。

让我们在实践中看看这些变化。为了娱乐，我们设想在甲板上有个叫 `heavy-loader`（重型起重机）的组件，它会 **假装** 在初始化时装载一吨数据。

我们将显示该组件的两个实例。我们使用 CSS 切换第一个实例的可见性，用 `ngIf` 把第二个实例添加到 DOM 和将其移除。

```
1.  <div><!-- visibility -->
2.    <button (click)="isVisible = !isVisible">show | hide</button>
3.    <heavy-loader [style.display]="isVisible ? 'inline' : 'none'">
4.      [logs]="logs"></heavy-loader>
5.    </div>
6.
7.  <div><!-- NgIf -->
8.    <button (click)="condition = !condition">if | !if</button>
9.    <heavy-loader *ngIf="condition" [logs]="logs"></heavy-loader>
10.
11. <h4>heavy-loader log:</h4>
12. <div *ngFor="let message of logs">{{message}}</div>
```

借助内置的 `ngOnInit` 和 `ngOnDestroy` 生命周期钩子，我们同时记录了组件的创建或销毁过程。下面是它的操作演示：



heavy-loader log:

heavy-loader 1 initialized, loading 10,000 rows of data from the server
heavy-loader 2 initialized, loading 10,000 rows of data from the server

开始的时候，两个组件都在 DOM 中。首先我们重复切换第一个组件的可见性。组件从未离开过 DOM 节点。当可见时，它总是同一个实例，而日志里什么都没有。

当我们切换使用 `ngIf` 的第二个实例时。我们每次都会创建新的实例，而日志中显示，我们为了创建和销毁它付出了沉重的代价。

如果我们真的期望像这样让组件“眨眼”，切换可见性就会是更好的选择。在大多数 UI 中，当我们“关闭”一个组件时，在相当长时间内都不大可能想再见到它——可能永远也不见。在这种情况下，我们会更喜欢 `ngIf`。

<template> 标签

结构型指令，比如 `ngIf`，使用 [HTML 5 的 template 标签](#) 完成它们的“魔法”。

在 Angular 应用之外，`<template>` 标签的默认 CSS 属性 `display` 是 `none`。它的内容存在于一个 [隐藏的 文档片段](#) 中。

而在 Angular 应用中，Angular 会 [移除](#) `<template>` 标签及其子元素。这些内容不见了，但是并没有被“忘记”，我们很快就明白了。

我们可以通过把短语 "Hip! Hip! Hooray!" 中间的 "hip" 包在一个 `<template>` 标签中来验证下这个效果。

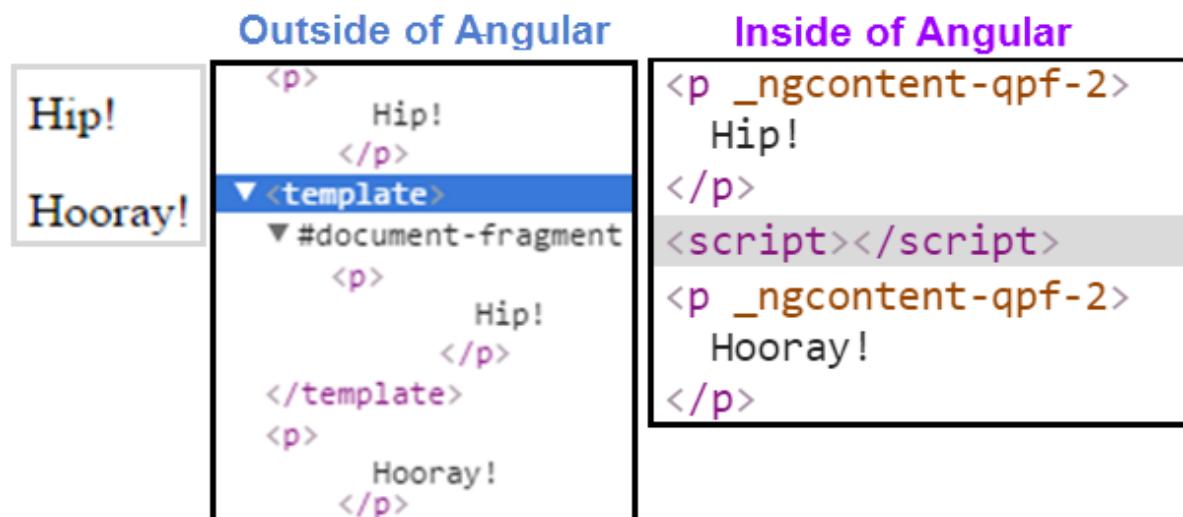
```
<p>  
  Hip!
```

```

</p>
<template>
  <p>
    Hip!
  </p>
</template>
<p>
  Hooray!
</p>

```

这时候显示的内容是 'Hip! Hooray!'，缺乏完美的热情（译注：因为少了一个词嘛）。在 Angular 的控制下，DOM 的效果是不同的。



显然，Angular 把 `<template>` 标签及其内容替换成了一个空白的 `<script>` 标签。这只是它的默认行为。当把 `ngSwitch` 家族的各种指令应用于 `<template>` 标签时，我们就会看到有些东西不一样了：

```

<div [ngSwitch]="status">
  <template [ngSwitchCase]="'in-mission'">In Mission</template>
  <template [ngSwitchCase]="'ready'">Ready</template>
  <template ngSwitchDefault>Unknown</template>
</div>

```

当这些 `ngSwitch` 的条件之一为真的时候，Angular 把模板的内容插入到了 DOM 中。

这和 `ngIf` 和 `ngFor` 有什么关系？很明显，我们在那些指令中并没有用到 `<template>` 标签。

星号 (*) 效果

下面也是那些指令。看出有什么不同了吗？

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

我们把那些指令名加上了星号 (*) 前缀。

这个星号是一种“语法糖”。它简化了 `ngIf` 和 `ngFor` ——无论是写还是读。

接下来这两个 `ngIf` 范例的效果完全相同，只是我们写成了另一种风格：

```
<!-- Examples (A) and (B) are the same -->
<!-- (A) *ngIf paragraph -->
<p *ngIf="condition">
    Our heroes are true!
</p>

<!-- (B) [ngIf] with template -->
<template [ngIf]="condition">
    <p>
        Our heroes are true!
    </p>
</template>
```

大多数都喜欢用风格 (A) 来写。

要知道，Angular 会把风格 (A) 写成风格 (B)。它把段落及其内容移到了 `<template>` 标签中。它把指令移到了 `<template>` 标签上，成为该标签的一个属性绑定——包装在方括号中。宿主组件的 `condition` 属性的布尔值决定该模板的内容是否应该被显示。

Angular 把 `*ngFor` 转换成一个类似的形式：

```
<!-- Examples (A) and (B) are the same -->

<!-- (A) *ngFor div -->
<div *ngFor="let hero of heroes">{{ hero }}</div>

<!-- (B) ngFor with template -->
<template ngFor let-hero [ngForOf]="heroes">
  <div>{{ hero }}</div>
</template>
```

基本的转换模式是一样的：创建一个 `<template>`，将内容重定位，并且把指令移到 `<template>` 上。

Angular 的 `ngFor` 微语法 里面有一些细微差别，它被展开成了另一个 `ngForOf` 属性绑定（可迭代者）和另一个模板输入变量 `hero`（每次迭代中的当前条目）。

制作一个结构型指令

我们来写自己的结构型指令：`Unless`，这是 `ngIf` 指令不那么邪恶的孪生兄弟。

当条件为 `true` 时 `ngIf` 才显示模板内容，与之不同的是，我们这个指令只有当条件是 `false` 时才显示这些内容。

创建指令很像创建组件。

- 导入 `Directive` 装饰器。
- 添加一个 CSS 属性选择器（括号中），来标记出我们的指令。
- 指定 `input` 属性用于绑定的公开名称（通常就是指令自己的名字）。
- 把这个装饰器应用到我们的实现类上。

下面是最初的样子：

unless.directive.ts (excerpt)

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

@Directive({ selector: '[myUnless]' })
export class UnlessDirective {
}
```

选择器中的括号 []

在 CSS 中，用于选择属性 (Attribute) 的选择器就是放在方括号中的名字。于是我们把指令名包裹在方括号中。参见 [小抄](#) 中的 [指令配置项](#)。

选择器名称前缀

我们建议在给选择器起名时加个前缀，以确保它不会和任何标准的 HTML 属性冲突，无论是现在还是未来。

我们 **并没有** 给 `unless` 指令名加上 `ng` 前缀。那个前缀是属于 Angular 的，我们肯定不会希望自己的指令和 Angular 内置的指令冲突。

我们用的前缀是 `my`。

我们需要访问模板，并且 **还** 需要一个渲染器来渲染它的内容。我们通过 `TemplateRef` 来访问模板。渲染器是 `ViewContainerRef`。我们把它们都作为私有变量注入到构造函数中。

```
constructor(
  private templateRef: TemplateRef<any>,
  private viewContainer: ViewContainerRef
) { }
```

这个指令的使用者将把一个布尔值绑定到指令的输入属性 `myUnless` 上。该指令会基于这个值添加或移除此模板。

我们现在先把 `myUnless` 属性定义成一个“只写”属性。

```
@Input() set myUnless(condition: boolean) {  
  if (!condition) {  
    this.viewContainer.createEmbeddedView(this.templateRef);  
  } else {  
    this.viewContainer.clear();  
  }  
}
```

`@Input()` 装饰器表明这个属性对于指令来说是个输入属性。

这里没什么特别的：如果条件为假，我们就渲染模板，否则就清空元素内容。

最终看起来是这样的：

unless.directive.ts

```
1. import { Directive, Input } from '@angular/core';  
2.  
3. import { TemplateRef, ViewContainerRef } from '@angular/core';  
4.  
5. @Directive({ selector: '[myUnless]' })  
6. export class UnlessDirective {  
7.  
  constructor(  
    private templateRef: TemplateRef<any>,  
    private viewContainer: ViewContainerRef  
  ) {}  
12.  
13.   @Input() set myUnless(condition: boolean) {  
14.     if (!condition) {
```

```

15.      this.viewContainer.createEmbeddedView(this.templateRef);
16.    } else {
17.      this.viewContainer.clear();
18.    }
19.  }
20. }

```

现在，我们就来把它加到 AppModule 的 declarations 数组中，试一试。我们首先把一些测试用的 HTML 添加到模板中：

```

<p *myunless="condition">
  condition is false and myUnless is true.
</p>

<p *myunless="!condition">
  condition is true and myUnless is false.
</p>

```

我们运行它，它的行为正如所预期的那样——跟 `ngIf` 相反。当 `condition` 为 `true` 时，顶部的段落被移除了（被替换为 `<script>` 标签），并且底部的段落显示了出来。

```

<script></script>
<p _ngcontent-tnn-1>
  condition is true and myUnless is false.
</p>

```

这个 `myUnless` 指令实在太简单了，我们肯定忘了点什么。那么 `ngIf` 会更复杂吗？

[看下源码](#)。它有很好的文档，况且，如果我们想了解某些东西的工作原理，也不用羞于“咨询”源码。

`ngIf` 也没多大不同嘛！它做了更多的检查来提升性能（除非必要，否则它不会清除或重新创建视图），但其它的部分都跟我们写的一样。

总结

本章相关的代码如下：

```
1. import { Directive, Input } from '@angular/core';
2.
3. import { TemplateRef, ViewContainerRef } from '@angular/core';
4.
5. @Directive({ selector: '[myUnless]' })
6. export class UnlessDirective {
7.
8.   constructor(
9.     private templateRef: TemplateRef<any>,
10.    private viewContainer: ViewContainerRef
11.  ) { }
12.
13.   @Input() set myUnless(condition: boolean) {
14.     if (!condition) {
15.       this.viewContainer.createEmbeddedView(this.templateRef);
16.     } else {
17.       this.viewContainer.clear();
18.     }
19.   }
20. }
```

我们学会了通过像 `ngFor` 和 `ngIf` 这样的结构型指令来操纵 HTML 的布局。我们还写出了我们的第一个结构型指令 `myUnless` 来做类似的事情。

Angular 提供了更多成熟的技术来管理布局，比如 **结构性组件** 可以接受外部内容，并把这些内容合并到组件自己的模板中。多页标签及其面板控件就是很好的例子。

我们将在未来的章节中还会讲述结构型指令。

测试

Angular 应用的测试技术与实践。

本章提供了一些测试 Angular 应用的提示和技巧。虽然这里讲述了一些常规测试理念和技巧，但是其重点是测试用 Angular 编写的应用。

目录

1. Angular 测试入门

2. 搭建测试环境

- 文件配置 : `karma.conf` , `karma-test-shim` , `systemjs.config`
- npm 包

3. 第一个 Karma 测试程序

4. Angular 测试工具

5. 例子应用及其测试

6. 简单的组件测试程序

- `configureTestingModule`
- `createComponent`

- **ComponentFixture, DebugElement, query(By.css)**

- **detectChanges**

- **autoDetectChanges**

7. 测试拥有服务依赖的组件

- 测试复制品

- 获取注入的服务

- **TestBed.get**

8. 测试拥有异步服务的组件

- **spies**

- **async**

- **whenStable**

- **fakeAsync**

- **tick**

- **jasmine.done**

9. 测试拥有外部模板的组件

- **async** in `beforeEach`

- **compileComponents**

10. 测试拥有导入 inputs 和导出 outputs 的组件

- **triggerEventHandler**

11. 在宿主组件中测试组件

12. 测试带路由器的组件

- **inject**

13. 测试带有路由和路由参数的组件

- **可观察** 测试复制品

14. 使用 **page** 对象来简化配置

15. 用模块的导入进行配置

16. 重载组件提供商

17. 测试带有 **RouterOutlet** 组件

- 模拟不需要的组件
- 模拟 **RouterLink**
- **By.directive** 和注入的指令

18. 使用 **NO_ERRORS_SCHEMA** 来“浅化”组件测试程序

19. 测试属性指令

20. 孤立的单元测试

- 服务
- 管道
- 组件

21. Angular 测试工具 APIs

- **独立函数** : `async` , `fakeAsync` , etc.
- **TestBed**

- [ComponentFixture](#)
- [DebugElement](#)
- [DebugElement](#)

22. 常见问题

以上主题繁多。幸运的是，你可以慢慢地阅读并立刻应用每一个主题。

在线例子

本章所有例子代码都在下面的在线例子中，以供参考、实验和下载。

- [例子应用](#)
- [第一个测试程序spec](#)
- [完整的应用测试程序spec](#)
- [示范spec包](#)

[回到顶部](#)

Angular 测试入门

编写测试程序来探索和确认应用的行为。

1. 测试 **守护** 由于代码变化而打破已有代码（“回归”）的情况。
2. 不管代码被正确使用还是错误使用，测试程序起到 **澄清** 代码的作用。
3. 测试程序 **暴露** 设计和实现可能出现的错误。测试程序从很多角度为代码亮出警报灯。当应用程序很难被测试时，其根本原因一般都是设计缺陷，这种缺陷最好立刻被修正，不要等到它变得很难被修复的时候才行动。

本章假设你熟悉测试。但是如果你不熟悉也没有关系。有很多书本和在线资源可以帮助你。

工具与技术

你可以用多种工具和技术来编写和运行 Angular 测试程序。本章介绍了一些大家已经知道能良好工作的选择。

技术	目的
Jasmine	<p>Jasmine 测试框架 提供了所有编写基本测试的工具。它自带 HTML 测试运行器，用来在浏览器中执行测试程序。</p>
Angular测试工具	<p>Angular 测试工具为被测试的 Angular 应用代码创建测试环境。在应用代码与 Angular 环境互动时，使用 Angular 测试工具来限制和控制应用的部分代码。</p>
Karma	<p>karma 测试运行器 是在开发应用的过程中 编写和运行单元测试的理想工具。它能成为项目开发和连续一体化进程的不可分割的一部分。本章讲述了如何用 Karma 设置和运行测试程序。</p>
Protractor	<p>使用 Protractor 来编写和运行 端对端 (e2e) 测试程序。端对端测试程序 像用户体验应用程序那样 探索它。在端对端测试中，一条进程运行真正的应用，另一条进程运行 Protractor 测试程序，模拟用户行为，判断应用在浏览器中的反应是否正确。</p>

搭建测试环境

很多人认为编写测试程序很有趣。很少有人享受搭建测试环境的过程。为了尽快开始有趣的部分，我们在本章后面一点再详细讲述搭建测试环境的细节。在这里，我们先进行最简单的讨论，再加上源代码供大家下载。

有两种快速方法立刻开始。

1. 根据 [快速起步的 github 库](#) 中的说明创建新的项目 .
2. 使用 [Angular CLI](#) 创建新的项目。

以上两种方法都安装在各自的模式下为应用预先配置的 **npm 包、文件和脚本**。它们的文件和规程有一点不同，但是它们的核心部分是一样的，并且在测试代码方面没有任何区别。

在本章中，应用及其测试程序是基于《快速起步》库的。

如果你的应用是基于《快速起步》库的，可以跳过本小节下面的内容，直接开始第一个测试。《快速起步》库一同了所有必须的配置。

文件配置

下面是文件配置的简介：

文件	描述
karma.conf.js	Karma 配置文件。它指定使用哪些插件、加载哪个应用和测试文件、使用哪些浏览器和如何报告测试结果。

它加载三种配置文件：

- `systemjs.config.js`
- `systemjs.config.extras.js`
- `karma-test-shim.js`

`karma-test-shim.js`

本垫片为 Angular 测试环境特别准备 Karma，并运行 Karma。在这个过程中，它加载 `systemjs.config.js` 文件。

`systemjs.config.js`

`SystemJS` 加载应用和测试文件。本脚本告诉 SystemJS 去哪儿寻找这些文件，以及如何加载它们。
`systemjs.config.js` 的版本和基于《快速起步》的应用使用的一样。

`systemjs.config.extras.js`

一个可有可无的文件，用来配置应用自己特殊的需求，补充 `systemjs.config.js` 里面的 SystemJS 配置。

原装 `systemjs.config.js` 无法预测这些需求。你可以在这里填补空白。

本章例子添加了 **模型封装桶** 到 SystemJS 的 `packages` 配置。

`systemjs.config.extras.js`

```
/** App specific SystemJS configuration */
System.config({
  packages: {
    // barrels
    'app/model': {main:'index.js', defaultExtension:'js'},
    'app/model/testing': {main:'index.js',
      defaultExtension:'js'}
  }
});
```

npm 包

例子中的测试程序是按照能在 `Jasmine` 和 `Karma` 中运行的规格编写的。以上两种“快速途径”配置测试环境，都在 `package.json` 的 `devDependencies` 中添加了合适的 `Jasmine` 和 `karma` 的 `npm` 包。你运行 `npm install` 时就会安装它们。

第一个 `karma` 测试

编写简单的测试程序，来确认以上的配置是否工作正常。

在应用的根目录 `app/` 创建新文件，名叫 `1st.spec.ts`。

用 `Jasmine` 编写的测试程序都被叫做 `specs`。文件名后缀必须是 `.spec.ts`，这是 `karma.conf.js` 和其它工具所坚持和遵守的规约。

将测试程序 `spec` 放到 `app/` 文件夹下的任何位置。`karma.conf.js` 告诉 `Karma` 在这个文件夹中寻找测试程序 `spec` 文件，原因在 [这里](#) 有所解释。

添加下面的代码到 `app/1st.spec.ts`。

app/1st.spec.ts

```
describe('1st tests', () => {
  it('true is true', () => expect(true).toBe(true));
});
```

运行 Karma

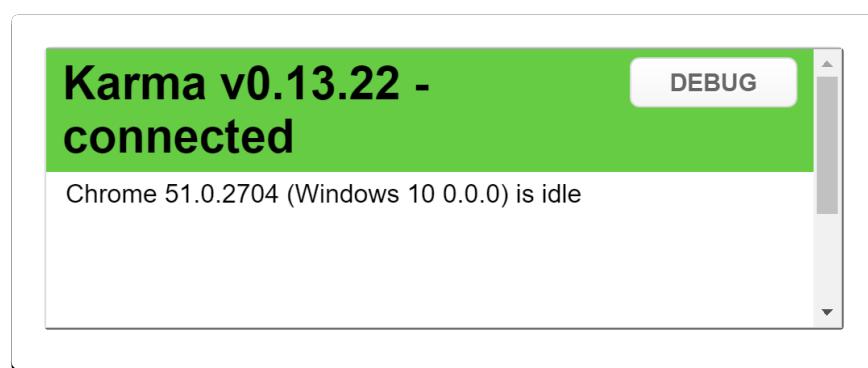
使用下面的命令从命令行中编译并在 Karma 中运行上面的测试程序。

```
npm test
```

该命令编译应用及其测试代码，并启动 Karma。两个进程都监视相关文件，往控制台输入信息和检测到变化时自动重新运行。

《快速开始》在 npm 的 `package.json` 中的 `scripts` 里定义了 `test` 命令。
Angular CLI 使用不同的命令来做同样的事情。对不同的环境采取不同的方案。

等一小段时间后，Karma 便打开浏览器并开始向控制台输出。



隐藏（不要关闭）浏览器，查看控制台的输出，应该看起来像这样：

```
> npm test
...
[0] 1:37:03 PM - Compilation complete. Watching for file changes.
...
```

```
[1] Chrome 51.0.2704: Executed 0 of 0 SUCCESS
    Chrome 51.0.2704: Executed 1 of 1 SUCCESS
    SUCCESS (0.005 secs / 0.005 secs)
```

编译器和 Karma 都会持续运行。编译器的输入信息前面有 [0] , Karma 的输出前面有 [1] 。

将期望从 true 变换为 false 。

编译器 监视器检测到这个变化并重新编译。

```
[0] 1:49:21 PM - File change detected. Starting incremental
compilation...
[0] 1:49:25 PM - Compilation complete. Watching for file changes.
```

Karma 监视器检测到编译器输出的变化，并重新运行测试。

```
[1] Chrome 51.0.2704 1st tests true is true FAILED
[1] Expected false to equal true.
[1] Chrome 51.0.2704: Executed 1 of 1 (1 FAILED) (0.005 secs / 0.005
secs)
```

正如所料，测试结果是 失败 。

将期望从 false 恢复为 true 。两个进程都检测到这个变化，自动重新运行， Karma 报告测试成功。

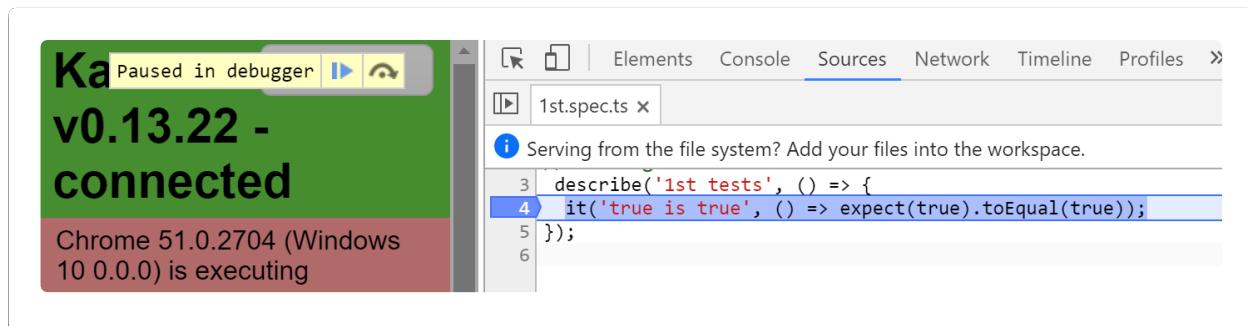
控制台的日志可能会非常长。注意最后一样。当一切正常时，它会显示 SUCCESS 。

调试测试程序

在浏览器中，像调试应用一样调试测试程序 spec 。

- 显示 Karma 的浏览器窗口（之前被隐藏了）。

- 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
- 打开浏览器的“ Developer Tools ” (F12 或者 Ctrl-Shift-I)。
- 选择“ sources ”页
- 打开 `1st.spec.ts` 测试文件 (Ctrl-P, 然后输入文件名字)。
- 在测试程序中设置断点。
- 刷新浏览器 ... 然后它就会停在断点上。



[回到顶部](#)

Angular 测试工具入门

许多测试程序探索应用的类在被 `Angular` 控制时，是如何与 `Angular` 和 `DOM` 互动的。

Angular 测试工具包含了 `TestBed` 类和一些辅助函数方法，在它们的帮助下，很容易编写上面那样的测试程序。

利用 **这些工具** 编写的测试程序是本章的主要焦点。但是它们不是你能写的唯一测试类型。

孤立的单元测试

孤立的单元测试 独立自己检测类的实例，不依靠 Angular 或者任何其它注入值。 测试器使用 `new` 创建类的测试实例，在需要时提供用于测试的构造函数参数复制品，并测试被测试实例的 API。

你可以，也应该为服务和管道编写孤立的单元测试。

app/shared/title-case.pipe.spec.ts (excerpt)

```
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.});
```

孤立的测试不会展示类是如何与 Angular 互动的。也就是说，它们不会展示组件的类是如何与自己的模板或者其它组件互动的。这样的测试程序需要 Angular 测试工具。

利用 Angular 测试工具 进行测试

Angular 测试工具 包含了 `TestBed` 类和在 `@angular/core/testing` 中一些辅助函数方法。

`TestBed` 创建 Angular 测试模块 - `@NgModule` 类 - 通过配置它，你为想要测试的类创造模块环境。通过 `TestBed` 创建被测试的组件的实例，并使用测试程序来测探这个实例。

在每个 spec 之前，`TestBed` 将自己重设为初始状态。这个初始状态包含了一套默认的、几乎所有情况都需要的测试模块配置，包括可声明类（组件、指令和管道）和提供商（其中一些是伪造的）。

之前 提到的测试垫片初始化测试模块配置到一个模块，这个模块和 `@angular/platform-browser` 中的 `BrowserModule` 类似。

默认配置只是测试应用的 **基础**。在 `TestBed.configureTestingModule` 中定义额外 imports、 declarations、 providers 和 schemas 的对象，构建适合你的应用程序的测试环境。可选的 `override...` 方法可以微调配置的各个方面。

`TestBed` 配置完成以后，用它创建 **被测试的组件** 的实例和测试 fixture，我们可以用它们检查和控制组件周围的环境。

app/banner.component.spec.ts (simplified)

```
beforeEach(() => {

  // refine the test module by declaring the test component
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  });

  // create component and test fixture
  fixture = TestBed.createComponent(BannerComponent);

  // get test component from the fixture
  comp = fixture.componentInstance;
});
```

Angular 测试可以在测试 DOM 中与 HTML 互动，模拟用户行为，让 Angular 执行特定任务（比如变换检测），并在被测试的组件和测试 DOM 中查看这些行为的效果。

app/banner.component.spec.ts (simplified)

```
it('should display original title', () => {

  // trigger change detection to update the view
  fixture.detectChanges();

  // query for the title <h1> by CSS element selector
  de = fixture.debugElement.query(By.css('h1'));

  // confirm the element's content
});
```

```
expect(de.nativeElement.textContent).toContain(comp.title);  
});
```

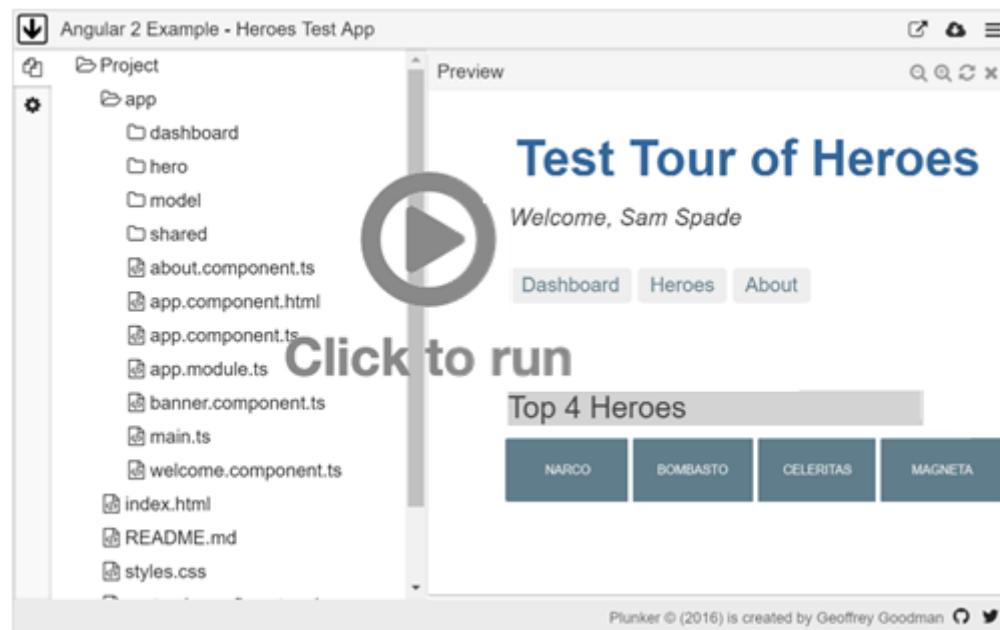
关于 Angular 测试工具的完整回顾将会在 [本章后面](#) 出现。现在开始深入到 Angular 测试，让我们以例子应用的组件开始。

[回到顶部](#)

例子应用和它的测试程序

本章测试了 [简化版本的英雄指南 教程应用程序](#)。

下面的在线例子展示了它是如何工作的，并提供了完整的源代码。



下面的在线例子在浏览器中运行该应用的所有测试程序，使用的是 `Jasmine` 测试运行器，而非 `Karma`。

它包含了本章讨论的测试程序和其它测试程序。本在线例子包含了整个应用和测试代码。给它一些时间来加载。

Angular 2 Example - Testing - app.specs

Project

- app
- testing
- browser-test-shim.js
- index.html
- README.md
- styles.css
- systemjs.config.extras.js
- systemjs.config.js
- tsconfig.json

Preview

Jasmine 2.4.1

74 specs, 0 failures, 2 pending specs finished in 12.633s

Hero

- has name
- has id
- can clone itself

HttpHeroService (mockBackend)

- can instantiate service when inject service
- can instantiate service with "new"
- can provide the mockBackend as XHRBackend

when getHeroes

- should have expected fake heroes (then)
- should have expected fake heroes (Observable.do)
- should be OK returning no heroes
- should treat 404 as an Observable error

TitleCasePipe

- transforms "abc" to "Abc"
- transforms "abc def" to "Abc Def"
- leaves "Abc Def" unchanged
- transforms "ihould" to "Ahould"

Plunker © (2016) is created by Geoffrey Goodman

[回到顶部](#)

测试组件

app/banner.component.ts 中的 BannerComponent 在屏幕顶部显示应用的标题。

app/banner.component.ts

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

@Component({
  selector: 'app-banner',
  template: '<h1>{{title}}</h1>'
})
export class BannerComponent {
  title = 'Test Tour of Heroes';
}
```

BannerComponent 有内联模板和插值表达式绑定。在真实应用场景中，这个组件可能过于简单，不值得测试。但是我们是第一次接触 TestBed，所以用它非常合适。

对应的 app/banner-component.spec.ts 在与组件相同的目录，参见 [这里](#) 的解释。

首先，使用 ES6 导入声明获取在 spec 中引用的符号。

app/banner.component.spec.ts (imports)

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By }                  from '@angular/platform-browser';
import { DebugElement }        from '@angular/core';

import { BannerComponent } from './banner.component';
```

下面是测试程序的配置和 `beforeEach` 的细节：

app/banner.component.spec.ts (setup)

```
let comp: BannerComponent;
let fixture: ComponentFixture<BannerComponent>;
let de: DebugElement;
let el: HTMLElement;

describe('BannerComponent', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ], // declare the test
      component
    });

    fixture = TestBed.createComponent(BannerComponent);

    comp = fixture.componentInstance; // BannerComponent test
    instance

    // query for the title <h1> by css element selector
    de = fixture.debugElement.query(By.css('h1'));
    el = de.nativeElement;

  });
});
```

`TestBed.configureTestingModule` 接受像 `@NgModule` 元数据的对象。这里的对象仅仅声明了要测试的组件 `BannerComponent`。

它没有 `imports`，因为 (a) 它拓展默认测试模块配置，已经有了所有 `BannerComponent` 需要的，(b) `BannerComponent` 不和其它任何组件互动。

createComponent

`TestBed.createComponent` 创建 `BannerComponent` 组件的实例，可以用来测试和返回 `fixture`。

`TestBed.createComponent` 关闭当前 `TestBed` 实例，让它不能再被配置。你不能再调用 `TestBed` 的配置方法，也不能调用 `configureTestingModule` 或者任何 `override...` 方法，否则 `TestBed` 会抛出错误。

不要再调用 `createComponent` 之后试图配置 `TestBed`。

ComponentFixture, DebugElement, and query(By.css)

`createComponent` 方法返回 `ComponentFixture`，用来控制和访问已创建的组件所在的测试环境。这个 `fixture` 提供了对组件实例自身的访问，同时还提供了用来访问组件的 DOM 元素的 `DebugElement` 对象。

`title` 属性被插值到 DOM 的 `<h1>` 标签中。用 CSS 选择器从 `fixture` 的 `DebugElement` 中 `query`<h1>` 元素。

`query` 方法接受 `predicate` 函数，并搜索 `fixture` 的整个 DOM 树，试图寻找 **第一个** 满足 `predicate` 函数的元素。

`queryAll` 方法返回一列数组，包含所有 `DebugElement` 中满足 `predicate` 的元素。

`predicate` 是返回布尔值的函数。`predicate` 查询接受 `DebugElement` 参数，如果元素符合选择条件便返回 `true`。

`By` 类是 Angular 测试工具之一，它生成有用的 predicate。它的 `By.css` 静态方法产生标准 CSS 选择器 predicate，与 JQuery 选择器相同的方式过滤。

最后，这个配置指定 `DebugElement` 中的 `nativeElement` DOM 元素到属性 `el`。测试程序将判断 `el` 是否包含期待的标题文本。

测试程序

再每个测试程序之前，Jasmin 都一次运行 `beforeEach` 函数：

app/banner.component.spec.ts (tests)

```
it('should display original title', () => {
  fixture.detectChanges();
  expect(el.textContent).toContain(comp.title);
});

it('should display a different test title', () => {
  comp.title = 'Test Title';
  fixture.detectChanges();
  expect(el.textContent).toContain('Test Title');
});
```

这些测试程序向 `DebugElement` 获取原生 HTML 元素，来满足自己的期望。

`detectChanges`：在测试中的 Angular 变化检测

每个测试程序都通过调用 `fixture.detectChanges()` 来通知 Angular 执行变化检测。第一个测试程序立刻这么做，触发数据绑定和并将 `title` 属性发送到 DOM 元素中。

第二个测试程序在更改组件的 `title` 属性 **之后** 才调用 `fixture.detectChanges()`。新值出现在 DOM 元素中。

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，自动触发变化检测。

`TestBed.createComponent` 不会 触发变化检测。该工具不会自动将组件的 `title` 属性值推送到数据绑定的元素，下面的测试程序展示了这个事实：

app/banner.component.spec.ts (no detectChanges)

```
it('no title in the DOM until manually call `detectChanges`', () => {
  expect(el.textContent).toEqual('');
});
```

此行为（或者缺乏的行为）是有意的。在 Angular 初始化数据绑定或者调用生命周期钩子之前，它给测试者机会来查看或者改变组件的状态。

自动变化检测

一些测试者偏向让 Angular 测试环境自动运行变化检测。这是可能的，只要配置 `TestBed` 的 `AutoDetect` 提供商即可：

app/banner.component.spec.ts (AutoDetect)

```
fixture = TestBed.configureTestingModule({
  declarations: [ BannerComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
})
```

下面是展示 **自动检测** 如何工作的三个测试程序：

app/banner.component.spec.ts (AutoDetect Tests)

```
it('should display original title', () => {
  // Hooray! No `fixture.detectChanges()` needed
  expect(el.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(el.textContent).toContain(oldTitle);

});

it('should display updated title after detectChanges', () => {
  comp.title = 'Test Title';
  fixture.detectChanges(); // detect changes explicitly
  expect(el.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一个重要的局限性。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 时间作出反应。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.detectChanges()`，来触发新一轮的变化检测周期。

与其怀疑测试工具会不会执行变化检测，本章中的例子 **总是显式** 调用 `detectChanges()`。即使是在不需要的时候，频繁调用 `detectChanges()` 没有任何坏处。

[回到顶部](#)

测试有依赖的组件

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.component.ts

```

import { Component, OnInit } from '@angular/core';
import { UserService } from './model';

```

```

@Component({
  selector: 'app-welcome',
  template: '<h3 class="welcome" ><i>{{welcome}}</i></h3>'
})
export class WelcomeComponent implements OnInit {
  welcome = '-- not initialized yet --';
  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.welcome = this.userService.isLoggedIn ?
      'welcome, ' + this.userService.user.name :
      'Please log in.';
  }
}

```

WelcomeComponent 有与服务进行交互的决策逻辑，这样的逻辑让这个组件值得测试。

下面是 spec 文件的测试模块配置， app/welcome.component.spec.ts :

app/welcome.component.spec.ts

```

 TestBed.configureTestingModule({
  declarations: [ WelcomeComponent ],
  // providers: [ UserService ] // NO! Don't provide the real
  // service!
  // 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 。

提供服务复制品

被测试的组件不一定要注入真正的服务。实际上，服务的复制品（ stubs, fakes, spies 或者 mocks ）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

可能自身有问题。

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` 复制品，会让测试更加容易和安全。

这个测试套件提供了最小化的 `UserService` stub 类，用来满足 `WelcomeComponent` 和它的测试的需求：

```
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，是从被测试的组件的注入器获取。组件注入器是 `fixture` 的 `DebugElement` 的属性。

WelcomeComponent's injector

```
// UserService actually injected into the component  
userService = fixture.debugElement.injector.get(UserService);
```

TestBed.get

你可以通过 `TestBed.get` 方法来从根注入器中获取服务。它更容易被记住，也更加简介。但是只有在 Angular 使用测试的根注入器中的那个服务实例来注入到组件时，它才

有效。幸运的是，在这个测试套件中，唯一的 `UserService` 提供商就是根测试模块，所以像下面这样调用 `TestBed.get` 很安全：

TestBed injector

```
// UserService from the root injector
userService = TestBed.get(UserService);
```

`inject` 辅助函数方法是另外一种从测试的根注入器注入一个或多个服务到测试的方法。

到“[重载组件提供商](#)”查看 `inject` 和 `TestBed.get` 无效，并且必须从组件的注入器获取服务的用例。

总是从注入器获取服务

出人意料的是，请不要引用测试代码里提供给测试模块的 `userServiceStub` 对象。它是行不通的！被注入组件的 `userService` 实例是彻底 不一样的 对象，是提供的 `userServiceStub` 的克隆。

```
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);
});
```

最后的设置和测试程序

这里是使用 `TestBed.get` 的完整 `beforeEach`：

app/welcome.component.spec.ts

```

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)
  de = fixture.debugElement.query(By.css('.welcome'));
  el = de.nativeElement;
});

```

下面是一些测试程序：

app/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 的 `it` 方法的第二个参数（比如 `'expected name'`）是可选附加参数。如果这个期待失败了，Jasmine 在期待失败信息后面显示这个附加参数。在拥有多个期待的 spec 中，它可以帮助澄清发生了什么错误，哪个期待失败了。

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。

[回到顶部](#)

测试有异步服务的组件

许多服务异步返回值。大部分数据服务向远程服务器发起 HTTP 请求，响应必然是异步的。

本例的 `About` 视图显示马克吐温的名言。 `TwainComponent` 组件处理视图，并委派 `TwainService` 向服务器发起请求。两者都在 `app/shared` 目录里，因为作者计划将来在其它页面也显示马克吐温的名言。下面是 `TwainComponent`：

app/shared/twain.component.ts

```
@Component({
  selector: 'twain-quote',
  template: '<p class="twain"><i>{{quote}}</i></p>'
})
export class TwainComponent implements OnInit {
  intervalId: number;
  quote = '...';
  constructor(private twainService: TwainService) { }

  ngOnInit(): void {
    this.twainService.getQuote().then(quote => this.quote = quote);
  }
}
```

`TwainService` 的实现细节现在并不重要。`ngOnInit` 的 `twainService.getQuote` 返回承诺，所以显然它是异步的。

一般来讲，测试程序不应该向远程服务器发请求。它们应该仿真这样的请求。

`app/shared/twain.component.spec.ts` 里的配置是其中一种伪造方法：

app/shared/twain.component.spec.ts (setup)

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ TwainComponent ],
    providers:    [ TwainService ],
  });

  fixture = TestBed.createComponent(TwainComponent);
  comp    = fixture.componentInstance;

  // TwainService actually injected into the component
  twainService = fixture.debugElement.injector.get(TwainService);
```

```
// Setup spy on the `getQuote` method
spy = spyOn(twainService, 'getQuote')
    .and.returnValue(Promise.resolve(testQuote));

// Get the Twain quote element by CSS selector (e.g., by class
name)
de = fixture.debugElement.query(By.css('.twain'));
el = de.nativeElement;
});
```

刺探 (Spy) 真实服务

本配置与 `welcome.component.spec` 配置类似。但是与其伪造服务对象，它注入了真实的服务（参见测试模块的 `providers`），并用 Jasmine 的 `spy` 替换关键的 `getQuote` 方法。

```
spy = spyOn(twainService, 'getQuote')
    .and.returnValue(Promise.resolve(testQuote));
```

这个 Spy 的设计是，所有调用 `getQuote` 的方法都会收到立刻解析的承诺，得到一条预设的名言。Spy 拦截了实际 `getQuote` 方法，所有它不会联系服务。

伪造服务实例和刺探真实服务都是好方法。挑选一种对当前测试套件最简单的方法。你可以随时改变主意。

下面是接下来带有注解的测试程序：

app/shared/twain.component.spec.ts (tests)

```
1. it('should not show quote before OnInit', () => {
2.     expect(el.textContent).toBe('', 'nothing displayed');
```

```

3.     expect(spy.calls.any()).toBe(false, 'getQuote not yet called');
4.   });
5.
6.   it('should still not show quote after component initialized', () =>
{
7.     fixture.detectChanges();
8.     // getQuote service is async => still has not returned with quote
9.     expect(el.textContent).toBe('...', 'no quote yet');
10.    expect(spy.calls.any()).toBe(true, 'getQuote called');
11.  });

12.
13.  it('should show quote after getQuote promise (async)', async(() => {
14.    fixture.detectChanges();

15.
16.    fixture.whenStable().then(() => { // wait for async getQuote
17.      fixture.detectChanges();           // update view with quote
18.      expect(el.textContent).toBe(testQuote);
19.    });
20.  }));
21.
22.  it('should show quote after getQuote promise (fakeAsync)', fakeAsync(() => {
23.    fixture.detectChanges();
24.    tick();                      // wait for async getQuote
25.    fixture.detectChanges(); // update view with quote
26.    expect(el.textContent).toBe(testQuote);
27.  }));

```

同步测试程序

前两个测试程序是同步的。在 Spy 的帮助下，它们验证了在 Angular 调用 `ngOnInit` 期间发生的第一次变化检测后，`getQuote` 被调用了。

两者都不能证明被显示的值是服务提供的。虽然 spy 返回了解析的承诺，名言本身还没有到来。

这个测试程序必须等待 JavaScript 引擎一整个回合，返回值才会有效。该测试程序必须要变成 **异步的**。

it 里的 async 函数方法

注意第三个测试程序的 `async` 方法。

app/shared/twain.component.spec.ts (async test)

```
it('should show quote after getQuote promise (async)', async(() => {
  fixture.detectChanges();

  fixture.whenStable().then(() => { // wait for async getQuote
    fixture.detectChanges();           // update view with quote
    expect(el.textContent).toBe(testQuote);
  });
}));
```

`async` 函数是 **Angular TestBed** 的一部分。通过将测试代码放到特殊的 **异步测试区域** 来运行，`async` 函数简化了异步测试程序的代码。

`async` 函数 **接受** 无参数的函数方法，**返回** 无参数的函数方法，变成 Jasmine 的 `it` 函数的参数。

`async` 函数的参数看起来和普通的 `it` 参数主体一样。没有任何地方显示异步特征。比如，它不返回承诺，并且没有 `done` 方法可调用，因为它是标准的 Jasmine 异步测试程序。

在测试程序（比如 `fixture.whenStable`）里面调用函数时，还是体现它们的异步行为。

`fakeAsync` 可选方法，[正如下面解释的](#)，进一步移除了异步行为，提供了更加直观的代码经验。

whenStable

测试程序必须等待 `getQuote` 在 JavaScript 引擎的下一回合中被解析。

本测试对 `twainService.getQuote` 返回的承诺没有直接的访问，因为它被埋没在 `TwainComponent.ngOnInit` 里，所以对于只测试组件 API 表面的测试来说，它是无法被访问的。

幸运的是，**异步测试区域** 可以访问 `getQuote` 承诺，因为它拦截所有调用 **异步** 方法所发出的承诺，不管它们在哪儿。

`ComponentFixture.whenStable` 方法返回它自己的承诺，它在 `getQuote` 承诺完成时被解析。实际上，“stable”的意思是当**所有待处理异步行为** 完成时的状态，在“stable”后 `whenStable` 承诺被解析。

然后测试程序继续运行，并开始另一轮的变化检测（`fixture.detectChanges`），通知 Angular 使用名言来更新 DOM。`getQuote` 辅助方法提取出显示元素文本，然后 `expect` 语句确认这个文本与预备的名言相符。

fakeAsync 函数方法

第四个测试程序用不同的方法验证同样的组件行为。

app/shared/twain.component.spec.ts (fakeAsync test)

```
it('should show quote after getQuote promise (fakeAsync)',  
fakeAsync(() => {  
  fixture.detectChanges();  
  tick(); // wait for async getQuote  
  fixture.detectChanges(); // update view with quote  
  expect(el.textContent).toBe(testQuote);  
}));
```

注意，在 `it` 的参数中，`async` 被 `fakeAsync` 替换。`fakeAsync` 是另一种 Angular 测试工具。

和 `async` 一样，它也 **接受** 无参数函数并 **返回** 一个函数，变成 Jasmine 的 `it` 函数的参数。

`fakeAsync` 函数通过在特殊的 **fakeAsync 测试区域** 运行测试程序，让测试代码更加简单直观。

对于 `async` 来说，`fakeAsync` 最重要的好处是测试程序看起来像同步的。里面没有任何承诺。没有 `then(...)` 链来打断控制流。

但是 `fakeAsync` 有局限性。比如，你不能从 `fakeAsync` 发起 XHR 请求。

tick 函数

`tick` 函数是 Angular 测试工具之一，是 `fakeAsync` 的同伴。它只能在 `fakeAsync` 的主体中被调用。

调用 `tick()` 模拟时间的推移，直到全部待处理的异步任务都已完成，在这个测试案例中，包含 `getQuote` 承诺的解析。

它不返回任何结果。没有任何承诺需要等待。直接执行与之前在 `whenStable.then()` 的回调函数里相同的代码。

虽然这个例子非常简单，但是它已经比第三个测试程序更易阅读。为了更充分的体会 `fakeAsync` 的好处，试想一下一连串的异步操作，被一长串的承诺回调链在一起。

jasmine.done

虽然 `fakeAsync`，甚至 `async` 函数大大的简化了异步测试，你仍然可以回退到传统的 Jasmine 异步测试技术上。

你仍然可以将接受 `done` 回调的函数传给 `it`。但是，你必须链接承诺、处理错误，并在适当的时候调用 `done`。

下面是上面两个测试程序的 `done` 版本：

app/shared/twain.component.spec.ts (done test)

```
it('should show quote after getQuote promise (done)', done => {
  fixture.detectChanges();
```

```
// get the spy promise and wait for it to resolve
spy.calls.mostRecent().returnValue.then(() => {
  fixture.detectChanges(); // update view with quote
  expect(el.textContent).toBe(testQuote);
  done();
});
});
```

虽然我们对 `TwainComponent` 里的 `getQuote` 承诺没有直接访问，但是 Spy 有，所以才可能等待 `getQuote` 完成。

虽然不推荐使用 `jasmine.done` 技术，但是它可能在 `async` 和 `fakeAsync` 都无法容纳一种特定的异步行为时，变得很有必要。这很少见，但是有可能发生。

[回到顶部](#)

测试有外部模板的组件

`TestBed.createComponent` 是一种同步的方法。它假设所有它需要的资源已经全部在内存。

到目前为止还是这样的。每个被测试程序的组件的 `@Component` 元数据都有 `template` 属性，制定 **内联模板**。没有组件有 `styleUrls` 属性。所有编译它们所需要的资源，在测试时都已经在内存里。

`DashboardComponent` 不一样。它有外部的模板和外部 CSS 文件，是在 `templateUrl` 和 `styleUrls` 属性分别配置的。

app/dashboard/dashboard-hero.component.ts (component)

```
@Component({
  moduleId: module.id,
  selector:    'dashboard-hero',
  templateUrl: 'dashboard-hero.component.html',
  styleUrls: [ 'dashboard-hero.component.css' ]
```

```

})
export class DashboardHeroComponent {
  @Input() hero: Hero;
  @Output() selected = new EventEmitter<Hero>();
  click() { this.selected.next(this.hero); }
}

```

编译器必须预先从文件系统读取这些文件，它才能创建组件实例。

`TestBed.compileComponents` 方法异步编译所有当前测试模块配置的组件。完成编译后，外部模板和 CSS 文件就已经被 **内联** 进来，然后 `TestBed.createComponent` 就可以异步的完成它的任务了。

WebPack 开发者不需要调用 `compileComponents`，因为在运行测试程序时，内联模板和 CSS 文件是自动编译流程的一部分。

`app/dashboard/dashboard-hero.component.spec.ts` 展示了预编译的过程：

app/dashboard/dashboard-hero.component.spec.ts (compileComponents)

```

// async beforeEach
beforeEach(async() => {
  TestBed.configureTestingModule({
    declarations: [ DashboardHeroComponent ],
  })
  .compileComponents(); // compile template and css
});

```

beforeEach 里的 async 函数

注意 `beforeEach` 里面对 `async` 的调用，因为异步方法 `TestBed.compileComponents` 而变得必要。`async` 函数将测试代码安排到特殊的 **异步测试区域** 来运行，该区域隐藏了

异步执行的细节，就像它被传递给 `[it 测试] (#async)` 一样。

compileComponents

在本例中，`TestBed.compileComponents` 编译了组件，那就是 `DashboardComponent`。它是这个测试模块唯一的声明组件。

本章后面的测试程序有更多声明组件，它们中间的一些导入应用模块，这些模块有更多的声明组件。一部分或者全部组件可能有外部模板和 CSS 文件。

`TestBed.compileComponents` 一次性异步编译所有组件。

`compileComponents` 方法返回承诺，可以用来在它完成时候，执行更多额外任务。

compileComponents 关闭配置

调用 `compileComponents` 关闭当前的 `TestBed` 实例，使它不能再被配置。你不能再调用任何 `TestBed` 配置方法、`configureTestingModule` 或者任何 `override...` 方法，否则 `TestBed` 将会抛出错误。

不要再调用 `compileComponents` 之后再配置 `TestBed`。在调用 `TestBed.createComponent` 来初始化被测试的组件之前，把 `compileComponents` 的调用放到最后一步。

`DashboardHeroComponent` spec 中，异步的 `beforeEach` 下面紧接着 **同步** 的 `beforeEach`，用来完成设置步骤和运行测试程序 ... 下面小节做了详细的解释。

测试带有导入 inputs 和导出 outputs 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。

`DashboardComponent` 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 `DashboardComponent` 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，设置每个组件的 `hero` input 属性到迭代的值，并监听组件的 `selected` 事件。

下面是组件的定义：

app/dashboard/dashboard-hero.component.ts (component)

```
@Component({
  moduleId: module.id,
  selector:    'dashboard-hero',
  templateUrl: 'dashboard-hero.component.html',
  styleUrls: [ 'dashboard-hero.component.css' ]
})
export class DashboardHeroComponent {
  @Input() hero: Hero;
  @Output() selected = new EventEmitter<Hero>();
  click() { this.selected.next(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。 有三种候选测试方案：

1. 把它当作被 `DashboardComponent` 使用的组件来测试
2. 把它当作独立的组件来测试
3. 把它当作被 `DashboardComponent` 的替代组件使用的组件来测试

简单看看 `DashboardComponent` 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) {  
}
```

`DashboardComponent` 依赖 Angular 路由器和 `HeroService` 服务。你必须使用测试复制品替换它们两个，似乎过于复杂了。路由器尤其具有挑战性。

下面 覆盖了如何测试带有路由器的组件。

当前的任务是测试 `DashboardHeroComponent` 组件，而非 `DashboardComponent`，所以无需做不必要的努力。让我们尝试第二和第三种方案。

独立测试 DashboardHeroComponent

下面是 spec 文件的设置。

app/dashboard/dashboard-hero.component.spec.ts (setup)

```
// async beforeEach  
beforeEach( async() => {  
  TestBed.configureTestingModule({  
    declarations: [ DashboardHeroComponent ],  
  })
```

```

.compileComponents(); // compile template and css
});

// synchronous beforeEach
beforeEach(() => {
  fixture = TestBed.createComponent(DashboardHeroComponent);
  comp = fixture.componentInstance;
  heroEl = fixture.debugElement.query(By.css('.hero')); // find
hero element

// pretend that it was wired to something that supplied a hero
expectedHero = new Hero(42, 'Test Name');
comp.hero = expectedHero;
fixture.detectChanges(); // trigger initial data binding
});

```

异步 `beforeEach` 已经在 [上面](#) 讨论过。在使用 `compileComponents` 异步编译完组件后，接下来的设置执行另一个 **同步** 的 `beforeEach`，使用 [之前](#) 解释过的基本知识。

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 `DashboardComponent` 在它的迭代器中通过属性绑定的赋值方式。

紧接着第一个测试程序：

app/dashboard/dashboard-hero.component.spec.ts (name test)

```

it('should display hero name', () => {
  const expectedPipedName = expectedHero.name.toUpperCase();

  expect(heroEl.nativeElement.textContent).toContain(expectedPipedName);

});

```

它验证了英雄名字通过绑定被传递到模板了。这里有个额外步骤。模板将英雄名字传给 Angular 的 `UpperCasePipe`，所以测试程序必须使用大写名字来匹配元素的值：

```
<div (click)="click()" class="hero">
  {{hero.name | uppercase}}
</div>
```

这个小测试演示了 Angular 测试是如何验证组件的视图表现的——这是 **孤立的单元测试** 无法实现的——它成本低，而且无需依靠更慢、更复杂的端对端测试。

第二个测试程序验证点击行为。点击英雄应该出发 `selected` 事件，可供宿主组件 (`DashboardComponent`) 监听：

app/dashboard/dashboard-hero.component.spec.ts (click test)

```
it('should raise selected event when clicked', () => {
  let selectedHero: Hero;
  comp.selected.subscribe(hero: Hero) => selectedHero = hero;

  heroEl.triggerEventHandler('click', null);
  expect(selectedHero).toBe(expectedHero);
});
```

这个组件公开 `EventEmitter` 属性。测试程序像宿主组件那样来描述它。

`heroEl` 是个 `DebugElement`，它代表了英雄所在的 `<div>`。测试程序用 "click" 事件名字来调用 `triggerEventHandler`。调用 `DashboardHeroComponent.click()` 时，"click" 事件绑定作出响应。

如果组件想期待的那样工作，`click()` 通知组件的 `selected` 属性发出 `hero` 对象，测试程序通过订阅 `selected` 事件而检测到这个值，所以测试应该成功。

triggerEventHandler

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。

本例中，测试程序用 null 事件对象触发 "click" 事件。

```
heroEl.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。

其它处理器将会更加严格。比如， RouterLink 指令期待事件对象，并且该对象具有 button 属性，代表了已被按下的鼠标按钮。如果该事件对象不具备上面的条件，指令便会抛出错误。

点击按钮、链接或者任意 HTML 元素是很常见的测试任务。把 click 触发过程封装到辅助方法中可以简化这个任务，比如下面的 click 辅助方法：

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() 不是 ANGULAR 测试工具

`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。

下面是使用了 `click` 辅助函数重新编写的上一个测试程序：

app/dashboard/dashboard-hero.component.spec.ts (click test revised)

```
it('should raise selected event when clicked', () => {
  let selectedHero: Hero;
  comp.selected.subscribe(hero: Hero) => selectedHero = hero;

  click(heroEl);    // triggerEventHandler helper
  expect(selectedHero).toBe(expectedHero);
});
```

在测试宿主组件中测试组件

在前面的方法中，测试本身扮演了宿主组件 `DashboardComponent` 的角色。一种挥之不去的疑虑仍然存在：当正常数据绑定到宿主组件时，`DashboardHeroComponent` 还会正常工作吗？

使用实际的 `DashboardComponent` 宿主来测试是可行的，但是这么做似乎不合算。像下面这样使用 **测试宿主组件** 来模拟 `DashboardComponent` 显得更加容易：

app/dashboard/dashboard-hero.component.spec.ts (test host)

```
@Component({
  template: `
    <dashboard-hero [hero]="hero" (selected)="onSelected($event)">
    </dashboard-hero>`
})
class TestHostComponent {
  hero = new Hero(42, 'Test Name');
```

```

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

```

测试宿主组件和 `DashboardComponent` 一样绑定 `DashboardHeroComponent`，但是不用理会 `Router`、`HeroService` 服务，甚至 `*ngFor` 循环。

测试宿主将组件的 `hero` 导入属性设置为它的模拟英雄。它将组件的 `selected` 事件绑定到它的 `onSelected` 处理器，使用 `selectedHero` 属性来记录发送来的英雄。然后测试检查这个属性来验证 `DashboardHeroComponent.selected` 事件确实发送了正确的英雄。

配置使用测试宿主的测试程序与配置孤立测试相似：

app/dashboard/dashboard-hero.component.spec.ts (test host setup)

```

beforeEach( async() => {
  TestBed.configureTestingModule({
    declarations: [ DashboardHeroComponent, TestHostComponent ], // declare both
  }).compileComponents();
});

beforeEach(() => {
  // create TestHostComponent instead of DashboardHeroComponent
  fixture = TestBed.createComponent(TestHostComponent);
  testHost = fixture.componentInstance;
  heroEl = fixture.debugElement.query(By.css('.hero')); // find hero
  fixture.detectChanges(); // trigger initial data binding
});

```

这个测试模块配置展示了两个非常重要的区别：

1. 它同时声明了 `DashboardComponent` 和 `TestHostComponent`。
2. 它创建了 `TestHostComponent`，而非 `DashboardHeroComponent`。

`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。

测试它们自己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.nativeElement.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。

[回到顶部](#)

测试带路由器的组件

测试实际的 `DashboardComponent` 似乎令人生畏，因为它注入了 `Router`。

```
app/dashboard/dashboard.component.ts (constructor)
```

```
constructor(
  private router: Router,
  private heroService: HeroService) {
}
```

它同时还注入了 `HeroService`，但是我们已经知道如何 [伪造](#) 它。`Router` 的 API 非常复杂，并且它缠绕了其它服务和许多应用的先决条件。

幸运的是，`DashboardComponent` 没有使用 `Router` 做很多事情。

app/dashboard/dashboard.component.ts (goToDetail)

```
gotoDetail(hero: Hero) {
  let url = `/heroes/${hero.id}`;
  this.router.navigateByUrl(url);
}
```

通常都是这样的。原则上，你测试的是组件，不是路由器，应该只关心在指定的条件下，组件是否导航到正确的地址。用模拟类来替换路由器是一种简单的方案。下面的代码应该可以：

app/dashboard/dashboard.component.spec.ts (Router Stub)

```
class RouterStub {
  navigateByUrl(url: string) { return url; }
}
```

现在我们来利用 `Router` 和 `HeroService` 的测试 stub 类来配置测试模块，并为接下来的测试创建 `DashboardComponent` 的测试实例。

app/dashboard/dashboard.component.spec.ts (compile and create)

```
beforeEach( async() => {
  TestBed.configureTestingModule({
    providers: [

```

```

    { provide: HeroService, useClass: FakeHeroService },
    { provide: Router,      useClass: RouterStub }
  ]
})
.compileComponents().then(() => {
  fixture = TestBed.createComponent(DashboardComponent);
  comp = fixture.componentInstance;
});

```

下面的测试程序点击显示的英雄，并利用 spy 来确认 `Router.navigateByUrl` 被调用了，而且传进的 url 是所期待的值。

app/dashboard/dashboard.component.spec.ts (navigate test)

```

it('should tell ROUTER to navigate when hero clicked',
  inject([Router], (router: Router) => { // ...

  const spy = spyOn(router, 'navigateByUrl');

  heroClick(); // trigger click on first inner <div class="hero">

  // args passed to router.navigateByUrl()
  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');
}));
```

inject 函数

注意第二个 `it` 参数里面的 `inject` 函数。

```

it('should tell ROUTER to navigate when hero clicked',
  inject([Router], (router: Router) => { // ...
}));
```

`inject` 函数是 Angular 测试工具之一。 它注入服务到测试函数，以供修改、监视和操纵。

`inject` 函数有两个参数：

1. 一列数组，包含了 Angular 依赖注入令牌
2. 一个测试函数，它的参数与注入令牌数组里的每个项目严格的一一对应。

使用 TESTBED 注入器来注入

`inject` 函数使用当前 `TestBed` 注入器，并且只返回这个级别提供的服务。 它不会返回组件提供商提供的服务。

这个例子通过当前的 `TestBed` 注入器来注入 `Router`。 对这个测试程序来说，这是没问题的，因为 `Router` 是（也必须是）由应用的根注入器来提供。

如果你需要组件自己的注入器提供的服务，调用 `fixture.debugElement.injector.get`：

Component's injector

```
// UserService actually injected into the component
userService = fixture.debugElement.injector.get(UserService);
```

使用组件自己的注入器来获取实际注入到组件的服务。

`inject` 函数关闭当前 `TestBed` 实例，使它无法再被配置。 你不能再调用任何 `TestBed` 配置方法、`configureTestingModule` 或者任何 `override...` 方法，否则 `TestBed` 将抛出错误。

不要再调用 `inject` 以后再试图配置 `TestBed`。

[回到顶部](#)

测试带有路由和路由参数的组件

点击 **Dashboard** 英雄触发导航到 `heros/:id`，其中 `:id` 是路由参数，它的值是进行编辑的英雄的 `id`。这个 URL 匹配到 `HeroDetailComponent` 的路由。

路由器将 `:id` 令牌的值推送到 `ActivatedRoute.params` 可观察属性里，Angular 注入 `ActivatedRoute` 到 `HeroDetailComponent` 中，然后组件提取 `id`，这样它就可以通过 `HeroDetailService` 获取相应的英雄。下面是 `HeroDetailComponent` 的构造函数：

app/hero/hero-detail.component.ts (constructor)

```
constructor(  
  private heroDetailsService: HeroDetailService,  
  private route: ActivatedRoute,  
  private router: Router) {  
}
```

`HeroDetailComponent` 在它的 `ngOnInit` 方法中监听 `ActivatedRoute.params` 的变化。

app/hero/hero-detail.component.ts (ngOnInit)

```
ngOnInit(): void {  
  // get hero when `id` param changes  
  this.route.params.pluck<string>('id')  
    .forEach(id => this.getHero(id))  
    .catch(() => this.hero = new Hero()); // no id; should edit new  
  hero  
}
```

`route.params` 之后的表达式链接了 可观察 操作符，它从 `params` 中提取 `id`，然后链接 `forEach` 操作符来订阅 `id` 变化事件。每次 `id` 变化时，用户被

导航到不同的英雄。

`forEach` 将新的 `id` 值传递到组件的 `getHero` 方法（这里没有列出来），它获取英雄并将它赋值到组件的 `hero` 属性。如果 `id` 参数无效，`pluck` 操作符就会失败，`catch` 将失败当作创建新英雄来处理。

路由器 章更详尽的讲述了 `ActivatedRoute.params`。

通过操纵被注入到组件构造函数的 `ActivatedRoute` 服务，测试程序可以探索 `HeroDetailComponent` 是如何对不同的 `id` 参数值作出响应的。

现在，你已经知道如何模拟 `Router` 和数据服务。模拟 `ActivatedRoute` 遵循类似的模式，但是有个额外枝节：`ActivatedRoute.params` 是 **可观察对象**。

可观察对象 的测试复制品

`hero-detail.component.spec.ts` 依赖 `ActivatedRouteStub` 来为每个测试程序设置 `ActivatedRoute.params` 值。它是跨应用、可复用的 **测试辅助类**。我们建议将这样的辅助类放到 `app` 目录下的名为 `testing` 的目录。本例把 `ActivatedRouteStub` 放到 `testing/router-stubs.ts`：

testing/router-stubs.ts (ActivatedRouteStub)

```
import { BehaviorSubject } from 'rxjs/BehaviorSubject';

@Injectable()
export class ActivatedRouteStub {

  // ActivatedRoute.params is observable
  private subject = new BehaviorSubject(this.testParams);
  params = this.subject.asObservable();

  // Test parameters
  private _testParams: {};
  get testParams() { return this._testParams; }
  set testParams(params: {}) {
    this._testParams = params;
  }
}
```

```

    this.subject.next(params);
}

// ActivatedRoute.snapshot.params
get snapshot() {
  return { params: this.testParams };
}
}

```

这个 stub 类有下列值得注意的特征：

- 这个 stub 类只实现 `ActivatedRoute` 的两个功能：`params` 和 `snapshot.params`。
- **BehaviorSubject** 驱使这个 stub 类的 `params` 可观察对象，并为每个 `params` 的订阅者返回同样的值，直到它接受到新值。
- `HeroDetailComponent` 链接它的表达式到这个 stub 类的 `params` 可观察对象，该对象现在被测试者的控制之下。
- 设置 `testParams` 属性导致 `subject` 将指定的值推送进 `params`。它触发上面描述过的 `HeroDetailComponent` 的 `params` 订阅，和导航的方式一样。
- 设置 `testParams` 属性同时更新这个 stub 类内部值，用于 `snapshot` 属性的返回。

snapshot 是组件使用路由参数的另一种流行的方法。

本章的路由器 stub 类是为了给你灵感。创建你自己的 stub 类，以适合你的测试需求。

可观察对象 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.testParams = { id: expectedHero.id };
    createComponent();
  }));

  it('should display that hero\'s name', () => {
    expect(page.nameDisplay.textContent).toBe(expectedHero.name);
  });
});

```

下一节 将解释 `createComponent` 方法和 `page` 对象，现在暂时跟着自己的直觉走。

当无法找到 `id` 时，组件应该重新导航到 `HeroListComponent`。该测试套件配置与 上面描述 的 `RouterStub` 一样，它在不实际导航的情况下刺探路由器。该测试程序提供了“坏”的 `id`，期望组件尝试导航。

app/hero/hero-detail.component.spec.ts (bad id)

```

describe('when navigate to non-existent hero id', () => {
  beforeEach(async(() => {
    activatedRoute.testParams = { id: 99999 };
    createComponent();
  }));

  it('should try to navigate back to hero list', () => {
    expect(page.gotoSpy.calls.any()).toBe(true, 'comp.gotoList called');
    expect(page.navSpy.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(comp.hero.id).toBe(0);  
  });  
  
  it('should display empty hero name', () => {  
    expect(page.nameDisplay.textContent).toBe('');  
  });  
});
```

到 [在线例子](#) 查看和下载 **所有** 本章应用程序测试代码。

使用 page 对象来简化配置

`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:

Save Cancel

但是它已经有很多模板复杂性。

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>
```

要彻底测试该组件，测试程序需要：

- 在 `*ngIf` 允许元素进入 DOM 之前，等待 `hero` 的到来
- 标题名字 `span` 和名字输入框元素的引用，用来检查它们的值
- 两个按钮的引用，用来点击
- 刺探（`spy`）服务和组件的方法

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。

通过简化组件属性的访问和封装设置属性的逻辑，`Page` 类可以轻松解决这个令人疯狂的难题。下面是为 `hero-detail.component.spec.ts` 准备的 `page` 类：

app/hero/hero-detail.component.spec.ts (Page)

```
class Page {
  gotoSpy:      jasmine.Spy;
  navSpy:       jasmine.Spy;
  saveSpy:      jasmine.Spy;

  saveBtn:      DebugElement;
  cancelBtn:    DebugElement;
  nameDisplay:  HTMLElement;
  nameInput:    HTMLInputElement;

  constructor() {
    // Use component's injector to see the services it injected.
    const compInjector = fixture.debugElement.injector;
    const hds          = compInjector.get(HeroDetailsService);
    const router       = compInjector.get(Router);

    this.gotoSpy      = spyOn(comp, 'gotoList').and.callThrough();
    this.navSpy       = spyOn(router, 'navigate');
    this.saveSpy      = spyOn(hds, 'saveHero').and.callThrough();
  }

  /** Add page elements after hero arrives */
  addPageElements() {
    if (comp.hero) {
      // have a hero so these elements are now in the DOM
      const buttons   =
        fixture.debugElement.queryAll(By.css('button'));
      this.saveBtn    = buttons[0];
      this.cancelBtn  = buttons[1];
      this.nameDisplay =
        fixture.debugElement.query(By.css('span')).nativeElement;
      this.nameInput   =
        fixture.debugElement.query(By.css('input')).nativeElement;
    }
  }
}
```

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);
  comp = fixture.componentInstance;
  page = new Page();

  // 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();
    page.addPageElements();
  });
}
```

上一节的 可观察对象测试 展示了 `createComponent` 和 `page` 如何让测试程序简短和即时。没有任何干扰：无需等待承诺的解析，也没有搜索 DOM 元素值进行比较。这里是一些更多的 `HeroDetailComponent` 测试程序，进一步的展示了这一点。

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.navSpy.calls.any()).toBe(true, 'router.navigate called');
});
```

```
it('should save when click save but not navigate immediately', () => {
  click(page.saveBtn);
  expect(page.saveSpy.calls.any()).toBe(true, 'HeroDetailsService.save called');
  expect(page.navSpy.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.navSpy.calls.any()).toBe(true, 'router.navigate called');
}));

it('should convert hero name to Title Case', fakeAsync(() => {
  const inputName = 'quick BROWN fox';
  const titleCaseName = 'Quick Brown Fox';

  // simulate user entering new name into the input box
  page.nameInput.value = inputName;

  // dispatch a DOM event so that Angular learns of input value change.
  page.nameInput.dispatchEvent(newEvent('input'));

  // tell Angular to update the output span through the title pipe
  fixture.detectChanges();

  expect(page.nameDisplay.textContent).toBe(titleCaseName);
}));
```

[回到顶部](#)

模块导入 imports 的配置

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 (config)

// async beforeEach
beforeEach( async() => {
  TestBed.configureTestingModule({
    declarations: [ DashboardHeroComponent ],
  })
  .compileComponents(); // compile template and css
});
```

`DashboardComponent` 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() => {
  TestBed.configureTestingModule({
    imports:      [ FormsModule ],
    declarations: [ HeroDetailComponent, TitleCasePipe ],
    providers:   [
      ...
    ]
});
```

```

    { provide: ActivatedRoute, useValue: activatedRoute },
    { provide: HeroService,      useClass: FakeHeroService },
    { provide: Router,          useClass: RouterStub},
  ]
})
.compileComponents();
});

```

因为许多应用组件需要 `FormsModule` 和 `TitleCasePipe`，所以开发者创建了 `SharedModule` 来合并它们和一些频繁需要的部件。测试配置也可以使用 `SharedModule`，请看下面另一种配置：

app/hero/hero-detail.component.spec.ts (SharedModule setup)

```

beforeEach( async() => {
  TestBed.configureTestingModule({
    imports:      [ SharedModule ],
    declarations: [ HeroDetailComponent ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService,      useClass: FakeHeroService },
      { provide: Router,          useClass: RouterStub},
    ]
  })
  .compileComponents();
});

```

它的导入声明少一些（未显示），稍微干净一些，小一些。

导入特征模块

`HeroDetailComponent` 是 `HeroModule` 特征模块的一部分，它组合了更多互相依赖的部件，包括 `SharedModule`。试试下面这个导入 `HeroModule` 的测试配置：

app/hero/hero-detail.component.spec.ts (HeroModule setup)

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    imports: [ HeroModule ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService, useClass: FakeHeroService },
      { provide: Router, useClass: RouterStub },
    ]
  })
  .compileComponents();
}));
```

这样特别清爽。只有 `providers` 里面的测试复制品被保留。连 `HeroDetailComponent` 声明都消失了。

事实上，如果里试图声明它，Angular 会抛出错误，因为 `HeroDetailComponent` 已经在 `HeroModule` 和测试模块的 `DynamicTestModule` 中声明。

导入组件的特征模块通常是最简单的配置测试的方法，尤其是当特征模块很小而且几乎自包含时 ... 特征模块应该是自包含的。

[回到顶部](#)

重载组件提供商

`HeroDetailComponent` 提供自己的 `HeroDetailService` 服务。

[app/hero/hero-detail.component.ts \(prototype\)](#)

```

@Component({
  moduleId: module.id,
  selector: 'app-hero-detail',
  templateUrl: 'hero-detail.component.html',
  styleUrls: ['hero-detail.component.css'],
  providers: [ HeroDetailsService ]
})
export class HeroDetailComponent implements OnInit {
  constructor(
    private heroDetailsService: HeroDetailsService,
    private route: ActivatedRoute,
    private router: Router) {
  }
}

```

在 `TestBed.configureTestingModule` 的 `providers` 中 stub 伪造组件的 `HeroDetailsService` 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 创建组件时，该组件有自己的注入器，它是 fixture 注入器的子级。Angular 使用这个子级注入器来注册组件的提供商（也就是 `HeroDetailsService`）。测试程序无法从 fixture 的注入器获取这个子级注入器。而且 `TestBed.configureTestingModule` 也无法配置它们。

Angular 始终都在创建真实 `HeroDetailsService` 的实例。

如果 `HeroDetailsService` 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，`HeroDetailsService` 将远程数据访问的责任交给了注入进来的 `HeroService`。

app/heroes/heroes.service.ts (prototype)

```

@Injectable()
export class HeroDetailsService {
  constructor(private heroService: HeroService) { }
}

```

```
/* . . . */  
}
```

之前的测试配置 将真实的 `HeroService` 替换为 `FakeHeroService` , 拦截了服务起请求，伪造了它们的响应。

如果我们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.configureTestingModule` 方法可以将组件的 `providers` 替换为容易管理的 **测试复制品**，参见下面的设置变化：

app/hero/hero-detail.component.spec.ts (Override setup)

```
beforeEach( async() => {  
    TestBed.configureTestingModule({  
        imports: [ HeroModule ],  
        providers: [  
            { provide: ActivatedRoute, useValue: activatedRoute },  
            { provide: Router, useClass: RouterStub },  
        ]  
    })  
  
    // override component's own provider  
    .overrideComponent(HeroDetailComponent, {  
        set: {  
            providers: [  
                { provide: HeroDetailsService, useClass:  
                    StubHeroDetailsService }  
            ]  
        }  
    })  
  
    .compileComponents();  
});
```

注意，`TestBed.configureTestingModule` 不再提供（伪造）`HeroService`，因为已经没有必要了。

overrideComponent 方法

注意这个 `overrideComponent` 方法。

`app/hero/hero-detail.component.spec.ts (overrideComponent)`

```
.overrideComponent(HeroDetailComponent, {
  set: {
    providers: [
      { provide: HeroDetailsService, useClass: StubHeroDetailsService }
    ]
  }
})
```

它接受两个参数：要重载的组件类型（`HeroDetailComponent`）和用于重载的元数据对象。[重载元数据对象](#) 是泛型类，就像这样：

```
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[];
...
```

StubHeroDetailsService

本例将组件的 providers 彻底替换为包含 StubHeroDetailsService 的数组。

StubHeroDetailsService 非常简单。它不需要 HeroService (假不假不重要)

app/hero/hero-detail.component.spec.ts (StubHeroDetailsService)

```
class StubHeroDetailsService {
  testHero = new Hero(42, 'Test Hero');

  getHero(id: number | string): Promise<Hero> {
    return Promise.resolve(true).then(() => Object.assign({}, this.testHero));
  }

  saveHero(hero: Hero): Promise<Hero> {
    return Promise.resolve(true).then(() =>
      Object.assign(this.testHero, hero));
  }
}
```

重载的测试程序

现在，测试程序可以通过操控 stub 的 testHero ，直接控制组件的英雄。

app/hero/hero-detail.component.spec.ts (override tests)

```
let hds: StubHeroDetailsService;

beforeEach(async() => {
  createComponent();
  // get the component's injected StubHeroDetailsService
  hds = fixture.debugElement.injector.get(HeroDetailsService);
});

it('should display stub hero\'s name', () => {
  expect(page.nameDisplay.textContent).toBe(hds.testHero.name);
});

it('should save stub hero change', fakeAsync(() => {
```

```
const origName = hds.testHero.name;
const newName = 'New Name';

page.nameInput.value = newName;
page.nameInput.dispatchEvent(newEvent('input')); // tell Angular

expect(comp.hero.name).toBe(newName, 'component hero has new
name');
expect(hds.testHero.name).toBe(origName, 'service hero unchanged
before save');

click(page.saveBtn);
tick(); // wait for async save to complete
expect(hds.testHero.name).toBe(newName, 'service hero has new name
after save');
expect(page.navSpy.calls.any()).toBe(true, 'router.navigate
called');
});
```

更多重载

`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。

[回到顶部](#)

测试带有 RouterOutlet 的组件

`AppComponent` 在 `<router-outlet>` 中显示导航组件。它还显示了导航条，包含了链接和它们的 `RouterLink` 指令。

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>

```

组件的类没有做任何事。

app/app.component.ts

```

import { Component } from '@angular/core';
@Component({
  moduleId: module.id,
  selector: 'my-app',
  templateUrl: 'app.component.html'
})
export class AppComponent { }

```

在不涉及路由的情况下，单元测试可以确认链接的设置是否正确。参见[下面的内容](#)，了解为什么值得这么做。

stub 伪造不需要的组件

该测试配置应该看起来很眼熟：

app/app.component.spec.ts (Stub Setup)

```

beforeEach( async() => {
  TestBed.configureTestingModule({
    declarations: [
      AppComponent,
      BannerComponent, welcomeStubComponent,

```

```

        RouterLinkStubDirective, RouterOutletStubComponent
    ]
})

.compileComponents()
.then(() => {
  fixture = TestBed.createComponent(AppComponent);
  comp    = fixture.componentInstance;
});
));
})
);

```

`AppComponent` 是被声明的测试对象。 使用一个真实的组件 (`BannerComponent`) 和几个 stub , 该配置扩展了默认测试模块。

- 原样使用 `BannerComponent` 非常简单而且无害。
- 真实的 `WelcomeComponent` 有被注入的服务。 `WelcomeStubComponent` 是无服务的替代品。
- 真实的 `RouterOutlet` 很复杂而且容易出错。 `testing/router-stubs.ts` 里的 `RouterOutletStubComponent` 是安全的替代品。

组件 stub 替代品很关键。 没有它们 , Angular 编译器无法识别 `<app>Welcome` 和 `<router-outlet>` 标签 , 抛出错误。

Stub 伪造 RouterLink

`RouterLinkStubDirective` 为测试作出了重要的贡献 :

testing/router-stubs.ts (RouterLinkStubDirective)

```

@Directive({
  selector: '[routerLink]',
  host: {
    '(click)': 'onClick()'
  }
})
export class RouterLinkStubDirective {
  @Input('routerLink') linkParams: any;
}

```

```

  navigatedTo: any = null;

  onClick() {
    this.navigatedTo = this.linkParams;
  }
}

```

host 元数据属性将宿主元素 (`<a>`) 的 `click` 事件与指令的 `onClick` 方法关联起来。绑定到 `[routerLink]` 的 URL 属性被传递到指令的 `linkParams` 属性。点击这个链接应该能触发 `onClick` 方法，从而设置 `navigatedTo` 属性。测试程序可以查看这个属性，来确认期望的 **点击导航** 行为。

By.directive 和注入的指令

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(() => {
  // trigger initial data binding
  fixture.detectChanges();

  // find DebugElements with an attached RouterLinkStubDirective
  linkDes = fixture.debugElement
    .queryAll(By.directive(RouterLinkStubDirective));

  // get the attached link directive instances using the DebugElement
  // injectors
  links = linkDes
    .map(de => de.injector.get(RouterLinkStubDirective) as
      RouterLinkStubDirective);
});

```

特别值得注意的两点：

1. 你还可以按指令定位元素，使用 `By.directive`，而不仅仅是通过 CSS 选择器。

2. 你可以使用组件的依赖注入器来获取附加的指令，因为 Angular 总是将附加组件添加到组件的注入器中。

下面是一些使用这个配置的测试程序：

app/app.component.spec.ts (selected tests)

```
it('can get RouterLinks from template', () => {
  expect(links.length).toBe(3, 'should have 3 links');
  expect(links[0].linkParams).toBe('/dashboard', '1st link should
go to Dashboard');
  expect(links[1].linkParams).toBe('/heroes', '1st link should go
to Heroes');
});

it('can click Heroes link in template', () => {
  const heroesLinkDe = linkDes[1];
  const heroesLink = links[1];

  expect(heroesLink.navigatedTo).toBeNull('link should not have
navigated yet');

  heroesLinkDe.triggerEventHandler('click', null);
  fixture.detectChanges();

  expect(heroesLink.navigatedTo).toBe('/heroes');
});
}
```

本例中的“click”测试程序其实毫无价值。它显得很有用，但是事实上，它测试的是 `RouterLinkStubDirective`，而非测试组件。这是指令 stub 的通病。

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。

这些测试有什么好处？

stub 伪造的 `RouterLink` 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，应用是否会成功的导航到目标组件。

对于这样局限的测试目标，stub 伪造 `RouterLink` 和 `RouterOutlet` 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。未来本章的更新将介绍如何使用 `RouterTestingModule` 来编写这样的测试程序。

[回到顶部](#)

使用 `NO_ERRORS_SCHEMA` 来“浅化”组件测试程序

以前的配置 声明了 `BannerComponent`，并 stub 伪造了两个其它组件，**仅仅是为了避免编译错误，不是为别的原因**。

没有它们，Angular 编译器无法识别 `app.component.html` 模板里的 `<app-banner>`、`<app-welcome>` 和 `<router-outlet>` 标签，并抛出错误。

添加 `NO_ERRORS_SCHEMA` 到测试模块的 `schemas` 元数据中，告诉编译器忽略不认识的元素和属性。这样你不再需要声明无关组件和指令。

这些测试程序比较 **浅**，因为它们只“深入”到你要测试的组件。这里是一套配置（拥有 `import` 语句），体现了相比使用 stub 伪造的配置来说，**浅** 测试程序的简单性。

```
import { NO_ERRORS_SCHEMA }          from '@angular/core';
import { AppComponent }              from './app.component';
import { RouterOutletStubComponent } from '../testing';
```

```

beforeEach( async() => {
  TestBed.configureTestingModule({
    declarations: [ AppComponent, RouterLinkStubDirective ],
    schemas:      [ NO_ERRORS_SCHEMA ]
  })

  .compileComponents()
  .then(() => {
    fixture = TestBed.createComponent(AppComponent);
    comp   = fixture.componentInstance;
  });
})];

```

这里 **唯一** 声明的是 **被测试的组件** (`AppComponent`) 和测试需要的 `RouterLinkStubDirective`。没有改变任何 **原测试程序**。

使用 `NO_ERRORS_SCHEMA` 的 **浅组件测试程序** 很大程度上简化了拥有复杂模板组件的单元测试。但是，编译器将不再提醒你一些错误，比如模板中拼写错误或者误用的组件和指令。

[回到顶部](#)

测试属性指令

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, Renderer } 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 renderer: Renderer, private el: ElementRef) {
    renderer.setElementProperty(el.nativeElement, 'customProperty',
      true);
  }

  ngOnChanges() {
    this.renderer.setStyle(
      this.el.nativeElement, '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.component.ts

```

import { Component } from '@angular/core';
@Component({
  template: `
    <h2 highlight="skyblue">About</h2>
    <twain-quote></twain-quote>
    <p>All about this sample</p>`
})
export class AboutComponent { }

```

使用 `AboutComponent` 来测试这个 `HighlightDirective` 的使用，只需要上面解释过的知识就够了，（尤其是 "浅测试程序" 方法）。

app/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 de = fixture.debugElement.query(By.css('h2'));
  expect(de.styles['backgroundColor']).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。

[孤立单元测试](#) 可能有用。但是像这样的属性指令一般都操纵 DOM。孤立单元测试不能控制 DOM，所以不推荐用它测试指令的功能。

更好的方法是创建一个展示所有使用该组件的方法的人工测试组件。

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 { }
```

`<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 =
11.     fixture.debugElement.queryAll(By.directive(HighlightDirective));
12.
13.   // the h2 without the HighlightDirective
14.   bareH2 = fixture.debugElement.query(By.css('h2:not([highlight])'));
15. });
16.
17. // color tests
18. it('should have three highlighted elements', () => {
19.   expect(des.length).toBe(3);
20. });
21. it('should color 1st <h2> background "yellow"', () => {

```

```
22.     expect(des[0].styles['backgroundColor']).toBe('yellow');
23.   });
24.
25.   it('should color 2nd <h2> background w/ default color', () => {
26.     const dir = des[1].injector.get(HIGHLIGHTDIRECTIVE) as
27.       HighlightDirective;
28.     expect(des[1].styles['backgroundColor']).toBe(dir.defaultColor);
29.   });
30.
31.   it('should bind <input> background to value color', () => {
32.     // easier to work with nativeElement
33.     const input = des[2].nativeElement as HTMLInputElement;
34.     expect(input.style.backgroundColor).toBe('cyan', 'initial
35.       backgroundColor');
36.
37.     // dispatch a DOM event so that Angular responds to the input value
38.     // change.
39.     input.value = 'green';
40.     input.dispatchEvent(newEvent('input'));
41.     fixture.detectChanges();
42.
43.     // customProperty tests
44.     it('all highlighted elements should have a true customProperty', () =>
45.     {
46.       const allTrue = des.map(de =>
47.         !!de.properties['customProperty']).every(v => v === true);
48.       expect(allTrue).toBe(true);
49.     });
50.
51.     it('bare <h2> should not have a customProperty', () => {
52.       expect(bareH2.properties['customProperty']).toBeUndefined();
53.     });
54.
```

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 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。
- 在缺乏浏览器时， DebugElement.styles 提供对元素样式的访问。但是，请随意使用 nativeElement，它可能更加容易、更加清晰。
- 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 提供了对指令的自定义属性的访问。

[回到顶部](#)

孤立的单元测试

使用 Angular 测试工具测试应用程序是本章的重点。

但是，使用 **孤立** 单元测试来探索应用类的内在逻辑往往更加有效率，它不依赖 Angular 。这种测试程序通常比较小、更易阅读、编写和维护。

它们不会：

- 从 Angular 测试库导入
- 配置模块
- 准备依赖注入 providers
- 调用 inject ，或者 async ，或者 fakeAsync

它们会：

- 使用标准的、与 Angular 无关的测试技巧
- 直接使用 new 创建实例

- 用测试复制品 (stub , spy 和 mock) 替代真正的依赖

同时采用这两种测试程序

优秀的开发者同时编写这两种测试程序来测试相同的应用部件，往往在同一个 spec 文件。编写简单的 **孤立** 单元测试程序来验证孤立的部分。编写 **Angular** 测试程序来验证与 Angular 互动、更新 DOM 、以及与应用其它部分互动的部分。

服务

服务是应用孤立测试的好例子。下面是未使用 Angular 测试工具的一些 `FancyService` 的同步和异步单元测试：

app/bag/bag.no-testbed.spec.ts

```
1. // Straight Jasmine - no imports from Angular test libraries
2.
3. describe('FancyService without the TestBed', () => {
4.   let service: FancyService;
5.
6.   beforeEach(() => { service = new FancyService(); });
7.
8.   it('#getValue should return real value', () => {
9.     expect(service.getValue()).toBe('real value');
10.    });
11.
12.   it('#getAsyncValue should return async value', done => {
13.     service.getAsyncValue().then(value => {
14.       expect(value).toBe('async value');
15.       done();
16.     });
17.   });
18.
19.   it('#getTimeoutValue should return timeout value', done => {
20.     service = new FancyService();
21.     service.getTimeoutValue().then(value => {
22.       expect(value).toBe('timeout value');
23.       done();
24.     });
25.   });
26.
```

```

26.
27.     it('#getObservableValue should return observable value', done => {
28.       service.getObservableValue().subscribe(value => {
29.         expect(value).toBe('observable value');
30.         done();
31.       });
32.     });
33.
34.   });

```

粗略行数表明，这些孤立单元测试比同等的 Angular 测试小 25%。这表明了它的好处，但是不是最关键的。主要的好处来自于缩减的配置和代码的复杂性。

比较下面两个同等的 `FancyService.getTimeoutValue` 测试程序：

```

1.   it('#getTimeoutValue should return timeout value', done => {
2.     service = new FancyService();
3.     service.getTimeoutValue().then(value => {
4.       expect(value).toBe('timeout value');
5.       done();
6.     });
7.   });

```

它们有类似的行数。但是，依赖 Angular 的版本有更多活动的部分，包括一些工具函数（`async` 和 `inject`）。两种方法都可行，而且如果你为了某些原因使用 Angular 测试工具，也并没有什么问题。反过来，为什么要为简单的服务测试程序添加复杂度呢？

选择你喜欢的方法。

带依赖的服务

服务通常依赖其它服务，Angular 通过构造函数注入它们。你可以 **不使用** TestBed 测试这些服务。在许多情况下，创建和手动 **注入** 依赖来的更加容易。

`DependentService` 是一个简单的例子：

app/bag/bag.ts

```

@Injectable()
export class DependentService {
  constructor(private dependentService: FancyService) { }
  getValue() { return this.dependentService.getValue(); }
}

```

它将唯一的方法，`getValue`，委托给了注入的 `FancyService`。

这里是几种测试它的方法。

app/bag/bag.no-testbed.spec.ts

```

1.  describe('DependentService without the TestBed', () => {
2.    let service: DependentService;
3.
4.    it('#getValue should return real value by way of the real
FancyService', () => {
5.      service = new DependentService(new FancyService());
6.      expect(service.getValue()).toBe('real value');
7.    });
8.
9.    it('#getValue should return faked value by way of a fakeService', () => {
10.      service = new DependentService(new FakeFancyService());
11.      expect(service.getValue()).toBe('faked value');
12.    });
13.
14.    it('#getValue should return faked value from a fake object', () => {
15.      const fake = { getValue: () => 'fake value' };
16.      service = new DependentService(fake as FancyService);
17.      expect(service.getValue()).toBe('fake value');
18.    });
19.
20.    it('#getValue should return stubbed value from a FancyService spy', () => {
21.      const fancy = new FancyService();
22.      const stubvalue = 'stub value';
23.      const spy = spyOn(fancy, 'getValue').and.returnValue(stubvalue);

```

```

24.     service = new DependentService(fancy);
25.
26.     expect(service.getValue()).toBe(stubValue, 'service returned stub
27.     value');
27.     expect(spy.calls.count()).toBe(1, 'stubbred method was called
28.     once');
28.     expect(spy.calls.mostRecent().returnValue).toBe(stubValue);
29.   });
30. });

```

第一个测试程序使用 `new` 创建 `FancyService` 实例，并将它传递给 `DependentService` 构造函数。

很少有这么简单的，注入的服务有可能很难创建和控制。你可以 mock 依赖，或者使用假值，或者用易于控制的替代品 stub 伪造相关服务。

这些 **孤立** 单元测试技巧是一个很好的方法，用来探索服务的内在逻辑，以及它与组件类简单的集成。当在 **运行时间环境下**，使用 Angular 测试工具来验证一个服务是如何与组件互动的。

管道

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: false})
/** 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.  });

```

同时也编写 Angular 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', fakeAsync(() => {
2.    const inputName = 'quick BROWN fox';
3.    const titleCaseName = 'Quick Brown Fox';
4.
5.    // simulate user entering new name into the input box
6.    page.nameInput.value = inputName;
7.
8.    // dispatch a DOM event so that Angular learns of input value
9.    // change.
10.   page.nameInput.dispatchEvent(newEvent('input'));
11.
12.   // Tell Angular to update the output span through the title pipe
13.   fixture.detectChanges();
14.
15.   expect(page.nameDisplay.textContent).toBe(titleCaseName);
16. });

```

组件

组件测试通常检查该组件类是如何与自己的模板或者其它合作组件交互的。Angular 测试工具是专门为这种测试设计的。

考虑这个 `ButtonComp` 组件。

app/bag/bag.ts (ButtonComp)

```

@Component({
  selector: 'button-comp',
  template: `
    <button (click)="clicked()">Click me!</button>
    <span>{{message}}</span>
  `})
export class ButtonComponent {
  isOn = false;
  clicked() { this.isOn = !this.isOn; }
  get message() { return `The light is ${this.isOn ? 'on' : 'off'}`; }
}

```

下面的 Angular 测试演示点击模板里的按钮后，引起了屏幕上的消息的更新。

app/bag/bag.spec.ts (ButtonComp)

```
it('should support clicking a button', () => {
  const fixture = TestBed.createComponent(ButtonComponent);
  const btn = fixture.debugElement.query(By.css('button'));
  const span =
    fixture.debugElement.query(By.css('span')).nativeElement;

  fixture.detectChanges();
  expect(span.textContent).toMatch(/is off/i, 'before click');

  click(btn);
  fixture.detectChanges();
  expect(span.textContent).toMatch(/is on/i, 'after click');
});
```

该判断验证了数据绑定从一个 HTML 控件 (`<button>`) 流动到组件，以及从组件回到 **不同的** HTML 控件 (``)。 通过的测试程序说明组件和它的模块是否设置正确。

孤立单元测试可以更快的在 API 边界探测组件，更轻松的探索更多条件。

下面是一套单元测试程序，用来验证面对多种输入时组件的输出。

app/bag/bag.no-testbed.spec.ts (ButtonComp)

```
describe('ButtonComp', () => {
  let comp: ButtonComponent;
  beforeEach(() => comp = new ButtonComponent());

  it('#ison should be false initially', () => {
    expect(comp.isOn).toBe(false);
  });

  it('#clicked() should set #ison to true', () => {
    comp.clicked();
    expect(comp.isOn).toBe(true);
  });
});
```

```

expect(comp.isOn).toBe(true);
});

it('#clicked() should set #message to "is on"', () => {
  comp.clicked();
  expect(comp.message).toMatch(/is on/i);
});

it('#clicked() should toggle #ison', () => {
  comp.clicked();
  expect(comp.isOn).toBe(true);
  comp.clicked();
  expect(comp.isOn).toBe(false);
});

```

孤立组件单元测试使用更少的代码以及几乎不存在的配置，提供了很多测试覆盖率。在测试复杂的组件时，这个优势显得更加明显，因为可能需要使用 Angular 测试工具进行精心准备。

但是，孤立测试无法确认 `ButtonComp` 是否与其模板正确的绑定，或者是否有数据绑定。使用 Angular 测试来应对它们。

[回到顶部](#)

Angular 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 测试工具包括 `TestBed`、 `ComponentFixture` 和一些其他函数，用来控制测试环境。`TestBed` 和 `ComponentFixture` 在这里分别解释了。

下面是一些独立函数的总结，以使用频率排序：

函数	描述

`async`

在特殊的 **async 测试区域** 运行测试程序（`it`）或者设置（`beforeEach`）的主体。参见 [上面的讨论](#)。

`fakeAsync`

在特殊的 **fakeAsync 测试区域** 运行测试程序（`it`）的主体，造就控制流更加线性的代码风格。参见 [上面的讨论](#)。

`tick`

在 **fakeAsync 测试区域** 内触发 **计时器** 和 **微任务** 队列，以模拟时间的推移和未完成异步任务的完成。

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，往前推移虚拟时间提供数字的毫秒数，清除在这段时间内的异步行为。参见 [上面的讨论](#)

`inject`

从当前 `TestBed` 注入器注入一个或多个服务到测试函数。参见 [上面](#)。

`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`

一个提供商令牌，用来设置 `auto-changeDetect` 的值，它默认值为 `false`。参见 [自动变化检测](#)

`get TestBed`

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 类总结

`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 `get TestBed()` 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	描述
<code>configureTestingModule</code>	<p>测试垫片（ <code>karma-test-shim</code> , <code>browser-test-shim</code> ）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（比如 <code>DebugDomRender</code> ），它们是所有测试程序都需要的。</p> <p>调用 <code>configureTestingModule</code> 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。</p>
<code>compileComponents</code>	<p>在你完成配置以后异步编译测试模块。如果 任何 测试组件有 <code>templateUrl</code> 或 <code>styleUrls</code> ，那么你 必须 调用这个方法。因为获取组件模块和样式文件必须是异步的。 参见 上面的描述 。</p> <p>一旦调用， <code>TestBed</code> 的配置就会在当前测试期间被冻结。</p>
<code>createComponent</code>	<p>基于当前 <code>TestBed</code> 的配置创建一个类型为 <code>T</code> 的组件实例。 一旦调用， <code>TestBed</code> 的配置就会在当前测试期间被冻结。</p>
<code>overrideModule</code>	<p>替换指定的 <code>NgModule</code> 的元数据。回想一下，模块可以导入其他模块。 <code>overrideModule</code> 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。</p>

`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` 函数通常很适合这个任务。但是如果 `inject` 不能提供服务，它会抛出错误。如果服务是可选的呢？

`TestBed.get` 方法接受一个可选的第二参数，它是在 Angular 找不到所需提供商时返回的对象。（在本例中为 `null`）：

```
service =  
  TestBed.get(FancyService,  
  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变换这个默认设置，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 对象

`TestBed.createComponent<T>` 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	描述
<code>componentInstance</code>	被 <code>TestBed.createComponent</code> 创建的组件类实例。

`debugElement`

与组件根元素关联的
DebugElement。

`debugElement` 在测试和调试期
间，提供对组件及其 DOM 元素的
访问。它是测试者至关重要的属性。
它最有用的成员在 [下面](#) 有所
介绍。

`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。

下面使对测试最有用的方法。

方法

描述

`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`。

如果组件还没有被初始化，立刻调用 `detectChanges` 将检测已有变化并触发 `ngOnInit`。

`checkNoChanges`

运行一次变化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 <code>true</code> 。如果有异步任务没有完成，则返回 <code>false</code> 。
whenStable	返回一个承诺，在 fixture 稳定时解析。 勾住这个承诺，以在异步行为或者异步变化检测之后继续测试。参见 上面
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	描述
<code>nativeElement</code>	与浏览器中 DOM 元素对应（ <code>WebWorkers</code> 时，值为 <code>null</code> ）。
<code>query</code>	调用 <code>query(predicate: Predicate<DebugElement>)</code>

返回子树所有层中第一个匹配 `predicate` 的 `DebugElement`。

`queryAll`

调用

`query(predicate:``Predicate<DebugElement>)`返回子树所有层中所有匹配 `predicate``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 的直接子级。通过 `children` 来降序探索元素树。

`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

如果在元素的 listeners 列表中有一个对应的 listener，则以事件名字触发。第二个参数是 **事件对象**，一般为事件处理器。参见 [上面](#)。

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相对。

`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`：

```
// 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));
```

很多制定应用程序指令注入 `Renderer` 并调用它其中一个方法 `set...`。

运行时的 `Renderer` 在测试环境中的替代品为 `DebugDomRender`。在调用 `set...` 方法时，`DebugDomRender` 更新 `DebugElement` 额外词典属性。

这些词典属性主要是为 Angular DOM 检测工具准备的，但是它们可能对测试者提供有用的信息。

词典	描述
properties	被 Renderer.setProperty 更新。很多 Angular 指令调用它，包括 NgModel。
attributes	被 Renderer.setElementAttribute 更新。Angular 的 [attribute] 绑定调用它。
classes	被 Renderer.setElementClass 更新。Angular 的 [class] 绑定调用它。
styles	被 Renderer.setStyle 更新。Angular 的 [style] 绑定调用它。

下面是 [在线 "Specs Bag" 例子](#) 中 Renderer 测试程序的例子

```
it('DebugDomRender should set attributes, styles, classes, and properties', () => {
  const fixture =
    TestBed.createComponent(BankAccountParentComponent);
  fixture.detectChanges();
  const comp = fixture.componentInstance;

  // the only child is debugElement of the BankAccount component
```

```
const el = fixture.debugElement.children[0];
const childComp = el.componentInstance as BankAccountComponent;
expect(childComp).toEqual(jasmine.any(BankAccountComponent));

expect(el.context).toBe(comp, 'context is the parent component');

expect(el.attributes['account']).toBe(childComp.id, 'account
attribute');
expect(el.attributes['bank']).toBe(childComp.bank, 'bank
attribute');

expect(el.classes['closed']).toBe(true, 'closed class');
expect(el.classes['open']).toBe(false, 'open class');

expect(el.properties['customProperty']).toBe(true,
'customProperty');

expect(el.styles['color']).toBe(comp.color, 'color style');
expect(el.styles['width']).toBe(comp.width + 'px', 'width style');
});
```

[返回顶部](#)

常见问题

为何将测试的 spec 配置文件放置到被测试文件的旁边？

我们推荐将单元测试的 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，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。
-

什么时候我应该把测试 spec 文件放到测试目录中？

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。

TYPESCRIPT配置

Angular 开发者的 TypeScript 配置

TypeScript 是 Angular 应用开发中使用的主语言。它是 JavaScript 的“方言”之一，为类型安全和工具化而做了设计期支持。

浏览器不能直接执行 TypeScript。它得先用 **tsc** 编译器转译 (transpile) 成 JavaScript，而且编译器需要进行一些配置。

本页面会覆盖 TypeScript 配置与环境的某些方面，这些对 Angular 开发者是很重要的。具体来说包括下列文件：

- `tsconfig.json` - TypeScript 编译器配置。
- `typings` - TypeScript 类型声明文件。

tsconfig.json

我们通常会往项目中加入一个 TypeScript 配置文件 (`tsconfig.json`)，来指导编译器如何生成 JavaScript 文件。

要了解关于 `tsconfig.json` 的详情，请参阅官方提供的 [TypeScript wiki](#)。

我们在 [快速起步](#) 中创建过如下的 `tsconfig.json`：

tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "moduleResolution": "node",  
    "sourceMap": true,  
    "emitDecoratorMetadata": true,  
    "experimentalDecorators": true,  
    "removeComments": false,  
    "noImplicitAny": false  
  }  
}
```

该文件中的选项和标志是写 Angular 应用程序的基础。

nolImplicitAny 与 suppressImplicitAnyIndexErrors

TypeScript 开发者们在 `nolImplicitAny` 标志应该是 `true` 还是 `false` 上存在分歧。这没有标准答案，我们以后还可以修改这个标志。但是我们的选择会在大项目中产生显著差异，所以它值得讨论一番。

当 `nolImplicitAny` 标志是 `false` (默认值) 时，如果编译器无法根据变量的用途推断出变量的类型，它就会悄悄的把变量类型默认为 `any`。这就是 **隐式 any** 的含义。

我们在“快速起步”中把 `nolImplicitAny` 标志初始化为 `false`，这是为了让学习 TypeScript 开发更简单点。

当 `nolImplicitAny` 标志是 `true` 并且 TypeScript 编译器无法推断出类型时，它仍然会生成 JavaScript 文件。但是它也会 **报告一个错误**。很多饱经沧桑的程序员更喜欢这种严格的设置，因为类型检查能在编译期间捕获更多意外错误。

即使 `nolImplicitAny` 标志被设置成了 `true`，你也可以把变量的类型设置为 `any`。

如果我们把 `nolImplicitAny` 标志设置为了 `true`，我们可能会得到 **隐式索引错**。大多数程序员可能觉得 **这种错误** 是个烦恼而不是助力。我们可以使用另一个标志来禁止它们。

```
"suppressImplicitAnyIndexErrors":true
```

TypeScript 类型定义 (typings)

很多 JavaScript 库，比如 jQuery、 Jasmine 测试库和 Angular，会通过新的特性和语法来扩展 JavaScript 环境。而 TypeScript 编译器并不能原生的识别它们。当编译器不能识别时，它就会抛出一个错误。

我们可以使用 [TypeScript 类型定义文件](#) —— `.d.ts` 文件——来告诉编译器要加载的库的类型定义。

TypeScript 敏感的编辑器借助这些定义文件来显示这些库中各个特性的类型定义。

很多库在自己的 npm 包中都包含了它们的类型定义文件，TypeScript 编译器和编辑器都能找到它们。Angular 库也是这样的。任何 Angular 应用程序的 `node_modules/@angular/core/` 目录下，都包含几个 `d.ts` 文件，它们描述了 Angular 的各个部分。

我们不需要为那些包含了 `d.ts` 文件的库获取类型定义文件——Angular 的所有包都是如此。

安装类型定义文件

遗憾的是，很多库——jQuery、 Jasmine 和 Lodash 等库——都 **没有** 在它们自己的 npm 包中包含 `d.ts` 文件。幸运的是，它们的作者或社区中的贡献者已经为这些库创建了独立的 `d.ts` 文件，并且把它们发布到了一个众所周知的位置。

We can install these typings via `npm` using the `@types/*` scoped package and Typescript (starting at 2.0) will automatically recognize them.

For instance, to install typings for `jasmine` we could do `npm install @types/jasmine --save-dev`.

我们在“快速起步”中指定过三个 **类型定义** 文件（`d.ts`）：

- `core-js` 是为 ES5 浏览器添加 ES2015/ES6 特性的类型定义

- [jasmine](#) 是 Jasmine 测试框架的类型定义
- [node](#) 是为了在 [nodejs](#) 环境中引用对象的代码提供的类型定义。 在 [webpack](#) 页面可以看到例子。

“快速起步”本身不需要这些类型定义，但是文档中的很多例子都需要。

从1.X升级

Angular 1 应用可以逐步升级到 Angular 2。

已经有了一个 Angular 1 的程序并不表示我们就不能享受 Angular 2 提供的一切。这是因为 Angular 2 带来了一些内置工具，来帮助我们把 Angular 1 的项目迁移到 Angular 2 平台。

有些应用可能比其它的升级起来简单，还有一些方法能让这项工作变得更简单。即使在正式开始升级过程之前，我们可以准备 Angular 1 的程序，让它向 Angular 2 看齐。这些准备步骤几乎都是关于如何让代码更加松耦合、更有可维护性，以及用现代开发工具提高速度的。这意味着，这种准备工作不仅能让最终的升级变得更简单，而且还能提升 Angular 1 程序的质量。

成功升级的关键之一是增量式的实现它，通过在同一个应用中一起运行这两个框架，并且逐个把 Angular 1 的组件迁移到 Angular 2 中。这意味着可以在不必打断其它业务的前提下，升级更大、更复杂的应用程序，因为这项工作可以多人协作完成，在一段时间内逐渐铺开。Angular 2 `upgrade` 模块的设计目标就是让你渐进、无缝的完成升级。

1. 准备工作

1. 遵循 Angular 风格指南
2. 使用模块加载器
3. 迁移到 TypeScript
4. 使用组件型指令

2. 通过升级适配器进行升级

1. 如何升级适配器
2. 引导 Angular 1 和 2 的混合 (hybrid) 应用
3. 从 Angular 1 的代码中使用 Angular 2 的组件
4. 从 Angular 2 的代码中使用 Angular 1 的组件
5. 把 Angular 1 的内容投影 (project) 进 Angular 2 组件中
6. 把 Angular 2 的内容透传 (transclude) 到 Angular 1 的组件型指令中
7. 让 Angular 1 提供的依赖可以被注入到 Angular 2
8. 让 Angular 2 提供的依赖可以被注入到 Angular 1

3. PhoneCat 准备工作教程

1. 切换到 TypeScript
2. 安装 Angular 2
3. 引导 Angular 1+2 的混合版 PhoneCat
4. 升级 Phone 服务
5. 升级组件
6. 切换到 Angular 2 的路由器并引导
7. 再见，Angular 1！

准备工作

Angular 1 应用程序的组织方式有很多种。当我们想把它们升级到 Angular 2 的时候，有些做起来会比其它的更容易些。即使在我们开始升级之前，也有一些关键的技术和模式可以让我们将来升级时更轻松。

遵循 Angular 风格指南

Angular 风格指南 收集了一些已证明能写出干净且可维护的 Angular 1 程序的模式与实践。它包含了很多关于如何书写和组织 Angular 代码的有价值信息，同样重要的是，**不应该** 采用的书写和组织 Angular 代码的方式。

Angular 2 是一个基于 Angular 1 中最好的部分构思出来的版本。在这种意义上，它的目标和 Angular 风格指南是一样的：保留 Angular 1 中好的部分，去掉坏的部分。当然，Angular 2 还做了更多。说这些的意思是：**遵循这个风格指南可以让你写出更接近 Angular 2 程序的 Angular 1 程序。**

特别是某些规则会让使用 Angular 2 的 `upgrade` 模块进行 **增量升级** 变得更简单：

- **单一规则** 规定每个文件应该只放一个组件。这不仅让组件更容易浏览和查找，而且还将允许我们逐个迁移它们的语言和框架。在这个范例程序中，每个控制器、工厂和过滤器都在它自己的源文件中。
- **按特性分目录的结构** 和 **模块化** 规则在较高的抽象层定义了一些相似的原则：应用程序中的不同部分应该被分到不同的目录和 Angular 模块中。

如果应用程序能用这种方式把每个特性分到一个独立目录中，它也就能每次迁移一个特性。对于那些还没有这么做的程序，强烈建议把应用这条规则作为准备步骤。而且这也不仅仅对升级有价值，它还是一个通用的规则，可以让你的程序更“坚实”。

使用模块加载器

当我们把应用代码分解成每个文件中放一个组件之后，我们通常会得到一个由大量相对较小的文件组成的项目结构。这比组织成少量大文件要整洁得多，但如果你不得不通过 `<script>` 标签在 HTML 页面中加载所有这些文件，那就不好玩了。尤其是当你不得不按正确的顺序维护这些标签时更是如此。这就是为什么开始使用 **模块加载器** 是一个好主意了。

使用模块加载器，比如 [SystemJS](#)、[Webpack](#) 或 [Browserify](#)，可以让我们在程序中使用 TypeScript 或 ES2015 语言内置的模块系统。我们可以使用 `import` 和 `export` 特性来明确指定哪些代码应该以及将被在程序的不同部分之间共享。对于 ES5 程序来说，我们可以改用 CommonJS 风格的 `require` 和 `module.exports` 特性代替。无论是论哪种情况，模块加载器都会按正确的顺序加载程序中用到的所有代码。

当我们的应用程序投入生产环境时，模块加载器也会让把所有这些文件打成完整的产品包变得更容易。

迁移到 TypeScript

Angular 2 升级计划的一部分是引入 TypeScript，即使在开始升级之前，引入 TypeScript 编译器也是有意义的。这意味着等真正升级的时候需要学习和思考的东西更少。它还意味着我们可以在 Angular 1 代码中开始使用 TypeScript 的特性。

因为 TypeScript 是 ECMAScript 2015 的一个超集，而 ES2015 又是 ECMAScript 5 的一个超集。这意味着除了安装一个 TypeScript 编译器，并把文件名都从 `*.js` 改成 `*.ts` 之外，其实什么都不用做。当然，如果仅仅这样做也没什么大用，也没什么令人兴奋之处。下面这些额外步骤可以让我们精神抖擞起来：

- 对那些使用了模块加载器的程序，TypeScript 的导入和导出（这实际上是 ECMAScript 2015 导入和导出）可以把代码组织到模块中。
- 类型注解可以逐步添加到已存在的函数和变量上，以固定它们的类型，并获得其优点：比如编译期错误检查、更好的支持自动完成，以及内联式文档等。
- 那些 ES2015 中新增的特性，比如箭头函数、`let`、`const`、默认函数参数、解构赋值等也能逐渐添加进来，让代码更有表现力。
- 服务和控制器可以转成 **类**。这样我们就能一步步接近 Angular 2 的服务和组件类了，这样等到我们开始升级时，也会更简单。

使用组件型指令

在 Angular 2 中，组件是用来构建用户界面的主要元素。我们把 UI 中的不同部分定义成组件，然后通过在模板中使用这些组件最终合成为 UI。

我们在 Angular 1 中也能这么做。那就是一种定义了自己的模板、控制器和输入 / 输出绑定的指令——跟 Angular 2 中对组件的定义是一样的。要迁移到 Angular 2，通过组件型指令构建的应用程序会比直接用 `ng-controller`、`ng-include` 和作用域继承等底层特性构建的要容易得多。

要与 Angular 2 兼容，Angular 1 的组件型指令应该配置下列属性：

- `restrict: 'E'`。组件通常会以元素的方式使用。
- `scope: {}` - 一个独立作用域。在 Angular 2 中，组件永远是从它们的环境中被隔离出来的，在 Angular 1 中，我们也应该这么做。
- `bindToController: {}`。组件的输入和输出应该绑定到控制器，而不是 `$scope`。
- `controller` 和 `controllerAs`。组件有它们自己的控制器。
- `template` 或 `templateUrl`。组件有它们自己的模板。

组件型指令还可能使用下列属性：

- `transclude: true`：如果组件需要从其它地方透传内容，就设置它。
- `require`：如果组件需要和父组件的控制器通讯，就设置它。

组件型指令 **不能** 使用下列属性：

- `compile`。它在 Angular 2 中将不再被支持。
- `replace: true`。Angular 永远不会用组件模板替换一个组件元素。这个特性在 Angular 1 中也同样不建议使用了。
- `priority` 和 `terminal`。虽然 Angular 1 的组件可能使用这些，但它们在 Angular 2 中已经没用了，并且最好不要再写依赖它们的代码。

Angular 1 中一个完全向 Angular 2 架构看齐过的组件型指令看起来有点像这样：

```
1.  export function heroDetailDirective() {
2.    return {
3.      restrict: 'E',
4.      scope: {},
```

```

5.     bindToController: {
6.       hero: '=',
7.       deleted: '&'
8.     },
9.     template: `
10.       <h2>{{ctrl.hero.name}} details!</h2>
11.       <div><label>id: </label>{{ctrl.hero.id}}</div>
12.       <button ng-click="ctrlonDelete()">Delete</button>
13.     `,
14.     controller: function() {
15.       this.onDelete = () => {
16.         this.deleted({hero: this.hero});
17.       };
18.     },
19.     controllerAs: 'ctrl'
20.   };
21. }

```

Angular 1.5 引入了 [组件 API](#)，它让像这样定义指令变得更简单了。为组件型指令使用这个 API 是一个好主意，因为：

- 它需要更少的样板代码。
- 它强制使用组件的最佳实践，比如 `controllerAs`。
- 对于指令中像 `scope` 和 `restrict` 这样的属性，它有良好的默认值。

如果使用这个组件 API 进行快捷定义，那么上面看到的组件型指令就变成了这样：

```

1.  export const heroDetail = {
2.    bindings: {
3.      hero: '<',
4.      deleted: '&'
5.    },
6.    template: `
7.      <h2>{{$ctrl.hero.name}} details!</h2>
8.      <div><label>id: </label>{{$ctrl.hero.id}}</div>
9.      <button ng-click="$ctrlonDelete()">Delete</button>
10.    `,
11.    controller: function() {
12.      this.onDelete = () => {
13.        this.deleted(this.hero);
14.      };
15.    }
16.  }

```

```
14.      };
15.    }
16.  };
```

控制器的生命周期钩子 `$onInit()` 、 `$onDestroy()` 和 `$onChanges()` 是 Angular 1.5 引入的另一些便利特性。它们都很接近于 [Angular 2 中的等价物](#)，所以，围绕它们组织组件生命周期的逻辑会更容易升级。

使用升级适配器进行升级

不管要升级什么，Angular 2 中的 `upgrade` 模块都会是一个非常有用的工具——除非是小到没功能的应用。借助它，我们可以在同一个应用程序中混用并匹配 Angular 1 和 2 的组件，并让它们实现无缝的互操作。这意味着我们不用必须一次性做完所有升级工作，因为在整个演进过程中，这两个框架可以很自然的和睦相处。

升级适配器如何工作

`upgrade` 模块提供的主要工具叫做 `UpgradeAdapter`。这是一个服务，它可以引导并管理同时支持 Angular 2 和 Angular 1 的混合式应用程序。

当使用 `UpgradeAdapter` 时，我们实际做的是 **同时运行两个版本的 Angular**。所有 Angular 2 的代码运行在 Angular 2 框架中，而 Angular 1 的代码运行在 Angular 1 框架中。所有这些都是真实的、全功能的框架版本。没有进行任何仿真，所以我们可以期待同时存在这两个框架的所有特性和天生的行为。

所有这些事情的背后，本质上是一个框架中管理的组件和服务能和来自另一个中的进行互操作。这发生在三个主要的领域：依赖注入、DOM 和变更检测。

依赖注入

无论是在 Angular 1 中还是在 Angular 2 中，依赖注入都处于前沿和中心的位置，但在两个框架的工作原理上，却存在着一些关键的不同之处。

Angular 1

Angular 2

依赖注入的令牌 (Token) 永远是字符串
(译注：指服务名称)。

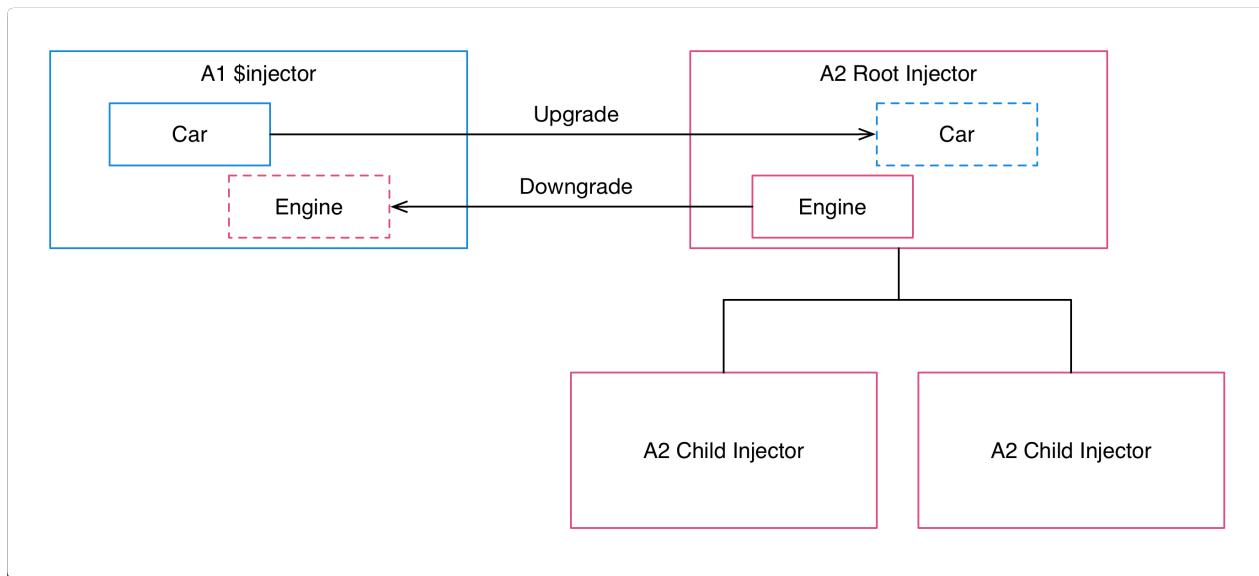
令牌 可能有不同的类型。通常是类，
也可能是字符串。

只有一个注入器。即使在多模块的应用程序中，每样东西也都被装入一个巨大的命名空间中。

有一组 树状多层注入器，有一个根注入器，每个组件也另外有一个注入器。

就算有这么多不同点，也并不妨碍我们在依赖注入时进行互操作。`UpgradeAdapter` 解决了这些差异，并让它们无缝的对接：

- 通过升级它们，我们就能让那些在 Angular 1 中能被注入的服务在 Angular 2 的代码中可用。在框架之间共享的是服务的同一个单例对象。在 Angular 2 中，这些外来服务总是被放在 **根注入器** 中，并可用于所有组件。它们总是具有 **字符串令牌** ——跟它们在 Angular 1 中的令牌相同。
- 通过降级它们，我们也能让那些在 Angular 2 中能被注入的服务在 Angular 1 的代码中可用。只有那些来自 Angular 2 根注入器的服务才能被降级。同样的，在框架之间共享的是同一个单例对象。当我们注册一个要降级的服务时，要明确指定一个打算在 Angular 1 中使用的 **字符串令牌**。



组件与 DOM

在混合式应用中，我们能同时发现那些来自 Angular 1 和 Angular 2 中组件和指令的 DOM。这些组件通过它们各自框架中的输入和输出绑定来互相通讯，它们由 `UpgradeAdapter` 桥接在一起。它们也能通过共享被注入的依赖彼此通讯，就像前面所说的那样。

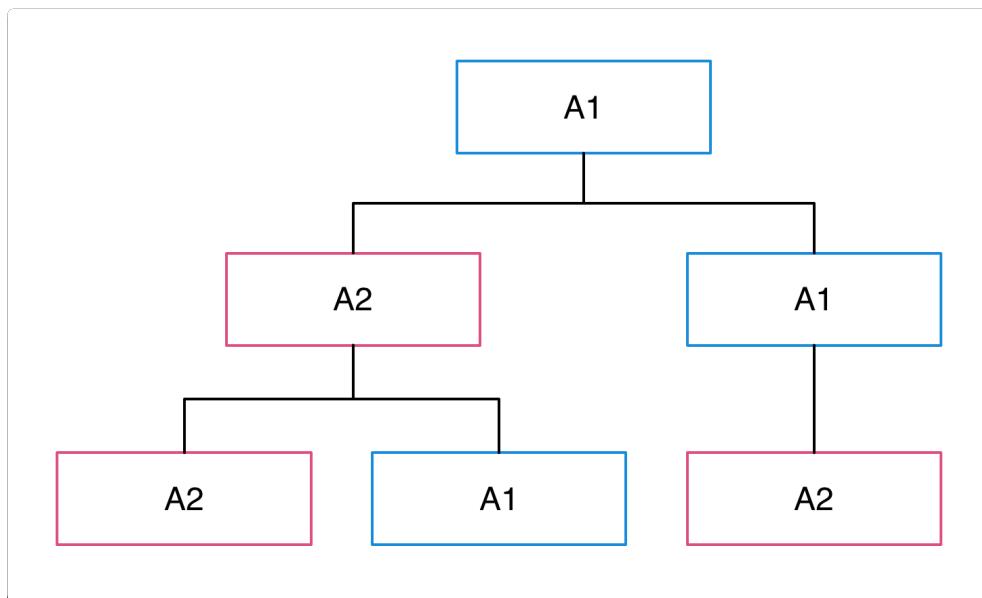
要弄明白在一个混合式应用的 DOM 中发生了什么，有两点很关键：

1. DOM 中的每个元素都只能被两个框架之一拥有。另一个框架会忽略它。如果一个元素被 Angular 1 拥有，Angular 2 就会当它不存在。反之亦然。
2. 应用的根节点 **总是来自 Angular 1 中的模板**。

所以，混合式应用总是像 Angular 1 程序那样启动，处理根模板的也是 Angular 1。然后，当这个应用的模板中使用到了 Angular 2 的组件时，Angular 2 才开始参与。这个组件的视图由 Angular 2 进行管理，而且它还可以使用一系列的 Angular 2 组件和指令。

更进一步说，我们可以按照需要，任意穿插使用这两个框架。使用下面的两种方式之一，我们可以自由穿梭于这两个框架的边界：

1. 通过使用来自另一个框架的组件：Angular 1 的模板中用到了 Angular 2 的组件，或者 Angular 2 的模板中使用了 Angular 1 的组件。
2. 通过透传 (transclude) 或投影 (project) 来自另一个框架的内容。`UpgradeAdapter` 牵线搭桥，把 Angular 1 的透传概念和 Angular 2 的内容投影概念关联起来。



无论什么时候，只要我们用到了来自另一个框架的组件，就会发生框架边界的切换。然而，这种切换只会发生在组件元素的 **子节点** 上。考虑一个场景，我们从 Angular 1 中像这样使用 Angular 2 的组件：

1. `<ng2-component></ng2-component>`

此时，`<ng2-component>` 这个 DOM 元素仍然由 Angular 1 管理，因为它是在 Angular 1 的模板中定义的。这也意味着你可以往它上面添加别的 Angular 1 指令，却 **不能** 添加 Angular 2 的指

令。只有在 `Ng2Component` 组件的模板中才是 Angular 2 的天下。同样的规则也适用于在 Angular 2 中使用 Angular 1 组件型指令的情况。

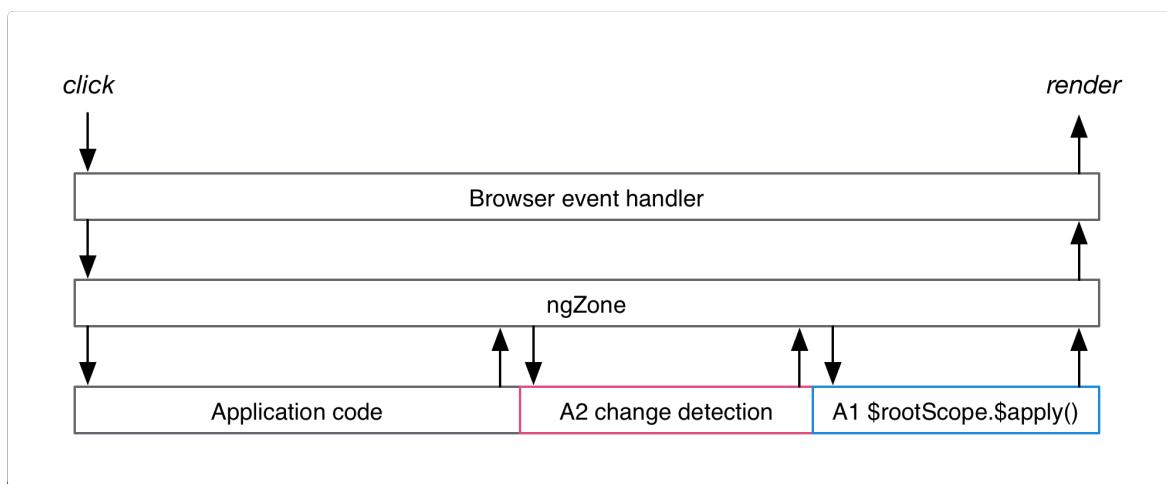
变更检测

Angular 1 中的变更检测全是关于 `scope.$apply()` 的。在每个事件发生之后，`scope.$apply()` 就会被调用。这或者由框架自动调用，或者在某些情况下由我们自己的代码手动调用。它是发生变更检测以及更新数据绑定的时间点。

在 Angular 2 中，事情有点不一样。虽然变更检测仍然会在每一个事件之后发生，却不再需要每次调用 `scope.$apply()` 了。这是因为所有 Angular 2 代码都运行在一个叫做 `Angular zone` 的地方。Angular 总是知道什么时候代码执行完了，也就知道了它什么时候应该触发变更检测。代码本身并不需要调用 `scope.$apply()` 或其它类似的东西。

在这种混合式应用的案例中，`UpgradeAdapter` 在 Angular 1 的方法和 Angular 2 的方法之间建立了桥梁。发生了什么呢？

- 应用中发生的每件事都运行在 Angular 2 的 zone 里。无论事件发生在 Angular 1 还是 Angular 2 的代码中，都是如此。
- `UpgradeAdapter` 将在每一次离开 Angular zone 时调用 Angular 1 的 `$rootScope.$apply()`。这样也就同样会在每个事件之后触发 Angular 1 的变更检测。



在实践中，这意味着我们不用在自己的代码中调用 `$apply()`，而不管这段代码是在 Angular 1 还是 Angular 2 中。`UpgradeAdapter` 都替我们做了。我们仍然可以调用 `$apply()`，也就是说我们不必从现有代码中移除此调用。但是在混合式应用中，那些调用没有任何效果。

当我们降级一个 Angular 2 组件，然后把它用于 Angular 1 中时，组件的输入属性就会被 Angular 1 的变更检测体系监视起来。当那些输入属性发生变化时，组件中相应的属性就会被设置。我们也能通过实现 `OnChanges` 接口来挂钩到这些更改，就像它未被降级时一样。

相应的，当我们把 Angular 1 的组件升级给 Angular 2 使用时，在这个组件型指令的 `scope`（或 `bindToController`）中定义的所有绑定，都将被挂钩到 Angular 2 的变更检测体系中。它们将和标准的 Angular 2 输入属性被同等对待，并当它们发生变化时设置回 `scope`（或控制器）上。

通过 Angular 2 的 NgModule 来使用升级适配器

Angular 1 还是 Angular 2 都有自己的模块概念，来帮你我们把应用组织成一些紧密相关的功能块。

它们在架构和实现的细节上有着显著的不同。在 Angular 1 中，我们会把 Angular 1 的资源添加到 `angular.module` 属性上。在 Angular 2 中，我们会创建一个或多个带有 `NgModule` 装饰器的类，这些装饰器用来在元数据中描述 Angular 资源。差异主要来自这里。

在混合式应用中，我们同时运行了两个版本的 Angular。这意味着我们至少需要 Angular 1 和 Angular 2 各提供一个模块。当我们使用 Angular 1 的模块进行引导时，就得把 Anuglar 2 的模块传给 `UpgradeAdapter`。我们来看看怎么做。

要了解 Angular 2 模块的更多信息，请参阅 [Angular 模块](#) 页。

引导 Angular 1+2 的混合式应用程序

使用 `UpgradeAdapter` 升级应用的第一步总是把它引导成一个同时支持 Angular 1 和 Angular 2 的混合式应用。

纯粹的 Angular 1 应用可以用两种方式引导：在 HTML 页面中的某处使用 `ng-app` 指令，或者从 JavaScript 中调用 `angular.bootstrap`。在 Angular 2 中，只有第二种方法是可行的，因为它没有 `ng-app` 指令。在混合式应用中也同样只能用第二种方法。所以，在将 Angular 1 应用切换到混合模式之前，把它改为用 JavaScript 引导的方式是一个不错的起点。

比如说我们有个由 `ng-app` 驱动的引导过程，就像这个：

```
1.  <!DOCTYPE HTML>
2.  <html>
3.    <head>
4.      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.3/angular.js">
</script>
```

```

5.      <script src="app/1-ng-app/app.module.js"></script>
6.    </head>
7.
8.    <body ng-app="heroApp" ng-strict-di>
9.      <div id="message" ng-controller="MainCtrl as mainCtrl">
10.        {{ mainCtrl.message }}
11.      </div>
12.    </body>
13.  </html>

```

我们可以从 HTML 中移除 `ng-app` 和 `ng-strict-di` 指令，改为从 JavaScript 中调用 `angular.bootstrap`，它能达到同样效果：

```
angular.bootstrap(document.body, ['heroApp'], {strictDi: true});
```

现在，把 Angular 2 引入项目中。根据“[快速起步](#)”中的指导，你可以有选择的从“[快速起步](#)”的 [Github 仓库](#) 中拷贝素材进来。

接下来，创建一个 `app.module.ts` 文件，并添加下列 `NgModule` 类：

```

1.  import { NgModule } from '@angular/core';
2.  import { BrowserModule } from '@angular/platform-browser';
3.
4.  @NgModule({
5.    imports: [ BrowserModule ]
6.  })
7.  export class AppModule {}

```

这个最小化的 `NgModule` 导入了 `BrowserModule`，该模块是每个基于浏览器的 Angular 应用都必须具备的。

用新的 `AppModule` 来导入和实例化 `UpgradeAdapter` 类，并调用它的 `bootstrap` 方法。该方法被设计成接受与 `angular.bootstrap` 完全相同的参数：

```

1.  import { UpgradeAdapter } from '@angular/upgrade';
2.

```

```

3.  /* . . . */
4.
5.  const upgradeAdapter = new UpgradeAdapter(AppModule);
6.
7.  upgradeAdapter.bootstrap(document.body, ['heroApp'], {strictDi: true});

```

恭喜！我们就要开始运行 Angular 1+2 的混合式应用程序了！所有现存的 Angular 1 代码会像以前一样正常工作，但是我们现在也同样可以运行 Angular 2 代码了。

注意，在 `angular.bootstrap` 和 `upgradeAdapter.bootstrap` 之间有一个显著的不同点：后者是**异步**工作的。该应用不是立即启动的。有时必须在这次 `bootstrap` 调用返回后才能传入。

当我们开始把组件移植到 Angular 2 时，我们还将使用 `UpgradeAdapter` ——不止是进行引导。在整个应用程序中使用此适配器的**同一个**实例是非常重要的，因为它保存了关于该应用程序当前状态的内部信息：

upgrade_adapter.ts

```

1.  import { UpgradeAdapter } from '@angular/upgrade';
2.  export const upgradeAdapter = new UpgradeAdapter(AppModule);

```

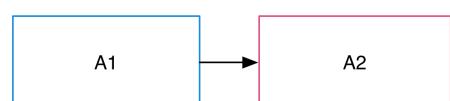
然后这个共享的实例就能被所有需要它的模块获取到：

```

1.  import { upgradeAdapter } from './upgrade_adapter';
2.
3.  /* . . . */
4.
5.  upgradeAdapter.bootstrap(document.body, ['heroApp'], {strictDi: true});

```

在 Angular 1 的代码中使用 Angular 2 的组件



一旦我们开始运行混合式应用，我们就可以开始逐渐升级代码了。做这件事的一种更常见的模式就是在 Angular 1 的上

下文中使用 Angular 2 的组件。该组件可能是全新的，也可能是把原本 Angular 1 的组件用 Angular 2 重写而成的。

假设我们有一个简单的用来显示英雄信息的 Angular 2 组件：

```
hero-detail.component.ts

1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'hero-detail',
5.   template: `
6.     <h2>Windstorm details!</h2>
7.     <div><label>id: </label>1</div>
8.   `
9. })
10. export class HeroDetailComponent {
```

如果我们想在 Angular 1 中使用这个组件，我们就得用“**升级适配器**”把它**降级**。如果我们这么做，就会得到一个 Angular 1 的**指令**，我们可以把它注册到 Angular 1 的模块中：

```
1. import { HeroDetailComponent } from './hero-detail.component';
2.
3. /* . . . */
4.
5. angular.module('heroApp', [])
6.   .directive('heroDetail',
7.     upgradeAdapter.downgradeNg2Component(HeroDetailComponent));
```

由于 `HeroDetailComponent` 是一个 Angular 2 组件，所以我们必须同时把它加入 `AppModule` 的 `declarations` 字段中。

```
1. import { HeroDetailComponent } from './hero-detail.component';
2.
3. @NgModule({
4.   imports: [ BrowserModule ],
```

```
5.     declarations: [ HeroDetailComponent ]
6.   })
7.   export class AppModule {}
```

所有 Angular 2 组件、指令和管道都必须声明在 NgModule 中。

这里我们得到的是一个叫做 `heroDetail` 的 Angular 1 指令，我们可以像用其它指令一样把它用在 Angular 1 模板中。

```
<hero-detail></hero-detail>
```

注意，它在 Angular 1 中是一个名叫 `heroDetail` 的元素型指令 (`restrict: 'E'`)。Angular 1 的元素型指令是基于它的 **名字** 匹配的。**Angular 2 组件中的 selector 元数据，在降级后的版本中会被忽略。**

当然，大多数组件都不像这个这么简单。它们中很多都有 **输入属性和输出属性**，来把它们连接到外部世界。Angular 2 的英雄详情组件带有像这样的输入属性与输出属性：

hero-detail.component.ts

```
1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2. import { Hero } from '../hero';
3.
4. @Component({
5.   selector: 'hero-detail',
6.   template: `
7.     <h2>{{hero.name}} details!</h2>
8.     <div><label>id: </label>{{hero.id}}</div>
9.     <button (click)="onDelete()">Delete</button>
10.    `
11. })
12. export class HeroDetailComponent {
13.   @Input() hero: Hero;
14.   @Output() deleted = new EventEmitter<Hero>();
```

```

15.     onDelete() {
16.       this.deleted.emit(this.hero);
17.     }
18.   }

```

这些输入属性和输出属性的值来自于 Angular 1 的模板，而 `UpgradeAdapter` 负责桥接它们：

```

1.  <div ng-controller="MainController as mainCtrl">
2.    <hero-detail [hero]="mainCtrl.hero"
3.      (deleted)="mainCtrl.onDelete($event)">
4.    </hero-detail>
5.  </div>

```

注意，虽然我们正在 Angular 1 的模板中，**但却在使用 Angular 2 的属性 (Attribute) 语法来绑定到输入属性与输出属性**。这是降级的组件本身要求的。而表达式本身仍然是标准的 Angular 1 表达式。

在降级过的组件属性中使用中线命名法

为降级过的组件使用 Angular 2 的属性 (Attribute) 语法规则时有一个值得注意的例外。它适用于由多个单词组成的输入或输出属性。在 Angular 2 中，我们要使用小驼峰命名法绑定这些属性：

`[myHero]="hero"`

但是从 Angular 1 的模板中使用它们时，我们得使用中线命名法：

`[my-hero]="hero"`

`$event` 变量能被用在输出属性里，以访问这个事件所发出的对象。这个案例中它是 `Hero` 对象，因为 `this.deleted.emit()` 函数曾把它传了出来。

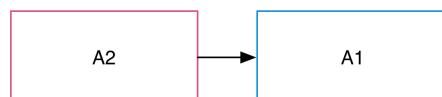
由于这是一个 Angular 1 模板，虽然它已经有了 Angular 2 中绑定的属性 (Attribute)，我们仍可以在这个元素上使用其它 Angular 1 指令。例如，我们可以用 `ng-repeat` 简单的制作该组件的多份拷贝：

```

1.  <div ng-controller="MainController as mainCtrl">
2.    <hero-detail [hero]="hero"
3.      (deleted)="mainCtrl.onDelete($event)"
4.      ng-repeat="hero in mainCtrl.heroes">
5.    </hero-detail>
6.  </div>

```

从 Angular 2 代码中使用 Angular 1 组件型指令



现在，我们已经能在 Angular 2 中写一个组件，并把它用于 Angular 1 代码中了。当我们从低级组件开始移植，并往上走时，这非常有用。但在另外一些情况下，从相反的方向进行移植会更加方便：从高级组件开始，然后往下走。这也同样能用 `UpgradeAdapter` 完成。我们可以 **升级** Angular 1 组件型指令，然后从 Angular 2 中用它们。

不是所有种类的 Angular 1 指令都能升级。该指令必须是一个严格的 **组件型指令**，具有 [上面的准备指南中描述的](#) 那些特征。确保兼容性的最安全的方式是 Angular 1.5 中引入的 **组件 API**。

可升级组件的简单例子是只有一个模板和一个控制器的指令：

hero-detail.component.ts

```

1.  export const heroDetail = {
2.    template: `
3.      <h2>Windstorm details!</h2>
4.      <div><label>id: </label>1</div>
5.    `,
6.    controller: function() {
7.    }
8.  };

```

我们可以使用 `UpgradeAdapter` 的 `upgradeNg1Component` 方法来把这个组件 **升级** 到 Angular 2。它接受 Angular 1 组件型指令的名字，并返回一个 Angular 2 **组件类**。像其它 Angular 2 组件一样，请把它声明在 `NgModule` 中：

app.module.ts

```

1. const HeroDetail = upgradeAdapter.upgradeNg1Component('heroDetail');
2.
3. @NgModule({
4.   imports: [ BrowserModule ],
5.   declarations: [ ContainerComponent, HeroDetail ]
6. })
7. export class AppModule {}

```

升级后的组件总会有一个元素选择器，它就是原 Angular 1 组件型指令的原始名字。

升级后的组件也可能有输入属性和输出属性，它们是在原 Angular 1 组件型指令的 scope/controller 绑定中定义的。当我们从 Angular 2 模板中使用该组件时，我们要使用 **Angular 2 模板语法** 来提供这些输入属性和输出属性，但要遵循下列规则：

	绑定定义	模板语法
属性 (Attribute) 绑定	myAttribute: '@myAttribute'	<my-component myAttribute="value">
表达式绑定	myOutput: '&myOutput'	<my-component (myOutput)="action()">
单向绑定	myValue: '<myValue'	<my-component [myValue]="anExpression">
双向绑定	myValue: '=myValue'	用作输入： <my-component [myValue]="anExpression"> 或 用作双向绑定： <my-component [(myValue)]="anExpression"

举个例子，假设我们在 Angular 1 中有一个表示“英雄详情”的组件型指令，它带有一个输入属性和一个输出属性：

hero-detail.component.ts

```

1.  export const heroDetail = {
2.    bindings: {
3.      hero: '<',
4.      deleted: '&'
5.    },
6.    template: `
7.      <h2>{{$ctrl.hero.name}} details!</h2>
8.      <div><label>id: </label>{{$ctrl.hero.id}}</div>
9.      <button ng-click="$ctrl.onDelete()">Delete</button>
10.     `,
11.    controller: function() {
12.      this.onDelete = () => {
13.        this.deleted(this.hero);
14.      };
15.    }
16.  };

```

我们可以把这个组件升级到 Angular 2，然后使用 Angular 2 的模板语法提供这个输入属性和输出属性：

container.component.ts

```

1.  import { Component } from '@angular/core';
2.  import { Hero } from '../hero';
3.
4.  @Component({
5.    selector: 'my-container',
6.    template: `
7.      <h1>Tour of Heroes</h1>
8.      <hero-detail [hero]="hero"
9.                  (deleted)="heroDeleted($event)">
10.     </hero-detail>
11.   `
12. })
13. export class ContainerComponent {
14.   hero = new Hero(1, 'Windstorm');
15.   heroDeleted(hero: Hero) {

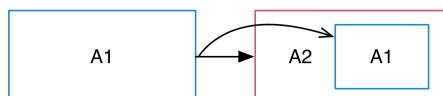
```

```

16.     hero.name = 'Ex-' + hero.name;
17.   }
18. }

```

把 Angular 1 的内容投影到 Angular 2 组件中



如果我们在 Angular 1 模板中使用降级后的 Angular 2 组件时，可能会需要把模板中的一些内容投影进那个组件。这也是可能的，虽然在 Angular 2 中并没有透传 (transclude) 这样的东西，但它有一个非常相似的概念，叫做 **内容投影**。 `UpgradeAdapter` 也能让这两个特性实现互操作。

Angular 2 的组件通过使用 `<ng-content>` 标签来支持内容投影。下面是这类组件的一个例子：

hero-detail.component.ts

```

1. import { Component, Input } from '@angular/core';
2. import { Hero } from '../hero';
3.
4. @Component({
5.   selector: 'hero-detail',
6.   template: `
7.     <h2>{{hero.name}}</h2>
8.     <div>
9.       <ng-content></ng-content>
10.    </div>
11.  `
12. })
13. export class HeroDetailComponent {
14.   @Input() hero: Hero;
15. }

```

当从 Angular 1 中使用该组件时，我们可以为它提供内容。正如它们将在 Angular 1 中被透传一样，它们也在 Angular 2 中被投影到了 `<ng-content>` 标签所在的位置：

```

1. <div ng-controller="MainController as mainCtrl">
2.   <hero-detail [hero]="mainCtrl.hero">
3.     <!-- Everything here will get projected -->

```

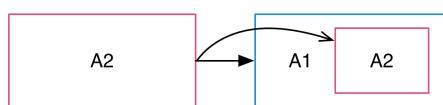
```

4.      <p>{{mainCtrl.hero.description}}</p>
5.    </hero-detail>
6.  </div>

```

当 Angular 1 的内容被投影到 Angular 2 组件中时，它仍然留在“Angular 1 王国”中，并被 Angular 1 框架管理着。

把 Angular 2 的内容透传进 Angular 1 的组件型指令



就像我们能把 Angular 1 的内容投影进 Angular 2 组件一样，我们也能把 Angular 2 的内容 **透传** 进 Angular 1 的组件，但不管怎样，我们都要使用它们升级过的版本。

如果一个 Angular 1 组件型指令支持透传，它就会在自己的模板中使用 `ng-transclude` 指令标记出透传到的位置：

hero-detail.component.ts

```

1.  export const heroDetailComponent = {
2.    bindings: {
3.      hero: '='
4.    },
5.    template: `
6.      <h2>{{$ctrl.hero.name}}</h2>
7.      <div>
8.        <ng-transclude></ng-transclude>
9.      </div>
10.     `
11.   };

```

该指令还需要启用一个 `transclude: true` 选项。当用 Angular 1.5 中的组件 API 定义组件型指令时，该选项默认是开启的。

如果我们升级这个组件，并把它用在 Angular 2 中，我们就能把准备透传的内容放进这个组件的标签中。

container.component.ts

```

1. import { Component } from '@angular/core';
2. import { Hero } from '../hero';
3.
4. @Component({
5.   selector: 'my-container',
6.   template: `
7.     <hero-detail [hero]="hero">
8.       <!-- Everything here will get transcluded -->
9.       <p>{{hero.description}}</p>
10.    </hero-detail>
11.  `
12. })
13. export class ContainerComponent {
14.   hero = new Hero(1, 'Windstorm', 'Specific powers of controlling winds');
15. }

```

让 Angular 1 中的依赖可被注入到 Angular 2

当运行一个混合式应用时，我们可能会遇到这种情况：我们需要把某些 Angular 1 的依赖注入到 Angular 2 代码中。这可能是因为某些业务逻辑仍然在 Angular 1 服务中，或者需要某些 Angular 1 的内置服务，比如 `$location` 或 `$timeout`。

在这些情况下，把一个 Angular 1 提供商 **升级到** Angular 2 也是有可能的。这就让它将来有可能被注入到 Angular 2 代码中的某些地方。比如，我们可能在 Angular 1 中有一个名叫 `HeroesService` 的服务：

heroes.service.ts

```

1. import { Hero } from '../hero';
2.
3. export class Heroesservice {
4.   get() {
5.     return [
6.       new Hero(1, 'Windstorm'),
7.       new Hero(2, 'Spiderman')
8.     ];
9.   }
10. }

```

我们可以用 `UpgradeAdapter` 的 `upgradeNg1Provider` 方法来升级该服务，只要给它传入服务的名字就行了。这会把该服务加到 Angular 2 的根注入器中。

app.module.ts

```
1. angular.module('heroApp', [])
2.   .service('heroes', HeroesService)
3.   .directive('heroDetail',
4.     upgradeAdapter.downgradeNg2Component(HeroDetailComponent));
5.
6. upgradeAdapter.upgradeNg1Provider('heroes');
```

然后我们可以使用与它在 Angular 1 中的原始名字相同的字符串型令牌，把它注入到 Angular 2 中：

hero-detail.component.ts

```
1. import { Component, Inject } from '@angular/core';
2. import { HeroesService } from './heroes.service';
3. import { Hero } from '../hero';
4.
5. @Component({
6.   selector: 'hero-detail',
7.   template:
8.     <h2>{{hero.id}}: {{hero.name}}</h2>
9.   `
10. })
11. export class HeroDetailComponent {
12.   hero: Hero;
13.   constructor(@Inject('heroes') heroes: HeroesService) {
14.     this.hero = heroes.get()[0];
15.   }
16. }
```

在这个例子中，我们升级了服务类。当我们注入它时，我们可以使用 TypeScript 类型注解来获得这些额外的好处。它没有影响该依赖的处理过程，同时还得到了启用静态类型检查的好处。任何 Angular 1 中的服务、工厂和提供商都能被升级——尽管这不是必须的。

让 Angular 2 的依赖能被注入到 Angular 1 中

除了能升级 Angular 1 依赖之外，我们还能 **降级** Angular 2 的依赖，以便我们能在 Angular 1 中使用它们。当我们已经开始把服务移植到 Angular 2 或在 Angular 2 中创建新服务，但同时还有一些用 Angular 1 写成的组件时，这会非常有用。

例如，我们可能有一个 Angular 2 的 `Heroes` 服务：

heroes.ts

```
1. import { Injectable } from '@angular/core';
2. import { Hero } from '../hero';
3.
4. @Injectable()
5. export class Heroes {
6.   get() {
7.     return [
8.       new Hero(1, 'Windstorm'),
9.       new Hero(2, 'Spiderman')
10.    ];
11.  }
12. }
```

仿照 Angular 2 组件，我们通过把该提供商加入 `NgModule` 的 `providers` 列表中来注册它。

app.module.ts

```
1. import { Heroes } from './heroes';
2.
3. @NgModule({
4.   imports: [ BrowserModule ],
5.   providers: [ Heroes ]
6. })
7. export class AppModule {}
```

现在，我们使用 `upgradeAdapter.downgradeNg2Provider()` 来把 Angular 2 的 `Heroes` 包装成 **Angular 1 的工厂函数**，并把这个工厂注册进 Angular 1 的模块中。依赖在 Angular 1 中的名字你可以自己定：

app.module.ts

```

1. angular.module('heroApp', [])
2.   .factory('heroes', upgradeAdapter.downgradeNg2Provider(Heroes))
3.   .component('heroDetail', heroDetailComponent);

```

此后，该服务就能被注入到 Angular 1 代码中的任何地方了：

hero-detail.component.ts

```

1. export const heroDetailComponent = {
2.   template:
3.     <h2>{{$ctrl.hero.id}}: {{$ctrl.hero.name}}</h2>
4.   ,
5.   controller: ['$heroes', function(heroes: Heroes) {
6.     this.hero = heroes.get()[0];
7.   }]
8. };

```

PhoneCat 升级教程

在本节和下节中，我们将看一个完整的例子，它使用 `upgrade` 模块准备和升级了一个应用程序。该应用就是来自 [原 Angular 1 教程](#) 中的 [Angular PhoneCat](#)。那是我们很多人当初开始 Angular 探险之旅的起点。现在，我们来看看如何把该应用带入 Angular 2 的美丽新世界。

这期间，我们将学到如何在实践中应用 [准备指南](#) 中列出的那些重点步骤：我们先让该应用向 Angular 2 看齐，然后为它引入 SystemJS 模块加载器和 TypeScript。

要跟随本教程，请先把 [angular-phonecat](#) 仓库克隆到本地，并跟我们一起应用这些步骤。

在项目结构方面，我们工作的起点是这样的：

```

angular-phonecat
├── bower.json
└── karma.conf.js

```

```
|- package.json  
|- app  
  |- core  
    |- checkmark  
      |- checkmark.filter.js  
      |- checkmark.filter.spec.js  
    |- phone  
      |- phone.module.js  
      |- phone.service.js  
      |- phone.service.spec.js  
    |- core.module.js  
  |- phone-detail  
    |- phone-detail.component.js  
    |- phone-detail.component.spec.js  
    |- phone-detail.module.js  
    |- phone-detail.template.html  
  |- phone-list  
    |- phone-list.component.js  
    |- phone-list.component.spec.js  
    |- phone-list.module.js  
    |- phone-list.template.html  
  |- img  
    |- ...  
  |- phones  
    |- ...  
  |- app.animations.js  
  |- app.config.js  
  |- app.css  
  |- app.module.js  
  |- index.html  
- e2e-tests  
  |- protractor-conf.js  
  |- scenarios.js
```

这确实是一个很好地起点。特别是，该结构遵循了 [Angular 1 风格指南](#)，要想成功升级，这是一个很重要的 [准备步骤](#)。

- 每个组件、服务和过滤器都在它自己的源文件中——就像 [单一规则](#) 所要求的。
- `core`、`phone-detail` 和 `phone-list` 模块都在它们自己的子目录中。那些子目录除了包含 HTML 模板之外，还包含 JavaScript 代码，它们共同完成一个特性。这是 [按特性分目录的结构](#) 和 [模块化](#) 规则所要求的。
- 单元测试都和应用代码在一起，它们很容易找到。就像[规则组织测试文件](#) 中要求的那样。

切换到 TypeScript

因为我们将使用 TypeScript 编写 Angular 2 的代码，所以在开始升级之前，我们把 TypeScript 的编译器设置好是很合理的。

我们还将开始逐步淘汰 Bower 包管理器，换成我们更喜欢的 NPM。后面我们将使用 NPM 来安装新的依赖包，并最终从项目中移除 Bower。

让我们先把 TypeScript 包安装到项目中。

```
npm i typescript --save-dev
```

我们还要把用来运行 TypeScript 编译器 `tsc` 和 `typings` 工具的脚本添加到 `package.json` 中：

package.json

```
1.  {
2.    "scripts": {
3.      "tsc": "tsc",
4.      "tsc:w": "tsc -w"
5.    }
6.  }
```

现在我们可以使用 `typings` 工具来安装 Angular 1 和 Jasmine 单元测试框架的类型定义文件。

```
npm install @types/jasmine @types/angular @types/angular-animate
@types/angular-cookies @types/angular-mocks @types/angular-resource
```

```
@types/angular-route @types/angular-sanitize --save-dev
```

我们还应该配置 TypeScript 编译器，以便它能理解我们的项目结构。我们要往项目目录下添加一个 `tsconfig.json` 文件，就像在“[快速起步](#)”中做过的那样。它将告诉 TypeScript 编译器，该如何编译我们的源文件。

tsconfig.json

```
1.  {
2.    "compilerOptions": {
3.      "target": "es5",
4.      "module": "commonjs",
5.      "moduleResolution": "node",
6.      "sourceMap": true,
7.      "emitDecoratorMetadata": true,
8.      "experimentalDecorators": true,
9.      "removeComments": false,
10.     "noImplicitAny": false,
11.     "suppressImplicitAnyIndexErrors": true
12.   }
13. }
```

我们告诉 TypeScript 编译器，把 TypeScript 文件转换成 ES5 代码，并打包进 CommonJS 模块中。

我们现在可以从命令行启动 TypeScript 编译器。它将监控 `.ts` 源码文件，并随时把它们编译成 JavaScript。然后这些编译出的 `.js` 文件被 SystemJS 加载到浏览器中。当我们继续往前走的时候，这个过程将在后台持续运行。

```
npm run tsc:w
```

我们要做的下一件事是把 JavaScript 文件转换成 TypeScript 文件。由于 TypeScript 是 ECMAScript 2015 的一个超集，而 ES2015 又是 ECMAScript 5 的超集，所以我们可以简单的把文件的扩展名从 `.js` 换成 `.ts`，它们还是会像以前一样工作。由于 TypeScript 编译器仍在运行，它会为每一个 `.ts` 文件生成对应的 `.js` 文件，而真正运行的是编译后的 `.js` 文件。如果你用 `npm start` 开启了本项目的 HTTP 服务器，你会在浏览器中看到一个功能完好的应用。

有了 TypeScript，我们就可以从它的一些特性中获益了。此语言可以为 Angular 1 应用提供很多价值。

首先，TypeScript 是一个 ES2015 的超集。任何以前用 ES5 写的程序（就像 PhoneCat 范例）都可以开始通过 TypeScript 纳入那些添加到 ES2015 中的新特性。这包括 `let`、`const`、箭头函数、函数默认参数以及解构 (destructure) 赋值。

我们能做的另一件事就是把 **类型安全** 添加到代码中。这实际上已经部分完成了，因为我们已经安装了 Angular 1 的类型定义。当我们正确调用 Angular 1 的 API 时，TypeScript 会帮我们检查它——比如往 Angular 模块中注册组件。

我们还能开始把 **类型注解** 添加到自己的代码中，来从 TypeScript 的类型系统中获得更多帮助。比如，我们可以给 `checkmark` 过滤器加上注解，表明它期待一个 `boolean` 类型的参数。这可以更清楚的表明此过滤器打算做什么

app/core/checkmark/checkmark.filter.ts

```

1.
2. angular.
3.   module('core').
4.     filter('checkmark', function() {
5.       return function(input: boolean) {
6.         return input ? '\u2713' : '\u2718';
7.       };
8.     });

```

在 `Phone` 服务中，我们可以明确的把 `$resource` 服务声明为 `angular.resource.IResourceService`，一个 Angular 1 类型定义提供的类型。

app/core/phone/phone.service.ts

```

1. angular.
2.   module('core.phone').
3.     factory('Phone', ['$resource',
4.       function($resource: angular.resource.IResourceService) {
5.         return $resource('phones/:phoneId.json', {}, {
6.           query: {
7.             method: 'GET',
8.             params: {phoneId: 'phones'},
9.             isArray: true
10.           }
11.         });
12.       }
13.     ]);

```

我们可以在应用的路由配置中使用同样的技巧，那里我们用到了 location 和 route 服务。一旦给它们提供了类型信息，TypeScript 就能检查我们是否在用类型的正确参数来调用它们了。

app/app.config.ts

```
1. angular.
2.   module('phonecatApp').
3.   config(['$locationProvider', '$routeProvider',
4.     function config($locationProvider: angular.ILocationProvider,
5.                   $routeProvider: angular.route.IRouteProvider) {
6.       $locationProvider.hashPrefix('!');
7.
8.       $routeProvider.
9.         when('/phones', {
10.             template: '<phone-list></phone-list>'
11.         }).
12.         when('/phones/:phoneId', {
13.             template: '<phone-detail></phone-detail>'
14.         }).
15.         otherwise('/phones');
16.     }
17.   ]);
```

我们用 typings 工具安装的这个 [Angular 1.x 类型定义文件](#) 并不是由 Angular 开发组维护的，但它也已经足够全面了。借助这些类型定义的帮助，它可以为 Angular 1.x 程序加上全面的类型注解。

如果我们想这么做，那么在 `tsconfig.json` 中启用 `noImplicitAny` 配置项就是一个好主意。这样，如果遇到什么还没有类型注解的代码，TypeScript 编译器就会显示一个警告。我们可以用它作为指南，告诉我们现在与一个完全类型化的项目距离还有多远。

我们能用的另一个 TypeScript 特性是 **类**。具体来讲，我们可以把控制器转换成类。这种方式下，我们离成为 Angular 2 组件类就又近了一步，它会令我们的升级之路变得更简单。

Angular 1 期望控制器是一个构造函数。这实际上就是 ES2015/TypeScript 中的类，这也就意味着只要我们把一个类注册为组件控制器，Angular 1 就会愉快的使用它。

新的“电话列表 (phone list) ”组件控制器类看起来是这样的：

app/phone-list/phone-list.component.ts

```

1.  class PhoneListController {
2.    phones: any[];
3.    orderProp: string;
4.    query: string;
5.
6.    static $inject = ['Phone'];
7.    constructor(Phone: any) {
8.      this.phones = Phone.query();
9.      this.orderProp = 'age';
10.    }
11.
12.  }
13.
14. angular.
15.   module('phoneList').
16.     component('phoneList', {
17.       templateUrl: 'phone-list/phone-list.template.html',
18.       controller: PhoneListController
19.     });

```

以前在控制器函数中实现的一切，现在都改由类的构造函数来实现了。类型注入注解通过静态属性 `$inject` 被附加到了类上。在运行时，它们变成了 `PhoneListController.$inject` 。

该类还声明了另外三个成员：电话列表、当前排序键的名字和搜索条件。这些东西我们以前就加到了控制器上，只是从来没有在任何地方显式定义过它们。最后一个成员从未真正在 TypeScript 代码中用过，因为它只是在模板中被引用过。但为了清晰起见，我们还是应该定义出此控制器应有的所有成员。

在电话详情控制器中，我们有两个成员：一个是用户正在查看的电话，另一个是正在显示的图像：

app/phone-detail/phone-detail.component.ts

```
1.  class PhoneDetailController {
```

```
2.     phone: any;
3.     mainImageUrl: string;
4.
5.     static $inject = ['$routeParams', 'Phone'];
6.     constructor($routeParams: angular.route.IRouteParamsService, Phone: any) {
7.       let phoneId = $routeParams['phoneId'];
8.       this.phone = Phone.get({phoneId}, (phone: any) => {
9.         this.setImage(phone.images[0]);
10.      });
11.    }
12.
13.    setImage(imageUrl: string) {
14.      this.mainImageUrl = imageUrl;
15.    }
16.  }
17.
18. angular.
19.   module('phoneDetail').
20.   component('phoneDetail', {
21.     templateUrl: 'phone-detail/phone-detail.template.html',
22.     controller: PhoneDetailController
23.   });

```

这已经让我们的控制器代码看起来更像 Angular 2 了。我们的准备工作做好了，可以引进 Angular 2 到项目中了。

如果项目中有任何 Angular 1 的服务，它们也是转换成类的优秀候选人，像控制器一样，它们也是构造函数。但是在本项目中，我们只有一个 `Phone` 工厂，这有点特别，因为它是一个 `ngResource` 工厂。所以我们不会在准备阶段中处理它，而是在下一节中直接把它转换成 Angular 2 服务。

安装 Angular 2

我们已经完成了准备工作，接下来就开始把 PhoneCat 升级到 Angular 2。我们将在 Angular 2 [升级模块](#) 的帮助下增量式的完成此项工作。等我们完成的那一刻，就能把 Angular 1 从项目中完全移除了，但其中的关键是在不破坏此程序的前提下一小块一小块的完成它。

该项目还包含一些动画，在此指南的当前版本我们先不升级它，等到后面的发行版再改。

我们来使用 SystemJS 模块加载器把 Angular 2 安装到项目中。看看“快速起步”中的指南，并从那里获得如下配置：

- 把 Angular 2 和其它新依赖添加到 `package.json` 中
- 把 SystemJS 的配置文件 `systemjs.config.js` 添加到项目的根目录。

这些完成之后，就运行：

```
npm install
```

我们可以通过 `index.html` 来把 Angular 2 的依赖快速加载到应用中，但首先，我们得做一些目录结构调整。这是因为我们正准备从 `node_modules` 中加载文件，然而目前项目中的每一个文件都是从 `/app` 目录下加载的。

把 `app/index.html` 移入项目的根目录，然后把 `package.json` 中的开发服务器根目录也指向项目的根目录，而不再是 `app` 目录：

package.json (start script)

```
1.  {
2.    "scripts": {
3.      "start": "http-server -a localhost -p 8000 -c-1 ./"
4.    }
5.  }
```

现在，我们能把项目根目录下的每一样东西发给浏览器了。但我们不想为了适应开发环境中的设置，被迫修改应用代码中用到的所有图片和数据的路径。因此，我们往 `index.html` 中添加一个 `<base>` 标签，它将导致各种相对路径被解析回 `/app` 目录：

index.html

```
<base href="/app/">
```

现在我们可以通过 SystemJS 加载 Angular 2 了。我们将把 Angular 2 的填充库 (polyfills) 和 SystemJS 的配置加到 `<head>` 区的末尾，然后，我们就用 `System.import` 来加载实际的应用：

index.html

```

1.  <script src="/node_modules/core-js/client/shim.min.js"></script>
2.  <script src="/node_modules/zone.js/dist/zone.js"></script>
3.  <script src="/node_modules/reflect-metadata/Reflect.js"></script>
4.  <script src="/node_modules/systemjs/dist/system.src.js"></script>
5.  <script src="/systemjs.config.js"></script>
6.  <script>
7.    System.import('/app');
8.  </script>

```

在我们从“快速起步”中拿来的 `systemjs.config.js` 文件中，我们还需要做一些调整，以适应我们的项目结构。在使用 SystemJS 而不是 `<base>` URL 加载时，我们需要把浏览器指向项目的根目录。

systemjs.config.js

```

1.  System.config({
2.    paths: {
3.      // paths serve as alias
4.      'npm:': '/node_modules/'
5.    },
6.    map: {
7.      app: '/app',
8.      /* . . . */
9.    },

```

创建 AppModule

现在，创建一个名叫 `AppModule` 的根 `NgModule` 类。我们已经有了一个名叫 `app.module.ts` 的文件，其中存放着 Angular 1 的模块。把它改名为 `app.module.ng1.ts`，同时也要在 `index.html` 中更新对应的脚本名。文件的内容保留：

app.module.ng1.ts

```

1.  'use strict';
2.
3.  // Define the `phonecatApp` Angular 1 module

```

```

4. angular.module('phonecatApp', [
5.   'ngAnimate',
6.   'ngRoute',
7.   'core',
8.   'phoneDetail',
9.   'phoneList',
10. ]);

```

然后创建一个新的 `app.module.ts` 文件，其中是一个最小化的 `NgModule` 类：

app.module.ts

```

1. import { NgModule } from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3.
4. @NgModule({
5.   imports: [
6.     BrowserModule,
7.   ],
8. })
9. export class AppModule {}

```

引导 PhoneCat 的 1+2 混合式应用

接下来，我们把该应用程序引导改装为一个同时支持 Angular 1 和 Angular 2 的 **混合式应用**。然后，就能开始把这些不可分割的小块转换到 Angular 2 了。

要引导一个混合式应用程序，我们首先得初始化一个 `UpgradeAdapter`，它 **提供了胶水**，用来把框架的两个不同版本粘在一起。我们在一个新文件 `app/main.ts` 中导入 `UpgradeAdapter` 类。这个文件已经在 `systemjs.config.js` 文件中被配置成了应用的入口点，所以它已经被浏览器加载了。

app/main.ts

```

1. import { UpgradeAdapter } from '@angular/upgrade';
2. declare var angular: any;
3.
4. import { AppModule } from './app.module';

```

为何将 Angular 声明为 any ?

如果能在 Angular 1 中使用强类型的 `angular` 引用就太好了！但是在不同时通过 `import * as angular from 'angular'` 导入 Angular 1 本身时，我们不能导入 Angular 2 的 UMD 类型库 `@types/angular`。

Angular 1 目前是被 `index.html` 里的脚本标签加载的，目前不值得将其切换到 ES6 导入。我们声明 `angular` 为无类型的 `any` 来防止类型错误。

然后我们可以制作一个适配器来实例化这个类：

```
let upgradeAdapter = new UpgradeAdapter(AppModule);
```

我们的应用现在是使用宿主页面中附加到 `<html>` 元素上的 `ng-app` 指令引导的。但在 Angular 2 中，它不再工作了。我们得切换成 JavaScript 驱动的引导方式。所以，从 `index.html` 中移除 `ng-app` 属性，并把这些加载 `main.ts` 中：

```
upgradeAdapter.bootstrap(document.documentElement, ['phonecatApp']);
```

这里使用的参数是应用的根元素（也就是以前我们放 `ng-app` 的元素），和我们准备加载的 Angular 1.x 模块。由于我们是通过 `UpgradeAdapter` 引导应用的，所以实际在运行的应用实际上是一个 Angular 1+2 的混合体。

我们现在同时运行着 Angular 1 和 Angular 2。漂亮！不过我们还没有运行什么实际的 Angular 2 组件，接下来我们就做这件事。

升级 Phone 服务

我们要移植到 Angular 2 的第一块是 `Phone` 工厂（位于 `app/js/core/phones.factory.ts`），并且让它能帮助控制器从服务器上加载电话信息。目前，它是用 `ngResource` 实现的，我们用它做两件事：

- 把所有电话的列表加载到电话列表组件中。

- 把一台电话的详情加载到电话详情组件中。

我们可以用 Angular 2 的服务类来替换这个实现，而把控制器继续留在 Angular 1 的地盘上。

在这个新版本中，我们导入了 Angular 2 的 HTTP 模块，并且用它的 `Http` 服务替换掉 `NgResource`。

再次打开 `app.module.ts` 文件，导入并把 `HttpModule` 添加到 `AppModule` 的 `imports` 数组中：

app.module.ts

```

1. import { HttpModule } from '@angular/http';
2.
3. @NgModule({
4.   imports: [
5.     BrowserModule,
6.     HttpModule,
7.   ],
8. })
9. export class AppModule {}

```

现在，我们已经准备好了升级 `Phones` 服务本身。我们将为 `phone.service.ts` 文件中基于 `ngResource` 的服务加上 `@Injectable` 装饰器：

app/core/phone/phone.service.ts (skeleton)

```

@Injectable()
export class Phone {
  /* . . . */
}

```

`@Injectable` 装饰器将把一些依赖注入相关的元数据附加到该类上，让 Angular 2 知道它的依赖信息。就像在 [依赖注入指南](#) 中描述过的那样，这是一个标记装饰器，我们要把它用在那些没有其它 Angular 2 装饰器，并且自己有依赖注入的类上。

在它的构造函数中，该类期待一个 `Http` 服务。`Http` 服务将被注入进来并存入一个私有字段。然后该服务在两个实例方法中被使用到，一个加载所有电话的列表，另一个加载一台指定电话的详情：

```

1.  @Injectable()
2.  export class Phone {
3.    constructor(private http: Http) { }
4.    query(): Observable<PhoneData[]> {
5.      return this.http.get(`phones/phones.json`)
6.        .map((res: Response) => res.json());
7.    }
8.    get(id: string): Observable<PhoneData> {
9.      return this.http.get(`phones/${id}.json`)
10.        .map((res: Response) => res.json());
11.    }
12.  }

```

该方法现在返回一个 `Phone` 类型或 `Phone[]` 类型的可观察对象 (Observable)。这是一个我们从未用过的类型，因此我们得为它新增一个简单的接口：

app/core/phone/phone.service.ts (interface)

```

export interface PhoneData {
  name: string;
  snippet: string;
  images: string[];
}

```

最终，该类的全部代码如下：

app/core/phone/phone.service.ts

```

1.  import { Injectable } from '@angular/core';
2.  import { Http, Response } from '@angular/http';
3.  import { Observable } from 'rxjs/Rx';
4.
5.  import 'rxjs/add/operator/map';
6.
7.  export interface PhoneData {
8.    name: string;
9.    snippet: string;
10.   images: string[];
11. }
12.

```

```

13.  @Injectable()
14.  export class Phone {
15.    constructor(private http: Http) { }
16.    query(): Observable<PhoneData[]> {
17.      return this.http.get(`phones/phones.json`)
18.        .map((res: Response) => res.json());
19.    }
20.    get(id: string): Observable<PhoneData> {
21.      return this.http.get(`phones/${id}.json`)
22.        .map((res: Response) => res.json());
23.    }
24.  }

```

注意，我们单独导入了 RxJS `Observable` 中的 `map` 操作符。我们需要对想用的所有 RxJS 操作符这么做，因为 Angular 2 默认不会加载所有 RxJS 操作符。

这个新的 `Phone` 服务具有和老的基于 `ngResource` 的服务相同的特性。因为它是 Angular 2 服务，我们通过 `NgModule` 的 `providers` 数组来注册它：

app.module.ts

```

1.  import { Phone } from './core/phone/phone.service';
2.
3.  @NgModule({
4.    imports: [
5.      BrowserModule,
6.      HttpModule,
7.    ],
8.    providers: [ Phone ]
9.  })
10. export class AppModule {}

```

`UpgradeAdapter` 有一个 `downgradeNg2Provider` 方法，用于让 Angular 2 的服务对 Angular 1 的代码可用。用它来插入 `Phone` 服务：

app/main.ts (excerpt)

```

import { Phone } from './core/phone/phone.service';

/* . . . */

```

```
angular.module('core.phone')
  .factory('phone', upgradeAdapter.downgradeNg2Provider(Phone));
```

现在，我们正在用 SystemJS 加载 `phone.service.ts`，我们应该从 `index.html` 中 移除该服务的 `<script>` 标签。这也是我们在升级所有组件时将会做的事。在从 Angular 1 向 2 升级的同时，我们也把代码从脚本移植为模块。

这时，我们可以把两个控制器从使用老的服务切换成使用新的。我们像降级过的 `phones` 工厂一样 `$inject` 它，但它实际上是一个 `Phones` 类的实例，并且我们可以据此注解它的类型：

app/phone-list/phone-list.component.ts

```
1. import { Phone, PhoneData } from '../core/phone/phone.service';
2. declare var angular: any;
3.
4. class PhoneListController {
5.   phones: PhoneData[];
6.   orderProp: string;
7.
8.   static $inject = ['phone'];
9.   constructor(phone: Phone) {
10.     phone.query().subscribe(phones => {
11.       this.phones = phones;
12.     });
13.     this.orderProp = 'age';
14.   }
15.
16. }
17.
18. angular.
19.   module('phoneList').
20.   component('phoneList', {
21.     templateUrl: 'app/phone-list/phone-list.template.html',
22.     controller: PhoneListController
23.   });
```

app/phone-detail/phone-detail.component.ts

```
1. import { Phone, PhoneData } from '../core/phone/phone.service';
```

```

2.  declare var angular: any;
3.
4.  class PhoneDetailController {
5.    phone: PhoneData;
6.    mainImageUrl: string;
7.
8.    static $inject = ['$routeParams', 'phone'];
9.    constructor($routeParams: angular.route.IRouteParamsService, phone: Phone)
10.   {
11.     let phoneId = $routeParams['phoneId'];
12.     phone.get(phoneId).subscribe(data => {
13.       this.phone = data;
14.       this.setImage(data.images[0]);
15.     });
16.   }
17.   setImage(imageUrl: string) {
18.     this.mainImageUrl = imageUrl;
19.   }
20. }
21.
22. angular.
23.   module('phoneDetail').
24.   component('phoneDetail', {
25.     templateUrl: 'phone-detail/phone-detail.template.html',
26.     controller: PhoneDetailController
27.   });

```

这里的两个 Angular 1 控制器在使用 Angular 2 的服务！控制器不需要关心这一点，尽管实际上该服务返回的是可观察对象 (Observable)，而不是承诺 (Promise)。无论如何，我们达到的效果都是把服务移植到 Angular 2，而不用被迫移植组件来使用它。

我们也能使用 Observable 的 toPromise 方法来在服务中把这些可观察对象转变成承诺，以进一步减小组件控制器中需要修改的代码量。

升级组件

接下来，我们把 Angular 1 的控制器升级成 Angular 2 的组件。我们每次升级一个，同时仍然保持应用运行在混合模式下。在做转换的同时，我们还将自定义首个 Angular 2 管道。

让我们先看看电话列表组件。它目前包含一个 TypeScript 控制器类和一个组件定义对象。重命名控制器类，并把 Angular 1 的组件定义对象更换为 Angular 2 `@Component` 装饰器，这样我们就把它变形为 Angular 2 的组件了。然后，我们还从类中移除静态 `$inject` 属性。

app/phone-list/phone-list.component.ts

```

1. import { Component } from '@angular/core';
2. import { Phone, PhoneData } from '../core/phone/phone.service';
3.
4. @Component({
5.   moduleId: module.id,
6.   selector: 'phone-list',
7.   templateUrl: 'phone-list.template.html'
8. })
9. export class PhoneListComponent {
10.   phones: PhoneData[];
11.   query: string;
12.   orderProp: string;
13.
14.   constructor(phone: Phone) {
15.     phone.query().subscribe(phones => {
16.       this.phones = phones;
17.     });
18.     this.orderProp = 'age';
19.   }
20. }
```

`selector` 属性是一个 CSS 选择器，用来定义组件应该被放在页面的哪。在 Angular 1，我们基于组件名字来匹配，但是在 Angular 2 中，我们要有一个专门指定的选择器。本组件将会对应元素名字 `phone-list`，和 Angular 1 版本一样。

现在，我们还需要将组件的模版也转换为 Angular 2 语法。在搜索控件中，我们要为把 Angular 1 的 `$ctrl` 表达式替换成 Angular 2 的双向绑定语法 `[(ngModel)]`：

app/phone-list/phone-list.template.html (search controls)

```

<p>
  Search:
  <input [(ngModel)]="query" />
</p>

<p>
```

```
Sort by:  

<select [(ngModel)]="orderProp">  

  <option value="name">Alphabetical</option>  

  <option value="age">Newest</option>  

</select>  

</p>
```

我们需要把列表中的 `ng-repeat` 替换为 `*ngFor` 以及它的 `let var of iterable` 语法，该语法在 [模板语法指南中讲过](#)。对于图片，我们可以把 `img` 标签的 `ng-src` 替换为一个标准的 `src` 属性 (property) 绑定。

app/phone-list/phone-list.template.html (phones)

```
<ul class="phones">  

  <li *ngFor="let phone of getPhones()"  

    class="thumbnail phone-list-item">  

    <a href="/#/phones/{{phone.id}}" class="thumb">  

      <img [src]="phone.imageUrl" [alt]="phone.name" />  

    </a>  

    <a href="/#/phones/{{phone.id}}" class="name">{{phone.name}}</a>  

    <p>{{phone.snippet}}</p>  

  </li>  

</ul>
```

Angular 2 中没有 `filter` 或 `orderBy` 过滤器

Angular 2 中并不存在 Angular 1 中内置的 `filter` 和 `orderBy` 过滤器。所以我们得自己实现进行过滤和排序。

我们把 `filter` 和 `orderBy` 过滤器改成绑定到控制器中的 `getPhones()` 方法，通过该方法，组件本身实现了过滤和排序逻辑。

app/phone-list/phone-list.component.ts

```
1.  getPhones(): PhoneData[] {  

2.    return this.sortPhones(this.filterPhones(this.phones));  

3.  }  

4.  

5.  private filterPhones(phones: PhoneData[]) {  

6.    if (phones && this.query) {
```

```

7.     return phones.filter(phone => {
8.       let name = phone.name.toLowerCase();
9.       let snippet = phone.snippet.toLowerCase();
10.      return name.indexOf(this.query) >= 0 || snippet.indexOf(this.query) >=
11.        0;
12.    });
13.    return phones;
14.  }
15.
16.  private sortPhones(phones: PhoneData[]) {
17.    if (phones && this.orderProp) {
18.      return phones
19.        .slice(0) // Make a copy
20.        .sort((a, b) => {
21.          if (a[this.orderProp] < b[this.orderProp]) {
22.            return -1;
23.          } else if ([b[this.orderProp] < a[this.orderProp]]) {
24.            return 1;
25.          } else {
26.            return 0;
27.          }
28.        });
29.    }
30.    return phones;
31.  }

```

新的 `PhoneListComponent` 使用 Angular 2 的 `ngModel` 指令，它位于 `FormsModule` 中。把 `FormsModule` 添加到 `NgModule` 的 `imports` 中，并声明新的 `PhoneListComponent` 组件：

app.module.ts

```

1.  import { FormsModule } from '@angular/forms';
2.  import { PhoneListComponent } from './phone-list/phone-list.component';
3.
4.  @NgModule({
5.    imports: [
6.      BrowserModule,
7.      HttpClientModule,
8.      FormsModule,
9.    ],
10.   declarations: [
11.     PhoneListComponent,

```

```

12.    ],
13.    providers: [ Phone ]
14.  })
15. export class AppModule {}

```

在入口点文件 `main.ts` 中，我们把该组件插入到 Angular 1 的模块中。

我们注册了一个 `phoneList` 指令 而不是组件，指令是 Angular 2 组件的降级版。

`UpgradeAdapter` 在两者之间架起了桥梁：

app/main.ts (excerpt)

```

import { PhoneListComponent } from './phone-list/phone-list.component';

/* . . . */

angular.module('phoneList')
.directive(
  'phoneList',
  upgradeAdapter downgradeNg2Component(PhoneListComponent) as
angular.IDirectiveFactory
);

```

`as angular.IDirectiveFactory` 这个强制类型转换告诉 TypeScript 编译器该降级方法的返回值是一个指令工厂。

从 `index.html` 中移除电话列表组件的 `<script>` 标签。

现在，剩下的 `phone-detail.component.ts` 文件变成了这样：

app/phone-detail/phone-detail.component.ts

```

1. import { Component, Inject } from '@angular/core';
2.
3. import { Phone, PhoneData } from '../core/phone/phone.service';
4. @Component({
5.   moduleId: module.id,
6.   selector: 'phone-detail',
7.   templateUrl: 'phone-detail.template.html',
8. })

```

```

9.  export class PhoneDetailComponent {
10.    phone: PhoneData;
11.    mainImageUrl: string;
12.
13.    constructor(@Inject('$routeParams')
14.                $routeParams: angular.route.IRouteParamsService,
15.                phone: Phone) {
16.      phone.get($routeParams['phoneId']).subscribe(phone => {
17.        this.phone = phone;
18.        this.setImage(phone.images[0]);
19.      });
20.    }
21.
22.    setImage(imageurl: string) {
23.      this.mainImageUrl = imageurl;
24.    }
25.  }

```

这和电话列表组件很相似。这里的窍门在于 `@Inject` 装饰器，它标记出了 `$routeParams` 依赖。

Angular 1 注入器具有 Angular 1 路由器的依赖，叫做 `$routeParams`。它被注入到了 `PhoneDetails` 中，但 `PhoneDetails` 现在还是一个 Angular 1 控制器。我们应该把它注入到新的 `PhoneDetailsComponent` 中。

不幸的是，Angular 1 的依赖不会自动在 Angular 2 的组件中可用。我们必须使用 `UpgradeAdapter` 来把 `$routeParams` 包装成 Angular 2 的服务提供商。在 `main.ts` 中这样写：

app/main.ts (\$routeParams)

```
upgradeAdapter.upgradeNg1Provider('$routeParams');
```

不要把升级得来的 Angular 1 服务提供商注册到 `NgModule` 中。

我们现在也要把该组件的模板转变成 Angular 2 的语法。这里是它完整的新模板：

app/phone-detail/phone-detail.template.html

```
1.  <div *ngIf="phone">
2.    <div class="phone-images">
3.      <img [src]="img" class="phone"
4.          [ngClass]={`${selected: img === mainImageUrl}`}
5.          *ngFor="let img of phone.images" />
6.    </div>
7.
8.    <h1>{{phone.name}}</h1>
9.
10.   <p>{{phone.description}}</p>
11.
12.   <ul class="phone-thumbs">
13.     <li *ngFor="let img of phone.images">
14.       <img [src]="img" (click)="setImage(img)" />
15.     </li>
16.   </ul>
17.
18.   <ul class="specs">
19.     <li>
20.       <span>Availability and Networks</span>
21.       <dl>
22.         <dt>Availability</dt>
23.         <dd *ngFor="let availability of phone.availability">{{availability}}
24.           </dd>
25.         </dl>
26.       </li>
27.       <li>
28.         <span>Battery</span>
29.         <dl>
30.           <dt>Type</dt>
31.           <dd>{{phone.battery?.type}}</dd>
32.           <dt>Talk Time</dt>
33.           <dd>{{phone.battery?.talkTime}}</dd>
34.           <dt>Standby time (max)</dt>
35.           <dd>{{phone.battery?.standbyTime}}</dd>
36.         </dl>
37.       </li>
38.       <li>
39.         <span>Storage and Memory</span>
40.         <dl>
41.           <dt>RAM</dt>
42.           <dd>{{phone.storage?.ram}}</dd>
43.           <dt>Internal Storage</dt>
44.           <dd>{{phone.storage?.flash}}</dd>
```

```
44.      </dl>
45.    </li>
46.    <li>
47.      <span>Connectivity</span>
48.      <dl>
49.        <dt>Network Support</dt>
50.        <dd>{{phone.connectivity?.cell}}</dd>
51.        <dt>WiFi</dt>
52.        <dd>{{phone.connectivity?.wifi}}</dd>
53.        <dt>Bluetooth</dt>
54.        <dd>{{phone.connectivity?.bluetooth}}</dd>
55.        <dt>Infrared</dt>
56.        <dd>{{phone.connectivity?.infrared | checkmark}}</dd>
57.        <dt>GPS</dt>
58.        <dd>{{phone.connectivity?.gps | checkmark}}</dd>
59.      </dl>
60.    </li>
61.    <li>
62.      <span>Android</span>
63.      <dl>
64.        <dt>OS Version</dt>
65.        <dd>{{phone.android?.os}}</dd>
66.        <dt>UI</dt>
67.        <dd>{{phone.android?.ui}}</dd>
68.      </dl>
69.    </li>
70.    <li>
71.      <span>Size and weight</span>
72.      <dl>
73.        <dt>Dimensions</dt>
74.        <dd *ngFor="let dim of phone.sizeAndWeight?.dimensions">{{dim}}</dd>
75.        <dt>weight</dt>
76.        <dd>{{phone.sizeAndWeight?.weight}}</dd>
77.      </dl>
78.    </li>
79.    <li>
80.      <span>Display</span>
81.      <dl>
82.        <dt>Screen size</dt>
83.        <dd>{{phone.display?.screenSize}}</dd>
84.        <dt>Screen resolution</dt>
85.        <dd>{{phone.display?.screenResolution}}</dd>
86.        <dt>Touch screen</dt>
87.        <dd>{{phone.display?.touchscreen | checkmark}}</dd>
88.      </dl>
89.    </li>
90.    <li>
```

```

91.      <span>Hardware</span>
92.      <dl>
93.          <dt>CPU</dt>
94.          <dd>{{phone.hardware?.cpu}}</dd>
95.          <dt>USB</dt>
96.          <dd>{{phone.hardware?.usb}}</dd>
97.          <dt>Audio / headphone jack</dt>
98.          <dd>{{phone.hardware?.audioJack}}</dd>
99.          <dt>FM Radio</dt>
100.         <dd>{{phone.hardware?.fmRadio | checkmark}}</dd>
101.         <dt>Accelerometer</dt>
102.         <dd>{{phone.hardware?.accelerometer | checkmark}}</dd>
103.     </dl>
104.   </li>
105.   <li>
106.       <span>Camera</span>
107.       <dl>
108.           <dt>Primary</dt>
109.           <dd>{{phone.camera?.primary}}</dd>
110.           <dt>Features</dt>
111.           <dd>{{phone.camera?.features?.join(' ', ')'}}</dd>
112.       </dl>
113.   </li>
114.   <li>
115.       <span>Additional Features</span>
116.       <dd>{{phone.additionalFeatures}}</dd>
117.   </li>
118. </ul>
119. </div>

```

这里有几个值得注意的改动：

- 我们从所有表达式中移除了 `$ctrl.` 前缀。
- 正如我们在电话列表中做过的那样，我们把 `ng-src` 替换成了标准的 `src` 属性绑定。
- 我们在 `ng-class` 周围使用了属性绑定语法。虽然 Angular 2 中有一个和 Angular 1 中 **非常相似的** `ngClass` 指令，但是它的值不会神奇的作为表达式进行计算。在 Angular 2 中，模板中的属性 (Attribute) 值总是被作为属性 (Property) 表达式计算，而不是作为字符串字面量。
- 我们把 `ng-repeat` 替换成了 `*ngFor`。

- 我们把 `ng-click` 替换成了一个到标准 `click` 事件的绑定。
- 我们把整个模板都包裹进了一个 `ngIf` 中，这导致只有当存在一个电话时它才会渲染。我们必须这么做，是因为组件首次加载时我们还没有 `phone` 变量，这些表达式就会引用到一个不存在的值。和 Angular 1 不同，当我们尝试引用未定义对象上的属性时，Angular 2 中的表达式不会默默失败。我们必须明确指出这种情况是我们所期望的。

把该组件添加到 `NgModule` 的 `declarations` 中：

app.module.ts

```

1. import { PhoneDetailComponent } from './phone-detail/phone-detail.component';
2.
3. @NgModule({
4.   imports: [
5.     BrowserModule,
6.     HttpModule,
7.     FormsModule,
8.   ],
9.   declarations: [
10.     PhoneListComponent,
11.     PhoneDetailComponent,
12.   ],
13.   providers: [ Phone ]
14. })
15. export class AppModule {}

```

在 `main.ts` 中，我们现在会注册一个 `pcPhoneDetail` 指令，而不再是组件。该指令是 `PhoneDetail` 组件的一个降级版。

app/main.ts (excerpt)

```

import { PhoneDetailComponent } from './phone-detail/phone-
detail.component';

/* . . . */

angular.module('phoneDetail')
.directive(
  'phoneDetail',
  upgradeAdapter downgradeNg2Component(PhoneDetailComponent) as

```

```
angular.IDirectiveFactory
);
```

我们现在应该从 `index.html` 中移除电话详情组件的 `<script>`。

添加 `CheckmarkPipe`

Angular 1 指令中有一个 `checkmark` 过滤器，我们把它转换成 Angular 2 的 管道。

升级适配器并没有什么方法能把过滤器转换成管道。但我们也并不需要它。把过滤器函数转换成等价的 Pipe 类非常简单。实现方式和以前一样，但把它们包装进 `transform` 方法中就可以了。把该文件改名成 `checkmark.pipe.ts`，以符合 Angular 2 中的命名约定：

app/core/checkmark/checkmark.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'checkmark'})
export class CheckmarkPipe implements PipeTransform {
  transform(input: boolean) {
    return input ? '\u2713' : '\u2718';
  }
}
```

当我们做这个修改时，也要同时从 `core` 模块文件中移除对该过滤器的注册。该模块的内容变成了：

app.module.ts

```
1. import { CheckmarkPipe } from './core/checkmark/checkmark.pipe';
2.
3. @NgModule({
4.   imports: [
5.     BrowserModule,
6.     HttpModule,
7.     FormsModule,
8.   ],
9.   declarations: [
10.     PhoneListComponent,
11.     PhoneDetailComponent,
```

```
12.      CheckmarkPipe
13.    ],
14.    providers: [ Phone ]
15.  })
16. export class AppModule {}
```

切换到 Angular 2 路由器和引导程序

此刻，我们已经把所有 Angular 1 程序中的部件替换了 Angular 2 中的等价物。

该应用仍然使用混合式应用的方式进行引导，但其实已经不需要了。

在最后两步中，我们彻底移除 Angular 1 的残余势力：

1. 切换到 Angular 2 路由器。
2. 作为纯 Angular 2 应用进行引导。

切换到 Angular 2 路由器

Angular 2 有一个 [全新的路由器](#)。

像所有的路由器一样，它需要在 UI 中指定一个位置来显示路由的视图。在 Angular 2 中，它是 `<router-outlet>`，并位于应用组件树顶部的 **根组件** 中。

我们还没有这样一个根组件，因为该应用仍然是像一个 Angular 1 应用那样被管理的。创建新的 `app.component.ts` 文件，放入像这样的 `AppComponent` 类：

```
app/app.component.ts

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

@Component({
  selector: 'phonecat-app',
  template: '<router-outlet></router-outlet>'
})
export class AppComponent { }
```

它有一个很简单的模板，只包含 `<router-outlet>`。该组件只负责渲染活动路由的内容，此外啥也不干。

该选择器告诉 Angular 2：当应用启动时就把这个根组件插入到宿主页面的 `<phonecat-app>` 元素中。

把这个 `<phonecat-app>` 元素插入到 `index.html` 中。用它来代替 Angular 1 中的 `ng-view` 指令：

index.html (body)

```
<body>
  <phonecat-app></phonecat-app>
</body>
</html>
```

创建 路由模块

无论在 Angular 1 还是 Angular 2 或其它框架中，路由器都需要进行配置。

Angular 2 路由器配置的详情最好去查阅下 [路由与导航](#) 文档。它建议你创建一个专门用于路由器配置的 `NgModule`（名叫 **路由模块**）。

app/app-routing.module.ts

```
1. import { NgModule } from '@angular/core';
2. import { Routes, RouterModule } from '@angular/router';
3. import { APP_BASE_HREF, HashLocationStrategy, LocationStrategy } from
   '@angular/common';
4.
5. import { PhoneDetailComponent } from './phone-detail/phone-detail.component';
6. import { PhoneListComponent } from './phone-list/phone-list.component';
7.
8. const routes: Routes = [
9.   { path: '', redirectTo: 'phones', pathMatch: 'full' },
10.  { path: 'phones', component: PhoneListComponent },
11.  { path: 'phones/:phoneId', component: PhoneDetailComponent }
12. ];
13.
14. @NgModule({
15.   imports: [ RouterModule.forRoot(routes) ],
16.   exports: [ RouterModule ],
```

```

17.     providers: [
18.       { provide: APP_BASE_HREF, useValue: '!' },
19.       { provide: LocationStrategy, useClass: HashLocationStrategy },
20.     ]
21.   })
22.   export class AppRoutingModule {}

```

该模块定义了一个 `routes` 对象，它带有两个路由，分别指向两个电话组件，以及为空路径指定的默认路由。它把 `routes` 传给 `RouterModule.forRoot` 方法，该方法会完成剩下的事。

一些额外的提供商让路由器使用“hash”策略解析 URL，比如 `#!/phones`，而不是默认的“Push State”策略。

现在，修改 `AppModule`，让它导入这个 `AppRoutingModule`，并同时声明根组件 `AppComponent`：

app/app.module.ts

```

1.  import { NgModule } from '@angular/core';
2.  import { BrowserModule } from '@angular/platform-browser';
3.  import { FormsModule } from '@angular/forms';
4.  import { HttpClientModule } from '@angular/http';

5.
6.  import { AppRoutingModule } from './app-routing.module';
7.  import { AppComponent } from './app.component';
8.  import { CheckmarkPipe } from './core/checkmark/checkmark.pipe';
9.  import { Phone } from './core/phone/phone.service';
10. import { PhoneDetailComponent } from './phone-detail/phone-detail.component';
11. import { PhoneListComponent } from './phone-list/phone-list.component';
12.

13. @NgModule({
14.   imports: [
15.     BrowserModule,
16.     FormsModule,
17.     HttpClientModule,
18.     AppRoutingModule
19.   ],
20.   declarations: [
21.     AppComponent,
22.     PhoneListComponent,
23.     CheckmarkPipe,
24.     PhoneDetailComponent
25.   ],

```

```

26.     providers: [
27.       Phone,
28.     ],
29.     bootstrap: [ AppComponent ]
30.   })
31. export class AppModule {}

```

Angular 2 路由器传递路由参数的方式不同。修改 `PhoneDetail` 组件的构造函数，来取得一个注入的 `ActivatedRoute` 对象。从 `ActivatedRoute.snapshot.params` 中提取 `phonId`，并像以前那样获取电话数据：

app/phone-detail/phone-detail.component.ts

```

1. import { Component }      from '@angular/core';
2. import { ActivatedRoute } from '@angular/router';
3.
4. import { Phone, PhoneData } from '../core/phone/phone.service';
5.
6. @Component({
7.   moduleId: module.id,
8.   selector: 'phone-detail',
9.   templateUrl: 'phone-detail.template.html'
10. })
11. export class PhoneDetailComponent {
12.   phone: PhoneData;
13.   mainImageUrl: string;
14.
15.   constructor(activatedRoute: ActivatedRoute, phone: Phone) {
16.     phone.get(activatedRoute.snapshot.params['phoneId'])
17.       .subscribe((p: PhoneData) => {
18.         this.phone = p;
19.         this.setImage(p.images[0]);
20.       });
21.   }
22.
23.   setImage(imageurl: string) {
24.     this.mainImageUrl = imageurl;
25.   }
26. }

```

为每个电话生成链接

在电话列表中，我们不用再被迫硬编码电话详情的链接了。 我们可以通过把每个电话的 `id` 绑定到 `routerLink` 指令来生成它们了，该指令的构造函数会为 `PhoneDetailComponent` 生成正确的 URL：

app/phone-list/phone-list.template.html (list with links)

```
<ul class="phones">
  <li *ngFor="let phone of getPhones()" class="thumbnail phone-list-item">
    <a [routerLink]="/phones", phone.id" class="thumb">
      
    </a>
    <a [routerLink]="/phones", phone.id" class="name">{{phone.name}}</a>
    <p>{{phone.snippet}}</p>
  </li>
</ul>
```

要了解详情，请查看 [路由与导航](#) 页。

作为 Angular 2 应用进行引导

你可能注意到了，有一个额外的元数据属性 `bootstrap` 添加进了 `AppModule` 中。

app/app.module.ts (bootstrap)

```
bootstrap: [ AppComponent ]
```

这是在告诉 Angular 2，它应该使用根组件 `AppComponent` 来引导该应用并且把它的视图插入宿主页面中。

现在，把该应用的引导方法从 `UpgradeAdapter` 的切换到 Angular 2 的方式。 因为这是一个浏览器应用，并且使用即时编译（JiT）进行编译的，因此要用 `platformBrowserDynamic` 函数来引导 `AppModule`：

main.ts

```
1. import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2.
3. import { AppModule } from './app.module';
4.
5. platformBrowserDynamic().bootstrapModule(AppModule);
```

我们现在运行的就是纯正的 Angular 2 应用了！

再见，Angular 1！

是时候把辅助训练的轮子摘下来了！让我们的应用作为一个纯粹、闪亮的 Angular 2 程序开始它的新生命吧。剩下的所有任务就是移除代码——这当然是每个程序员最喜欢的任务！

请从 `main.ts` 中移除所有到 `UpgradeAdapter` 的引用，并移除 Angular 1 的引导代码。

都完成了之后，`main.ts` 看起来应该像这样：

app/main.ts

```
1. import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2.
3. import { AppModule } from './app.module';
4.
5. platformBrowserDynamic().bootstrapModule(AppModule);
```

我们还要完全移除了下列文件。它们是 Angular 1 的模块配置文件和类型定义文件，在 Angular 2 中不需要了：

- `app/app.module.ts`
- `app/app.config.ts`
- `app/core/core.module.ts`
- `app/core/phone/phone.module.ts`
- `app/phone-detail/phone-detail.module.ts`
- `app/phone-list/phone-list.module.ts`

Angular 1 的外部类型定义文件还需要被反安装。我们现在只需要 Jasmine 的那些。

```
npm uninstall @types/angular --save-dev
```

最后，从 `index.html` 和 `karma.conf.js` 中，移除所有到 Angular 1 脚本的引用，比如 `jQuery`。当这些全部做完时，`index.html` 看起来应该是这样的：

index.html

```
1.  <!doctype html>
2.  <html lang="en">
3.    <head>
4.      <meta charset="utf-8">
5.      <base href="/app/">
6.      <title>Google Phone Gallery</title>
7.      <link rel="stylesheet"
     href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
     />
8.      <link rel="stylesheet" href="app.css" />
9.
10.     <script src="/node_modules/core-js/client/shim.min.js"></script>
11.     <script src="/node_modules/zone.js/dist/zone.js"></script>
12.     <script src="/node_modules/reflect-metadata/Reflect.js"></script>
13.     <script src="/node_modules/systemjs/dist/system.src.js"></script>
14.     <script src="/systemjs.config.js"></script>
15.     <script>
16.       System.import('/app');
17.     </script>
18.   </head>
19.   <body>
20.     <phonecat-app></phonecat-app>
21.   </body>
22. </html>
```

这是我们最后一次看到 Angular 1 了！它曾经带给我们很多帮助，不过现在，是时候说再见了。

附录：升级 PhoneCat 的测试

测试不仅要在升级过程中被保留，它还是确保应用在升级过程中不会被破坏的一个安全指示器。要达到这个目的，E2E 测试尤其有用。

E2E 测试

PhoneCat 项目中同时有基于 Protractor 的 E2E 测试和一些基于 Karma 的单元测试。对这两者来说，E2E 测试的转换要容易得多：根据定义，E2E 测试通过与应用中显示的这些 UI 元素互动，从 外部 访问我们的应用来进行测试。E2E 测试实际上并不关心这些应用中各部件的内部结构。这也意味着，虽然我们已经修改了此应用程序，但是 E2E 测试套件仍然应该能像以前一样全部通过。因为从用户的角度来说，我们并没有改变应用的行为。

在转成 TypeScript 期间，我们不用做什么就能让 E2E 测试正常工作。只有当我们想做些修改而把组件及其模板升级到 Angular 2 时才需要做些处理。这是因为 E2E 测试有一些匹配器是 Angular 1 中特有的。对于 PhoneCat 来说，为了让它能在 Angular 2 下工作，我们得做下列修改：

老代码	新代码	说明
<pre>by.repeater('phone in \$ctrl.phones').column('phone.name')</pre>	<pre>by.css('.phones .name')</pre>	repeater 匹配器 依赖于 Angular 1 中的 ng- repeat
<pre>by.repeater('phone in \$ctrl.phones')</pre>	<pre>by.css('.phones li')</pre>	repeater 匹配器 依赖于 Angular 1 中的 ng- repeat
<pre>by.model('\$ctrl.query')</pre>	<pre>by.css('input')</pre>	model 匹配器 依赖于 Angular

1 中的
ng-
model

by.model('\$ctrl.orderProp')

by.css('select')

model
匹配器
依赖于
Angular
1 中的
ng-
model

by.binding('\$ctrl.phone.name')

by.css('h1')

binding
匹配器
依赖于
Angular
1 的数
据绑定

当引导方式从 `UpgradeAdapter` 切换到纯 Angular 2 的时，Angular 1 就从页面中完全消失了。此时，我们需要告诉 Protractor，它不用再找 Angular 1 应用了，而是从页面中查找 **Angular 2** 应用。于是在 `protractor-conf.js` 中做下列修改：

```
useAllAngular2AppRoots: true,
```

同样，我们的测试代码中有两个 Protractor API 调用内部使用了 `$location`。该服务没有了，我们就得把这些调用用一个 WebDriver 的通用 URL API 代替。第一个 API 是“重定向 (redirect)”规约：

e2e-tests/scenarios.ts

```
1. it('should redirect `index.html` to `index.html#/phones`', function() {
2.   browser.get('index.html');
3.   browser.waitForAngular();
4.   browser.getCurrentUrl().then(function(url: string) {
5.     expect(url.endsWith('/phones')).toBe(true);
```

```
6.      });
7.  });

```

然后是“电话链接 (phone links)”规约：

e2e-tests/scenarios.ts

```
1.  it('should render phone specific links', function() {
2.    let query = element(by.css('input'));
3.    // https://github.com/angular/protractor/issues/2019
4.    let str = 'nexus';
5.    for (let i = 0; i < str.length; i++) {
6.      query.sendKeys(str.charAt(i));
7.    }
8.    element.all(by.css('.phones li a')).first().click();
9.    browser.getCurrentUrl().then(function(url: string) {
10.      expect(url.endsWith('/phones/nexus-s')).toBe(true);
11.    });
12.  });

```

单元测试

另一方面，对于单元测试来说，需要更多的转化工作。实际上，它们需要随着产品代码一起升级。

在转成 TypeScript 期间，严格来讲没有什么改动是必须的。但把单元测试代码转成 TypeScript 仍然是个好主意，产品代码从 TypeScript 中获得的那些增益也同样适用于测试代码。

比如，在这个电话详情组件的规约中，我们不仅用到了 ES2015 中的箭头函数和块作用域变量这些特性，还为所用的一些 Angular 1 服务提供了类型定义。

app/phone-detail/phone-detail.component.spec.ts

```
1.  describe('phoneDetail', () => {
2.
3.    // Load the module that contains the `phoneDetail` component before each
4.    // test
5.    beforeEach(angular.mock.module('phoneDetail'));

```

```

6.    // Test the controller
7.    describe('PhoneDetailController', () => {
8.      let $httpBackend: angular.IHttpBackendService;
9.      let ctrl: any;
10.     let xyzPhoneData = {
11.       name: 'phone xyz',
12.       images: ['image/url1.png', 'image/url2.png']
13.     };
14.
15.     beforeEach(inject(($componentController: any,
16.                           _$httpBackend_: angular.IHttpBackendService,
17.                           $routeParams: angular.route.IRouteParamsService) => {
18.       $httpBackend = _$httpBackend_;
19.       $httpBackend.expectGET('phones/xyz.json').respond(xyzPhoneData);
20.
21.       $routeParams['phoneId'] = 'xyz';
22.
23.       ctrl = $componentController('phoneDetail');
24.     }));
25.
26.     it('should fetch the phone details', () => {
27.       jasmine.addCustomEqualityTester(angular.equals);
28.
29.       expect(ctrl.phone).toEqual({});
30.
31.       $httpBackend.flush();
32.       expect(ctrl.phone).toEqual(xyzPhoneData);
33.     });
34.
35.   });
36.
37. });

```

一旦我们开始了升级过程并引入了 SystemJS，还需要对 Karma 进行配置修改。我们需要让 SystemJS 加载所有的 Angular 2 新代码，

karma-test-shim.js

```

1.  // /*global jasmine, __karma__, window*/
2.  Error.stackTraceLimit = 0; // "No stacktrace"" is usually best for app
   testing.
3.
4.  // Uncomment to get full stacktrace output. Sometimes helpful, usually not.

```

```
5. // Error.stackTraceLimit = Infinity; //
6.
7. jasmine.DEFAULT_TIMEOUT_INTERVAL = 1000;
8.
9. var builtPath = '/base/app/';
10.
11. __karma__.loaded = function () { };
12.
13. function isJsFile(path) {
14.   return path.slice(-3) == '.js';
15. }
16.
17. function isSpecFile(path) {
18.   return /\.spec\.(.*\.)?js$/.test(path);
19. }
20.
21. function isBuiltFile(path) {
22.   return isJsFile(path) && (path.substr(0, builtPath.length) == builtPath);
23. }
24.
25. var allSpecFiles = Object.keys(window.__karma__.files)
26.   .filter(isSpecFile)
27.   .filter(isBuiltFile);
28.
29. System.config({
30.   baseURL: '/base',
31.   // Extend usual application package list with test folder
32.   packages: { 'testing': { main: 'index.js', defaultExtension: 'js' } },
33.
34.   // Assume npm: is set in `paths` in systemjs.config
35.   // Map the angular testing umd bundles
36.   map: {
37.     '@angular/core/testing': 'npm:@angular/core/bundles/core-testing.umd.js',
38.     '@angular/common/testing': 'npm:@angular/common/bundles/common-
39.       testing.umd.js',
40.     '@angular/compiler/testing': 'npm:@angular/compiler/bundles/compiler-
41.       testing.umd.js',
42.     '@angular/platform-browser/testing': 'npm:@angular/platform-
43.       browser/bundles/platform-browser-testing.umd.js',
44.     '@angular/platform-browser-dynamic/testing': 'npm:@angular/platform-
45.       browser-dynamic/bundles/platform-browser-dynamic-testing.umd.js',
46.     '@angular/http/testing': 'npm:@angular/http/bundles/http-testing.umd.js',
47.     '@angular/router/testing': 'npm:@angular/router/bundles/router-
48.       testing.umd.js',
49.     '@angular/forms/testing': 'npm:@angular/forms/bundles/forms-
50.       testing.umd.js',
51.   },
52. }
```

```
46.    });
47.
48.    System.import('systemjs.config.js')
49.      .then(importSystemJsExtras)
50.      .then(init TestBed)
51.      .then(initTesting);
52.
53.    /** Optional SystemJS configuration extras. Keep going w/o it */
54.    function importSystemJsExtras(){
55.      return System.import('systemjs.config.extras.js')
56.        .catch(function(reason) {
57.          console.log(
58.            'Warning: System.import could not load the optional
59. "systemjs.config.extras.js". Did you omit it by accident? Continuing without
60. it.'
61.          );
62.          console.log(reason);
63.        });
64.
65.    function init TestBed(){
66.      return Promise.all([
67.        System.import('@angular/core/testing'),
68.        System.import('@angular/platform-browser-dynamic/testing')
69.      ])
70.
71.      .then(function (providers) {
72.        var coreTesting = providers[0];
73.        var browserTesting = providers[1];
74.
75.        coreTesting.TestBed.initTestEnvironment(
76.          browserTesting.BrowserDynamicTestingModule,
77.          browserTesting.platformBrowserDynamicTesting());
78.      })
79.    }
80.
81.    // Import all spec files and start karma
82.    function initTesting () {
83.      return Promise.all(
84.        allSpecFiles.map(function (moduleName) {
85.          return System.import(moduleName);
86.        })
87.      )
88.      .then(__karma__.start, __karma__.error);
89.    }
90.
```

这个 shim 文件首先加载了 SystemJS 的配置，然后是 Angular 2 的测试支持库，然后是应用本身的规约文件。

然后需要修改 Karma 配置，来让它使用本应用的根目录作为基础目录 (base directory)，而不是 app。

karma.conf.js

```
basePath: './',
```

一旦这些完成了，我们就能加载 SystemJS 和其它依赖，并切换配置文件来加载那些应用文件，而 **不用** 在 Karma 页面中包含它们。我们要让这个 shim 文件和 SystemJS 去加载它们。

karma.conf.js

```
1. // System.js for module loading
2. 'node_modules/systemjs/dist/system.src.js',
3.
4. // Polyfills
5. 'node_modules/core-js/client/shim.js',
6. 'node_modules/reflect-metadata/Reflect.js',
7.
8. // zone.js
9. 'node_modules/zone.js/dist/zone.js',
10. 'node_modules/zone.js/dist/long-stack-trace-zone.js',
11. 'node_modules/zone.js/dist/proxy.js',
12. 'node_modules/zone.js/dist/sync-test.js',
13. 'node_modules/zone.js/dist/jasmine-patch.js',
14. 'node_modules/zone.js/dist/async-test.js',
15. 'node_modules/zone.js/dist/fake-async-test.js',
16.
17. // RxJS.
18. { pattern: 'node_modules/rxjs/**/*.js', included: false, watched: false },
19. { pattern: 'node_modules/rxjs/**/*.js.map', included: false, watched: false
  },
20.
21. // Angular 2 itself and the testing library
22. {pattern: 'node_modules/@angular/**/*.js', included: false, watched: false},
23. {pattern: 'node_modules/@angular/**/*.js.map', included: false, watched:
  false},
24.
25. {pattern: 'systemjs.config.js', included: false, watched: false},
```

```

26.   'karma-test-shim.js',
27.
28.   {pattern: 'app/**/*.module.js', included: false, watched: true},
29.   {pattern: 'app/*!(.module|.spec).js', included: false, watched: true},
30.   {pattern: 'app/!(bower_components)/**/*!(.module|.spec).js', included: false,
31.     watched: true},
32.   {pattern: 'app/**/*.spec.js', included: false, watched: true},
33.   {pattern: '**/*.html', included: false, watched: true},

```

由于 Angular 2 组件中的 HTML 模板也同样要被加载，所以我们得帮 Karma 一把，帮它在正确的路径下找到这些模板：

karma.conf.js

```

1. // proxied base paths for loading assets
2. proxies: {
3.   // required for component assets fetched by Angular's compiler
4.   "/phone-detail": '/base/app/phone-detail',
5.   "/phone-list": '/base/app/phone-list'
6. },

```

如果产品代码被切换到了 Angular 2，单元测试文件本身也需要切换过来。对勾 (checkmark) 管道的规约可能是最简单的，因为它没有任何依赖：

app/core/checkmark/checkmark.pipe.spec.ts

```

1. import { CheckmarkPipe } from './checkmark.pipe';
2.
3. describe('CheckmarkPipe', function() {
4.
5.   it('should convert boolean values to unicode checkmark or cross', function()
6.   {
7.     const checkmarkPipe = new CheckmarkPipe();
8.     expect(checkmarkPipe.transform(true)).toBe('\u2713');
9.     expect(checkmarkPipe.transform(false)).toBe('\u2718');
10.    });
11.  });

```

Phone 服务的测试会牵扯到一点别的。我们需要把模拟版的 Angular 1 \$httpBackend 服务切换到模拟版的 Angular 2 Http 后端。

app/core/phone/phone.service.spec.ts

```
1. import { inject, TestBed } from '@angular/core/testing';
2. import {
3.   Http,
4.   BaseRequestOptions,
5.   ResponseOptions,
6.   Response
7. } from '@angular/http';
8. import { MockBackend, MockConnection } from '@angular/http/testing';
9. import { Phone, PhoneData } from './phone.service';
10.
11. describe('Phone', function() {
12.   let phone: Phone;
13.   let phonesData: PhoneData[] = [
14.     {name: 'Phone X', snippet: '', images: []},
15.     {name: 'Phone Y', snippet: '', images: []},
16.     {name: 'Phone Z', snippet: '', images: []}
17. ];
18.   let mockBackend: MockBackend;
19.
20.   beforeEach(() => {
21.     TestBed.configureTestingModule({
22.       providers: [
23.         Phone,
24.         MockBackend,
25.         BaseRequestOptions,
26.         { provide: Http,
27.           useFactory: (backend: MockBackend, options: BaseRequestOptions) =>
28.             new Http(backend, options),
29.             deps: [MockBackend, BaseRequestOptions]
30.           }
31.         ]
32.       });
33.     });
34.   beforeEach(inject([MockBackend, Phone], (_mockBackend_: MockBackend,
35.   _phone_: Phone) => {
36.     mockBackend = _mockBackend_;
37.     phone = _phone_;
38.   }));
39.
```

```

39.     it('should fetch the phones data from `/phones/phones.json`', (done: () =>
40.       void) => {
41.         mockBackend.connections.subscribe((conn: MockConnection) => {
42.           conn.mockRespond(new Response(new ResponseOptions({body:
43.             JSON.stringify(phonesData)})));
44.         });
45.         phone.query().subscribe(result => {
46.           expect(result).toEqual(phonesData);
47.           done();
48.         });
49.       });

```

对于组件的规约，我们可以模拟出 `Phone` 服务本身，并且让它提供电话的数据。我们可以对这些组件使用 Angular 的组件单元测试 API。

app/phone-detail/phone-detail.component.spec.ts

```

1.  import { ActivatedRoute } from '@angular/router';
2.
3.  import { Observable } from 'rxjs/Rx';
4.
5.  import { async, TestBed } from '@angular/core/testing';
6.
7.  import { PhoneDetailComponent } from './phone-detail.component';
8.  import { Phone, PhoneData } from '../core/phone/phone.service';
9.  import { CheckmarkPipe } from '../core/checkmark/checkmark.pipe';
10.
11. function xyzPhoneData(): PhoneData {
12.   return {
13.     name: 'phone xyz',
14.     snippet: '',
15.     images: ['image/url1.png', 'image/url2.png']
16.   };
17. }
18.
19. class MockPhone {
20.   get(id: string): Observable<PhoneData> {
21.     return Observable.of(xyzPhoneData());
22.   }
23. }
24.

```

```

25.
26. class ActivatedRouteMock {
27.   constructor(public snapshot: any) {}
28. }
29.
30.
31. describe('PhoneDetailComponent', () => {
32.
33.
34.   beforeEach(async(() => {
35.     TestBed.configureTestingModule({
36.       declarations: [ CheckmarkPipe, PhoneDetailComponent ],
37.       providers: [
38.         { provide: Phone, useClass: MockPhone },
39.         { provide: ActivatedRoute, useValue: new ActivatedRoute({ params:
40.           { 'phoneId': 1 } }) }
41.       ]
42.     })
43.     .compileComponents();
44.   });
45.
46.   it('should fetch phone detail', () => {
47.     const fixture = TestBed.createComponent(PhoneDetailComponent);
48.     fixture.detectChanges();
49.     let compiled = fixture.debugElement.nativeElement;
50.
51.     expect(compiled.querySelector('h1').textContent).toContain(xyzPhoneData().name
52.   });
53. });

```

app/phone-list/phone-list.component.spec.ts

```

1. import { NO_ERRORS_SCHEMA } from '@angular/core';
2. import { ActivatedRoute } from '@angular/router';
3. import { Observable } from 'rxjs/Rx';
4. import { async, ComponentFixture, TestBed } from '@angular/core/testing';
5. import { SpyLocation } from '@angular/common/testing';
6.
7. import { PhoneListComponent } from './phone-list.component';
8. import { Phone, PhoneData } from '../core/phone/phone.service';
9.
10. class ActivatedRouteMock {

```

```
11.     constructor(public snapshot: any) {}
12. }
13.
14. class MockPhone {
15.     query(): Observable<PhoneData[]> {
16.         return Observable.of([
17.             {name: 'Nexus S', snippet: '', images: []},
18.             {name: 'Motorola DROID', snippet: '', images: []}
19.         ]);
20.     }
21. }
22.
23. let fixture: ComponentFixture<PhoneListComponent>;
24.
25. describe('PhoneList', () => {
26.
27.     beforeEach(async(() => {
28.         TestBed.configureTestingModule({
29.             declarations: [ PhoneListComponent ],
30.             providers: [
31.                 { provide: ActivatedRoute, useValue: new ActivatedRouteMock({ params:
32. { 'phoneId': 1 } }) },
33.                 { provide: Location, useClass: SpyLocation },
34.                 { provide: Phone, useClass: MockPhone },
35.             ],
36.             schemas: [ NO_ERRORS_SCHEMA ]
37.         })
38.         .compileComponents();
39.     }));
40.
41.     beforeEach(() => {
42.         fixture = TestBed.createComponent(PhoneListComponent);
43.     });
44.
45.     it('should create "phones" model with 2 phones fetched from xhr', () => {
46.         fixture.detectChanges();
47.         let compiled = fixture.debugElement.nativeElement;
48.         expect(compiled.querySelectorAll('.phone-list-item').length).toBe(2);
49.         expect(
50.             compiled.querySelector('.phone-list-item:nth-child(1)').textContent
51.         ).toContain('Motorola DROID');
52.         expect(
53.             compiled.querySelector('.phone-list-item:nth-child(2)').textContent
54.         ).toContain('Nexus S');
55.     });
56.    xit('should set the default value of orderProp model', () => {
```

```

57.     fixture.detectChanges();
58.     let compiled = fixture.debugElement.nativeElement;
59.     expect(
60.       compiled.querySelector('select option:last-child').selected
61.     ).toBe(true);
62.   });
63.
64. });

```

最后，当我们切换到 Angular 2 路由时，我们需要重新过一遍这些组件测试。对详情组件来说，我们需要提供一个 Angular 2 `RouteParams` 的 mock 对象，而不再用 Angular 1 中的 `$routeParams`。

app/phone-detail/phone-detail.component.spec.ts

```

1. import { ActivatedRoute } from '@angular/router';
2.
3. /* . . . */
4.
5. class ActivatedRouteMock {
6.   constructor(public snapshot: any) {}
7. }
8.
9. /* . . . */
10.
11. beforeEach(async(() => {
12.   TestBed.configureTestingModule({
13.     declarations: [ CheckmarkPipe, PhoneDetailComponent ],
14.     providers: [
15.       { provide: Phone, useClass: MockPhone },
16.       { provide: ActivatedRoute, useValue: new ActivatedRouteMock({ params:
17.         { 'phoneId': 1 } }) }
18.     ]
19.   })
20.   .compileComponents();
21. }));

```

对于电话列表组件来说，我们需要为路由器本身略作设置，以便它的路由链接（`routerLink`）指令能够正常工作。

app/phone-list/phone-list.component.spec.ts

```
1. import { NO_ERRORS_SCHEMA } from '@angular/core';
2. import { ActivatedRoute } from '@angular/router';
3. import { observable } from 'rxjs/Rx';
4. import { async, ComponentFixture, TestBed } from '@angular/core/testing';
5. import { SpyLocation } from '@angular/common/testing';
6.
7. import { PhoneListComponent } from './phone-list.component';
8. import { Phone, PhoneData } from '../core/phone/phone.service';
9.
10. /* . . . */
11.
12. beforeEach(async(() => {
13.   TestBed.configureTestingModule({
14.     declarations: [ PhoneListComponent ],
15.     providers: [
16.       { provide: ActivatedRoute, useValue: new ActivatedRouteMock({ params:
17.         { 'phoneId': 1 } }) },
18.       { provide: Location, useClass: SpyLocation },
19.       { provide: Phone, useClass: MockPhone },
20.     ],
21.     schemas: [ NO_ERRORS_SCHEMA ]
22.   })
23.   .compileComponents();
24. }));
25.
26. beforeEach(() => {
27.   fixture = TestBed.createComponent(PhoneListComponent);
28. });
```

WEBPACK简介

使用基于 Webpack 的工具创建 Angular 应用

Webpack 是一个广受欢迎的模块打包器，这个工具用来把程序源码打包到一些方便易用的 **块** 中，以便把这些代码从服务器加载到浏览器中。

它是我们文档中到处使用的 **SystemJS** 的一个优秀替代品。这篇指南会带我们尝尝 Webpack 的滋味，并学习如何在 Angular 程序中使用它。

目录

[什么是 Webpack ?](#)

- [入口与输出](#)
- [加载器](#)
- [插件](#)

[配置 Webpack](#)

- [公共配置](#)
- [开发环境配置](#)
- [生产环境配置](#)
- [测试环境配置](#)

[试一下](#)

总结

什么是 Webpack ?

Webpack 是一个强力的模块打包器。 所谓 **包 (bundle)** 就是一个 JavaScript 文件，它把一堆 **资源 (assets)** 合并在一起，以便它们可以在同一个文件请求中发回给客户端。 包中可以包含 JavaScript、 CSS 样式、 HTML 以及很多其它类型的文件。

Webpack 会遍历你应用中的所有源码，查找 `import` 语句，构建出依赖图谱，并产出一个（或多个）**包**。通过“加载器 (loaders)”插件，Webpack 可以对各种非 JavaScript 文档进行预处理和最小化 (Minify)，比如 TypeScript、 SASS 和 LESS 文件等。

我们通过一个 JavaScript 配置文件 `webpack.config.js` 来决定 Webpack 做什么以及如何做。

入口与输出

我们给 Webpack 提供一个或多个 **入口** 文件，来让它查找与合并那些从这些入口点发散出去的依赖。在下面这个例子中，我们的入口点是该应用的根文件 `src/app.ts`：

webpack.config.js (single entry)

```
entry: {  
  app: 'src/app.ts'  
}
```

Webpack 探查那个文件，并且递归遍历它的 `import` 依赖。

src/app.ts

```
import { Component } from '@angular/core';  
  
@Component({  
  ...  
})  
export class AppComponent {}
```

这里，Webpack 看到我们正在导入 `@angular/core`，于是就将这个文件加入到它的依赖列表里，为（有可能）把该文件打进包中做准备。它打开 `@angular/core` 并追踪由 **该文件的 import 语句构成的网络**，直到构建出从 `app.ts` 往下的整个依赖图谱。

然后它把这些文件 **输出** 到当前配置所指定的 **包文件** `app.js` 中：

webpack.config.js (single output)

```
output: {  
  filename: 'app.js'  
}
```

这个 `app.js` 输出包是个单一的 JavaScript 文件，它包含程序的源码及其所有依赖。后面我们将再 `index.html` 中用 `<script>` 标签来加载它。

多重包

我们可能不会希望把所有东西打进一个巨型包，而更喜欢把多变的应用代码从相对稳定的第三方提供商模块中分离出来。

所以要修改配置，以获得两个入口点：`app.ts` 和 `vendor.ts`：

webpack.config.js (two entries)

```
entry: {  
  app: 'src/app.ts',  
  vendor: 'src/vendor.ts'  
},  
  
output: {  
  filename: '[name].js'  
}
```

Webpack 会构造出两个独立的依赖图谱，并产出 **两个** 包文件：一个叫做 `app.js`，它只包含我们的应用代码；另一个叫做 `vendor.js`，它包含所有的提供商依赖。

在输出文件名中出现的 `[name]` 是一个 Webpack 的 **占位符**，它将被替换为入口点的名字，分别是 `app` 和 `vendor`。

我们需要一个插件来完成此项工作，本章 [后面](#) 的部分会覆盖它。

我们以前见过 `app.ts`，就不再赘述了。还要再写一个 `vendor.ts`，让它导入我们要用的提供商模块：

src/vendor.ts

```
// Angular
import '@angular/platform-browser';
import '@angular/platform-browser-dynamic';
import '@angular/core';
import '@angular/common';
import '@angular/http';
import '@angular/router';

// RXJS
import 'rxjs';

// other vendors for example jQuery, Lodash or Bootstrap
// You can import js, ts, css, sass, ...
```

加载器 (Loader)

Webpack 可以打包任何类型的文件：JavaScript、TypeScript、CSS、SASS、LESS、图片、HTML 以及字体文件等等。Webpack 本身并不知道该如何处理这些非 JavaScript 文件。我们要通过 **加载器** 来告诉它如何把这些文件处理成 JavaScript 文件。在这里，我们为 TypeScript 和 CSS 文件配置了加载器。

webpack.config.js (two entries)

```
loaders: [
  {
    test: /\.ts$/,
    loaders: 'ts'
  },
  {
    test: /\.css$/,
    loaders: 'style!css'
  }
]
```

当 Webpack 遇到像这样的 `import` 语句时.....

```
import { AppComponent } from './app.component.ts';
import 'uiframework/dist/uiframework.css';
```

.....它会使用 `test` 后面的正则表达式进行模式匹配。如果一个模式匹配上文件名，Webpack 就用它所关联的加载器处理这个文件。

第一个 `import` 文件匹配上了 `.ts` 模式，于是 Webpack 就用 `awesome-typescript-loader` 加载器处理它。导入的文件没有匹配上第二个模式，于是它的加载器就被忽略了。

第二个 `import` 匹配上了第二个 `.css` 模式，它有两个用感叹号字符 (!) 串联起来的加载器。Webpack 会 **从右到左** 逐个应用串联的加载器，于是它先应用了 `css` 加载器（用来平面化 CSS 的 `@import` 和 `url(...)` 语句），然后应用了 `style` 加载器（用来把 css 追加到页面上的 `<style>` 元素中）。

插件

Webpack 有一条构建流水线，它被划分成多个经过精心定义的阶段 (phase)。我们可以把插件（比如 `uglify` 代码最小化插件）挂到流水线上：

```
plugins: [
  new webpack.optimize.UglifyJsPlugin()
]
```

配置 Webpack

经过简短的培训之后，我们准备为 Angular 应用构建一份自己的 Webpack 配置了。

从设置开发环境开始。

创建一个 新的项目文件夹

```
mkdir angular-webpack
cd angular-webpack
```

把下列文件添加到根目录下：

```
1.  {
2.    "name": "angular2-webpack",
3.    "version": "1.0.0",
4.    "description": "A webpack starter for Angular",
5.    "scripts": {
6.      "start": "webpack-dev-server --inline --progress --port 8080",
7.      "test": "karma start",
8.      "build": "rimraf dist && webpack --config config/webpack.prod.js -
   -progress --profile --bail"
9.    },
10.   "licenses": [
11.     {
12.       "type": "MIT",
13.       "url":
14.         "https://github.com/angular/angular.io/blob/master/LICENSE"
15.     },
16.     "dependencies": {
17.       "@angular/common": "~2.1.1",
```

```
18.   "@angular/compiler": "~2.1.1",
19.   "@angular/core": "~2.1.1",
20.   "@angular/forms": "~2.1.1",
21.   "@angular/http": "~2.1.1",
22.   "@angular/platform-browser": "~2.1.1",
23.   "@angular/platform-browser-dynamic": "~2.1.1",
24.   "@angular/router": "~3.1.1",
25.   "core-js": "^2.4.1",
26.   "rxjs": "5.0.0-beta.12",
27.   "zone.js": "^0.6.25"
28. },
29. "devDependencies": {
30.   "@types/core-js": "^0.9.34",
31.   "@types/node": "^6.0.45",
32.   "@types/jasmine": "^2.5.35",
33.   "angular2-template-loader": "^0.4.0",
34.   "awesome-typescript-loader": "^2.2.4",
35.   "css-loader": "^0.23.1",
36.   "extract-text-webpack-plugin": "^1.0.1",
37.   "file-loader": "^0.8.5",
38.   "html-loader": "^0.4.3",
39.   "html-webpack-plugin": "^2.15.0",
40.   "jasmine-core": "^2.4.1",
41.   "karma": "^1.2.0",
42.   "karma-jasmine": "^1.0.2",
43.   "karma-phantomjs-launcher": "^1.0.2",
44.   "karma-sourcemap-loader": "^0.3.7",
45.   "karma-webpack": "^1.8.0",
46.   "null-loader": "^0.1.1",
47.   "phantomjs-prebuilt": "^2.1.7",
48.   "raw-loader": "^0.5.1",
49.   "rimraf": "^2.5.2",
50.   "style-loader": "^0.13.1",
51.   "typescript": "^2.0.3",
52.   "webpack": "^1.13.0",
53.   "webpack-dev-server": "^1.14.1",
54.   "webpack-merge": "^0.14.0"
55. }
56. }
```

这些文件及其内容多数都在其它 Angular 文档中见过了，应该比较熟悉。

要了解 `package.json`，请到 [npm 包](#) 一章。除了那一章列出的包之外，我们还需要用到 Webpack 的包。

要了解 `tsconfig.json`，参见 [TypeScript 配置](#) 一章。

打开终端 / 控制台窗口，并通过 `npm install` 命令安装这些 npm 包。

公共配置

我们可以为开发、产品和测试环境定义分别各自的配置文件。但三者总会有一些公共配置。于是我们把那些公共的配置收集到一个名叫 `webpack.common.js` 的独立文件中。

来看下入口文件，用一个小节的时间了解下它的内容：

config/webpack.common.js

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var helpers = require('./helpers');

module.exports = {
  entry: {
    'polyfills': './src/polyfills.ts',
    'vendor': './src/vendor.ts',
    'app': './src/main.ts'
  },
  resolve: {
    extensions: ['', '.ts', '.js']
  },
  module: {
    loaders: [
      {
        test: /\.ts$/,
        loader: 'ts-loader',
        exclude: /node_modules/
      }
    ]
  }
};
```

```

{
  test: /\.ts$/,
  loaders: ['awesome-typescript-loader', 'angular2-template-
loader']
},
{
  test: /\.html$/,
  loader: 'html'
},
{
  test: /\.(png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico)$/,
  loader: 'file?name=assets/[name].[hash].[ext]'
},
{
  test: /\.css$/,
  exclude: helpers.root('src', 'app'),
  loader: ExtractTextPlugin.extract('style', 'css?sourceMap')
},
{
  test: /\.css$/,
  include: helpers.root('src', 'app'),
  loader: 'raw'
}
],
},
plugins: [
  new webpack.optimize.CommonsChunkPlugin({
    name: ['app', 'vendor', 'polyfills']
  }),

  new HtmlWebpackPlugin({
    template: 'src/index.html'
  })
];
};

```

Webpack 是一个基于 NodeJS 的工具，所以它的配置文件就是一个 JavaScript 的 CommonJS 模块文件，它像常规写法一样以 `require` 语句开始。

这个配置项导出了几个对象，从前面说过的 **入口点 (entry)** 开始：

config/webpack.common.js

```
entry: {  
  'polyfills': './src/polyfills.ts',  
  'vendor': './src/vendor.ts',  
  'app': './src/main.ts'  
},
```

我们正在把应用拆成三个包：

- polyfills - 我们在大多数现代浏览器中运行 Angular 程序时需要的标准填充物。
- vendor - 我们需要的提供商文件：Angular、Lodash、bootstrap.css
- app - 我们的应用代码。

加载填充物 (POLYFILLS)

早点加载 Zone.js，紧跟在其它 ES6 和 metadata 垫片 (shim) 之后。

本程序将需要 import 十多个 (如果不是上百个的话) JavaScript 和 TypeScript 文件。我们 可以 带着明确的扩展名来写这些 import 语句，就像下面的例子中这样：

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

但是大多数 import 语句完全不会去引用扩展名。所以我们要告诉 Webpack 如何通过查找匹配的文件来 **解析** 模块文件的加载请求：

- 一个明确的扩展名 (通过一个空白的扩展名字符串 "" 标记出来)，或者
- .js 扩展名 (为了查找标准的 JavaScript 文件和预编译过的 TypeScript 文件)，或者
- .ts 扩展名。

config/webpack.common.js

```
resolve: {
  extensions: ['', '.ts', '.js']
},
```

我们以后还可能会添加 `.css` 和 `.html` ——如果希望 Webpack 也用无扩展名的方式去解析 **那些** 扩展名的话。

接下来我们开始指定加载器：

config/webpack.common.js

```
module: {
  loaders: [
    {
      test: /\.ts$/,
      loaders: ['awesome-typescript-loader', 'angular2-template-loader']
    },
    {
      test: /\.html$/,
      loader: 'html'
    },
    {
      test: /\.(png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico)$/,
      loader: 'file?name=assets/[name].[hash].[ext]'
    },
    {
      test: /\.css$/,
      exclude: helpers.root('src', 'app'),
      loader: ExtractTextPlugin.extract('style', 'css?sourceMap')
    },
    {
      test: /\.css$/,
      include: helpers.root('src', 'app'),
      loader: 'raw'
    }
  ]
}
```

```
]  
},
```

- awesome-typescript-loader - 一个用于把 TypeScript 代码转译成 ES5 的加载器，它会由 `tsconfig.json` 文件提供指导
- angular2-template-loader - 用于加载 Angular 组件的模板和样式
- html - 为组件模板准备的加载器
- images/fonts - 图片和字体文件也能被打包。
- css - 第一个模式匹配应用级样式，第二个模式匹配组件局部样式（就是在组件元数据的 `styleUrls` 属性中指定的那些）。

第一个模式排除了 `/src/app` 目录下的 `.css` 文件，因为那里放着我们的组件局部样式。它只包含了那些位于 `/src` 及其上级目录的 `.css` 文件，那里是应用级样式。`ExtractTextPlugin`（后面会讲到）使用 `style` 和 `css` 加载器来处理这些文件。

第二个模式过滤器是给组件局部样式的，并通过 `raw` 加载器把它们加载成字符串——那是 Angular 期望通过元数据的 `styleUrls` 属性来指定样式的形式。

多重加载器也能使用数组形式串联起来。

最后我们来添加两个插件：

config/webpack.common.js

```
plugins: [  
  new webpack.optimize.CommonsChunkPlugin({  
    name: ['app', 'vendor', 'polyfills']  
  }),
```

```
new HtmlWebpackPlugin({
  template: 'src/index.html'
})
]
```

COMMONSCHUNKPLUGIN

我们希望 `app.js` 包中只包含应用代码，而 `vendor.js` 包中只包含提供商代码。

应用代码中 `import` 了提供商代码。Webpack 还没有智能到自动把提供商代码排除在 `app.js` 包之外。`CommonsChunkPlugin` 插件能完成此工作。

这里标记出了三个 **块** 之间的等级体系：`app` -> `vendor` -> `polyfills`。当 Webpack 发现 `app` 与 `vendor` 有共享依赖时，就把它们从 `app` 中移除。在 `vendor` 和 `polyfills` 之间有共享依赖时也同样如此（虽然它们没啥可共享的）。

HTMLWEBPACKPLUGIN

Webpack 生成了一些 js 和 css 文件。虽然我们可以手动 把它们插入到 `index.html` 中，但那样既枯燥又容易出错。Webpack 可以通过 `HtmlWebpackPlugin` 自动为我们注入那些 `script` 和 `link` 标签。

环境相关的配置

`webpack.common.js` 配置做了大部分繁重的工作。通过合并它们特有的配置，我们可以基于 `webpack.common` 为目标环境创建独立的、环境相关的配置文件。

这些文件越小越简单越好。

开发环境配置

下面是开发环境的而配置文件 `webpack.dev.js` :

config/webpack.dev.js

```
var webpackMerge = require('webpack-merge');
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var commonConfig = require('./webpack.common.js');
var helpers = require('./helpers');

module.exports = webpackMerge(commonConfig, {
  devtool: 'cheap-module-eval-source-map',

  output: {
    path: helpers.root('dist'),
    publicPath: 'http://localhost:8080/',
    filename: '[name].js',
    chunkFilename: '[id].chunk.js'
  },

  plugins: [
    new ExtractTextPlugin('[name].css')
  ],

  devServer: {
    historyApiFallback: true,
    stats: 'minimal'
  }
});
```

开发环境下的构建依赖于 Webpack 的开发服务器，我们在靠近文件底部的地方配置了它。

虽然我们告诉 Webpack 把输出包放到 `dist` 目录，但实际上开发服务器把这些包都放在了内存里，而不会把它们写到硬盘中。所以在 `dist` 目录下是找不到任何文件的（至少现在这个开发环境下构建时没有）。

`HtmlWebpackPlugin` (由 `webpack.common.js` 引入) 插件使用了 `publicPath` 和 `filename` 设置，来向 `index.html` 中插入适当的 `<script>` 和 `<link>` 标签。

我们这些 CSS 默认情况下会被埋没在 JavaScript 包中。 ExtractTextPlugin 会把它们提取成外部 .css 文件，这样 HtmlWebpackPlugin 插件就会转而把一个 <link> 标签写进 index.html 了。

要了解本文件中这些以及其它配置项的详情，请参阅 Webpack 文档。

抓取本指南底部的应用代码，并试一试：

```
npm start
```

产品环境配置

产品环境 下的配置和 开发环境 下的配置很相似……除了一些关键的改动。

config/webpack.prod.js

```
var webpack = require('webpack');
var webpackMerge = require('webpack-merge');
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var commonConfig = require('./webpack.common.js');
var helpers = require('./helpers');

const ENV = process.env.NODE_ENV = process.env.ENV = 'production';

module.exports = webpackMerge(commonConfig, {
  devtool: 'source-map',

  output: {
    path: helpers.root('dist'),
    publicPath: '/',
    filename: '[name].[hash].js',
    chunkFilename: '[id].[hash].chunk.js'
  },

  htmlLoader: {
    minimize: false // workaround for ng2
  },

  plugins: [
```

```

new webpack.NoErrorsPlugin(),
new webpack.optimize.DedupePlugin(),
new webpack.optimize.UglifyJsPlugin({ //  

https://github.com/angular/angular/issues/10618  

  mangle: {  

    keep_fnames: true  

  }  

}),
new ExtractTextPlugin('[name].[hash].css'),
new webpack.DefinePlugin({
  'process.env': {
    'ENV': JSON.stringify(ENV)
  }
})
];
);

```

我们不会在产品环境下使用开发服务器，而是希望把应用程序及其依赖都部署到一个真实的产品服务器中。

这次，输出的包文件就被真的放到 `dist` 目录下了。

Webpack 生成的文件名中带有“缓存无效哈希 (cache-busting hash)”。感谢 `HtmlWebpackPlugin` 插件，当这些哈希值变化时，我们不用去更新 `index.html` 了。

还有一些别的插件：

- **NoErrorsPlugin** - 如果出现任何错误，就终止构建。
- **DedupePlugin** - 检测完全相同（以及几乎完全相同）的文件，并把它们从输出中移除。
- **UglifyJsPlugin** - 最小化 (minify) 生成的包。
- **ExtractTextPlugin** - 把内嵌的 css 抽取成外部文件，并为其文件名添加“缓存无效哈希”。
- **DefinePlugin** - 用来定义环境变量，以便我们在自己的程序中引用它。

感谢 **DefinePlugin** 和顶部定义的 `ENV` 变量，我们就可以像这样启用 Angular 的产品模式了：

```
if (process.env.ENV === 'production') {  
    enableProdMode();  
}
```

抓取本指南底部的应用代码，并试一试：

```
npm run build
```

测试环境配置

我们并不需要使用很多配置项来运行单元测试。也不需要在开发环境和产品环境下引入的那些加载器和插件。如果有可能拖慢执行速度，甚至都不需要在单元测试中加载和处理应用全局样式文件，所以我们用一个 `null` 加载器来处理所有 CSS。

我们可以把测试环境的配置合并到 `webpack.common` 配置中，并且改写不想要或不需要的部分。但是从一个全新的配置开始可能更简单。

config/webpack.test.js

```
var helpers = require('./helpers');  
  
module.exports = {  
    devtool: 'inline-source-map',  
  
    resolve: {  
        extensions: ['', '.ts', '.js']  
    },  
  
    module: {  
        loaders: [  
            {  
                test: /\.ts$/,  
                loaders: ['awesome-typescript-loader', 'angular2-template-  
loader']  
            },  
            {  
                test: /\.html$/,  
                loader: 'html'  
            }  
        ]  
    }  
};
```

```

    loader: 'html'

  },
  {
    test: /\.png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico$/,
    loader: 'null'
  },
  {
    test: /\.css$/,
    exclude: helpers.root('src', 'app'),
    loader: 'null'
  },
  {
    test: /\.css$/,
    include: helpers.root('src', 'app'),
    loader: 'raw'
  }
]
}
}

```

这里是我们的 Karma 配置：

config/karma.conf.js

```

var webpackConfig = require('./webpack.test');

module.exports = function (config) {
  var _config = {
    basePath: '',
    frameworks: ['jasmine'],
    files: [
      {pattern: './config/karma-test-shim.js', watched: false}
    ],
    preprocessors: {
      './config/karma-test-shim.js': ['webpack', 'sourcemap']
    },
  }
}

```

```

    webpack: webpackConfig,

    webpackMiddleware: {
        stats: 'errors-only'
    },

    webpackServer: {
        noInfo: true
    },

    reporters: ['progress'],
    port: 9876,
    colors: true,
    LogLevel: config.LOG_INFO,
    autowatch: false,
    browsers: ['PhantomJS'],
    singleRun: true
};

config.set(_config);
};

```

我们告诉 Karma 使用 Webpack 来运行测试。

我们不用预编译 TypeScript , Webpack 随时在内存中转译我们的 TypeScript 文件，并且把产出的 JS 直接反馈给 Karma 。 硬盘上没有任何临时文件。

`karma-test-shim` 告诉 Karma 哪些文件需要预加载，首要的是：带有“测试版提供商”的 Angular 测试框架是每个应用都希望预加载的。

config/karma-test-shim.js

```

Error.stackTraceLimit = Infinity;

require('core-js/es6');
require('core-js/es7/reflect');

require('zone.js/dist/zone');
require('zone.js/dist/long-stack-trace-zone');
require('zone.js/dist/proxy');

```

```

require('zone.js/dist/sync-test');
require('zone.js/dist/jasmine-patch');
require('zone.js/dist/async-test');
require('zone.js/dist/fake-async-test');

var appContext = require.context('../src', true, /\.spec\.ts/);

appContext.keys().forEach(appContext);

var testing = require('@angular/core/testing');
var browser = require('@angular/platform-browser-dynamic/testing');

testing.TestBed.initTestEnvironment(browser.BrowserDynamicTestingModule,
  browser.platformBrowserDynamicTesting());

```

注意，我们 **并没有** 明确加载这些应用代码。只是告诉 Webpack 查找并加载我们的测试文件（文件名以 `.spec.ts` 结尾）。每个规约（spec）文件都导入了所有（也只有）它测试所需的应用源码。Webpack 只加载 **那些** 特定的应用文件，而忽略所有其它我们不会测试到的。

抓取本指南底部的应用代码，并试一试：

```
npm test
```

试一试

这里是一个小型应用的全部源码，我们可以用本章中学到的 Webpack 技术打包它们。

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <base href="/">
5. <title>Angular with Webpack</title>
6. <meta charset="UTF-8">
7. <meta name="viewport" content="width=device-width, initial-scale=1">

```
8.      </head>
9.      <body>
10.         <my-app>Loading...</my-app>
11.      </body>
12.     </html>
```

```
1.  import { Component } from '@angular/core';
2.
3.  import '../../../../../public/css/styles.css';
4.
5.  @Component({
6.    selector: 'my-app',
7.    templateUrl: './app.component.html',
8.    styleUrls: ['./app.component.css']
9.  })
10. export class AppComponent { }
```

app.component.html 显示了这个可下载的 Angular Logo 。

```
1. // Angular
2. import '@angular/platform-browser';
3. import '@angular/platform-browser-dynamic';
4. import '@angular/core';
5. import '@angular/common';
6. import '@angular/http';
7. import '@angular/router';
8.
9. // RxJS
10. import 'rxjs';
11.
12. // Other vendors for example jQuery, Lodash or Bootstrap
13. // You can import js, ts, css, sass, ...
```

重点：

- 在 `index.html` 中没有 `<script>` 或 `<link>` 标签。`HtmlWebpackPlugin` 会在运行时动态插入它们。
- `app.component.ts` 中的 `AppComponent` 类简单的用一个 `import` 语句导入了应用级 `CSS`。
- `AppComponent` 组件本身有它自己的 HTML 模板和 CSS 文件。Webpack 通过调用 `require()` 方法加载它们。Webpack 还把那些组件内部的文件打包进了 `app.js` 中。我们在自己的源码中看不到这些调用，这些工作是由幕后的 `angular2-template-loader` 插件完成的。
- `vendor.ts` 由 `import` 提供商依赖的语句组成，它最终决定了 `vender.js` 的内容。本应用也导入这些模块，如果没有 `CommonsChunkPlugin` 插件检测出这种重叠，并且把它们从 `app.js` 中移除，它们就会同时出现在 `app.js` 包中。

总结

我们学到了刚好够用来在开发、测试、产品环境下构建一个小型 Angular 应用的 Webpack 配置知识。

但我们还能做得更多。搜索互联网来获得专家的建议，并扩展你对 Webpack 的认识。

[回到顶部](#)

烹饪宝典

Angular 烹饪宝典

这本“烹饪宝典”提供了在实施中一些常见问题的答案。

每个“烹饪宝典”章节都是一套“菜谱”，每套菜谱都聚焦于一个特定的 Angular 特性或者来自应用方面的挑战，比如数据绑定、跨组件通讯和通过 HTTP 与远程服务器交互等。

这本烹饪宝典只是一个开始。我们正在编写更多“菜谱”。

“烹饪宝典”中的每个章节都链接了一个在线例子，包含“菜谱”的完整代码。

我们有意保持着“菜谱”的简洁，并坚持以代码为中心。

每个“菜谱”都与开发者指南或者 API 指南里面的某个章节相关联。

在这些相关章节里，你可以学到更多关于代码片段背后的目的、上下文和设计选择等深层知识。

反馈

本“烹饪宝典”还在 [持续建设中](#)。

欢迎您的反馈！请点击 Banner 右上角的图标给我们留言。

在 [angular.io](#) github 仓库提交 [文档](#) 问题和 Pull Requests。

在 [angular](#) 提交 [Angular 本身](#) 的问题。

预 (AOT) 编译器

学习如何使用预编译器

这个烹饪指南描述如何通过在构建过程中进行预编译 (Ahead of Time - AoT) 来从根本上提升性能。

目录

- [概览](#)
- [预编译 vs 即时编译](#)
- [用 AoT 进行编译](#)
- [引导](#)
- [摇树优化 \(Tree Shaking \)](#)
- [加载捆文件](#)
- [启动应用服务器](#)
- [源码](#)
- [Source Code](#)
- [英雄指南](#)

概览

Angular 应用主要包含组件和它们的 HTML 模板。在浏览器可以渲染应用之前，组件和模板必须要被 **Angular 编译器** 转换为可以执行的 JavaScript。

观看编译器作者 Tobias Bosch 在 AngularConnect 2016 大会里，对 [Angular 编译器](#) 的演讲。

你可以在浏览器中使用 **即时编译器** (Just-in-Time - JIT) 在运行期间编译该应用，也就是在应用加载时。这是本文档中展示过的标准开发方式。它很不错，但是有自己的缺点。

JIT 编译导致运行期间的性能损耗。由于需要在浏览器中的这个编译过程，视图需要花更长时间才能渲染出来。由于应用包含了 Angular 编译器以及大量实际上并不需要的库代码，所以文件体积也会更大。更大的应用需要更长的时间进行传输，加载也更慢。

编译可以揭露一些组件模板绑定错误。JIT 变异在运行时才揭露它们，有点太晚了。

预编译 (AoT) 通过在构建时编译，在早期截获模板错误，提高应用性能。

预编译 (AoT) vs 即时编译 (JIT)

事实上只有一个 Angular 编译器，AoT 和 JIT 之间的差别仅仅在于编译的时机和所用的工具。使用 AoT，编译器仅仅使用一组库在构建期间运行一次；使用 JIT，编译器在每个用户的每次运行期间都要用不同的库运行一次。

为什么需要 AoT 编译？

渲染得更快

使用 AoT，浏览器下载预编译版本的应用程序。浏览器直接加载运行代码，所以它可以立即渲染该应用，而不用等应用完成首次编译。

需要的异步请求更少

编译器把外部 html 模板和 css 样式表内联到了该应用的 JavaScript 中。消除了用来下载那些源文件的 Ajax 请求。

需要下载的 Angular 框架体积更小

如果应用已经编译过了，自然不需要再下载 Angular 编译器了。该编译器差不多占了 Angular 自身体积的一半儿，所以，省略它可以显著减小应用的体积。

提早检测模板错误

AoT 编译器在构建过程中检测和报告模板绑定错误，避免用户遇到这些错误。

更安全

AoT 编译远在 HTML 模版和组件被服务到客户端之前，将它们编译到 JavaScript 文件。没有模版可以阅读，没有高风险客户端 HTML 或 JavaScript 可利用，所以注入攻击的机会较少。

用 AoT 进行编译

为离线编译做准备

本烹饪书以“[快速起步](#)”作为起始点。只要单独对 `app.component` 文件的类文件和 html 文件做少量修改就可以了。

```
<button (click)="toggleHeading()">Toggle Heading</button>
<h1 *ngIf="showHeading">My First Angular App</h1>

<h3>List of Heroes</h3>
<div *ngFor="let hero of heroes">{{hero}}</div>
```

用下列命令安装少量新的 npm 依赖：

```
npm install @angular/compiler-cli @angular/platform-server --save
```

你要用 `@angular/compiler-cli` 包中提供的 `ngc` 编译器来代替 TypeScript 编译器 (`tsc`)。

`ngc` 是一个 `tsc` 的高仿替代品，它们的配置方式几乎完全一样。

`ngc` 需要自己的带有 AoT 专用设置的 `tsconfig.json`。把原始的 `tsconfig.json` 拷贝到一个名叫 `tsconfig-aot.json` 的文件中，然后像这样修改它：

`tsconfig-aot.json`

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "es2015",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true
  },
  "files": [
    "app/app.module.ts",
    "app/main.ts"
  ],
  "angularCompilerOptions": {
    "genDir": "aot",
    "skipMetadataEmit": true
  }
}
```

`compilerOptions` 部分只修改了一个属性：** 把 `module` 设置为 `es2015` 。 这一点非常重要，我们会在后面的 [摇树优化](#) 部分解释为什么。

`ngc` 区真正新增的内容是底部的 `angularCompilerOptions` 。 它的 `"genDir"` 属性告诉编译器把编译结果保存在新的 `aot` 目录下。

`"skipMetadataEmit": true` 属性阻止编译器为编译后的应用生成元数据文件。 当输出成 TypeScript 文件时，元数据并不是必须的，因此不需要包含它们。

编译该应用

在命令行中执行下列命令，借助刚安装好的 `ngc` 编译器来启动 AoT 编译：

```
node_modules/.bin/ngc -p tsconfig-aot.json
```

Windows 用户应该双引号 `ngc` 命令：

```
"node_modules/.bin/ngc" -p tsconfig-aot.json
```

`ngc` 希望 `-p` 选项指向一个 `tsconfig.json` 文件，或者一个包含 `tsconfig.json` 文件的目录。

在 `ngc` 完成时，会在 `aot` 目录下看到一组 **NgFactory** 文件（该目录是在 `tsconfig-aot.json` 的 `genDir` 属性中指定的）。

这些工厂文件对于编译后的应用是必要的。每个组件工厂都可以在运行时创建一个组件的实例，其中带有一个原始的类文件和一个用 JavaScript 表示的组件模板。注意，原始的组件类依然是由所生成的这个工厂进行内部引用的。

如果你好奇，可以打开 `aot/app.component.ngfactory.ts` 来看看原始 Angular 模板语法被编译成 TypeScript 时的中间结果。

JiT 编译器在内存中同样会生成这一堆 **NgFactory**，但它们大部分是不可见的。AoT 编译器则会生成在单独的物理文件中。

不要编辑这些 **NgFactory**！重新编译时会替换这些文件，你做的所有修改都会丢失。

引导

AoT 也改变了应用的引导方式。

引导的方式从引导 `AppModule` 改成了引导生成的模块工厂：`AppModuleNgFactory`。

从使用 JIT 编译时的 `platformBrowserDynamic.bootstrap` 换成了 `platformBrowser().bootstrapModuleFactory`，并把 `AppModuleNgFactory` 传进去。

这里是 AoT 版本 `main.ts` 中的引导过程，下一个是你所熟悉的 JIT 版本。

```
1. import { platformBrowser }      from '@angular/platform-browser';
2.
3. import { AppModuleNgFactory }   from '../aot/app/app.module.ngfactory';
4.
5. platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

确保用 `ngc` 进行重新编译！

摇树优化 (Tree Shaking)

AoT 编译为接下来通过一个叫做 **摇树优化** 的过程做好了准备。 摆树优化器从上到下遍历依赖图谱，并且 **摇掉** 用不到的代码，这些代码就像是圣诞树中那些死掉的松针一样。

通过移除源码和库代码中用不到的部分，摇树优化可以大幅缩减应用的下载体积。 事实上，在小型应用中大部分的缩减都是因为筛掉了那些没用到的 Angular 特性。

例如，这个演示程序中没有用到 `@angular/forms` 库中的任何东西，那么也就没有理由去下载这些与表单有关的 Angular 代码了。 摆树优化可以帮你确保这一点。

摇树优化和 AoT 编译是单独的步骤。 摆树优化不仅针对 JavaScript 代码。 AoT 编译会把应用中的大部分都转换成 JavaScript，这种转换会让应用更容易被“摇树优化”。

Rollup

这个烹饪宝典中用来示范的摇树优化工具是 **Rollup**。

Rollup 会通过跟踪 `import` 和 `export` 语句来对本应用进行静态分析。 它所生成的最终代码 **捆** 中会排除那些被导出过但又从未被导入的代码。

Rollup 只能对 ES2015 模块摇树，因为那里有 `import` 和 `export` 语句。

回忆一下，`tsconfig-aot.json` 中曾配置为生成 `ES2015` 的模块。代码本身是否用到了 `ES2015` 语法（例如 `class` 和 `const`）并不重要，重要的是这些代码使用的应该是 `import` 和 `export` 语句，而不是 `require` 语句。

通过下列命令安装 Rollup 依赖：

```
npm install rollup rollup-plugin-node-resolve rollup-plugin-commonjs  
rollup-plugin-uglify --save-dev
```

接下来，在项目根目录新建一个配置文件（`rollup-config.js`），来告诉 Rollup 如何处理应用。本烹饪书配置文件是这样的：

rollup-config.js

```
import rollup from 'rollup'  
import nodeResolve from 'rollup-plugin-node-resolve'  
import commonjs from 'rollup-plugin-commonjs';  
import uglify from 'rollup-plugin-uglify'  
  
export default {  
  entry: 'app/main.js',  
  dest: 'dist/build.js', // output a single application bundle  
  sourceMap: false,  
  format: 'iife',  
  plugins: [  
    nodeResolve({jsnext: true, module: true}),  
    commonjs({  
      include: 'node_modules/rxjs/**',  
    }),  
    uglify()  
  ]  
}
```

它会告诉 Rollup，该应用的入口点是 `app/main.js`。 `dest` 属性告诉 Rollup 要在 `dist` 目录下创建一个名叫 `build.js` 的捆文件。 然后是插件。

Rollup 插件

这些可选插件过滤并转换 Rollup 的输入和输出。

RxJS

Rollup 期望应用的源码使用 `ES2015` 模块。但并不是所有外部依赖都发布成了 `ES2015` 模块。事实上，大多数都不是。它们大多数都发布成了 **CommonJS** 模块。

可观测对象库 **RxJS** 是 Angular 所依赖的基础之一，它就是发布了 `ES5 JavaScript` 的 **CommonJS** 模块。

幸运的是，有一个 Rollup 插件，它会修改 **RxJS**，以使用 Rollup 所需的 `ES import` 和 `export` 语句。然后 Rollup 就可以把该应用中用到的那部分 `RxJS` 代码留在“捆”文件中了。

rollup-config.js (CommonJs to ES2015 Plugin)

```
commonjs({  
  include: 'node_modules/rxjs/**',  
}),
```

最小化

Rollup 做摇树优化时会大幅减小代码体积。最小化过程则会让它更小。本烹饪宝典依赖于 Rollup 插件 **uglify** 来最小化并混淆代码。

rollup-config.js (CommonJs to ES2015 Plugin)

```
uglify()
```

在生产环境中，我们还应该打开 Web 服务器的 gzip 特性来把代码压缩得更小。

运行 Rollup

通过下列命令执行 Rollup 过程：

```
node_modules/.bin/rollup -c rollup-config.js
```

Rollup 可能会记录很多行下面这样的警告信息：

```
The `this` keyword is equivalent to `undefined` at the top  
level of an ES module, and has been rewritten
```

你可以放心的忽略这些警告。

加载捆文件

加载所生成的应用捆文件，并不需要使用像 SystemJS 这样的模块加载器。 移除与 SystemJS 有关的那些脚本吧。 改用 `script` 标签来加载这些捆文件：

index.html (load bundle)

```
<body>  
  <my-app>Loading...</my-app>  
</body>  
  
<script src="dist/build.js"></script>
```

启动应用服务器

你需要一个 Web 服务器来作为应用的宿主。像与文档中其它部分一样，用 **Lite Server** 吧：

```
npm run lite
```

启动了服务器、打开浏览器，应用就出现了。

AoT 快速开始源代码

下面是相关源代码：

```
1.  <button (click)="toggleHeading()">Toggle Heading</button>
2.  <h1 *ngIf="showHeading">My First Angular App</h1>
3.
4.  <h3>List of Heroes</h3>
5.  <div *ngFor="let hero of heroes">{{hero}}</div>
```

英雄指南

上面的例子是快速开始应用的一个简单的变体。在本节中，你将在一个更多内容的应用 - [英雄指南](#) 上使用从 AoT 编译和摇树优化学到的知识。

开发器使用 JIT, 产品期使用 AoT

目前，AoT 编译和摇树优化对开发来说，占用的时间太多了。这将在未来得到改变。当前的最佳实践是在开发器使用 JIT 编译，然后在发布产品前切换到 AoT 编译。

幸运的是，**如果** 你处理了几个关键不同点，源代码可以在没有任何变化时，采取两种方式的任何一种都能编译。

Index.html

JiT 和 AoT 应用的设置和加载非常不一样，以至于它们需要自己单独的 `index.html` 文件。下面是它们的比较：

```
1.  <!DOCTYPE html>
2.  <html>
3.    <head>
4.      <base href="/">
5.      <title>Angular Tour of Heroes</title>
6.      <meta name="viewport" content="width=device-width, initial-
scale=1">
7.
8.      <link rel="stylesheet" href="styles.css">
9.
10.     <script src="shim.min.js"></script>
11.     <script src="zone.min.js"></script>
12.     <script>window.module = 'aot';</script>
13.   </head>
14.
15.   <body>
16.     <my-app>Loading...</my-app>
17.   </body>
18.   <script src="dist/build.js"></script>
19. </html>
```

它们不能在同一个目录。将 AoT 版本放置到 `/aot` 目录。

JiT 版本依靠 `SystemJS` 来加载单个模块，并需要 `reflect-metadata` 垫片。所以它们出现在它的 `index.html` 中。

AoT 版本用一个单独的脚本来加载整个应用 - `aot/dist/build.js`。它不需要 `SystemJS` 和 `reflect-metadata` 垫片，所以它们不会出现在 `index.html` 中。

相对组件的模板路径

AoT 编译器要求 `@Component` 外部模板和 CSS 文件的路径是相对组件的。意思是，`@Component.templateUrl` 的值是一个相对组件类文件 `foo.component.html` 的路径，不管 `foo.component.ts` 在项目的哪个目录。

JiT 应用的 URLs 更加灵活，但是为了与 AoT 编译兼容，坚持使用 **相对组件** 路径。

JiT 编译的应用，使用 SystemJS 加载器，**相对组件路径** 必须要设置

`@Component.moduleId` 属性 为 `module.id` 。 `module` 对象在 AoT 编译的应用运行时的值为 `undefined` 。 应用将会失败，除非你像这样，在 `index.html` 中指定一个全局 `module` 值：

```
<script>window.module = 'aot';</script>
```

设置一个全局 `module` 是暂时的权宜之计。

TypeScript 配置

JiT 编译的应用编译为 `commonjs` 模块。 AoT 编译的应用编译为 **ES2015/ES6** 模块，用来支持摇树优化。而且 AoT 需要它自己的 TypeScript 配置设置。

你将需要单独的 TypeScript 配置文件，像这些：

```
1.  {
2.    "compilerOptions": {
3.      "target": "es5",
4.      "module": "es2015",
5.      "moduleResolution": "node",
6.      "sourceMap": true,
7.      "emitDecoratorMetadata": true,
8.      "experimentalDecorators": true,
9.      "removeComments": false,
10.     "noImplicitAny": true,
11.     "suppressImplicitAnyIndexErrors": true,
12.     "typeRoots": [
13.       "../node_modules/@types/"
14.     ]
15.   },
```

```
16.  
17.     "files": [  
18.         "app/app.module.ts",  
19.         "app/main-aot.ts"  
20.     ],  
21.  
22.     "angularCompilerOptions": {  
23.         "genDir": "aot",  
24.         "skipMetadataEmit" : true  
25.     }  
26. }
```

摇树优化

Rollup 和以前一样，仍然进行摇树优化。

rollup-config.js

```
import rollup from 'rollup'  
import nodeResolve from 'rollup-plugin-node-resolve'  
import commonjs from 'rollup-plugin-commonjs';  
import uglify from 'rollup-plugin-uglify'  
  
//paths are relative to the execution path  
export default {  
    entry: 'app/main-aot.js',  
    dest: 'aot/dist/build.js', // output a single application bundle  
    sourceMap: true,  
    sourceMapFile: 'aot/dist/build.js.map',  
    format: 'iife',  
    plugins: [  
        nodeResolve({jsnext: true, module: true}),  
        commonjs({  
            include: ['node_modules/rxjs/**']  
        }),  
        uglify()  
    ]  
}
```

运行应用

面向大众的运行 AoT 构建的英雄指南应用的说明还没有准备好。

下面的说明假设你克隆了 [angular.io Github 库](#)，并按照该库的 README.md 准备了开发环境。

英雄指南 源代码在 `public/docs/_examples/toh-6/ts` 目录。

和其他 JIT 例子一样，使用 `npm start` 命令，运行 JIT 编译的应用：

AoT 编译假设上面介绍的一些支持文件都以准备好。

```
1.  <!DOCTYPE html>
2.  <html>
3.    <head>
4.      <base href="/">
5.      <title>Angular Tour of Heroes</title>
6.      <meta name="viewport" content="width=device-width, initial-
scale=1">
7.
8.      <link rel="stylesheet" href="styles.css">
9.
10.     <script src="shim.min.js"></script>
11.     <script src="zone.min.js"></script>
12.     <script>window.module = 'aot';</script>
13.   </head>
14.
15.   <body>
16.     <my-app>Loading...</my-app>
17.   </body>
18.   <script src="dist/build.js"></script>
19. </html>
```

使用下面的 npm 脚本，扩展 `package.json` 文件的 `scripts` 部分：

```
"build:aot": "ngc -p tsconfig-aot.json && rollup -c rollup-config.js",
"lite:aot": "lite-server -c aot/bs-config.json",
```

使用下面的 node 脚本，拷贝 AoT 发布文件到 /aot/ 目录：

```
node copy-dist-files
```

直到 `zone.js` 或者支持老版本浏览器的 `core-js` 垫片有更新，你不需要再这样做。

现在 AoT 编译应用，并使用 `lite` 服务器启动它：

```
npm run build:aot && npm run lite:aot
```

检查包裹

看看 Rollup 之后生成的 JavaScript 包，非常神奇。代码已经被最小化，所以你不会从中直接学到任何知识。但是 `source-map-explorer` 工具非常有用。

安装：

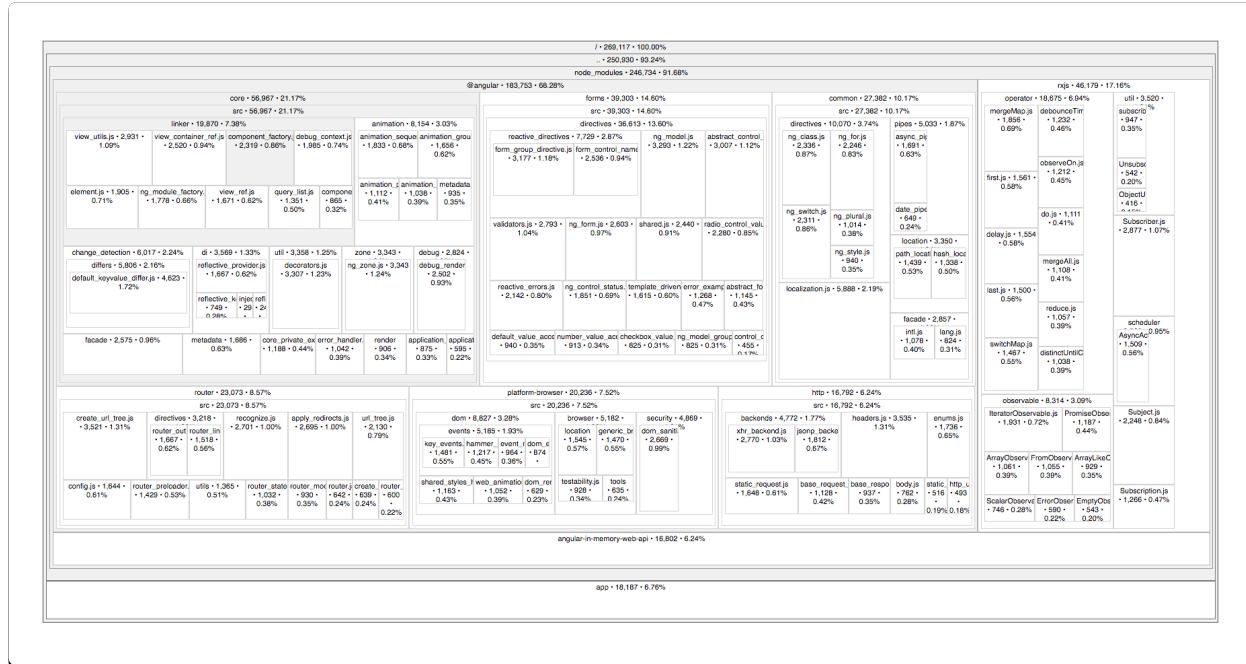
```
npm install source-map-explorer --save-dev
```

运行下面的命令来生成源映射。

```
node_modules/.bin/source-map-explorer aot/dist/build.js
```

`source-map-explorer` 分析从包生成的源映射，并画出一个依赖地图，显示包中包含哪些应用程序和 Angular 模块和类。

下面是英雄指南的地图：



ANGULAR从1到2快速参考

学习如何把 Angular 1 中的概念和技术对应到 Angular 2 中

在 Angular 1 和 Anuglar 2 之间，有很多不同的概念和语法。本章提供了一个快速的参考指南，指出一些常用的 Angular 1 语法及其在 Angular 2 中的等价物。

可到 [在线例子](#) 中查看 Angular 2 语法。

内容

本页内容覆盖了：

- [模板基础](#) - 绑定变量与局部变量。
- [模板指令](#) - 内置指令 `ngIf` 和 `ngClass`。
- [过滤器 / 管道](#) - 内置 [过滤器 \(filter\)](#)，在 Angular 2 中叫 [管道 \(pipe\)](#)。
- [模块 / 控制器 / 组件](#) - Angular 2 中的 [模块](#) 和 Angular 1 中的略有不同；而 [控制器](#) 在 Angular 2 中叫组件。
- [样式表](#) - 在 Angular 2 中关于 CSS 的更多选项。

模板基础

模板是 Angular 应用中的门面部分，它是用 HTML 写的。下表中是一些 Angular 1 中的关键模板特性及其在 Angular 2 中的等价语法。

Angular 1	Angular 2
<p>绑定 / 插值表达式</p> <pre>Your favorite hero is: {{vm.favoriteHero}}</pre> <p>在 Angular 1 中，花括号中的表达式代表单向绑定。它把元素的值绑定到了与模板相关控制器的属性上。</p> <p>当使用 <code>controller as</code> 语法时，该绑定需要用控制器的别名（<code>vm</code>）为前缀，这是因为我们不得不通过它来指定绑定源。</p>	<p>绑定 / 插值表达式</p> <pre>Your favorite hero is: {{favoriteHero}}</pre> <p>在 angular 2 中，花括号中的模板表达式同样代表单向绑定。它把元素的值绑定到了组件的属性上。它绑定的上下文变量是隐式的，并且总是关联到组件。所以，它不需要一个引用变量。</p> <p>要了解更多，请参见模板语法中的 插值表达式 部分。</p>

过滤器

```
<td>{{movie.title | uppercase}}</td>
```

要在 Angular 1 中过滤输出，使用管道字符 (|) 以及一个或多个过滤器。

在这个例子中，我们把 title 属性过滤成了大写形式。

管道

```
<td>{{movie.title | uppercase}}</td>
```

在 Angular 2 中，我们使用相似的语法——用管道字符 (|) 来过滤输出，但是现在直接把它叫做 **管道** 了。很多（但不是所有）Angular 1 中的内置过滤器也成了 Angular 2 中的内置管道。

请参见下面 [过滤器 / 管道](#) 了解更多信息。

局部变量

```
<tr ng-repeat="movie in vm.movies">
  <td>{{movie.title}}</td>
</tr>
```

这里的 movie 是一个用户定义的局部变量

输入变量

```
<tr *ngFor="let movie of movies">
  <td>{{movie.title}}</td>
</tr>
```

在 Angular 2 中，我们有了真正的模板输入变量，它需要使用 let 关键字进行明确定义。

要了解更多信息，请参见模板语法中的 [ngFor 微语法](#) 部分。

[返回顶部](#)

模板指令

Angular 1 为模板提供了七十多个内置指令。在 Angular 2 中，它们很多都已经不需要了，因为 Angular 2 有了一个更加强大、快捷的绑定系统。下面是一些 Angular 1 中的关键指令及其在 Angular 2 中的等价物。

Angular 1

Angular 2

ng-app

```
<body ng-app="movieHunter">
```

应用的启动过程被称为 **引导**。

虽然可以从代码中引导 Angular 1 应用，但很多应用都是通过 ng-app 指令进行声明式引导的，只要给它一个应用模块的名字（movieHunter）就可以了。

引导

main.ts

```
import { platformBrowserDynamic } from
'@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

app.module.ts

```

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

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

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }

```

Angular 2 没有引导指令。我们总是通过显式调用一个 `bootstrap` 函数，并传入应用模块的名字（`AppComponent`）来启动应用。

要了解更多，参见“[快速起步](#)”。

ng-class

```

<div ng-class="{active: isActive}">
<div ng-class="{active: isActive,
               shazam:
               isImportant}">

```

在 Angular 1 中，`ng-class` 指令会基于一个表达式来包含 / 排除某些 CSS 类。该表达式通常是一个“键 - 值”型的控制对象，对象中的每一个键代表一个 CSS 类名，每一个值定义为一个返回布尔值的模板表达式。

在第一个例子中，当 `isActive` 为真时，`active` 类会被应用到元素上。

就像第二个例子中展示的，可以指定多个 CSS 类。

ngClass

```

<div [ngClass]="{active: isActive}">
<div [ngClass]="{active: isActive,
               shazam: isImportant}">
<div [class.active]="isActive">

```

在 Angular 2 中，`ngClass` 指令用类似的方式工作。它根据一个表达式包含 / 排除某些 CSS 类。

在第一个例子中，如果 `isActive` 为真，则 `active` 类被应用到那个元素上。

就像第二个例子中所展示的那样，可以同时指定多个类。

Angular 2 还有 **类绑定**，它是单独添加或移除一个类的好办法——就像第三个例子中展示的。

要了解更多信息，参见模板语法中的 [属性、CSS 类和样式绑定](#) 部分。

ng-click

```

<button ng-click="vm.toggleImage()">
<button ng-
click="vm.toggleImage($event)">

```

在 Angular 1 中，`ng-click` 指令指定当元素被点击时的自定义行为。

绑定到 click 事件

```

<button (click)="toggleImage()">
<button (click)="toggleImage($event)">

```

Angular 1 基于事件的指令在 Angular 2 中已经不存在了。不过，可以使用 **事件绑定** 来定义从模板视图到组件的单向数据绑定。

在第一个例子中，如果用户点击了这个按钮，那么控制器的 `toggleImage()` 方法就会被执行，这个控制器是被 `controller as` 中指定的 `vm` 别名所引用的。

第二个例子演示了传入 `$event` 对象，它提供了事件的详情，并被传到控制器。

要使用事件绑定，把目标事件的名字放在圆括号中，并且使用等号右侧引号中的模板语句对它赋值。然后 Angular 2 为这个目标时间设置事件处理器。当事件被触发时，这个处理器就会执行模板语句。

在第一个例子中，当用户点击此按钮时，相关组件中的 `toggleImage()` 方法就会被执行。

第二个例子演示了如何传入 `$event` 对象，它为组件提供了此事件的详情。

要查看 DOM 事件的列表，请参见 [网络事件](#)。

要了解更多，请参见模板语法中的 [事件绑定](#) 部分。

ng-controller

```
<div ng-controller="MovieListCtrl as vm">
```

在 Angular 1 中，`ng-controller` 指令把控制器附加到视图上。使用 `ng-controller`（或把控制器定义为路由的一部分）把视图及其控制器的代码联系在一起。

Component 装饰器

```
@Component({
  moduleId: module.id,
  selector: 'movie-list',
  templateUrl: 'movie-list.component.html',
  styleUrls: [ 'movie-list.component.css' ],
})
```

在 Angular 2 中，模板不用再指定它相关的控制器。反过来，组件会在组件类的装饰器中指定与它相关的模板。

要了解更多，请参见 [架构概览](#)。

ng-hide

在 Angular 1 中，`ng-hide` 指令会基于一个表达式显示或隐藏相关的 HTML 元素。参见 [ng-show](#) 了解更多。

绑定 hidden 属性

在 Angular 2 中，并没有一个内置的 `hide` 指令，可以改用属性绑定。参见 [ng-show](#) 了解更多。

ng-href

```
<a ng-href="angularDocsUrl">Angular Docs</a>
```

`ng-href` 指令允许 Angular 1 对 `href` 属性进行预处理，以便它能在浏览器获取那个 URL 之前，使用一个返回适当 URL 的绑定表达式替换它。

在 Angular 1 中，`ng-href` 通常用来作为导航的一部分，激活一个路由。

```
<a ng-href="#movies">Movies</a>
```

绑定到 href 属性

```
<a [href]="angularDocsUrl">Angular Docs</a>
```

在 Angular 2 中，并没有内置的 `href` 指令，改用属性绑定。我们把元素的 `href` 属性放在方括号中，并把它设成一个引号中的模板表达式。

要了解属性绑定的更多知识，参见 [模板语法](#)。

在 Angular 2 中，`href` 不再用作路由，而是改用第三个例子中所展示的 `routerLink` 指令。

路由在 Angular 2 中的处理方式不同。

```
<a [routerLink]=["'/movies']>Movies</a>
```

要了解关于路由的更多信息，请参见 [路由与导航](#)。

ng-if

```
<table ng-if="movies.length">
```

在 Angular 1 中，`ng-if` 指令会根据一个表达式来移除或重建 DOM 中的一部分。如果表达式为假，元素就会被从 DOM 中移除。

在这个例子中，除非 `movies` 数组的长度大于 0，否则 `table` 元素就会被从 DOM 中移除。

*ngIf

```
<table *ngIf="movies.length">
```

Angular 2 中的 `*ngIf` 指令与 Angular 1 中的 `ng-if` 指令一样，它根据表达式的值移除或重建 DOM 中的一部分。

在这个例子中，除非 `movies` 数组的长度大于 0，否则 `table` 元素就会被从 DOM 中移除。

在这个例子中 `ngIf` 前的星号 (*) 是必须的。要了解更多信息，参见 [结构型指令](#)。

ng-model

```
<input ng-model="vm.favoriteHero"/>
```

在 Angular1 中，`ng-model` 指令把一个表单控件绑定到了模板相关控制器的一个属性上。这提供了**双向绑定**功能，因此，任何对视图中值的改动，都会同步到模型中，对模型的改动，也会同步到视图中。

ngModel

```
<input [(ngModel)]="favoriteHero" />
```

在 Angular 2 中，**双向绑定**使用 `[]()` 标记出来，它被形象的比作“盒子中的香蕉”。这种语法是一个简写形式，用来同时定义一个属性绑定（从组件到视图）和一个事件绑定（从视图到组件），因此，我们得到了双向绑定。

要了解使用 `ngModel` 进行双向绑定的更多知识，参见 [模板语法](#)。

ng-repeat

```
<tr ng-repeat="movie in vm.movies">
```

在 Angular1 中，`ng-repeat` 指令会为指定集合中的每一个条目重复渲染相关的 DOM 元素。

在这个例子中，对 `movies` 集合中的每一个 `movie` 对象重复渲染了这个表格行元素 (`tr`)。

*ngFor

```
<tr *ngFor="let movie of movies">
```

Angular 2 中的 `*ngFor` 指令类似于 Angular 1 中的 `ng-repeat` 指令。它为指定集合中的每一个条目重复渲染了相关的 DOM 元素。更准确的说，它把被界定出来的元素（这个例子中是 `tr`）及其内容转成了一个模板，并使用那个模板来为列表中的每一个条目实例化一个视图。

请注意其它语法上的差异：在 `ngFor` 前面的星号 (*) 是必须的；`let` 关键字把 `movie` 标记成一个输入变量；列表中使用的介词是 `of`，而不再是 `in`。

要了解更多信息，参见 [结构性指令](#)。

ng-show

```
<h3 ng-show="vm.favoriteHero">
  Your favorite hero is:
  {{vm.favoriteHero}}
</h3>
```

在 Angular 1 中，`ng-show` 指令根据一个表达式来显示或隐藏相关的 DOM 元素。

在这个例子中，如果 `favoriteHero` 变量为真，`div` 元素就会显示出来。

绑定到 hidden 属性

```
<h3 [hidden]="!favoriteHero">
  Your favorite hero is: {{favoriteHero}}
</h3>
```

在 Angular 2 中，并没有内置的 `show` 指令，可以改用属性绑定。要隐藏或显示一个元素，绑定到它的 `hidden` 属性就可以了。

要想有条件的显示一个元素，就把该元素的 `hidden` 属性放到一个方括号里，并且把它设置为引号中的模板表达式，它的结果应该是与 [显示时相反](#) 的值。

在这个例子中，如果 `favoriteHero` 变量不是真值，`div` 元素就会被隐藏。

要了解关于属性绑定的更多信息，参见 [模板表达式](#)。

ng-src

```

```

在 Angular 1 中，`ng-style` 指令根据一个绑定表达式设置一个 HTML 元素的 CSS 样式。该表达式通常是一个“键 - 值”形式的控制对象，对象的每个键都是一个 CSS 的样式名，每个值都是一个能计算为此样式的合适值的表达式。

在这个例子中，`color` 样式被设置为 `colorPreference` 变量的当前值。

ngStyle

```
<div [ngStyle]={{color: colorPreference}}>
<div [style.color]={{colorPreference}}>
```

在 Angular 2 中，`ngStyle` 指令的工作方式与此类似。它根据一个表达式设置 HTML 元素上的 CSS 样式。

在第一个例子中，`color` 样式被设置成了 `colorPreference` 变量的当前值。

Angular 2 还有 [样式绑定](#) 语法，它是单独设置一个样式的好方法。它展示在第二个例子中。

要了解样式绑定的更多知识，参见 [模板语法](#)。

要了解关于 ngStyle 指令的更多知识，参见 [模板语法](#)。

ng-switch

```
<div ng-switch="vm.favoriteHero &&
  vm.checkMovieHero(vm.favoriteHero)">
  <div ng-switch-when="true">
    Excellent choice!
  </div>
  <div ng-switch-when="false">
    No movie, sorry!
  </div>
  <div ng-switch-default>
    Please enter your favorite
    hero.
  </div>
</div>
```

在 Angular1 中， ng-switch 指令根据一个表达式的当前值把元素的内容替换成几个模板之一。

在这个例子中，如果 favoriteHero 没有设置，则模板显示“ Please enter ... ”。如果 favoriteHero 设置过，它就会通过调用一个控制其方法来检查它是否电影里的英雄。如果该方法返回 true ，模板就会显示“ Excellent choice! ”。如果该方法返回 false ，该模板就会显示“ No movie, sorry! ”。

ngSwitch

```
<span [ngSwitch]="favoriteHero &&
  checkMovieHero(favoriteHero)">
  <p *ngSwitchCase="true">
    Excellent choice!
  </p>
  <p *ngSwitchCase="false">
    No movie, sorry!
  </p>
  <p *ngSwitchDefault>
    Please enter your favorite hero.
  </p>
</span>
```

在 Angular2 中， ngSwitch 指令的工作方式与此类似。它会显示那个与 ngSwitch 表达式的当前值匹配的那个 *ngSwitchCase 所在的元素。

在这个例子中，如果 favoriteHero 没有设置，则 ngSwitch 的值是 null ，我们会看到 *ngSwitchDefault 中的段落“ Please enter ... ”。如果 favoriteHero 被设置了，它就会通过调用一个组件方法来检查电影英雄。如果该方法返回 true ，我们就会看到“ Excellent choice! ”。如果该方法返回 false ，我们就会看到“ No movie, sorry! ”。

在这个例子中， ngSwitchCase 和 ngSwitchDefault 前面的星号 (*) 是必须的。

要了解关于 ngswitch 指令的更多信息，参见 [模板语法](#)。

[回到顶部](#)

过滤器 / 管道

Angular 2 中的 管道 为模板提供了格式化和数据转换功能，类似于 Angular 1 中的 过滤器 。 Angular 1 中的很多内置过滤器在 Angular 2 中都有对应的管道。要了解管道的更多信息，参见 [Pipes](#) 。

Angular 1

currency

```
<td>{{movie.price | currency}}</td>
```

Angular 2

currency

```
<td>{{movie.price |
  currency:'USD':true}}</td>
```

把一个数字格式化成货币。

Angular 2 的 currency 管道和 1 中很相似，只是有些参数变化了。

date

```
<td>{{movie.releaseDate | date}}</td>
```

基于要求的格式把日期格式化成字符串。

date

```
<td>{{movie.releaseDate | date}}</td>
```

Angular 2 的 date 管道和 · 中很相似。参见 [备注](#) 来了解字符串日期值。

filter

```
<tr ng-repeat="movie in movieList | filter: {title:listFilter}">
```

基于过滤条件从指定的集合中选取出来一个子集。

没了

在 Angular 2 中，出于性能的考虑，并没有一个类似的管道。过滤逻辑应该在组件中用代码实现。如果它将被复用在几个模板中，可以考虑构建一个自定义管道。

json

```
<pre>{{movie | json}}</pre>
```

把一个 JavaScript 对象转换成一个 JSON 字符串。这对调试很有用。

json

```
<pre>{{movie | json}}</pre>
```

Angular 2 的 json 管道做完全相同的事。

limitTo

```
<tr ng-repeat="movie in movieList | limitTo:2:0">
```

从集合中选择从 (第二参数指定的) 起始索引号 (0) 开始的最多 (第一参数指定的) 条目数 (2) 个条目。

slice

```
<tr *ngFor="let movie of movies | slice:0:2">
```

SlicePipe 做同样的事，但是 **两个参数的顺序是相反的**，以便于 JavaScript 中的 slice 方法保持一致。第一个参数是起始索引号，第二个参数是限制的数量。和 Angular 1 中一样，如果们改用组件中的代码实现此操作，性能将会提升。

lowercase

```
<div>{{movie.title | lowercase}}</div>
```

lowercase

```
<td>{{movie.title | lowercase}}</td>
```

把该字符串转成小写形式。

Angular 2 的 lowercase 管道和 1 中的功能完全相同。

number

```
<td>{{movie.starRating | number}}</td>
```

把数字格式化为文本。

number

```
<td>{{movie.starRating | number}}</td>
<td>{{movie.starRating | number:'1.1-2'}}</td>
<td>{{movie.approvalRating | percent:'1.0-2'}}</td>
```

Angular 2 的 number 管道很相似。但在指定小数点位置时，它提供了更多的功能，如第二个范例所示。

Angular 2 还有一个 percent 管道，它把一个数组格式化为本地化的 (local) 百分比格式，如第三个范例所示。

orderBy

```
<tr ng-repeat="movie in movieList | orderBy : 'title'">
```

使用表达式中所指定的方式对集合进行排序。在这个例子中，movieList 被根据 movie 的 title 排序了。

没了

在 Angular 2 中，出于性能的考虑，并没有一个类似的管道。排序逻辑应该在组件中用代码实现。如果它将被复用在几个模板中，可以考虑构建一个自定义管道。

[回到顶部](#)

模块 / 控制器 / 组件

无论在 Angular 1 还是 Angular 2 中，我们都要借助“模块”来把应用拆分成一些紧密相关的功能块。

在 Angular 1 中，我们在 **控制器** 中写代码，来为视图提供模型和方法。在 Angular 2 中，我们创建 **组件**。

因为很多 Angular 1 的代码是用 JavaScript 写的，所以在 Angular 1 列显示的是 JavaScript 代码，而 Angular 2 列显示的是 TypeScript 代码。

Angular 1

IIFE

```
(function () {
  ...
})();
```

Angular 2

没了

在 Angular 2 中我们不用担心这个问题，因为使用 ES 2015 的模块，模块会替我们处理命名空间问题。

要了解关于模块的更多信息，参见 [架构概览](#)。

在 Angular 1 中，我们通常会定义一个立即调用的函数表达式 (IIFE) 来包裹控制器代码。这样让控制器代码不会污染全局命名空间。

Angular 模块

```
angular.module("movieHunter",
  ["ngRoute"]);
```

在 Angular 1 中，Angular 模块用来对控制器、服务和其它代码进行跟踪。第二个参数定义该模块依赖的其它模块列表。

Angular modules

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

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

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Angular 2 的模块用 `NgModule` 装饰器进行定义，有如下用途：

- `imports`：指定当前模块依赖的其它模块列表
- `declaration`：用于记录组件、管道和指令。

要了解关于模块的更多信息，参见 [Angular Modules](#)。

控制器注册

```
angular
  .module("movieHunter")
  .controller("MovieListCtrl",
    ["movieService",
     MovieListCtrl]);
```

在 Angular 1 中，在每个控制器中都有一些代码，用于找到合适的 Angular 模块并把该控制器注册进去。

第一个参数是控制器的名称，第二个参数定义了所有将注入到该控制器的依赖的字符串名称，以及一个到控制器函数的引用。

组件装饰器

```
@Component({
  moduleId: module.id,
  selector: 'movie-list',
  templateUrl: 'movie-
list.component.html',
  styleUrls: [ 'movie-
list.component.css' ],
})
```

在 Angular 2 中，我们往组件类上添加了一个装饰器，以提供任何需要的元数据。组件装饰器把该类声明为组件，并提供了关于该组件的元数据，比如它的选择器（或标签）和模板。

这就是把模板关联到代码的方式，它定义在组件类中。

要了解关于模板的更多信息，参见架构概览中的 [组件](#) 部分。

控制器函数

```
function MovieListCtrl(moviesService) {  
}
```

在 Angular1 中，我们在控制器函数中写模型和方法的代码。

组件类

```
export class MovieListComponent {  
}
```

在 Angular 2 中，我们写组件类。

注意：如果你正在用 TypeScript 写 Angular 1，那么必须用 `export` 关键字来导出组件类。

要了解关于组件的更多信息，参见架构概览中的 [组件](#) 部分。

依赖注入

```
MovieListCtrl.$inject = ['MovieService'];  
function MovieListCtrl(moviesService) {  
}
```

在 Angular 1 中，我们把所有依赖都作为控制器函数的参数。在这个例子中，我们注入了一个 `MovieService`。

我们还通过在第一个参数明确告诉 Angular 它应该注入一个 `MovieService` 的实例，以防止在最小化时出现问题。

依赖注入

```
constructor(movieService: MovieService)  
{  
}
```

在 Angular 2 中，我们把依赖作为组件构造函数的参数传入。在这个例子中，我们注入了一个 `MovieService`。即使在最小化之后，第一个参数的 TypeScript 类型也会告诉 Angular 它该注入什么。

要了解关于依赖注入的更多信息，参见架构概览中的 [依赖注入](#) 部分。

[回到顶部](#)

样式表

样式表美化我们的应用程序。在 Angular 1 中，我们为整个应用程序指定样式表。当应用程序成长一段时间之后，应用程序中很多部分的样式会被合并，导致无法预计的后果。在 Angular 2 中，我们仍然会为整个应用程序定义样式，不过现在也可以把样式表封装在特定的组件中。

Angular 1

Link tag

```
<link href="styles.css" rel="stylesheet"  
/>>
```

Angular 2

Link 标签

```
<link rel="stylesheet"  
href="styles.css">
```

在 Angular 1 中，我们在 `index.html` 的 `head` 区使用 `link` 标签来为应用程序定义样式。

在 Angular2 中，我们可以继续在 `index.html` 中使用 `link` 标签来为应用程序定义样式。但是也能在组件中封装样式。

StyleUrls

在 Angular 2 中，我们可以在 `@Component` 的元数据中使用 `styles` 或 `styleUrls` 属性来为一个特定的组件定义样式表。

```
styleUrls: [ 'movie-list.component.css'  
],
```

这让我们可以为各个组件设置合适的样式，而不用担心它被泄漏到程序中的其它部分。

[回到顶部](#)

ANGULAR模块常见问题

对 @NgModule 常见问题的解答

Angular 模块常见问题 (FAQ)

Angular 模块 可以帮我们把应用组织成一些紧密相关的代码块。

Angular 模块 章涵盖了此概念，并带你一步步的从最基本的 `@NgModule` 到包罗万象的惰性加载模块。

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。

本《Angular 模块常见问题》假设你已经读完了 [Angular 模块 章](#)。

声明 (declarations)

- 我应该把哪些类加到 `declarations` 中？
- 什么是 **可声明的** ？
- 我 **不应该** 把哪些类加到 `declarations` 中？
- 为什么要把同一个组件列在 `NgModule` 的不同属性中？
- "`Can't bind to 'x' since it isn't a known property of 'y'`" 是什么意思？

导入 (imports)

- 我应该导入什么？
- 我应该导入 `BrowserModule` 还是 `CommonModule` ？
- 如果我两次导入了同一个模块会怎么样？

导出 (exports)

- 我应该导出什么？
- 我**不**应该导出什么？
- 我能再次导出 (re-export) 所导入的类和模块吗？
- **forRoot** 方法是什么？

服务提供商 (providers)

- 为什么特性模块中提供的服务是到处可见的？
- 为什么**惰性加载**模块中的服务提供商只对那个模块本身可见？
- 如果两个模块提供了**同一个**服务会怎样？
- 如何把服务的范围限制在某个模块中？
- 我应该把全应用级的提供商添加到根模块 **AppModule** 还是根组件 **AppComponent** ？
- 我应该把其它提供商添加到模块中还是组件中？
- 为什么让 **SharedModule** 为惰性加载模块提供服务是个馊主意？
- 为什么惰性加载模块要创建一个子注入器？
- 我要怎样才能知道一个模块或服务已经被加载过？

入口组件

- 什么是**入口组件**？
- **引导**组件和**入口**组件有什么不同？
- 什么情况下我应该把组件添加到 **entryComponent** 中？
- 为什么Angular需要 **entryComponents**？

一般问题

- 我需要哪些类型的模块？我应该如何使用它们？
- Angular 模块和 JavaScript 模块有什么不同？
- 什么是“模板引用”？
- Angular 如何是在模板中查找组件、指令和管道的？
- 什么是 Angular 编译器（ Compiler ）？
- 你能总结一下 **NgModule API** 吗？

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。

哪些类不應該加到 `declarations` 中？

只有 **可声明的** 类才能加到模块的 `declarations` 列表中。

不要声明：

- 已经被其它模块声明过的类，无论是在应用模块、`@angular` 模块还是第三方模块中。
 - 一组从其它模块中导入的指令。例如，不要声明来自 `@angular/forms` 的 `FORMS_DIRECTIVES`。
 - 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？

我们经常看到 `AppComponent` 被同时列在 `declarations` 和 `bootstrap` 中。我们还可能看到 `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，我们无法从它出现在一个列表中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（所以我们没法声明它）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。

比如，如果这个“`x`”是 `ngModel`，你可能忘了从 `@angular/forms` 中导入 `FormsModule`。

也可能你在该应用的特性模块中声明了“`x`”，但是忘了从那个模块导出它。除非你把这个“`x`”类加入了 `exports` 列表中，否则它对其他模块将是不可见的。

我应该导入什么？

一句话：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。

如果你在惰性加载模块中导入 `BrowserModule` , Angular 就会抛出一个错误。

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？

不会有问题是。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那个层级，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 **公开** 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。模块甚至还可以导出它未曾导入过的模块。

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 `HttpModule`，因为它不导出任何东西。它唯一的用途是把 `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 `HttpModule`，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。我们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。

`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` 都是方法的约定名称，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。

当我们导入一个模块时，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 `HttpModule`，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。**这是一个不受欢迎的意外。**

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。

我们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 `HttpModule`，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 `HttpModule`。

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插座（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？

在根模块 `AppModule` 中注册全应用级提供商，而不是 `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。但这个“几乎”对于带路有的应用仍然是不够的。

当有路由时，`AppComponent` 服务并不在根部。惰性加载的模块就不能用它们。在“Angular 模块”章的范例应用中，如果我们在 `AppComponent` 中注册 `UserService`，那么 `HeroComponent` 就不能注入它。一旦用户导航到“Heroes”特性区，该应用就会失败。

我应该把其它提供商注册到模块中还是组件中？

通常，优先把模块中具体特性的提供商注册到模块中（`@NgModule.providers`），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，那么我们应该把 `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 SharedModule 为惰性加载模块提供服务是个馊主意？

这个问题在 [Angular 模块](#) 章出现过，那时我们在讨论不要把提供商放进 `SharedModule` 的重要性。

假设把 `UserService` 列在了模块的 `providers` 中（我们没有这么做）。假设每个模块都导入了这个 `SharedModule`（我们是这么做的）。

当应用启动时，Angular 主动加载了 `AppModule` 和 `ContactModule`。

导入的 `SharedModule` 的每个实例都会提供 `UserService`。Angular 把它们中的一个注册进了应用的根注入器中（参见 [前面](#)）。然后，某些组件要求注入 `UserService`，Angular 就会在应用的根注入器中查找它，并交付一个全应用级的单例对象 `UserService`。这没问题。

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。

这绝对是一个错误。

自己验证一下吧。运行这个 [在线例子](#)。修改 `SharedModule`，由它来提供 `UserService` 而不再由 `CoreModule`。然后在“Contact”和“Heroes”链接之间切换几次。由于 Angular 每次都创建一个新的 `UserService` 实例，所以用户名变得不正常了。

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，我们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，我们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`）就实现了一个像本页范例中的 `CoreModule` 构造函数那样的守卫。

```
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。

也有少量组件只会被动态加载，并且 **永远不会** 被组件的模板所引用。

用于引导的根 `AppComponent` 就是一个 **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。

Angular 总是会动态加载 `AppComponent` ——无论把它的 **类型** 列在了 `@NgModule.bootstrap` 函数中，还是命令式的调用该模块的 `ngDoBootstrap` 方法来引导它。

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。

编译器无法通过在其它组件的模板中查找来发现这些 **入口组件**。 我们必须通过把它们加入 `entryComponents` 列表中来让编译器知道它们的存在。

Angular 会自动把两种组件添加到模块的 `entryComponents` 中：

1. 那些出现在 `@NgModule.bootstrap` 列表中的组件
2. 那些被路由定义引用的组件

我们并不需要显式的引用这些组件——虽然引用了也没坏处。

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。

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。

为什么 Angular 需要 入口组件？

[入口组件](#) 也是被声明的。为什么 Angular 编译器不为 `@NgModule.declarations` 中的每个组件都生成一份代码呢？那样就不需要入口组件了。

原因在于 **摇树优化**。对于产品化应用，我们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。它应该排除那些我们从未用过的组件，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我们都用不到。如果我们从未引用它们，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，它只为我们用到的那些组件生成代码。

它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。一些建议和向导具有更加广泛的吸引力。

下面这些初步的指南仅来自在少量应用中使用 Angular 模块时的早期体验。仅供参考。

SHARED MODULE

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。

它可以重新导出其它 **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。

它 **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](#) 。

这里的范例违背了此建议，它声明和导出了两个只用在 `AppModule` 模块的 `AppComponent` 组件中的组件。如果你想严格遵循这条指南，应该把这两个组件改为声明在 `AppModule` 中。

特性模块

围绕特定的业务领域、工作流和工具集来为应用创建 [特性模块](#) 。

特性模块一般可分成下面这四种：

- 领域特性模块
- 路由特性模块
- 服务特性模块
- 窗口部件特性模块

真实世界中的模块通常会偏离这些指导原则，而混杂多种不同的类型。这些只是指导原则，不是硬性要求。但除非你有充分的理由不这么做，最好还是遵循它们。

特性模块

指导原则

领域

领域特性模块 **专注于一个特定的应用领域** 来提供用户体验，比如编辑消费者信息或下订单。

它们通常有一个顶级组件，并作为该特性的根组件。内部则是它的一些子组件。

领域特性模块几乎总是由 `declarations` 构成。只有顶级组件会被导出。

领域特性模块很少会有 `providers` 。如果要这么做，那它们所提供的服务的生命周期就应该与该模块的生命周期相同。

不要在领域特性模块中提供全应用级的单例服务。

领域特性模块的典型用法是 **只被更大的特性模块 导入一次**。

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。

比如“Angular 模块”章的 **ContactModule**。

路由特性模块

路由特性模块 属于 **领域特性模块** 的一种，它的顶层组件是 **路由器导航时的路由目标**。

根据这个定义，所有惰性加载的模块都是路由特性模块。

这里的 ContactModule 、 HeroModule 和 CrisisModule 都是路由特性模块。

路由特性模块 **不应该导出任何东西**，这是因为它们中的任何组件都不可能出现在外部组件的模板中。

惰性加载的路由特性模块也不应该被任何模块 **导出**。那么做会触发一次主动加载，破坏了我们惰性加载的目的。HeroModule 和 CrisisModule 是惰性加载的。它们没有出现在 AppModule 的 imports 中。

而主动加载的路由特性模块必须被其它模块导入，以便编译器了解它有哪些组件。ContactModule 就是主动加载的，因此它也被列在了 AppModule 的 imports 中。

路由特性模块很少会有 providers，理由 **前面解释过**。如果要那么做，它所提供的服务就应该与模块具有相同的生命周期。

不要在路由特性模块及其导入的模块中提供 **全应用级** 的单例服务。

路由模块

路由模块 为其它模块 提供路由配置

。

路由模块将路由配置从它的关联模块分离开来。

它通常：

- 定义路由
- 添加路由配置到模块的 imports 中
- 重新导出 RouterModule
- 添加守卫和解析器服务提供商到模块的 providers 。

路由模块的名字应该和它的关联模块平行，比如使用“ Routing ”前缀，
foo.module.ts 中的 FooModule 有名为 FooRoutingModule 的路由模块，所属文件名为
foo-routing.module.ts 。

如果关联模块是 **根** AppModule ，那么在 AppRoutingModule 的 imports 中，添加
RouterModule.forRoot(routes) 来配置路由。所有其它路由模块都是子级，导入
RouterModule.forChild(routes) 。

路由模块顺便重新导出 RouterModule ，这样关联模块的组件可以访问路由指令，比如 RouterLink 和 RouterOutlet 。

路由模块 **不应该有自己的 declarations** ! 。组件、指令和管道是 **特性模块的责任** ，不属于路由模块。

路由模块应该 **只** 被它的关联模块导入。

AppRoutingModule 、
ContactRoutingModule 和
`HeroRoutingModule 是很好的例
子。

参见 " [你需要 路由模块 吗
？](#)"

服务

服务模块 用于 **提供工具类服务** , 比
如数据访问和消息等。

理想情况下 , 它们应该完全由
providers 组成 , 不应该包括
declarations 。 CoreModule 和
Angular 的 HttpClientModule 就是很好的
例子。

服务模块应该 **只被** 根模块
 AppModule 导入。

不要 在任何特性模块中导入它们。
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。

窗口部件

窗口部件 模块导出能用供外部模块
使用的 **组件、指令和管道** 。

CommonModule 和 SharedModule
都是窗口部件模块。很多第三方 UI
组件库都是窗口部件模块。

部件模块应该只有 declarations ,
并导出里面的绝大多数声明。

窗口部件模块很少会有 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。

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。

下表是对各种 **特性模块** 的关键特征汇总。

真实世界中的模块可能会违背这些分类法，混杂使用它们。

特性模块	声明 'declarations'	提供商 'providers'	导出什么	被谁导入	范例
领域	有	罕见	顶级组件	特性模块和 AppModule	ContactModule (路由之前的那个例子)
路由	有	罕见	无	无	ContactModule 、 HeroModule 、 CrisisModule
路由	无	有	无	AppModule	HttpModule 、 CoreModule

服务	无	有	无	AppModule	HttpModule 、 CoreModule
窗口部件	有	罕见	有	特性模块	CommonModule 、 SharedModule

Angular 模块和 JavaScript 模块有什么区别？

Angular 和 JavaScript 是两种不同但互补的模块体系。

在现代 JavaScript 中，[每个文件都是模块](#)。在每个文件中，我们写一个 `export` 语句将模块的一部分公开。

```
export class AppComponent { ... }
```

然后，我们可以在其它模块中 `import` 那部分：

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

这种模块化方式是 **JavaScript** 语言中的特性。

而 **Angular 模块** 是 **Angular 本身** 的特性。

Angular 的 `NgModule` 也有自己的 `imports` 和 `exports` 来达到类似的目的。

我们可以 **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。我们可以 **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。

Angular 的模块类与 JavaScript 的模块类有三个主要的不同点：

1. 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。
2. JavaScript 模块把所有成员类都定义在一个巨型文件，Angular 模块则把自己的类都列在 `@NgModule.declarations` 数组中。
3. 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在其它模块中导入的。它不会声明或导出任何其它类型的类。

Angular 模块还有些别的特殊之处。不同于 JavaScript 模块，Angular 模块可以通过把服务提供商添加到 `@NgModule.providers` 数组中来扩展 **整个** 应用提供的服务。

这些提供的服务不仅仅从属于当前模块，其作用范围也不局限于模块中声明的类。它们 **在哪里** 都能用。

这里是一个带有 `imports`、`exports` 和 `declarations` 的 **Angular 模块** 类。

```
@NgModule({
  imports:      [ CommonModule, FormsModule ],
  declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],
  exports:      [ ContactComponent ],
  providers:    [ ContactService ]
})
export class ContactModule { }
```

当然，我们同样得用 **JavaScript** 模块来写 **Angular 模块**，就像在最终版 `contact.module.ts` 文件中所见到的：

app/contact/contact.module.ts

```
import { NgModule }           from '@angular/core';
import { CommonModule }       from '@angular/common';
import { FormsModule }        from '@angular/forms';

import { AwesomePipe }        from './awesome.pipe';

import { ContactComponent }   from './contact.component';
import { ContactService }     from './contact.service';
import { HighlightDirective } from './highlight.directive';

@NgModule({
  imports:      [ CommonModule, FormsModule ],
  declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],
  exports:      [ ContactComponent ],
  providers:    [ ContactService ]
})
export class ContactModule { }
```

```

{ ContactComponent }    from './contact.component';
import { ContactService }      from './contact.service';
import { HighlightDirective } from './highlight.directive';

@NgModule({
  imports:      [ CommonModule, FormsModule ],
  declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],
  exports:       [ ContactComponent ],
  providers:     [ ContactService ]
})
export class ContactModule { }

```

ANGULAR 如何在模板中查找组件、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 会把我们所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。

我们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。

NgModule API

下面是 `NgModule` 元数据中属性的汇总表 :

属性	描述
<code>declarations</code>	<p><code>可声明类</code> 的列表 , 也就是属于当前模块的 组件 、 指令 和 管道 类。</p> <p>这些声明的类对组件内部可见 , 但是对其它模块不可见 , 除非 (a) 这些类从当前模块中 导出过 , 并且 (b) 其它模块导入了当前模块。</p>
	<p>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。</p>
	<p>不要重新声明从其它模块中导入的类。</p>

providers

依赖注入提供商的列表。

Angular 会在当前模块执行环境的根注入器中注册这些提供商。那是应用程序在启动时为其加载的所有模块提供的根注入器。

Angular 可以把这些提供商提供的服务注入到应用中的任何组件中。如果该模块提供了 `HeroService` 或启动时被加载的任何模块提供了 `HeroService`，那么 Angular 就会把同一个 `HeroService` 实例注入到应用中的任何组件中。

惰性加载模块有自己的子注入器，通常它是应用的根注入器的直接子级。

惰性加载的服务，其作用范围仅限于惰性加载模块的注入器中。如果惰性加载的模块也提供了 `HeroService`，那么在该模块的环境中创建的任何组件（比如通过路由器导航），都会得到该服务的一个局部实例，而不是来自应用程序根注入器的那个全局实例。

外部模块中的组件仍然会取得由应用的根注入器创建的那个实例。

imports

支撑模块的列表。

特别是包含当前模块中的组件模板引用过的组件、指令或管道的那些模块的列表。

在两种情况下组件模板可以 [引用](#) 其它组件、指令或管道：或者所引用的

类是声明在当前模块中的，或者那个类已经从其它模块中导入进来了。

组件可以使用 `NgIf` 和 `NgFor` 指令，只是因为它所在的模块导入了 Angular 的 `CommonModule`（也可能是通过导入 `BrowserModule` 而间接导入的）。

通过 `CommonModule`，我们可以导入很多标准指令。但是也有一些熟悉的指令是属于其它模块的。比如组件只有导入了 Angular 的 `FormsModule` 才能在组件模板中用 `[(ngModel)]` 进行绑定。

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。

这些导出的可声明对象就是模块的 **公开 API**。如果 (a) 其它模块导入了当前模块，并且 (b) 当前模块导出了 `HeroComponent`，那么其它模块中的组件就可以 **引用** 来自当前模块的 `HeroComponent`。

可声明对象默认情况下是私有的。如果当前模块 **没有** 导出 `HeroComponent`，那么没有任何其它模块能看到它。

导入一个模块 **并不会** 自动重新导出这个模块导出的东西。模块 'B' 即使导入了模块 A，而模块 A 中导入过 `CommonModule`，它也没法使用 `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开组件、指令和管道都会被导出。

[重新导出](#) 可以让模块的传递性更加明确。如果模块 A 重新导出了 CommonModule，然后模块 B 导入了模块 A，那么模块 B 中的组件就能使用 NgIf 了，虽然模块 B 本身并没有导入过 CommonModule。

bootstrap

能被引导的组件列表。

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。

引导组件会自动成为 entryComponent。

entryComponents

那些 **没有** 在任何可访问的组件的模板中 [引用过](#) 的组件列表。

大多数开发人员从来不会设置该属性，为什么呢？

[Angular 编译器](#) 必须知道在应用中实际用过的每一个组件。通过遍历组件模板中的引用树，编译器可以自动找出大多数的组件。

但是至少有一个组件不会被任何模板引用：根组件 `AppComponent`，因为我们就是用它来引导本应用程序的。这也就是为什么它被称为 **入口组件**。

路由组件同样是 **入口组件**，因为它们也不会被从模板中引用。路由器创建会它们，并把它们扔到 DOM 中的 `<router-outlet>`。

引导组件 和 **路由组件** 都是 **入口组件**，我们一般不用再把它们添加到模块的 `entryComponents` 列表中。

Angular 会自动把模块的 `bootstrap` 列表中的组件添加到 `entryComponents` 列表中。

`RouterModule` 同样会把路由组件添加到 `entryComponents` 列表中。

这样，那些无法自动发现的组件就只剩下两个来源了：

1. 使用某种命令式技巧引导的组件。
2. 使用路由器之外的手段动态加载到 DOM 中的组件。

所有这些高级技巧是只有极少数开发人员才会去用的。如果你是其中的一位，那么你就不得不自行把这些组件添加到 `entryComponents` 列表中——无论是用程序添加还是手动添加。

组件通讯

在不同的指令和组件之间共享信息

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。

要深入了解组件通讯的各个基本概念，在 [组件通讯 Component Communication](#) 文档中可以找到详细的描述和例子。

目录

[使用输入型绑定，把数据从父组件传到子组件](#)

[通过 setter 拦截输入属性值的变化](#)

[使用 ngOnChanges 拦截输入属性值的变化](#)

[父组件监听子组件的事件](#)

[父组件与子组件通过 本地变量 local variable 互动](#)

[父组件调用 ViewChild](#)

[父组件和子组件通过服务来通讯](#)

[参见 在线例子 。](#)

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。

```

1. import { Component, Input } from '@angular/core';
2.
3. import { Hero } from './hero';
4.
5. @Component({
6.   selector: '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。

```

1. import { Component } from '@angular/core';
2.
3. import { HEROES } from './hero';
4.
5. @Component({
6.   selector: 'hero-parent',
7.   template: `
8.     <h2>{{master}} controls {{heroes.length}} heroes</h2>
9.     <hero-child *ngFor="let hero of heroes"
10.       [hero]="hero"
11.       [master]="master">
```

```

12.      </hero-child>
13.    )
14.  })
15. export class HeroParentComponent {
16.   heroes = HEROES;
17.   master: string = 'Master';
18. }

```

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

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。

```

1. // ...
2. let _heroNames = ['Mr. IQ', 'Magneta', 'Bombasto'];
3. let _masterName = 'Master';
4.
5. it('should pass properties to children properly', function () {
6.   let parent = element.all(by.tagName('hero-parent')).get(0);
7.   let heroes = parent.all(by.tagName('hero-child'));
8.
9.   for (let i = 0; i < _heroNames.length; i++) {
10.     let childTitle =
        heroes.get(i).element(by.tagName('h3')).getText();

```

```

11.     let childDetail =
12.       heroes.get(i).element(by.tagName('p')).getText();
13.       expect(childTitle).toEqual(_heroNames[i] + ' says:');
14.       expect(childDetail).toContain(_masterName);
15.   });
16. // ...

```

[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。

```

1. import { Component, Input } from '@angular/core';
2.
3. @Component({
4.   selector: 'name-child',
5.   template: `
6.     <h3>"{{name}}"</h3>
7.   `
8. })
9. export class NameChildComponent {
10.   _name: string = '<no name set>';
11.
12.   @Input()
13.   set name(name: string) {
14.     this._name = (name && name.trim()) || '<no name set>';
15.   }
16.
17.   get name() { return this._name; }
18. }

```

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。

```

1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'name-parent',
5.   template: `
6.     <h2>Master controls {{names.length}} names</h2>
7.     <name-child *ngFor="let name of names"
8.       [name]="name">
9.     </name-child>
10.    `
11. })
12. export class NameParentComponent {
13.   // Displays 'Mr. IQ', '<no name set>', 'Bombasto'
14.   names = ['Mr. IQ', ' ', ' Bombasto '];
15. }

```

Master controls 3 names

"Mr. IQ"

"<no name set>"

"Bombasto"

测试

端到端测试：输入属性的 setter，分别使用空名字和非空名字。

```

1. // ...
2. it('should display trimmed, non-empty names', function () {
3.   let _nonEmptyNameIndex = 0;
4.   let _nonEmptyName = '"Mr. IQ"';

```

```

5.   let parent = element.all(by.tagName('name-parent')).get(0);
6.   let hero = parent.all(by.tagName('name-
7.     child')).get(_nonEmptyNameIndex);
8.
9.   let displayName = hero.element(by.tagName('h3')).getText();
10.  expect(displayName).toEqual(_nonEmptyName);
11. });
12.
13. it('should replace empty name with default name', function () {
14.   let _emptyNameIndex = 1;
15.   let _defaultName = "<no name set>";
16.   let parent = element.all(by.tagName('name-parent')).get(0);
17.   let hero = parent.all(by.tagName('name-
18.     child')).get(_emptyNameIndex);
19.
20.   let displayName = hero.element(by.tagName('h3')).getText();
21.   expect(displayName).toEqual(_defaultName);
22. });
23. // ...

```

[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。

```

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

2.

3. @Component({
4.   selector: 'version-child',
5.   template: `
6.     <h3>Version {{major}}.{{minor}}</h3>
7.     <h4>Change log:</h4>
8.     <ul>
9.       <li *ngFor="let change of changeLog">{{change}}</li>
10.    </ul>
11.  `
12. })
13. export class VersionChildComponent implements OnChanges {
14.   @Input() major: number;
15.   @Input() minor: number;
16.   changeLog: string[] = [];
17.
18.   ngOnChanges(changes: {[propKey: string]: SimpleChange}) {
19.     let log: string[] = [];
20.     for (let propName in changes) {
21.       let changedProp = changes[propName];
22.       let from = JSON.stringify(changedProp.previousValue);
23.       let to = JSON.stringify(changedProp.currentValue);
24.       log.push(` ${propName} changed from ${from} to ${to}`);
25.     }
26.     this.changeLog.push(log.join(', '));
27.   }
28. }

```

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

```

1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'version-parent',
5.   template: `
6.     <h2>Source code version</h2>

```

```

7.      <button (click)="newMinor()">New minor version</button>
8.      <button (click)="newMajor()">New major version</button>
9.      <version-child [major]="major" [minor]="minor"></version-child>
10.
11.  })
12.  export class VersionParentComponent {
13.    major: number = 1;
14.    minor: number = 23;
15.
16.    newMinor() {
17.      this.minor++;
18.    }
19.
20.    newMajor() {
21.      this.major++;
22.      this.minor = 0;
23.    }
24.  }

```

下面是点击按钮的结果。

Source code version

New minor version
New major version

Version 1.23

Change log:

- major changed from {} to 1, minor changed from {} to 23

测试

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。

```
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 = 'major changed from {} to 1, minor changed from {} 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('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('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
41.         25 to 0';
42.
43.     expect(actual.label).toBe(labelAfterMajor);
44.     expect(actual.count).toBe(4);
45.     expect(actual.logs.getText(3)).toBe(logAfterMajor);
46.   });
47.
48. function getActual() {
49.   let versionTag = element(by.tagName('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。

```

1. import { Component, EventEmitter, Input, Output } from
2.     '@angular/core';
3. @Component({

```

```

4.   selector: 'my-voter',
5.   template: `
6.     <h4>{{name}}</h4>
7.     <button (click)="vote(true)" [disabled]="voted">Agree</button>
8.     <button (click)="vote(false)" [disabled]="voted">Disagree</button>
9.   `
10. }
11. export class VoterComponent {
12.   @Input() name: string;
13.   @Output() onVoted = new EventEmitter<boolean>();
14.   voted = false;
15.
16.   vote(agreed: boolean) {
17.     this.onVoted.emit(agreed);
18.     this.voted = true;
19.   }
20. }

```

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。

```

1. import { Component }      from '@angular/core';
2.
3. @Component({
4.   selector: 'vote-taker',
5.   template: `
6.     <h2>Should mankind colonize the Universe?</h2>
7.     <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>
8.     <my-voter *ngFor="let voter of voters"
9.       [name]="voter"
10.      (onVoted)="onVoted($event)">
11.    </my-voter>
12.   `
13. })
14. export class VoteTakerComponent {
15.   agreed = 0;
16.   disagreed = 0;

```

```

17.     voters = ['Mr. IQ', 'Ms. Universe', 'Bombasto'];
18.
19.     onVoted(agreed: boolean) {
20.       agreed ? this.agreed++ : this.disagreed++;
21.     }
22.   }

```

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：

The screenshot shows a user interface for a component named 'vote-taker'. It consists of three separate sections, each with a title and two buttons: 'Agree' and 'Disagree'. The sections are labeled 'Mr. IQ', 'Ms. Universe', and 'Bombasto'. The 'Agree' button is highlighted in blue, while the 'Disagree' button is greyed out.

Should mankind colonize the Universe?

Agree: 0, Disagree: 0

Mr. IQ

Agree **Disagree**

Ms. Universe

Agree **Disagree**

Bombasto

Agree **Disagree**

测试

测试确保点击 Agree 和 Disagree 按钮时，计数器被正确更新。

```

1. // ...
2. it('should not emit the event initially', function () {
3.   let voteLabel = element(by.tagName('vote-taker'))
4.   .element(by.tagName('h3')).getText();
5.   expect(voteLabel).toBe('Agree: 0, Disagree: 0');
6. });
7.
8. it('should process Agree vote', function () {
9.   let agreeButton1 = element.all(by.tagName('my-voter')).get(0)

```

```

10.     .all(by.tagName('button')).get(0);
11.     agreeButton1.click().then(function() {
12.       let voteLabel = element(by.tagName('vote-taker'))
13.         .element(by.tagName('h3')).getText();
14.       expect(voteLabel).toBe('Agree: 1, Disagree: 0');
15.     });
16.   });
17.
18. it('should process Disagree vote', function () {
19.   let agreeButton1 = element.all(by.tagName('my-voter')).get(1)
20.     .all(by.tagName('button')).get(1);
21.   agreeButton1.click().then(function() {
22.     let voteLabel = element(by.tagName('vote-taker'))
23.       .element(by.tagName('h3')).getText();
24.     expect(voteLabel).toBe('Agree: 1, Disagree: 1');
25.   });
26. });
27. // ...

```

[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。

```

1. import { Component, OnDestroy, OnInit } from '@angular/core';
2.
3. @Component({
4.   selector: 'countdown-timer',
5.   template: '<p>{{message}}</p>'
6. })
7. export class CountdownTimerComponent implements OnInit, OnDestroy {

```

```

8.
9.     intervalId = 0;
10.    message = '';
11.    seconds = 11;
12.
13.    clearTimer() { clearInterval(this.intervalId); }
14.
15.    ngOnInit() { this.start(); }
16.    ngOnDestroy() { this.clearTimer(); }
17.
18.    start() { this.countDown(); }
19.    stop() {
20.        this.clearTimer();
21.        this.message = `Holding at T-${this.seconds} seconds`;
22.    }
23.
24.    private countDown() {
25.        this.clearTimer();
26.        this.intervalId = window.setInterval(() => {
27.            this.seconds -= 1;
28.            if (this.seconds === 0) {
29.                this.message = 'Blast off!';
30.            } else {
31.                if (this.seconds < 0) { this.seconds = 10; } // reset
32.                this.message = `T-${this.seconds} seconds and counting`;
33.            }
34.        }, 1000);
35.    }
36. }

```

让我们来看看计时器组件的宿主组件 `CountdownLocalVarParentComponent`。

```

1. import { Component }           from '@angular/core';
2. import { CountdownTimerComponent } from './countdown-
   timer.component';
3.
4. @Component({
5.   selector: 'countdown-parent-lv',
6.   template: `
7.     <h3>Countdown to Liftoff (via local variable)</h3>

```

```

8.   <button (click)="timer.start()">Start</button>
9.   <button (click)="timer.stop()">Stop</button>
10.  <div class="seconds">{{timer.seconds}}</div>
11.  <countdown-timer #timer></countdown-timer>
12.  `,
13.  styleUrls: ['demo.css']
14. }
15. export class CountdownLocalVarParentComponent { }

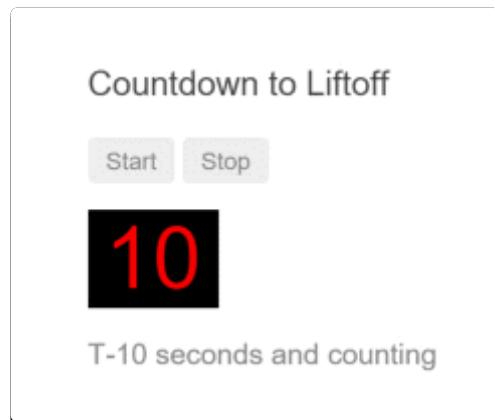
```

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。

在这个例子中，我们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。



测试

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：

```
1. // ...
```

```

2.   it('timer and parent seconds should match', function () {
3.     let parent = element(by.tagName(parentTag));
4.     let message = parent.element(by.tagName('countdown-
5.       timer')).getText();
6.     browser.sleep(10); // give `seconds` a chance to catchup with
7.     // `message`
8.     let seconds = parent.element(by.className('seconds')).getText();
9.     expect(message).toContain(seconds);
10.    });
11.   it('should stop the countdown', function () {
12.     let parent = element(by.tagName(parentTag));
13.     let stopButton = parent.all(by.tagName('button')).get(1);
14.     stopButton.click().then(function() {
15.       let message = parent.element(by.tagName('countdown-
16.         timer')).getText();
17.       expect(message).toContain('Holding');
18.     });
19.   });
20.   // ...

```

[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。

我们将会用同一个 **倒计时** 范例来解释这种技术。我们没有改变它的样子或行为。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` :

```
1. import { AfterViewInit, ViewChild } from '@angular/core';
2. import { Component } from '@angular/core';
3. import { CountdownTimerComponent } from './countdown-
   timer.component';
4.
5. @Component({
6.   selector: 'countdown-parent-vc',
7.   template: `
8.     <h3>Countdown to Liftoff (via ViewChild)</h3>
9.     <button (click)="start()">Start</button>
10.    <button (click)="stop()">Stop</button>
11.    <div class="seconds">{{ seconds() }}</div>
12.    <countdown-timer></countdown-timer>
13.    `,
14.   styleUrls: ['demo.css']
15. })
16. export class CountdownViewChildParentComponent implements
   AfterViewInit {
17.
18.   @ViewChild(CountdownTimerComponent)
19.   private timerComponent: CountdownTimerComponent;
20.
21.   seconds() { return 0; }
22.
23.   ngAfterViewInit() {
24.     // Redefine `seconds()` to get from the
25.     `CountdownTimerComponent.seconds` ...
26.     // but wait a tick first to avoid one-time devMode
27.     // unidirectional-data-flow-violation error
28.     setTimeout(() => this.seconds = () => this.timerComponent.seconds,
29.     0);
30.   }
31.
32.   start() { this.timerComponent.start(); }
```

```
31.     stop() { this.timerComponent.stop(); }  
32. }
```

把子组件的视图插入到父组件类需要做一点额外的工作。

需要通过 `ViewChild` 装饰器导入这个引用，并挂上 `AfterViewInit` 生命周期钩子。

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，所以我们先把秒数显示为 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。我们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

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。

```

1. import { Injectable } from '@angular/core';
2. import { Subject } from 'rxjs/Subject';
3.
4. @Injectable()
5. export class MissionService {
6.
7.   // Observable string sources
8.   private missionAnnouncedSource = new Subject<string>();
9.   private missionConfirmedSource = new Subject<string>();
10.
11.  // Observable string streams
12.  missionAnnounced$ = this.missionAnnouncedSource.asObservable();
13.  missionConfirmed$ = this.missionConfirmedSource.asObservable();
14.
15.  // Service message commands
16.  announceMission(mission: string) {
17.    this.missionAnnouncedSource.next(mission);
18.  }
19.
20.  confirmMission(astronaut: string) {
21.    this.missionConfirmedSource.next(astronaut);
22.  }
23. }
```

`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。

```

1. import { Component } from '@angular/core';
2.
3. import { MissionService } from './mission.service';
4.
5. @Component({
```

```

6.   selector: 'mission-control',
7.   template: `
8.     <h2>Mission Control</h2>
9.     <button (click)="announce()">Announce mission</button>
10.    <my-astronaut *ngFor="let astronaut of astronauts"
11.      [astronaut]="astronaut">
12.    </my-astronaut>
13.    <h3>History</h3>
14.    <ul>
15.      <li *ngFor="let event of history">{{event}}</li>
16.    </ul>
17.  `,
18.  providers: [MissionService]
19. }
20. export class MissionControlComponent {
21.   astronauts = ['Lovell', 'Swigert', 'Haise'];
22.   history: string[] = [];
23.   missions = ['Fly to the moon!',
24.                 'Fly to mars!',
25.                 'Fly to vegas!'];
26.   nextMission = 0;
27.
28.   constructor(private missionService: MissionService) {
29.     missionService.missionConfirmed$.subscribe(
30.       astronaut => {
31.         this.history.push(` ${astronaut} confirmed the mission`);
32.       });
33.   }
34.
35.   announce() {
36.     let mission = this.missions[this.nextMission++];
37.     this.missionService.announceMission(mission);
38.     this.history.push(`Mission "${mission}" announced`);
39.     if (this.nextMission >= this.missions.length) { this.nextMission =
40.       0; }
41.   }

```

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。

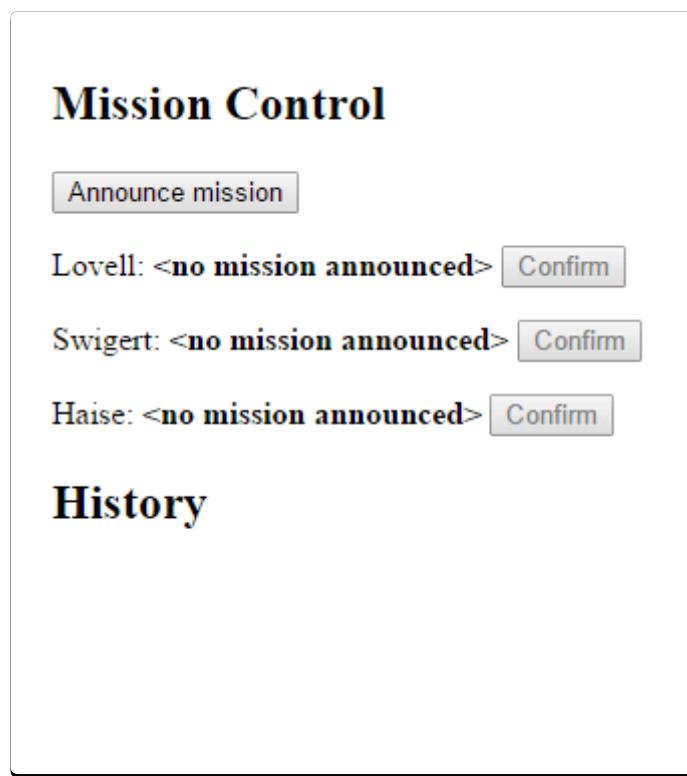
```
1. import { Component, Input, OnDestroy } from '@angular/core';
2.
3. import { MissionService } from './mission.service';
4. import { Subscription } from 'rxjs/Subscription';
5.
6. @Component({
7.   selector: 'my-astronaut',
8.   template: `
9.     <p>
10.       {{astronaut}}: <strong>{{mission}}</strong>
11.       <button
12.         (click)="confirm()"
13.         [disabled]="!announced || confirmed">
14.           Confirm
15.         </button>
16.       </p>
17.     `
18. })
19. export class AstronautComponent implements OnDestroy {
20.   @Input() astronaut: string;
21.   mission = '<no mission announced>';
22.   confirmed = false;
23.   announced = false;
24.   subscription: Subscription;
25.
26.   constructor(private missionService: MissionService) {
27.     this.subscription = missionService.missionAnnounced$.subscribe(
28.       mission => {
29.         this.mission = mission;
30.         this.announced = true;
31.         this.confirmed = false;
32.       });
33.   }
34.
35.   confirm() {
36.     this.confirmed = true;
37.     this.missionService.confirmMission(this.astronaut);
38.   }
39.
40.   ngOnDestroy() {
```

```
41.     // prevent memory leak when component destroyed
42.     this.subscription.unsubscribe();
43. }
44. }
```

注意，通过 `subscription` 服务订阅任务，并在 `AstronautComponent` 被销毁的时候退订。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，因为作为父组件，它控制着 `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。



测试

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。

```
1. // ...
2. it('should announce a mission', function () {
3.   let missionControl = element(by.tagName('mission-control'));
4.   let announceButton =
5.     missionControl.all(by.tagName('button')).get(0);
6.   announceButton.click().then(function () {
7.     let history = missionControl.all(by.tagName('li'));
8.     expect(history.count()).toBe(1);
9.     expect(history.get(0).getText()).toMatch(/Mission.* announced/);
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('mission-control'));
27.    let confirmButton =
28.      missionControl.all(by.tagName('button')).get(buttonIndex);
29.    confirmButton.click().then(function () {
30.      let history = missionControl.all(by.tagName('li'));
31.      expect(history.count()).toBe(expectedLogCount);
32.      expect(history.get(expectedLogCount - 1).getText()).toBe(astronaut
33.        + _confirmedLog);
34.    });
35.  }
36. // ...
```

[回到顶部](#)

相对于组件的路径

为组件的模板和样式指定相对于组件的路径

为组件模板和样式表文件提供 相对于组件的 URL

组件通常都是引用外部的模板和样式表文件。我们在 `@Component` 的元数据中通过 `templateUrl` 和 `styleUrls` 属性来标识出它们的位置：

```
@Component({
  selector: 'absolute-path',
  templateUrl: 'app/some.component.html',
  styleUrls:  ['app/some.component.css']
})
```

默认情况下，我们 **必须** 指定一个一直到应用程序根目录的完整路径。我们称之为 **绝对路径**，因为它 **绝对的** 以应用程序的根目录为基准。

使用 **绝对路径** 有两个问题：

1. 我们不得不记住到应用程序根目录的完整路径。
2. 当我们在应用的文件结构中移动这个组件时，将不得不更新这个 URL

如果能用 **相对** 于组件类文件的路径来指定模板和样式表的位置，那么编写和维护组件就会变得容易得多。

没问题！

如果把应用构建成 commonjs 模块，并用一个合适的包加载器（比如 systemjs 或 webpack）加载那些模块，就可以用相对路径。在下方可以学到原理。

Angular CLI(命令行界面) 使用这些技术，并默认采用这里所说的 **组件相对路径** 方法。用 CLI 用户可以跳过本章，或者继续阅读来了解它是怎么工作的。

组件相对 路径

目标是把模板和样式表的 URL 指定为 相对 于组件类的路径，因此得名 **组件相对路径**。

成功的关键是遵循一个约定：把相对组件的文件放进众所周知的位置。

建议把组件的模板和组件特有的样式表文件作为组件类文件的“兄弟”。这里在 `app` 目录下依次有 `SomeComponent` 的三个文件。

```
app
  └── some.component.css
  └── some.component.html
  └── some.component.ts
  ...
```

当应用规模增长后，还会有更多的文件和目录，目录深度也会增加。如果组件的所有文件总是像形影不离的兄弟那样共进退，那该多好啊！

设置 moduleId

采用这种文件结构约定，可以为模板和样式表文件指定相对于组件类文件的位置——只要简单的在 `@Component` 元数据中设置 `moduleId` 属性就可以了，就像这样：

```
moduleId: module.id,
```

从 `templateUrl` 和 `styleUrls` 中把基准路径 `app/` 去掉了。结果是这样的：

```
@Component({
  moduleId: module.id,
  selector: 'relative-path',
  templateUrl: 'some.component.html',
  styleUrls: ['some.component.css']
})
```

Webpack 用户可能更喜欢 [一个替代方案](#)。

源码

参见 [live example](#)，并从中下载源码或只在这里阅读相关源码。

```
1. import { Component } from '@angular/core';
2.
3. ////////////// Using Absolute Paths //////////
4.
5. @Component({
6.   selector: 'absolute-path',
7.   templateUrl: 'app/some.component.html',
8.   styleUrls: ['app/some.component.css']
9. })
10. export class SomeAbsoluteComponent {
11.   class = 'absolute';
12.   type = 'Absolute template & style URLs';
13.   path = 'app/path.component.html';
14. }
15.
16. ////////////// Using Relative Paths //////////
17.
18. @Component({
19.   moduleId: module.id,
20.   selector: 'relative-path',
```

```

21.     templateUrl: 'some.component.html',
22.     styleUrls:  ['some.component.css']
23.   })
24.
25.   export class SomeRelativeComponent {
26.     class = 'relative';
27.     type = 'Component-relative template & style URLs';
28.     path = 'path.component.html';
29.
30.   }

```

附录：为什么组件相对路径不是默认方式

组件相对路径 明显比 绝对路径 高级一点。为什么 Angular 默认采用了 绝对路径 呢？为什么 我们 不得不设置 `moduleId` 呢？Angular 为什么不能自己设置它？

首先，如果只使用相对路径而省略掉 `moduleId`，我们来看看会发生什么。

`EXCEPTION: Failed to load some.component.html`

Angular 找不到这个文件，所以它抛出一个错误。

为什么 Angular 不能相对于组件类文件的路径来自动计算模板和样式表的 URL 呢？

因为如果没有开发人员的帮助，组件的位置是检测不到的。Angular 应用可能被用多种方式加载：独立文件、SystemJS 包、CommonJS 包等等。用来生成模块的格式可以是任何格式。甚至可能完全没有写成模块化代码。

由于存在这么多打包和模块加载策略，所以 Angular 不可能知道在运行期这些文件的正确位置。

Angular 能够确定的唯一的位置是首页 `index.html` 的 URL，也就是应用的根目录。所以，默认情况下，它只能计算相对于 `index.html` 的模板和样式表路径。这就是为什么我们以前用 `app/` 基准路径的前缀来写文件的 URL。

但是，**如果** 遵循建议的指导原则，用 `commonjs` 格式编写模块，并使用一个 **不错的** 模块加载器，我们要知道，有一个可用的半全局变量 `module.id`，它包含组件类模块文件

的绝对 URL。

这种认知让我们得以通过设置 `moduleId` 来告诉 Angular **组件类** 文件在哪里：

```
moduleId: module.id,
```

Webpack: 加载模板和样式表

Webpack 开发者可以采用 `moduleId` 的另一个替代方案。

通过让组件元数据中的 `template` 和 `styles` / `styleUrls` 属性以 `./` 开头，并使其指向相对于组件的 URL，可以在运行期间为它们加载模板和样式表。

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

import './../public/css/styles.css';

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent { }
```

Webpack 将会在幕后执行一次 `require` 来加载这些模板和样式。要了解更多，请参阅 [这里](#)。

参见 [Webpack 简介](#)。

依赖注入

依赖注入技术

依赖注入是一个用来管理代码依赖的强大模式。在这本“烹饪宝典”中，我们会讨论 Angular 依赖注入的许多特性。

目录

[应用程序全局依赖](#)

[外部模块配置](#)

[@Injectable 与嵌套服务的依赖](#)

[把服务作用域限制到一个子组件树](#)

[多个服务实例 \(沙箱 \)](#)

[使用 @Optional 和 @Host 装饰器来限定依赖查找方式](#)

[注入组件的 DOM 元素](#)

[使用提供商定义依赖](#)

- [provide 对象](#)
- [useValue - 值提供商](#)
- [useClass - 类提供商](#)
- [useExisting - 别名提供商](#)

- [useFactory - 工厂提供商](#)

[使用对象字面量定义提供商](#)

[提供商可选令牌](#)

- [类 - 接口](#)
- [Opaque 令牌](#)

[注入到一个派生类](#)

[通过注入来查找父组件](#)

- [通过已知组件类型查找父组件](#)
- [无法通过自己的基类查找父组件](#)
- [通过类 - 接口查找父组件](#)
- [在父组件树里查找一个父组件 \(@SkipSelf\)](#)
- [provideParent 助手函数](#)

[使用类的前向引用 \(forwardRef\) 打破循环依赖](#)

要获取本“烹饪宝典”的代码，[参见 live example](#)。

应用程序全局依赖

在应用程序根组件 `AppComponent` 中注册那些被应用程序全局使用的依赖提供商。

在下面的例子中，通过 `@Component` 元数据的 `providers` 数组导入和注册了几个服务 (`LoggerService`，`UserContext` 和 `UserService`)。

app/app.component.ts (excerpt)

```

import { LoggerService }      from './logger.service';
import { UserContextService } from './user-context.service';
import { UserService }        from './user.service';

@Component({
  moduleId: module.id,
  selector: 'my-app',
  templateUrl: 'app.component.html',
  providers: [ LoggerService, UserContextService, UserService ]
})
export class AppComponent {
  /* . . . */
}

```

所有这些服务都是用类实现的。服务类能充当自己的提供商，这就是为什么只要把它们列在 `providers` 数组里就算注册成功了。

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 [下面](#) 可以学到更多关于提供商的知识。

现在我们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。

app/hero-bios.component.ts (component constructor injection)

```

constructor(logger: LoggerService) {
  logger.logInfo('Creating HeroBiosComponent');
}

```

app/user-context.service.ts (service constructor injection)

```

constructor(private userService: UserService, private loggerService:
LoggerService) {

```

}

外部模块配置

经常在 `NgModule` 中需要注册提供商，而不是在应用程序根组件中。

在下列两种情况下这么注册：(1) 希望服务在整个应用的每个角落都可以被注入，或者
(2) 必须在应用 **启动前** 注册一个全局服务。

下面的例子时第二种情况下，配置一个非默认的 `location strategy` 的路由器，把它加入到 `AppModule` 的 `providers` 数组中。

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。

比如，我们在 `AppComponent` 里注入的 `LoggerService` 和 `UserContext`。

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()`

注意在 `UserContextService` 类里面的 `@Injectable()` 装饰器。

user-context.service.ts (@Injectable)

```
@Injectable()  
export class UserContextService {  
}
```

该装饰器让 Angular 有能力识别这两个依赖 `LoggerService` 和 `UserService` 的类型。

严格来说，这个 `@Injectable()` 装饰器只在一个服务类有 **自己的依赖** 的时候，才是 **不可缺少** 的。`LoggerService` 不依赖任何东西，所以该日志服务在没有 `@Injectable()` 的时候应该也能工作，生成的代码也更少一些。

但是在给它添加依赖的那一瞬间，该服务就会停止工作，要想修复它，就必须添加 `@Injectable()`。为了保持一致性和防止将来的麻烦，推荐从一开始就加上 `@Injectable()`。

虽然推荐在所有服务中使用 `@Injectable()`，但你也不需要一定要这么做。一些开发者就更喜欢在真正需要的地方才添加，这也是一个合理的策略。

`AppComponent` 类有两个依赖，但它没有 `@Injectable()`。它不需要 `@Injectable()`，这是因为组件类有 `@Component` 装饰器。在用 TypeScript 的 Angular 应用里，有一个 **单独的** 装饰器—**任何** 装饰器—来标识依赖的类型就够了。

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。有时候我们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。这里通过列入 `providers` 数组，在 `HeroesBaseComponent` 中提供了 `HeroService`：

app/sorted-heroes.component.ts (HeroesBaseComponent excerpt)

```
1.  @Component({
2.    selector: '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. }
```

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。我们称作 **沙盒化**，因为每个服务和组件实例都在自己的沙盒里运行。

想象一下，一个 `HeroBiosComponent` 组件显示三个 `HeroBioComponent` 的实例。

ap/hero-bios.component.ts

```
1.  @Component({
2.    selector: 'hero-bios',
3.    template: `
4.      <hero-bio [heroId]="1"></hero-bio>
5.      <hero-bio [heroId]="2"></hero-bio>
6.      <hero-bio [heroId]="3"></hero-bio>`,
7.    providers: [HeroService]
8.  })
9.  export class HeroBiosComponent {
10. }
```

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。

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。

app/hero-bio.component.ts

```

1.  @Component({
2.    selector: '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.

```

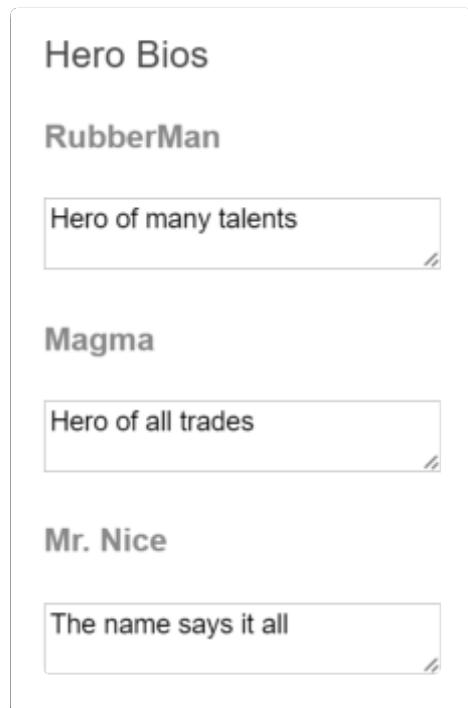
```

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](#) 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。



使用 `@Optional` 和 `@Host` 装饰器来限定依赖查找方式

我们学过，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，我们确实 **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。我们先思考一下，更多有趣的案例还在后面。

示范

`HeroBiosAndContactsComponent` 是前面见过的 `HeroBiosComponent` 的修改版。

app/hero-bios.component.ts (HeroBiosAndContactsComponent)

```

1.  @Component({
2.    selector: 'hero-bios-and-contacts',
3.    template: `
4.      <hero-bio [heroId]="1"> <hero-contact></hero-contact> </hero-bio>
5.      <hero-bio [heroId]="2"> <hero-contact></hero-contact> </hero-bio>
6.      <hero-bio [heroId]="3"> <hero-contact></hero-contact> </hero-
7.      bio>`,
8.    providers: [HeroService]
9.  })
10. export class HeroBiosAndContactsComponent {
11.   constructor(logger: LoggerService) {
12.     logger.logInfo('Creating HeroBiosAndContactsComponent');
13.   }

```

注意看模板：

```

template: `
<hero-bio [heroId]="1"> <hero-contact></hero-contact> </hero-bio>
<hero-bio [heroId]="2"> <hero-contact></hero-contact> </hero-bio>
<hero-bio [heroId]="3"> <hero-contact></hero-contact> </hero-bio>`,

```

我们在 `<hero-bio>` 标签中插入了 `<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。

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，看起来像这样：

RubberMan

Phone #: 123-456-7899
Hero of many talents

下面的 `HeroContactComponent`，示范了在本节一直在讨论的限定型装饰器 (@Optional 和 @Host)：

app/hero-contact.component.ts

```
1.  @Component({
2.    selector: 'hero-contact',
3.    template: `
4.      <div>Phone #: {{phoneNumber}}
5.      <span *ngIf="hasLogger">!!!</span></div>`
6.  })
7.  export class HeroContactComponent {
8.
9.    hasLogger = false;
10.
11.   constructor(
12.     @Host() // limit to the host component's instance of the
13.     heroCacheService
14.   ) {
15.     this.heroCache = heroCacheService;
16.   }
17. }
```

```

14.
15.      @Host()      // limit search for logger; hides the application-
   wide logger
16.      @Optional() // ok if the logger doesn't exist
17.      private loggerService: LoggerService
18.    )
19.    if (loggerService) {
20.      this.hasLogger = true;
21.      loggerService.logInfo('HeroContactComponent can log!');
22.    }
23.  }
24.
25.  get phoneNumber() { return this.heroCache.hero.phone; }
26.
27. }

```

注意看构造函数的参数

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

```

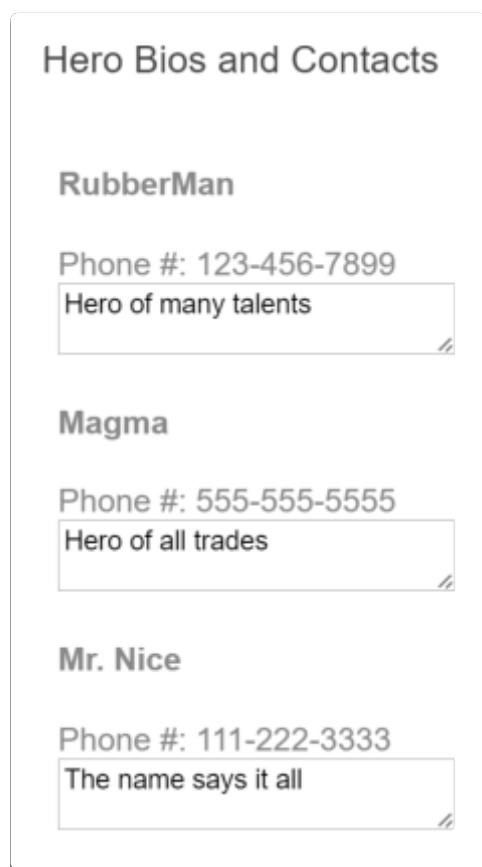
`@Host()` 函数是 `heroCache` 属性的装饰器，确保从其父组件 `HeroBioComponent` 得到一个缓存服务。如果该父组件不存在这个服务，Angular 就会抛出错误，即使组件树里的再上级有某个组件拥有这个服务，Angular 也会抛出错误。

另一个 `@Host()` 函数是属性 `loggerService` 的装饰器，我们知道在应用程序中，只有一个 `LoggerService` 实例，也就是在 `AppComponent` 级提供的服务。该宿主 `HeroBioComponent` 没有自己的 `LoggerService` 提供商。

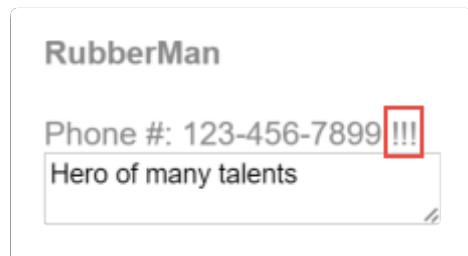
如果没有同时使用 `@Optional()` 装饰器的话，Angular 就会抛出错误。多亏了 `@Optional()`，Angular 把 `loggerService` 设置为 `null`，并继续执行组件而不会抛出错误。

我们将很快回到 `elementRef` 属性。

下面是 `HeroBiosAndContactsComponent` 的执行结果：



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。



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。

为了说明这一点，我们在 `属性型指令` `HighlightDirective` 的基础上，编写了一个简化版本。

app/highlight.directive.ts

```
1. import { Directive, ElementRef, HostListener, Input } from
   '@angular/core';
2.
3. @Directive({
4.   selector: '[myHighlight]'
5. })
6. export class HighlightDirective {
7.
8.   @Input('myHighlight') highlightColor: string;
9.
10.  private el: HTMLElement;
11.
12.  constructor(el: ElementRef) {
13.    this.el = el.nativeElement;
14.  }
15.
16.  @HostListener('mouseenter') onMouseEnter() {
17.    this.highlight(this.highlightColor || 'cyan');
18.  }
19.
20.  @HostListener('mouseleave') onMouseLeave() {
21.    this.highlight(null);
22.  }
23.
24.  private highlight(color: string) {
25.    this.el.style.backgroundColor = color;
```

```
26.      }
27.    }
```

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。

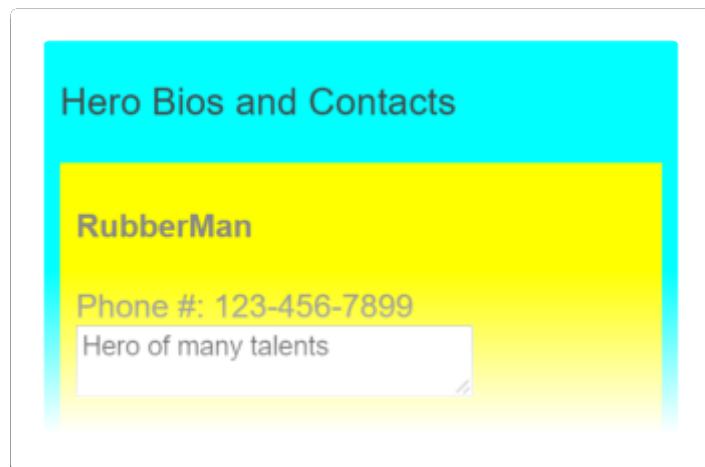
Angular 把构造函数参数 `el` 设置为注入的 `ElementRef`，该 `ElementRef` 代表了宿主的 DOM 元素，它的 `nativeElement` 属性把该 DOM 元素暴露给了指令。

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。

app/app.component.html (highlight)

```
<div id="highlight" class="di-component" myHighlight>
  <h3>Hero Bios and Contacts</h3>
  <div myHighlight="yellow">
    <hero-bios-and-contacts></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

在这个部分，我们学习如何编写提供商来提供被依赖的服务。

背景知识

我们通过给依赖注入器提供 **令牌** 来获取服务。

我们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：

app/herobios.component.ts (组件构造器注入)

```
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**。

让我们把注意力转回到提供商。

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。我们必须自行注册属于 **自己的** 的提供商，通常用 `组件` 或者 `指令` 元数据中的 `providers` 数组进行注册。

app/app.component.ts (提供商)

```
providers: [ LoggerService, UserContextService, UserService ]
```

定义提供商

简单的类提供商是最典型的例子。只要在 providers 数值里面提到该类就可以了。

app/herobios.component.ts (类提供商)

```
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。我们还需要其它的交付方式，这意味着我们也需要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。

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)
```

它看起来很简单：一些属性和一个日志输出。但代码的背后有很多可讨论的地方。

hero-of-the-month.component.ts

```
1. import { Component, Inject } from '@angular/core';
2.
3. import { DateLoggerService,
4.         MinimalLogger }    from './date-logger.service';
5. import { Hero }                  from './hero';
6. import { HeroService }          from './hero.service';
7. import { LoggerService }        from './logger.service';
8. import { RUNNERS_UP,
9.         runnersUpFactory }   from './runners-up';
10.
11. @Component({
```

```

12.     selector: 'hero-of-the-month',
13.     template: template,
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:
21.         [Hero, HeroService] }
22.     ]
23.   })
24.   export class HeroofTheMonthComponent {
25.     logs: string[] = [];
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** 对象有一个主属性（即 `useValue`），用来标识该提供商如何新建和返回依赖。

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。我们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：

```
{ 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 `OpaqueToken`。

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：

```
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：

```
{ 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，来反衬我们更喜欢的缩写形式。

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。

app/date-logger.service.ts

```
@Injectable()
export class DateLoggerService extends LoggerService implements MinimalLogger {
  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**。

```
{ 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。我们在这里就是为了这个目的使用的别名。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：

app/date-logger.service.ts (MinimalLogger)

```
// class used as a restricting interface (hides other public members)
export abstract class MinimalLogger {
  logInfo: (msg: string) => void;
  logs: string[];
}
```

构造函数的 `logger` 参数是一个 `MinimalLogger` 类型，所有在 TypeScript 里面，它只有两个成员可见：

```
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。

```
{ 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。我们立刻用 `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，并把它们的名字拼接起来。请到 [live example](#) 查看全部原代码。

备选提供商令牌：类 - 接口 和 OpaqueToken

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

在前面的 **每月英雄** 的例子中，我们用了 `MinimalLogger` 类作为 `LoggerService` 提供商的令牌。

```
{ provide: MinimalLogger, useExisting: LoggerService },
```

该 `MinimalLogger` 是一个抽象类。

```
// class used as a restricting interface (hides other public members)
export abstract class MinimalLogger {
  logInfo: (msg: string) => void;
  logs: string[];
}
```

我们通常从一个抽象类继承。但 `LoggerService` 并不继承 `MinimalLogger`。没有类会继承它。只把它当接口来使用。

请再看下 `DateLoggerService` 的声明

```
export class DateLoggerService extends LoggerService implements
MinimalLogger
```

`DateLoggerService` 继承(扩展)了 `LoggerService`，而不是 `MinimalLogger`。该 `DateLoggerService` 实现了 `MinimalLogger`，就像 `MinimalLogger` 是一个接口一样。

我们称这种用法的类叫做 **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** 定义它的消费者允许调用的成员。窄的接口有助于解耦该类的具体实现和它的消费者。该 `MinimalLogger` 只定义了两个 `LoggerClass` 的成员。

为什么 MINIMALLOGGER 是一个类而不是一个接口

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 ... 一个类。

把类当做接口使用，可以为我们在一个 JavaScript 对象上提供类似于接口的特性。

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：

```
var MinimalLogger = (function () {
  function MinimalLogger() {}
  return MinimalLogger;
}());
exports("MinimalLogger", MinimalLogger);
```

只要不实现它，不管添加多少成员，它永远不会增长大小。

OpaqueToken

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。

`OpaqueToken` 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。

```
{ provide: TITLE,           useValue: 'Hero of the Month' },
{ provide: RUNNERS_UP,     useFactory: runnersUpFactory(2), deps:
[Hero, HeroService] }
```

这样创建 `TITLE` 令牌：

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

export const TITLE = new OpaqueToken('title');
```

注入到一个派生类

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。

在这个生造的例子中，`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，并将它们按照数据库返回的顺序显示出来。

app/sorted-heroes.component.ts (HeroesBaseComponent)

```

1.  @Component({
2.    selector: '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. }
```

强烈推荐简单的构造函数。它们应该 只 用来初始化变量。这个规则会帮助我们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（比如连接服务）。这就是为什么我们要在 `ngOnInit` 里面调用 `HeroService`，而不是在构造函数中。

我们在下面解释这个神秘的 `afterGetHeroes`。

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。（我们说过这是生造的，仅用来解释这种机制）。

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。

app/sorted-heroes.component.ts (SortedHeroesComponent)

```

1.  @Component({
2.    selector: 'sorted-heroes',
3.    template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
4.    providers: [HeroService]
5.  })
6.  export class SortedHeroesComponent extends HeroesBaseComponent {
7.    constructor(heroService: HeroService) {
8.      super(heroService);
9.    }
10.
11.   protected afterGetHeroes() {
12.     this.heroes = this.heroes.sort((h1, h2) => {
13.       return h1.name < h2.name ? -1 :
14.         (h1.name > h2.name ? 1 : 0);
15.     });
16.   }
17. }
```

现在，请注意 `_afterGetHeroes` 方法。我们第一反应是在 `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。我们喜欢更加松耦合的技术，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，但它没有公开的 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 `Query`，`QueryList`，`ViewChildren`，和 `ContentChildren`）。

但没有公开的 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

我们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 ) { }
}
```

安全起见，我们添加了 `@Optional` 装饰器，但是 [live example](#) 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。为了合理（咳咳）的商业理由，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。我们没有东西可查。

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`]`{#alex}` , 我们看到 `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。 [live example](#) 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`。

我们编写一个 **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) }],
```

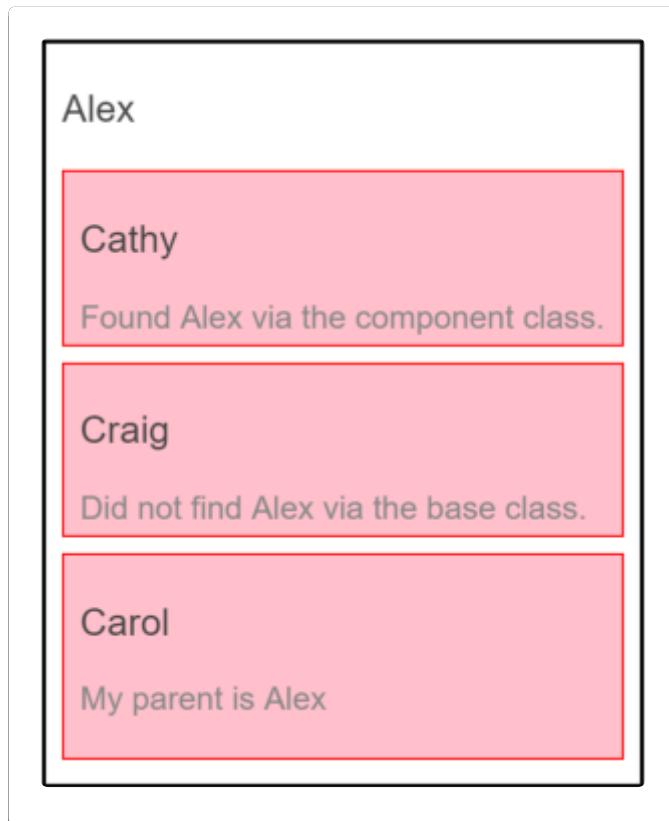
`Parent` 是该提供商的 **类 - 接口** 令牌。`AlexComponent` 引用了自身，造成循环引用，使用 `forwardRef` 打破了该循环。

`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，我们应该建立一个 **帮助函数**。

眼下，请注意 Barry 的构造函数：

```
constructor( @SkipSelf() @Optional() public parent: Parent ) { }
```

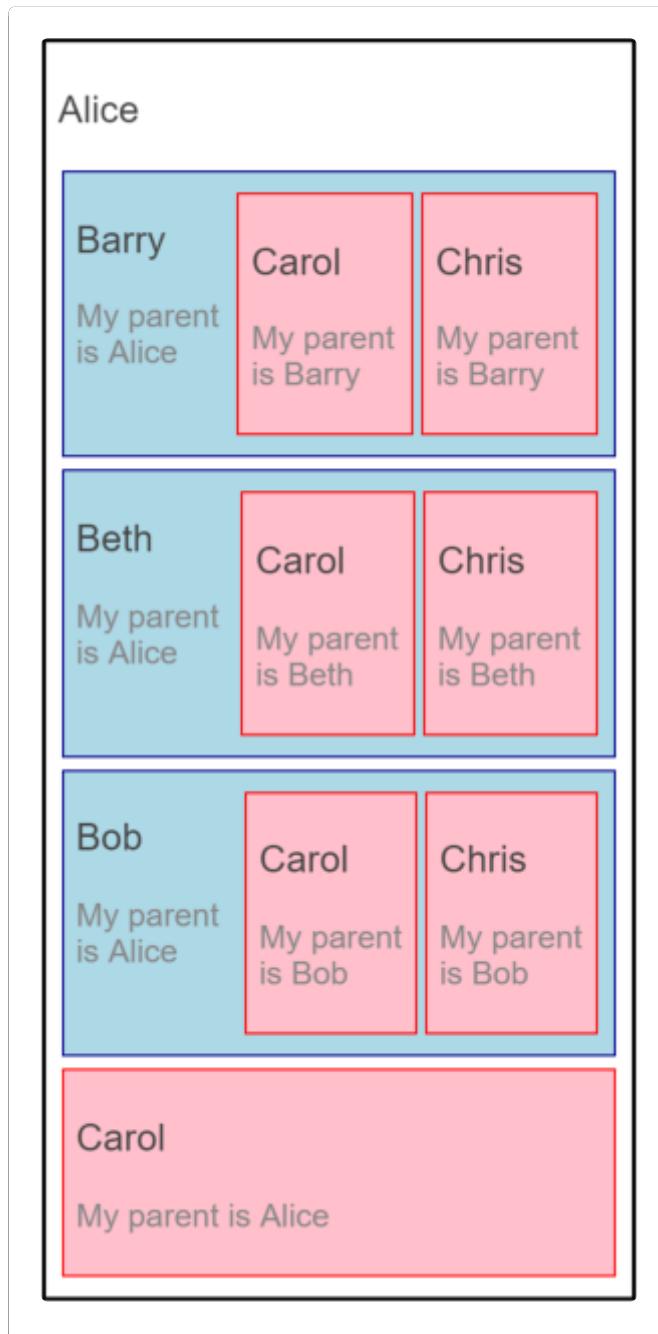
除额外添加了一个的 `@SkipSelf` 外，它和 Carol 的构造函数一样。

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

我们 [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。

我们的例子定义了一个 **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：

```
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：

```
// 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：

```
providers: [ provideParent(AliceComponent) ]
```

我们可以做得更好。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**。

```
// 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：

```
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，特别是当我们遵循 **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，我们就陷入困境了：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，我们面临同样的困境，就像在 `AlexComponent` 的 `providers` 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。

我们使用 `forwardRef` 来打破这种循环：

parent-finder.component.ts (AlexComponent providers)

```
providers: [{ provide: Parent, useExisting: forwardRef(() =>
  AlexComponent) }],
```

动态表单

用 FormGroup 渲染动态表单

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。

在此烹饪宝典中，我们会展示如何利用 `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。

在本例中，我们使用动态表单，为正在找工作的英雄们创建一个在线申请表。英雄管理局会不断修改申请流程，我们要在 **不修改应用代码** 的情况下，动态创建这些表单。

目录

[程序启动](#)

[问卷问题模型](#)

[表单组件](#)

[问卷元数据](#)

[动态模板](#)

参见 [在线例子](#)。

程序启动

让我们从创建一个名叫 `AppModule` 的 `NgModule` 开始。

在本例子中，我们将使用响应式表单（Reactive Forms）。

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，我们得往 `AppModule` 中引入 `ReactiveFormsModule` 模块。

我们在 `main.ts` 中启动 `AppModule`。

```
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,
12.                 DynamicFormQuestionComponent ],
13.   bootstrap: [ AppComponent ]
14. })
15. export class AppModule {
16.   constructor() {
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。问卷问题是基础的对象模型。

下面是我们建立的最基础的问卷问题基类，名叫 `QuestionBase`。

app/question-base.ts

```

1.  export class QuestionBase<T>{
2.    value: T;
3.    key: string;
4.    label: string;
5.    required: boolean;
6.    order: number;
7.    controlType: string;
8.
9.    constructor(options: {
10.      value?: T,
11.      key?: string,
12.      label?: string,
13.      required?: boolean,
14.      order?: number,
15.      controlType?: string
16.    } = {}) {
17.      this.value = options.value;
18.      this.key = options.key || '';
19.      this.label = options.label || '';
20.      this.required = !!options.required;
21.      this.order = options.order === undefined ? 1 : options.order;
22.      this.controlType = options.controlType || '';
23.    }
24.  }

```

在这个基础上，我们派生出两个新类 `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。

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。

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，我们定义了 `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，并允许我们设置默认值和验证规则。

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

现在我们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。

```

1.  <div>
2.    <form (ngSubmit)="onSubmit()" [formGroup]="form">
3.
4.      <div *ngFor="let question of questions" class="form-row">
5.        <df-question [question]="question" [form]="form"></df-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 `<df-question>` 组件元素。`<df-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。

```

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'"
10.             [formControlName]="question.key">
11.        <option *ngFor="let opt of question.options" [value]= "opt.key">
12.          {{opt.value}}</option>
13.        </select>
14.
15.      <div class="errorMessage" *ngIf="!isValid">{{question.label}} is
16.        required</div>
</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，我们依赖 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。我们之所以能在模板中使用它们，是因为我们往 `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，我们会从数据库里获得这些问题列表。

关键是，我们完全根据 `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、更新和删除 `questions` 数组中的对象就可以了。

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.
29.     return questions;
30.   }
31. }
```

```
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: 'my-app',
7.   template: `
8.     <div>
9.       <h2>Job Application for Heroes</h2>
10.      <dynamic-form [questions]="questions"></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，虽然我们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。

完整的表单看起来是这样的：

The form is titled "Job Application for Heroes". It contains three input fields:

- "First name": Input field containing "Bombasto".
- "Email": Input field currently empty.
- "Bravery Rating": A dropdown menu with the option "Solid" selected.

A "Save" button is located at the bottom of the form.

[回到顶部](#)

表单验证

验证用户在表单中的输入

我们可以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。

在本烹饪书中，我们展示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 [表单章节](#) 了解关于这些选择的更多知识。

目录

[简单的模板驱动表单](#)

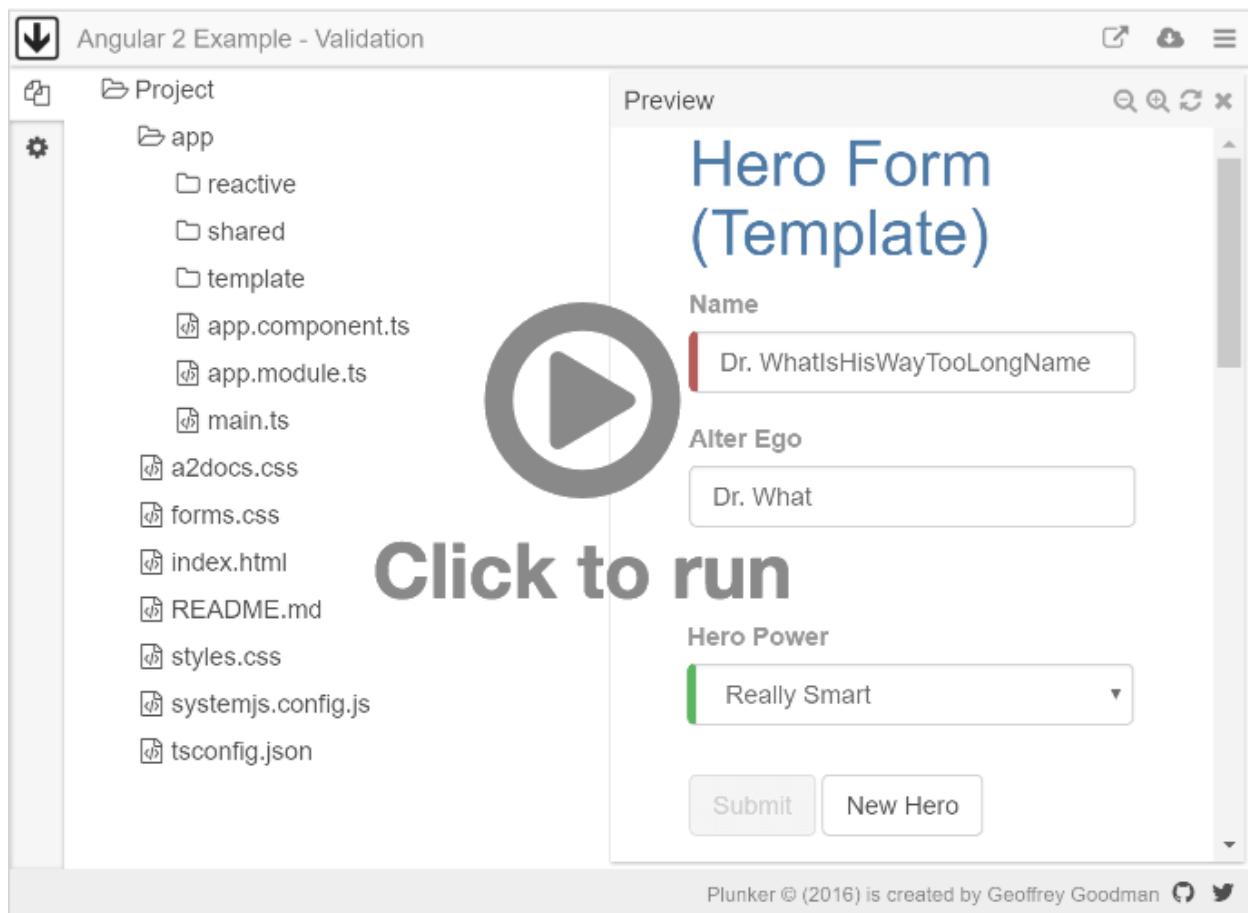
[代码中拥有验证信息的模板驱动表单](#)

[代码中拥有验证的响应式表单](#)

[自定义验证](#)

[测试](#)

[查看在线例子，并下载整个烹饪书的源代码](#)



简单的模板驱动表单

在模板驱动表单方法中，你在组件的模板中组织 表单元素 。

你可以添加 Angular 表单指令（通常为以 `ng` 开头的指令）来帮助 Angular 构建对应的内部控制模型，以实现表单功能。 控制模型在模板中是 隐式 的。

要验证用户输入，你添加 `HTML 验证属性` 到元素中。 Angular 拦截这些元素，添加验证器函数到控制模型中。

Angular 暴露关于控制状态的信息，包括用户是否已经“触摸”了控制器，或者用户已经作了更新和控制器的值是否还有效。

在第一个模板验证例子中，我们添加了更多 `HTML`，来读取控制器状态并适当更新显示。 下面是模板 `HTML` 中提取的，一个绑定到英雄名字的输入框控制器：

`template/hero-form-template1.component.html (Hero name)`

```
<label for="name">Name</label>

<input type="text" id="name" class="form-control"
       required minlength="4" maxlength="24"
       name="name" [(ngModel)]="hero.name"
       #name="ngModel" >

<div *ngIf="name.errors && (name.dirty || name.touched)"
      class="alert alert-danger">
    <div [hidden]="!name.errors.required">
      Name is required
    </div>
    <div [hidden]="!name.errors.minLength">
      Name must be at least 4 characters long.
    </div>
    <div [hidden]="!name.errors.maxLength">
      Name cannot be more than 24 characters long.
    </div>
</div>
```

请注意一下几点：

- `<input>` 元素具有 HTML 验证属性：`required`、`minlength`、和 `maxlength`。
- 我们设置输入框的 `name` 属性为 `"name"`，这样 Angular 可以跟踪这个输入元素，并将其内部控制器模型的一个名为 `name` 的 Angular 表单控制关联起来。
- 我们使用 `[(ngModel)]` 指令，将输入框双向数据绑定到 `hero.name` 属性。
- 我们将模板变量 (`#name`) 赋值为 `"ngModel"` (总是 `ngModel`)。它为我们提供了与这个控制器关联的 Angular `NgModel` 指令的引用，我们在模板中使用它，以检查控制器状态，比如 `valid` 和 `dirty`。
- `<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。我们已经为 `required`、`minlength`、和 `maxlength` 准备了消息。

整个模板为表单上的每种数据输入控制器重复这种布局。

为何检查 DIRTY 和 TOUCHED ?

当用户创建一个新英雄时，在还没有机会输入之前，我们不应该显示任何错误。检查 `dirty` 和 `touched` 防止了这种过早的错误显示。

参见 [表单](#) 章，学习关于 `dirty` 和 `touched` 的知识。

组件类管理用于数据绑定的英雄模型，它还有其他支持视图的代码。

template/hero-form-template1.component.ts (class)

```
1.  export class HeroFormTemplate1Component {
2.
3.    powers = ['Really smart', 'Super Flexible', 'Weather Changer'];
4.
5.    hero = new Hero(18, 'Dr. whatIsHisWayTooLongName', this.powers[0],
6.      'Dr. what');
7.
8.    submitted = false;
9.
10.   onsubmit() {
11.     this.submitted = true;
12.   }
13.
14.   addHero() {
15.     this.hero = new Hero(42, '', '');
16.   }
}
```

在处理简单的、拥有标准验证规则的静态表单时，使用这种模板驱动验证方法。

下面是第一个版本的使用模板驱动方法的 `HeroFormTemplateComponent`：

```
1.  <div class="container">
2.    <div [hidden]="submitted">
3.      <h1>Hero Form 1 (Template)</h1>
4.      <form #heroForm="ngForm" *ngIf="active" (ngSubmit)="onSubmit()">
5.        <div class="form-group">
6.          <label for="name">Name</label>
7.
8.          <input type="text" id="name" class="form-control"
9.                 required minlength="4" maxlength="24"
10.                name="name" [(ngModel)]="hero.name"
11.               #name="ngModel" >
12.
13.          <div *ngIf="name.errors && (name.dirty || name.touched)">
14.            class="alert alert-danger">
15.              <div [hidden]="!name.errors.required">
16.                Name is required
17.              </div>
18.              <div [hidden]="!name.errors.minLength">
19.                Name must be at least 4 characters long.
20.              </div>
21.              <div [hidden]="!name.errors.maxLength">
22.                Name cannot be more than 24 characters long.
23.              </div>
24.            </div>
25.          </div>
26.
27.          <div class="form-group">
28.            <label for="alterEgo">Alter Ego</label>
29.            <input type="text" id="alterEgo" class="form-control"
30.                   name="alterEgo"
31.                   [(ngModel)]="hero.alterEgo" >
32.          </div>
33.
34.          <div class="form-group">
35.            <label for="power">Hero Power</label>
36.            <select id="power" class="form-control"
37.                   name="power"
38.                   [(ngModel)]="hero.power" required
39.                   #power="ngModel" >
40.             <option *ngFor="let p of powers" [value]="p">{{p}}</option>
41.           </select>
42.
```

```

43.          <div *ngIf="power.errors && power.touched" class="alert alert-
danger">
44.              <div [hidden]="!power.errors.required">Power is
        required</div>
45.          </div>
46.      </div>
47.
48.          <button type="submit" class="btn btn-default"
49.                  [disabled]="!heroForm.form.valid">Submit</button>
50.          <button type="button" class="btn btn-default"
51.                  (click)="addHero()">New Hero</button>
52.      </form>
53.  </div>
54.
55.  <hero-submitted [hero]="hero" [(submitted)]="submitted"></hero-
submitted>
56. </div>

```

验证消息在代码中的模板驱动表单

虽然布局很直观，但是我们处理验证消息的方法有明显的缺陷：

- 它使用了很多 HTML 来表现所有可能出现的错误情况。如果有太多控制器和太多验证规则，我们就失去了控制。
- 我们不喜欢在 HTML 里面插入这么多 JavaScript。
- 这些消息是静态的字符串，被硬编码到模板中。我们通常要求在代码中可以塑造的动态消息。

只需要对模板和组件做出一些修改，我们可以将逻辑和消息移到组件中。

下面也是关于英雄名字的控制器，从修改后的模板（"Template 2"）中抽取出来，与原来的版本相比：

```
1.      <label for="name">Name</label>
```

```

2.
3.      <input type="text" id="name" class="form-control"
4.              required minlength="4" maxlength="24"
5.              forbiddenName="bob"
6.              name="name" [(ngModel)]="hero.name" >
7.
8.      <div *ngIf="formErrors.name" class="alert alert-danger">
9.          {{ formErrors.name }}
</div>

```

<input> 元素的 HTML 几乎一样。但是下列有值得注意的区别：

- 硬编码的错误消息 <div> 消失了。
- 添加了一个新属性 `forbiddenName`，它实际上是一个自定义验证指令。如果用户名字中的任何地方输入“bob”，该指令将控制器标记为无效（试试）。我们在本烹饪书后面介绍了[自定义验证指令](#)。
- 模板变量 `#name` 消失了，因为我们不再需要为这个元素引用 Angular 控制器。
- 绑定到新的 `formErrors.name` 属性，就可以处理所有名字验证错误信息了。

组件类

原来的组件代码还是一样。我们[添加](#)了新的代码，来获取 Angular 表单控制器和撰写错误信息。

第一步是获取 Angular 通过查询模板而生成的表单控制器。

回头看组件模板顶部，我们在 <form> 元素中设置 `#heroForm` 模板变量：

`template/hero-form-template1.component.html (form tag)`

```
<form #heroForm="ngForm" *ngIf="active" (ngSubmit)="onSubmit()">
```

`heroForm` 变量是 Angular 从模板衍生出来的控制模型的引用。我们利用 `@ViewChild` 来告诉 Angular 注入这个模型到组件类的 `currentForm` 属性：

template/hero-form-template2.component.ts (heroForm)

```
heroForm: NgForm;
@ViewChild('heroForm') currentForm: NgForm;

ngAfterViewChecked() {
  this.formchanged();
}

formChanged() {
  if (this.currentForm === this.heroForm) { return; }
  this.heroForm = this.currentForm;
  if (this.heroForm) {
    this.heroForm.valueChanges
      .subscribe(data => this.onValueChanged(data));
  }
}
```

一些细节：

- Angular 的 `@ViewChild` 使用传入的模板变量的字符串名字（这里是 `'heroForm'`），来查询对应的模板变量。
- `heroForm` 对象在组件的生命周期内变化了好几次，最值得注意的是当我们添加一个新英雄时的变化。我们必须定期重新检测它。
- 当视图有任何变化时，Angular 调用 `ngAfterViewChecked` 生命周期钩子方法。这是查看是否有新 `heroForm` 对象的最佳时机。
- 当出现新 `heroForm` 模型时，我们订阅它的 `valueChanged` 可观察属性。

`onValueChanged` 处理器在每次用户键入后查找验证错误。

template/hero-form-template2.component.ts (handler)

```

onvalueChanged(data?: any) {
  if (!this.heroForm) { return; }
  const form = this.heroForm.form;

  for (const field in this.formErrors) {
    // clear previous error message (if any)
    this.formErrors[field] = '';
    const control = form.get(field);

    if (control && control.dirty && !control.valid) {
      const messages = this.validationMessages[field];
      for (const key in control.errors) {
        this.formErrors[field] += messages[key] + ' ';
      }
    }
  }

  formErrors = {
    'name': '',
    'power': ''
  };
}

```

`onValueChange` 处理器拦截用户数据输入。包含当前元素值得 `data` 对象被传入处理器。处理器忽略它们。相反，它迭代组件的 `formErrors` 对象。

`formErrors` 是一个词典，包含了拥有验证规则和当前错误消息的英雄控件。只有两个英雄属性有验证规则，`name` 和 `power`。当英雄数据有效时，这些消息的值为空字符串。

对于每个控件，这个处理器：

- 如果有之前的错误信息，清楚它们
- 获取控件对应的 Angular 表单控制器
- 如果这样的控制器存在，并且它被更新（“dirty”）以及它无效 ...
- 处理器便为所有控制器的错误合成一条错误消息。

很显然，我们需要一些错误消息，每个验证的属性都需要一套，每个验证规则需要一条消息：

template/hero-form-template2.component.ts (messages)

```
validationMessages = {
  'name': {
    'required': 'Name is required.',
    'minlength': 'Name must be at least 4 characters long.',
    'maxlength': 'Name cannot be more than 24 characters long.',
    'forbiddenName': 'Someone named "Bob" cannot be a hero.'
  },
  'power': {
    'required': 'Power is required.'
  }
};
```

现在，每次用户作出变化时，`onValueChange` 处理器检查验证错误并按情况发出错误消息。

这是增强吗？

很显然，模板变得小多了，组件代码变得大多了。当只有三个控件并且其中只有两个有验证规则时，我们很难看出好处。

假设增加需要验证的控件和规则后会怎么样。通常，HTML 比代码更难阅读和维护。初始的模板已经很大了，如果我们添加更多验证消息 `<div>`，它会迅速更大。

将验证消息移到组件后，模板的增长变得更加缓慢，幅度也小一些。不管有多少个验证规则，每个控件的行数是差不多的。组件也按比例增长，每增加一个控件增加一行，每个验证消息一行。

两条线容易维护。

现在消息在代码中，我们有更多的灵活度。我们更加智能的撰写消息。我们可以将消息重构出组件，比如到一个服务类，从服务端获取消息。简而言之，有很多机会增强消息处理，因为文本和逻辑都已经从模板移到代码中。

FormModule 和模板驱动表单

Angular 有两种不同的表单模块 - `FormsModule` 和 `ReactiveFormsModule` - 它们与表单开发的两种方法对应。两种模块都从同一个 `@angular/forms` 库。

我们一直在探讨 **模板驱动** 方法，它需要 `FormsModule`。下面是如何在 `HeroFormTemplateModule` 中导入它：

template/hero-form-template.module.ts

```
import { NgModule }      from '@angular/core';
import { FormsModule }  from '@angular/forms';

import { SharedModule }           from '../shared/shared.module';
import { HeroFormTemplate1Component } from './hero-form-
template1.component';
import { HeroFormTemplate2Component } from './hero-form-
template2.component';

@NgModule({
  imports:      [ SharedModule, FormsModule ],
  declarations: [ HeroFormTemplate1Component,
    HeroFormTemplate2Component ],
  exports:       [ HeroFormTemplate1Component,
    HeroFormTemplate2Component ]
})
export class HeroFormTemplateModule { }
```

我们还没有讲 `SharedModule` 或者它的 `SubmittedComponent`，它们出现在本烹饪书的每一个表单模板中。

它们与表单验证没有紧密的关系。如果你感兴趣，参见 [在线例子](#)。

响应式表单

在模板驱动方法中，你在模板中标出表单元素、验证属性和 Angular `FormsModule` 中的 `ng...` 指令。在运行时间，Angular 解释模板并从 `表单控制器模型` 衍生它。

响应式表单 采用不同的方法。你在代码中创建表单控制器模型，并用表单元素和来自 Angular `ReactiveFormsModule` 中的 `form...` 指令来编写模板。在运行时间，Angular 根据你的指示绑定模板元素到你的控制器模型。

这个方法需要做一些额外的工作。**你必须编写并管理控制器模型 ***

作为回报，你可以：

- 随时添加、修改和删除验证函数
- 在组件内动态操纵控制器模型
- 使用孤立单元测试来 [测试](#) 验证和控制器逻辑

第三个烹饪书例子用 **响应式表单** 风格重新编写英雄表格。

切换到 `ReactiveFormsModule`

响应式表单类和指令来自于 Angular 的 `ReactiveFormsModule`，不是 `FormsModule`。本例中，应用模块的“响应式表单”特性是这样的：

app/reactive/hero-form-reactive.module.ts

```
import { NgModule }           from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';

import { SharedModule }        from '../shared/shared.module';
import { HeroFormReactiveComponent } from './hero-form-
reactive.component';

@NgModule({
  imports:      [ SharedModule, ReactiveFormsModule ],
  declarations: [ HeroFormReactiveComponent ],
  exports:      [ HeroFormReactiveComponent ]
})
export class HeroFormReactiveModule { }
```

“响应式表单”特性模块和组件在 `app/reactive` 目录。让我们关注那里的 `HeroFormReactiveComponent`，先看它的模板。

组件模板

我们先修改 `<form>` 标签，让 Angular 的 `FormGroup` 指令绑定到组件类的 `heroForm` 属性。`heroForm` 是组件类创建和维护的控制器模型。

```
<form [FormGroup]="heroForm" *ngIf="active"
(ngSubmit)="onSubmit()">
```

接下来，我们修改模板 HTML 元素，来匹配 **响应式表单** 样式。下面又是“name”部分的模板，响应式表单修改版本和模板驱动版本的比较：

```
1.  <label for="name">Name</label>
2.
3.  <input type="text" id="name" class="form-control"
       formControlName="name" required>
4.
5.
6.  <div *ngIf="formErrors.name" class="alert alert-danger">
7.    {{ formErrors.name }}
8.  </div>
```

关键变化：

- 验证属性没有了（除了 `required`），因为我们将在代码中验证。
- 保留了 `required`，不是为了验证目的（我们将在代码中再行解释），而是为了 CSS 样式和可访问性。

未来版本的响应式表单将会在控制器有 `required` 验证器函数时，添加 `required` HTML 验证属性到 DOM 元素（也可能添加 `aria-required` 属性）。

在此之前，添加 `required` 属性 **以及** 添加 `Validator.required` 函数到控制器模型，像我们下面这样做：

- `formControlName` 替换了 `name` 属性；它起到了关联输入框和 Angular 表单控制器的同样作用。
- 双向 `[(ngModel)]` 绑定消失了。响应式表单方法不使用数据绑定从表单控制器移入和移出数据。我们在代码中做这些。

不适用表单数据绑定是响应式模式的原则，而非技术局限。

组件类

组件类现在负责定义和管理表单控制器模型。

Angular 不再从模板衍生控制器模型，所以我们不能再查询它。我们利用 `FormBuilder` 来显式创建 Angular 表单控制器模型。

下面是负责该进程的代码部分，与被它取代的模板驱动代码相比：

```
1.      heroForm: FormGroup;
2.      constructor(private fb: FormBuilder) { }
3.
4.      ngOnInit(): void {
5.          this.buildForm();
6.      }
7.
8.      buildForm(): void {
9.          this.heroForm = this.fb.group({
10.              'name': [this.hero.name, [
11.                  validators.required,
12.                  validators.minLength(4),
13.                  validators.maxLength(24),
14.              ],
15.          ],
16.      });
17.  }
```

```

14.         forbiddenNameValidator(/bob/i)
15.     ]
16.   ],
17.   'alterEgo': [this.hero.alterEgo],
18.   'power': [this.hero.power, Validators.required]
19. );
20.
21.   this.heroForm.valueChanges
22.     .subscribe(data => this.onValueChanged(data));
23.
24.   this.onValueChanged(); // (re)set validation messages now
25. }

```

- 我们注入 `FormBuilder` 到构造函数中。
- 我们在 `ngOnInit` 生命周期钩子方法 中调用 `buildForm` 方法，因为这正是我们拥有英雄数据的时刻。我们将在 `addHero` 方法中再次调用它。

真实的应用很可能从数据服务异步获取英雄，这个任务最好在 `ngOnInit` 生命周期钩子中进行。

- `buildForm` 方法使用 `FormBuilder` (`fb`) 来声明表单控制器模型。然后它将相同的 `onValueChange` (有一行代码不一样) 处理器附加到表单的 `valueChanged` 事件，并立刻为新的控制器模型设置错误消息。

FORMBUILDER 声明

`FormBuilder` 声明对象指定了本例英雄表单的三个控制器。

每个控制器的设置都是控制器名字和数组值。第一个数组元素是英雄控件对应的当前值。第二个值（可选）是验证器函数或者验证器函数数组。

大多数验证器函数是 Angular 以 `Validators` 类的静态方法的形式提供的原装验证器。Angular 有一些原装验证器，与标准 HTML 验证属性一一对应。

"name" 控制器上的 `forbiddenNames` 验证器是自定义验证器，在下面单独的 [小结](#) 有所讨论。

到 [即将到来](#) 的响应式表单章，学习更多关于 `FormBuilder` 的知识。

提交英雄值的更新

在双向数据绑定时，用户的修改自动从控制器流向数据模型属性。响应式表单不适用数据绑定来更新数据模型属性。开发者决定 **何时** 与 **如何** 从控制器的值更新数据模型。

本例更新模型两次：

1. 当用户提交标单时
2. 当用户选择添加新英雄

`onSubmit` 方法直接使用表单的值得合集来替换 `hero` 对象：

```
onSubmit() {
  this.submitted = true;
  this.hero = this.heroForm.value;
}
```

本例非常“幸运”，因为 `heroForm.value` 属性 **正好** 与英雄数据对象属性对应。

`addHero` 方法放弃未处理的变化，并创建一个崭新的 `hero` 模型对象。

```
addHero() {
  this.hero = new Hero(42, '', '');
```

```

    this.buildForm();
}

```

然后它再次调用 `buildForm`，用一个新对象替换了之前的 `heroForm` 控制器模型。

`<form>` 标签的 `[FormGroup]` 绑定使用这个新的控制器模型更新页面。

下面是完整的响应式表单的组件文件，与两个模板驱动组件文件对比：

```

1. import { Component, OnInit }                      from '@angular/core';
2. import { FormGroup, FormBuilder, Validators } from '@angular/forms';
3.
4. import { Hero }                               from '../shared/hero';
5. import { forbiddenNameValidator } from '../shared/forbidden-
name.directive';
6.
7. @Component({
8.   moduleId: module.id,
9.   selector: 'hero-form-reactive3',
10.  templateUrl: 'hero-form-reactive.component.html'
11. })
12. export class HeroFormReactiveComponent implements OnInit {
13.
14.   powers = ['Really Smart', 'Super Flexible', 'Weather Changer'];
15.
16.   hero = new Hero(18, 'Dr. WhatIsHisName', this.powers[0], 'Dr.
what');
17.
18.   submitted = false;
19.
20.   onSubmit() {
21.     this.submitted = true;
22.     this.hero = this.heroForm.value;
23.   }
24.   addHero() {
25.     this.hero = new Hero(42, '', '');
26.     this.buildForm();
27.
28.     this.active = false;
29.     setTimeout(() => this.active = true, 0);

```

```
30. }
31.
32.     heroForm: FormGroup;
33.     constructor(private fb: FormBuilder) { }
34.
35.     ngOnInit(): void {
36.         this.buildForm();
37.     }
38.
39.     buildForm(): void {
40.         this.heroForm = this.fb.group({
41.             'name': [this.hero.name, [
42.                 validators.required,
43.                 validators.minLength(4),
44.                 validators.maxLength(24),
45.                 forbiddenNamevalidator(/bob/i)
46.             ]
47.         ],
48.             'alterEgo': [this.hero.alterEgo],
49.             'power': [this.hero.power, validators.required]
50.         });
51.
52.         this.heroForm.valueChanges
53.             .subscribe(data => this.onValueChanged(data));
54.
55.         this.onValueChanged(); // (re)set validation messages now
56.     }
57.
58.
59.     onValueChanged(data?: any) {
60.         if (!this.heroForm) { return; }
61.         const form = this.heroForm;
62.
63.         for (const field in this.formErrors) {
64.             // clear previous error message (if any)
65.             this.formErrors[field] = '';
66.             const control = form.get(field);
67.
68.             if (control && control.dirty && !control.valid) {
69.                 const messages = this.validationMessages[field];
70.                 for (const key in control.errors) {
71.                     this.formErrors[field] += messages[key] + ' ';
72.                 }
73.             }
74.         }
75.     }
76.
```

```

73.      }
74.    }
75.  }
76.
77.  formErrors = {
78.    'name': '',
79.    'power': ''
80.  };
81.
82.  validationMessages = {
83.    'name': {
84.      'required': 'Name is required.',
85.      'minlength': 'Name must be at least 4 characters long.',
86.      'maxlength': 'Name cannot be more than 24 characters long.',
87.      'forbiddenName': 'Someone named "Bob" cannot be a hero.'
88.    },
89.    'power': {
90.      'required': 'Power is required.'
91.    }
92.  };
93.}

```

运行 [在线例子](#)，查看响应式表单是的行为，并与本章中的例子文件作比较。

自定义验证

本烹饪书例子有一个自定义 `forbiddenNameValidator` 函数，在模板驱动和响应式表单中都有使用。它在 `app/shared` 目录，在 `SharedModule` 中被声明。

下面是 `forbiddenNameValidator` 函数：

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 name = control.value;
  const no = nameRe.test(name);
  return no ? {'forbiddenName': {name}} : null;
};

}

```

该函数其实是一个工厂函数，接受一个正则表达式，用来检测 **指定** 的禁止的名字，并返回验证器函数。

在本例中，禁止的名字是 "bob"；验证器拒绝任何带有 "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，我们可以用来插入错误信息（`{name}`）。

在 **即将到来** 的自定义表单验证章节，学习更多关于验证器函数的知识。

自定义验证指令

在响应式表单组件中，我们在 `'name'` 控制器的验证函数列表的底部添加了一个配置了的 `forbiddenNameValidator`。

reactive/hero-form-reactive.component.ts (name validators)

```

'name': [this.hero.name, [
  Validators.required,
  Validators.minLength(4),
  Validators.maxLength(24),
  forbiddenNameValidator(/bob/i)
],
],

```

在模板驱动组件的模板中，我们在 name 的输入框元素中添加了自定义 **属性指令** 的选择器（`forbiddenName`），并配置它来拒绝“ bob ”。

template/hero-form-template2.component.html (name input)

```
<input type="text" id="name" class="form-control"
       required minlength="4" maxlength="24" forbiddenName="bob"
       name="name" [(ngModel)]="hero.name" >
```

对应的 `ForbiddenValidatorDirective` 包装了 `forbiddenNamevalidator`。

Angular 表单接受指令在验证流程中的作用，因为指令注册自己到 `NG_VALIDATORS` 提供商中，该提供商拥有可扩展的验证指令集。

shared/forbidden-name.directive.ts (providers)

```
providers: [{provide: NG_VALIDATORS, useExisting:
  ForbiddenValidatorDirective, multi: true}]
```

指令的剩余部分没有什么特殊的，所以我们将它展示在下面，不作任何注解。

shared/forbidden-name.directive.ts (directive)

```
1.  @Directive({
2.    selector: '[forbiddenName]',
3.    providers: [{provide: NG_VALIDATORS, useExisting:
  ForbiddenValidatorDirective, multi: true}]
4.  })
5.  export class ForbiddenvalidatorDirective implements Validator,
  OnChanges {
6.    @Input() forbiddenName: string;
7.    private valFn = Validators.nullValidator;
8.
9.    ngOnChanges(changes: SimpleChanges): void {
10.      const change = changes['forbiddenName'];
11.      if (change) {
```

```
12.     const val: string | RegExp = change currentValue;
13.     const re = val instanceof RegExp ? val : new RegExp(val, 'i');
14.     this.valFn = forbiddenNameValidator(re);
15.   } else {
16.     this.valFn = validators.nullValidator;
17.   }
18. }
19.
20. validate(control: AbstractControl): {[key: string]: any} {
21.   return this.valFn(control);
22. }
23. }
```

参见 [属性型指令](#) 章节。

测试注意事项

我们可以为 **响应式表单** 的验证器和控制器逻辑编写 **独立单元测试**。

独立单元测试 直接检测组件类，与组件和它的模板的交互、 DOM、 其他以来和 Angular 本省都无关。

这样的测试具有简单设置 #，快速编写和容易维护的特征。它们不需要 `Angular TestBed` 或异步测试工序。

这对 **模板驱动** 表单来说是不可能的。 模板驱动方法依靠 Angular 来生成控制器模型并从 HTML 验证属性中衍生验证规则。 你必须使用 `Angular TestBed` 来创建组件测试实例， 编写异步测试并与 DOM 交互。

虽然这种测试并不困难，但是它需要更多时间、工作和能力 - 这些因素往往降低测试代码覆盖率和测试质量。

国际化(I18N)

把应用的模板文本翻译成多种语言

Angular 的 **国际化** ("i18n") 工具可以帮助我们使用多个语言发布应用。

目录

- [Angular 和 i18n 模板翻译](#)
- [使用 i18n 属性标记文本](#)
- [使用 ng-xi18n 工具创建翻译源文件](#)
- [翻译](#)
- [合并翻译完成的文件到应用中
 - \[JiT 配置\]\(#\)
 - \[AoT 配置\]\(#\)](#)

试试 这个翻译为法语版 JiT 编译应用的 [在线例子](#)。

Angular 和 i18n 模板翻译

应用程序国际化很具有挑战性，多方面的努力，需要持久的奉献和决心。 Angular 的 **i18n** 国际化工具可以帮助你。

本章描述了 **i18n** 是如何协助翻译组件模板文本到多种语言的。

国际化 工作者通常将一个可翻译的文本叫作“信息”。本章使用了“文本”和“信息”，它们可以互换，也可以组合“文本信息”。

i18n 模板翻译流程有四个阶段：

1. 在组件模板中标记需要翻译的静态文本信息。
2. Angular 的 i18n 工具将标记的信息提取到一个行业标准的翻译源文件。
3. 翻译人员编辑该文件，翻译提取出来的文本信息到目标语言，并将该文件还给你。
4. Angular 编译器导入完成翻译的文件，使用翻译的文本替换原始信息，并生成新的目标语言版本的应用程序。

你可以为每种支持的语言构建和部署单独的应用程序版本。

使用 i18n 属性标记文本

Angular 的 `i18n` 属性是可翻译内容的标记。将它放到每个固定文本需要翻译的元素标签中。

i18n 不是 Angular 指令。它是一个自定义 **属性**，Angular 工具和编译器认识它。它将在完成翻译 **之后**，被编译器移除。

在例子中，`<h1>` 标签显示了一句简单的英文问候语，它将被翻译为法语：

app/app.component.html

```
<h1>Hello i18n!</h1>
```

添加 `i18n` 属性到该标签上，把它标记为需要翻译的文本。

app/app.component.html

```
<h1 i18n>Hello i18n!</h1>
```

翻译人员可能需要待翻译文本的描述才能翻译准确。为 i18n 属性添加描述：

app/app.component.html

```
<h1 i18n="An introduction header for this sample">Hello i18n!</h1>
```

文本的准确 **意思** 可能需要一些应用上下文。在制定的字符串中添加上下文含义，用 | 将其与描述文字隔开。

app/app.component.html

```
<h1 i18n="User welcome|An introduction header for this sample">Hello  
i18n!</h1>
```

如果所有地方出现的文本具有 **相同** 含义时，它们应该有 **相同** 的翻译，但是如果在某些地方它具有 **不同含义**，那么它应该有不同的翻译。Angular 的提取工具在翻译源文件中保留 **含义** 和 **描述**，以支持符合特定上下文的翻译。

使用 ng-xi18n 工具创建翻译源文件

使用 `ng-xi18n` 提取工具来将 `i18n` 标记的文本提取到一个符合行业标准格式的翻译源文件。

它是在 `@angular/compiler-cli` npm 包中的一个 Angular CLI 工具。如果你还没有安装这个 CLI 和它的 `platform-server`，安装它们：

```
npm install @angular/compiler-cli @angular/platform-server --save
```

在应用的项目根目录打开一个终端窗口，并输入 `ng-xi18n` 命令：

```
./node_modules/.bin/ng-xi18n
```

工具默认生成一个名为 `messages.xlf` 的翻译文件，格式为 [XML 本土化互换文件格式 \(XLIFF, version 1.2\)](#)。

```
./node_modules/.bin/ng-xi18n --i18nFormat=xmb
```

Windows 用户可能需要双引号这个命令：

```
"./node_modules/.bin/ng-xi18n"
```

建议在 `package.json` 文件的 `scripts` 部分添加一个便利的快捷方式，让这个命令更容易被记住和运行：

```
"scripts": {  
  "i18n": "ng-xi18n",  
  ...  
}
```

现在你只需要输入：

```
npm run i18n
```

其他翻译格式

你可以通过添加 `--i18nFormat=xmb` 开关，来生成名为 `messages.xmb` 的翻译文件，它的格式为 [XML 信息捆绑包 \(XMB\)](#)。

本例采用 XLIFF 格式。

翻译

`ng-xi18n` 命令在项目根目录生成一个名为 `messages.xlf` 的翻译源文件。下一步是将英文模板文本翻译到目标语言的翻译文件。本烹饪书创建了一个法语翻译文件。

新建一个本土化目录

你很有可能翻译到更多其他语言，所以为全部国际化工作做适当的调整项目目录结构是理所当然的。

其中一种方法是为本土化和相关资源（比如国际化文件）创建一个专门的目录。

本土化和国际化是 [不同但是很相近的概念](#)。

本例遵循这个建议。项目根目录有 `locale` 目录。目录的资源的文件名都匹配有一个语言代码后缀，参见 [语言代码对照表](#)。

将 `messages.xlf` 移到 `locale` 目录，这里将存放所有语言与翻译相关的文件。然后为法语复制这个文件，名为 `messages.fr.xlf`。

对所有目标语言都采用同样的约定。

翻译

在现实世界中，`messages.fr.xlf` 文件会被发给法语翻译，他们使用 [这些 XLIFF 文件编辑器](#) 中的一种来翻译它。

我们不需要任何编辑器或者法语知识就可以轻易的翻译本例子文件。打开 `messages.fr.xlf` 并找到 `<trans-unit>` 节点：

`locale/messages.fr.xlf (<trans-unit>)`

```
<trans-unit id="af2ccf4b5dba59616e92cf1531505af02da8f6d2"  
datatype="html1">  
  <source>Hello i18n!</source>  
  <target/>  
  <note priority="1" from="description">An introduction header for  
this sample</note>  
  <note priority="1" from="meaning">User welcome</note>  
</trans-unit>
```

这个 XML 元素代表了你使用 `i18n` 属性标记的 `<h1>` 问候语标签的翻译。

翻译中利用 `source`、`description` 和 `meaning` 元素的信息，替换 `<target/>` 标签为法语问候语：

locale/messages.fr.xlf (<trans-unit>, after translation)

```
<trans-unit id="af2ccf4b5dba59616e92cf1531505af02da8f6d2"  
datatype="html1">  
  <source>Hello i18n!</source>  
  <target>Bonjour i18n!</target>  
  <note priority="1" from="description">An introduction header for  
this sample</note>  
  <note priority="1" from="meaning">User welcome</note>  
</trans-unit>
```

注意 `id` 是工具生成的。不要修改它。它的值取决于两个因素：信息的内容和其指定的含义。改变任何一个因素，`id` 就会改变。

每当你添加或者编辑应用信息，重复提取流程。小心不要丢失之前的翻译。专门的软件可以帮助你管理变更流程。

翻译前的应用程序

如下所示，是完成前面的步骤后的例子应用 和 它的翻译文件：

```
1.  <h1 i18n="User welcome|An introduction header for this sample">Hello  
i18n!</h1>
```

合并已经翻译的文件

要合并已经翻译的文件到组件模板，使用翻译过的文件编译应用。不管文件是 `.xlf` 格式还是其他 Angular 接受的格式（`.xlif` 和 `.xtb`），流程是一样的。

你为 Angular 编译器提供下列三种新信息：

- 翻译文件
- 翻译文件的格式
- 目标 **语言环境 ID** (例如 `fr` 或 `en-US`)

你如何提供这些信息取决于你使用的是 JIT (即时) 编译器还是 AoT (预先) 编译器。

- 使用 **JIT** 时，在引导时提供
- 使用 **AoT** 时，在 `ngc` 命令的选项里提供

用 JIT 编译器合并

JIT (即时) 编译器在应用程序加载时，在浏览器中编译应用。在使用 JIT 编译器的环境中翻译是一个动态的流程，包括：

1. 决定当前用户的语言，
2. 导入合适的语言翻译文件到一个字符串常量，
3. 新建对应的翻译提供商来指导 JIT 编译器，
4. 使用这些提供商来启动应用。

打开 `index.html` 并这样修改加载脚本：

```
index.html (launch script)

<script>
  // Get the locale id somehow
  document.locale = 'fr';

  // Map to the text plugin
  System.config({
    map: {
      text: 'systemjs-text-plugin.js'
    }
  });

  // Launch the app
  System.import('app').catch(function(err){ console.error(err); });
</script>
```

在本例中，用户的语言在 `index.html` 中被硬编码到一个全局的 `document.locale` 变量中。

SystemJS 文本插件

注意 SystemJS 将 `text` 映射为 `systemjs-text-plugin.js`。在这个文本插件的帮助下，SystemJS 可以读取任何原始文件并将其内容作为字符串返回。你需要使用它来导入语言翻译文件。

SystemJS 没有自带原始文本插件，但是我们很容易添加它。在根目录新建下面的 `systemjs-text-plugin.js` 文件：

```
systemjs-text-plugin.js

/*
  SystemJS Text plugin from
  https://github.com/systemjs/plugin-text/blob/master/text.js
*/
exports.translate = function(load) {
  if (this.builder && this.transpiler) {
```

```

load.metadata.format = 'esm';
return 'exp' + 'ort var __useDefault = true; exp' + 'ort default
' + JSON.stringify(load.source) + ';';
}

load.metadata.format = 'amd';
return 'def' + 'ine(function() {\nreturn ' +
JSON.stringify(load.source) + ';\n});';
}

```

新建翻译提供商

三种提供商帮助 JIT 编译在编译应用时，将模板文本翻译到某种语言：

- `TRANSLATIONS` 是含有翻译文件内容的字符串。
- `TRANSLATIONS_FORMAT` 是文件的格式：`xlf`、`xlif` 或 `xtb`。
- `LOCALE_ID` 是目标语言的语言环境。

在下面的 `app/i18n-providers.ts` 文件的 `getTranslationProviders`` 函数中，根据用户的语言环境 和对应的翻译文件构建这些提供商：

app/i18n-providers.ts

```

1. import { TRANSLATIONS, TRANSLATIONS_FORMAT, LOCALE_ID } from
  '@angular/core';

2.

3. export function getTranslationProviders(): Promise<Object[]> {
4.

5.   // Get the locale id from the global
6.   const locale = document['locale'] as string;
7.

8.   // return no providers if fail to get translation file for locale
9.   const noProviders: Object[] = [];

10.

11.  // No locale or U.S. English: no translation providers
12.  if (!locale || locale === 'en-US') {
13.    return Promise.resolve(noProviders);

```

```

14. }
15.
16. // Ex: 'locale/messages.fr.xlf'
17. const translationFile = `./locale/messages.${locale}.xlf`;
18.
19. return getTranslationsWithSystemJs(translationFile)
20. .then( (translations: string) => [
21.     { provide: TRANSLATIONS, useValue: translations },
22.     { provide: TRANSLATIONS_FORMAT, useValue: 'xlf' },
23.     { provide: LOCALE_ID, useValue: locale }
24. ])
25. .catch(() => noProviders); // ignore if file not found
26. }
27.
28. declare var System: any;
29.
30. function getTranslationsWithSystemJs(file: string) {
31.     return System.import(file + '!text'); // relies on text plugin
32. }

```

- 它从在 `index.html` 中设置的全局 `document.locale` 变量中获取语言环境。
- 如果没有语言环境或者语言是美国英语（`en-US`），则就无需翻译。该函数以 `Promise` 的形式返回一个空的 `noProviders` 数组。它必须要返回 `Promise`，因为这个函数可能异步从服务器读取翻译文件。
- 根据 [上面描述](#) 的名字和本土化的约定，它根据语言环境创建一个合约文件名。
- `getTranslationsWithSystemJs` 方法读取翻译并以字符串的形式返回其内容。注意它在文件名上附加 `!text`，告诉 SystemJS 使用 [文本插件](#)。
- 回调函数使用这三种翻译提供商创建一个提供商数组。
- 最后，`getTranslationProviders` 返回以承诺的形式返回全部流程的结果。

使用翻译提供商引导应用

Angular 的 `bootstrapModule` 方法接受 **可选的** 第二参数，它可以影响编译器的行为。

从 `getTranslationProviders` 返回的翻译提供商创建 `options` 对象，并将其传给 `bootstrapModule`。打开 `app/main.ts` 并这样修改引导代码：

app/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { getTranslationProviders } from './i18n-providers';

import { AppModule } from './app.module';

getTranslationProviders().then(providers => {
  const options = { providers };
  platformBrowserDynamic().bootstrapModule(AppModule, options);
});
```

注意，它等待 `getTranslationProviders` 承诺的解析完成后，才引导应用。

现在，应用已经被国际化为英语版和法语版，而且我们有了清晰的添加更多语言的方法。

使用 AoT 编译器时的国际化

JiT 编译器在浏览器中动态编译应用时，将其翻译到目标语言。这样很灵活，但是对用户来讲，可能速度太慢。

AoT（预先）编译器是一种构建流程，出产尺寸小、速度快和可执行的应用程序包。在使用 AoT 编译器的环境中国际化，你为每种语言预先构建一个单独的应用程序包。然后，在宿主网络页面（`index.html`）中，你再决定用户需要哪种语言，并选择合适的应用程序包。

本烹饪书不介绍如何构建多种应用程序包和如何根据用户的语言设置推送它们。它介绍了一些必要的步骤，来告诉 AoT 采用用翻译文件。

使用 AoT 编译器时的国际化，需要一些针对 AoT 的设置。以 [合并翻译文件之前](#) 的应用项目开始，并参见 [AoT 烹饪书](#)，把它变成与 AoT 兼容的项目。

接下来，为每种支持的语言（包括英语）运行一次 `ngc` 编译命令。结果是每种语言都有自己单独的应用版本。

通过添加下面三种选项到 `ngc` 命令来告诉 AoT 编译器如何翻译：

- `--i18nFile` : 翻译文件的路径
- `--locale` : 语言环境的名字
- `--i18nFormat` : 翻译文件的格式

本法语例子的命令为：

```
./node_modules/.bin/ngc --i18nFile=./locale/messages.fr.xlf --locale=fr  
--i18nFormat=xlf
```

Windows 用户可能需要双引号这个命令：

```
"./node_modules/.bin/ngc" --i18nFile=./locale/messages.fr.xlf -  
-locale=fr --i18nFormat=xlf
```

设置文档标题

使用 Title 服务来设置文档标题或窗口标题

应用程序应该能让浏览器标题栏显示我们想让它显示的内容。本 烹饪宝典 解释怎么做。

参见 [在线例子](#)。

要看到浏览器标题的变化，点击预览窗口右上角的 'X' 按钮，弹出窗口。



<title> 的问题

显而易见的方法是把组件的属性绑定到 HTML 的 <title> 标签上，像这样：

```
<title>{{This_Does_Not_Work}}</title>
```

抱歉，这样不行。我们应用程序的根组件是一个包含在 <body> 标签里的元素。该 HTML 的 <title> 在文档的 <head> 元素里，在 <body> 之外，Angular 的数据绑定无法访问到它。

可以从浏览器获得 document 对象，并且手动设置标题。但是这样看起来很脏，而且将无法在浏览器之外运行应用程序。

在浏览器外运行应用程序意味着：利用服务器端预先渲染，为应用程序实现几乎实时的首次渲染，同时还能支持 SEO(搜索引擎优化)。意味着你可以在一个 Web Worker 中运行你的应用程序，通过多线程技术增强应用程序的响应性。还意味着你可以在 Electron.js 或者 Windows Universal 里面运行，发布到桌面环境。

使用 Title 服务

幸运的是，Angular 在 **浏览器平台** 的包中，提供了一个 `Title` 服务，弥补了这种差异。`Title` 服务是一个简单的类，提供了一个 API，用来获取和设置当前 HTML 文档的标题。

- `getTitle(): string` —— 获取当前 HTML 文档的标题。
- `setTitle(newTitle: string)` —— 设置当前 HTML 文档的标题。

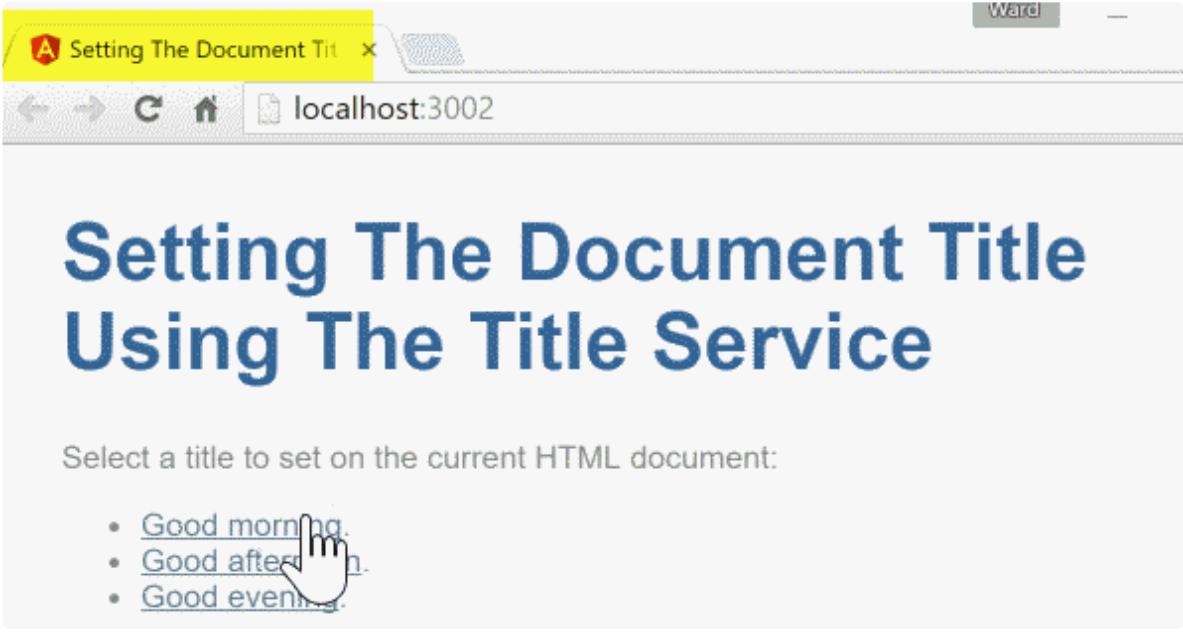
我们来把 `Title` 服务注入到根组件 `AppComponent`，并暴露出可供绑定的 `setTitle` 方法让别人来调用该服务：

app/app.component.ts (class)

```
export class AppComponent {
  public constructor(private titleService: Title) { }

  public setTitle( newTitle: string ) {
    this.titleService.setTitle( newTitle );
  }
}
```

我们把这个方法绑定到三个 A 标签，瞧瞧！



The screenshot shows a browser window with the address bar displaying "localhost:3002". The main content area has a yellow header bar with the text "A Setting The Document Tit" and a red "X" button. Below this, the page title is "Setting The Document Title Using The Title Service". A sub-section below the title says "Select a title to set on the current HTML document:" followed by a list of three options: "Good morning.", "Good aftern", and "Good evening.". A hand cursor is hovering over the second option, "Good aftern".

这里是完整的方案(代码)。

```
1. import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2.
3. import { AppModule } from './app.module';
4.
5. platformBrowserDynamic().bootstrapModule(AppModule);
```

为什么要在 bootstrap 里面提供这个 Title 服务

我们通常会推荐在应用程序的根组件 `AppComponent` 中提供应用程序级的服务。

但这里，我们推荐在引导过程中注册这个 Title 服务，这个位置是为设置 Angular 运行环境而保留的。

我们的做法正是如此。这里的 `Title` 服务是 Angular 浏览器平台的一部分。如果在其它平台上引导应用程序，就得提供另一个专为那个平台准备的 `Title` 服务。

[回到顶部](#)

从TYPESCRIPT到JAVASCRIPT

把 Angular 的 TypeScript 范例转换为 ES5 JavaScript

在 Angular 中，所有能用 TypeScript 完成的事，也都能用 JavaScript 完成。从一个语言换成另一个语言，最多只会影响源代码的组织方式和访问 Angular API 的方法。

TypeScript 是个广受欢迎的 Angular 2 语言选项，你在网络上和本站上看到的很多范例代码都是用 TypeScript 写的。本烹饪宝典包含如何把这些代码编译到 ES5 的方法，这样它们就可以被用到 JavaScript 版的 Angular 2 程序里了。

目录

[模块化：导入与导出](#)

[类和“类的元数据”](#)

[Input（输入）与 Output（输出）元数据](#)

[依赖注入](#)

[Host（宿主）与 Query（查询）元数据](#)

[运行并比较本烹饪宝典里的在线 TypeScript 和 JavaScript 代码](#)

导入与导出

TypeScript

ES5 JavaScript

导入 Angular 代码

在 TypeScript 代码中，Angular 是利用 TypeScript 的 import 语句来导入类、函数和其它成员的。

通过全局变量 ng 来访问 Angular 代码

在 JavaScript 中，当使用 [Angular 库](#) 时，我们可以通过全局的 ng 对象来访问 Angular 代码。在本对象内嵌的很多成员中，我们会发现 TypeScript 能从 @angular 库导入的所有对应物。

```
import {
  platformBrowserDynamic
} from
  '@angular/platform-
  browser-dynamic';
import {
  LocationStrategy,
  HashLocationStrategy
} from
  '@angular/common';
```

```
var platformBrowserDynamic =
  ng.platformBrowserDynamic.platformBrowserDynamic;
var LocationStrategy =
  ng.common.LocationStrategy;
var HashLocationStrategy =
  ng.common.HashLocationStrategy;
```

导入与导出应用程序代码

在 TypeScript 版的 Angular 程序里，每个文件都是一个 TypeScript 模块。当需要让一个模块在其它模块中可见时，我们要 `export` 它。

```
export class
  HeroComponent {
  title = 'Hero
  Detail';
  getName() {return
  'Windstorm'; }
}
```

然后，我们就可以在其它模块中 `import` 这些在别处导出的东西。

```
import { HeroComponent
} from
  './hero.component';
```

共享应用程序代码

在 JavaScript 版的 Angular 程序里，我们在页面中通过 `<script>` 标签来加载每个文件。每个文件都要通过全局作用域 `window` 来互相共享自己的那些东西。

我们经常在 `window` 对象上附加一个应用程序命名空间对象（比如 `"app"`），然后把所要共享一切都附加到这个对象上。也可以把我们的代码包装进一个 [立即调用函数表达式 IIFE](#)。这些实践可以防止我们的代码污染全局作用域。

```
(function(app) {
  function HeroComponent() {
    this.title = "Hero Detail";
  }

  app.HeroComponent = HeroComponent;

})(window.app = window.app || {});
```

然后我们就可以用这个共享的命名空间来访问来自其它文件中的东西了。

```
(function(app) {
  var HeroComponent = app.HeroComponent;
})(window.app = window.app || {});
```

注意，页面中 `<script>` 标签的排列顺序非常重要。我们必须先加载“定义”共享成员的文件，然后再加载“使用”该共享成员的文件。

另外，我们可以在 JavaScript 版的 Angular 2 项目中，使用模块加载器（比如 Webpack 或者 Browserify）。在这样的项目中，我们使用 CommonJS 模块和 `require` 函数来加载 Angular 2 框架代码。然后用 `module.exports` 和 `require` 来导出和导入应用程序代码。

类和“类的元数据”

TypeScript

类

在 Angular 中，我们把绝大多数 TypeScript 代码都放到 TypeScript 类中。

```
export class HeroComponent {
  title = 'Hero Detail';
  getName() {return 'Windstorm';}
}
```

ES5 JavaScript

构造函数和原型

ES5 JavaScript 不支持类，我们使用构造器模式 (constructor pattern) 来代替它，它可以和类一样与 Angular 协同工作。

```
function HeroComponent() {
  this.title = "Hero Detail";
}

HeroComponent.prototype.getName
=
  function() {return
  'Windstorm';};
```

使用装饰器提供元数据

大部分 Angular 2 中的类都会附加一个或多个 TypeScript 装饰器，用来提供配置和元数据。比如组件必须要有一个 `@Component` 装饰器。

通过注解数组来提供元数据

在 JavaScript 中，我们可以为构造函数附加一个 注解 数组来提供元数据。数组里的每一个条目都对应一个 TypeScript 装饰器。

在下面的例子中，我们新建了一个 `Component` 的新实例，与 `@Component` TypeScript 装饰器相对应。

```

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

@Component({
  selector: 'hero-view',
  template:
    '<h1>Hero: {{getName()}}'
  )
export class HeroComponent {
  title = 'Hero Detail';
  getName() {return 'Windstorm';
}
}

```

```

function HeroComponent() {
  this.title = "Hero Detail";
}

HeroComponent.annotations = [
  new ng.core.Component({
    selector: 'hero-view',
    template:
      '<h1>Hero: {{getName()}}'
    )
  ];
}

HeroComponent.prototype.getName =
  function() {return
  'windstorm';};

```

利用便捷 API Class 提供元数据

创建一个构造函数并用元数据来装饰它，这种模式非常常见，以至于 Angular 为它提供了一个可选的便捷 API。该 API 让我们能在单一表达式中定义任何东西。

利用该 API，我们先调用 `ng.core.Component` 函数，再链式调用一个 `Class` 方法。该 `Class` 方法的参数是一个对象，用来定义组件的构造函数和实例方法。

```

var HeroComponent =
  ng.core.Component({
    selector: 'hero-view-2',
    template:
      '<h1>Name: {{getName()}}'
    )
  .Class({
    constructor: function() {
    },
    getName: function() {
      return 'windstorm';
    }
  });

```

其它装饰器也都有类似的 API。比如你可以像这样定义指令：

```
var MyDirective =
  ng.core.Directive({
    ...
  }).Class({
    ...
  });

```

或者一个管道：

```
var MyPipe = ng.core.Pipe({
  name: 'myPipe'
}).Class({
  ...
});

```

接口

当定义一个想实现某个方法的类时，常用 TypeScript 接口来确保方法的签名正确。组件生命周期方法，比如 `ngOnInit` 就是一个例子。`ngOnInit` 是在 `OnInit` 接口里面定义的。

```
import { Component, OnInit } from '@angular/core';
class HeroComponent implements OnInit {
  name: string;
  ngOnInit() {
    this.name = 'windstorm';
  }
}
```

实现方法，但不用管接口

TypeScript 的接口纯粹是为了方便开发者而设计的，它不能被 Angular 2 在运行时使用。这就意味着在 JavaScript 代码里，我们不需要使用任何东西来代替接口。我们只要直接实现它的方法就行了。

```
function HeroComponent() {}
HeroComponent.prototype.ngOnInit =
  function() {
    this.name = 'windstorm';
  };

```

Input (输入) 和 Output (输出) 元数据

TypeScript

Input (输入) 和 Output (输出) 装饰器

ES5 JavaScript

组件元数据里的输入属性和输出属性

在 TypeScript 里，属性装饰器经常被用来为组件和指令提供额外的元数据。

我们使用 `@Input` 和 `@Output` 装饰器来装饰 **输入属性与输出属性**。如果想让它们的名字与类属性名不同，我们可以（可选的）指定输入和输出的绑定名。

```
@Component({
  selector: 'my-confirm',
  template: `
    <button
      (click)="onokclick()"
      {{okMsg}}
    </button>
    <button
      (click)="onNotokclick()"
      {{notokMsg}}
    </button>
  `,
})
class ConfirmComponent {
  @Input() okMsg: string;
  @Input('cancelMsg') notokMsg: string;
  @Output() ok = new EventEmitter();
  @Output('cancel') notok = new EventEmitter();

  onokclick() {
    this.ok.next(true);
  }
  onNotokclick() {
    this.notok.next(true);
  }
}
```

在 TypeScript 里，我们同样可以使用 `inputs` 和 `outputs` 元数据数组，来代替 `@Input` 和 `@Output` 属性装饰器。

在 ES5 JavaScript 里并没有属性装饰器的等价物，但我们可以往 `Component`（或者 `Directive`）的元数据中添加对应的信息。

在这个例子中，我们添加了 `inputs` 和 `outputs` 数组属性，它们包含了输入属性名和输出属性名。如果我们需要一个不同于属性名的绑定名，可以使用 `propertyName: bindingName` 的语法。

```
var ConfirmComponent =
  ng.core.Component({
    selector: 'my-confirm',
    inputs: [
      'okMsg',
      'notokMsg: cancelMsg'
    ],
    outputs: [
      'ok',
      'notok: cancel'
    ],
    template:
      '<button
        (click)="onokclick()">' +
      '{{okMsg}}' +
      '</button>' +
      '<button
        (click)="onNotokclick()">' +
      '{{notokMsg}}' +
      '</button>'
  }).Class({
    constructor: function() {
      this.ok = new
        ng.core.EventEmitter();
      this.notok = new
        ng.core.EventEmitter();
    },
    onokclick: function() {
      this.ok.next(true);
    },
    onNotokclick: function() {
      this.notok.next(true);
    }
});
```

依赖注入

按类型注入

Angular 2 通常使用 TypeScript 的类型信息来确定需要注入什么。

```
@Component({
  selector: 'hero-di',
  template: `<h1>Hero: {{name}}</h1>`
})
class HeroComponent {
  name: string;
  constructor(dataService: DataService) {
    this.name =
      dataService.getHeroName();
  }
}
```

使用参数指令来注入

由于在 ES5 JavaScript 里没有类型信息，所以我们必须使用其它方法来标记出“可供注入”的东西。

我们可以为构造函数附加一个 `parameters` 数组。数组中的每个条目都是一个“依赖注入令牌”，用来标识出希望被注入进来的东西。通常，对于“类”依赖，这个令牌就是它的构造函数。

```
app.HeroDIClass = HeroComponent;

function HeroComponent(dataService) {
  this.name =
    dataService.getHeroName();
}

HeroComponent.parameters = [
  app.DataService
];
HeroComponent.annotations = [
  new ng.core.Component({
    selector: 'hero-di',
    template: '<h1>Hero: {{name}}</h1>'
  })
];
```

当我们使用便捷 API `Class` 时，也可以把构造器包装到一个数组里面来提供参数令牌。

```

var HeroComponent =
  ng.core.Component({
    selector: 'hero-di-inline',
    template: '<h1>Hero: {{name}}</h1>'
  })
  .Class({
    constructor: [
      app.DataService,
      function(service) {
        this.name =
          service.getHeroName();
      }
    ]
  );

```

使用 @Inject 装饰器来注入

当被注入的东西不是一个类型时，要使用 @Inject() 装饰器来提供注入令牌。

在下面这个例子中，我们用令牌 "heroName" 注入了一个字符串。

```

@Component({
  selector: 'hero-di-inject',
  template: `<h1>Hero: {{name}}</h1>`
})
class HeroComponent {
  constructor(
    @Inject('heroName')
    private name: string) {
  }
}

```

利用普通字符串令牌来注入

在 JavaScript 中，我们要把令牌字符串添加到注入参数的数组中。

```

function HeroComponent(name) {
  this.name = name;
}
HeroComponent.parameters = [
  'heroName'
];
HeroComponent.annotations = [
  new ng.core.Component({
    selector: 'hero-di-inject',
    template: '<h1>Hero: {{name}}</h1>'
  })
];

```

另外，我们可以使用 Inject 方法新建一个令牌，然后把它加到注解里的 constructor 数组中，就像这样：

```
var HeroComponent =  
  ng.core.Component({  
    selector: 'hero-di-inline2',  
    template: '<h1>Hero: {{name}}</h1>'  
  })  
  .Class({  
    constructor:  
      [new  
        ng.core.Inject('heroName'),  
        function(name) {  
          this.name = name;  
        }]  
  });
```

额外的注入装饰器

我们可以往构造函数中附加额外的装饰器来调整注入行为。比如使用 `@Optional` 来标记依赖是可选的，用 `@Attribute` 来注入宿主元素的属性 (Attribute)，用 `@ContentChild` 来注入投影内容的子查询，用 `@ViewChild` 来注入视图的子查询。

利用嵌套数组来添加额外的注入元数据

在 JavaScript 中，为了达到同样的效果，要使用构造器数组表示法，其中的注入信息优先于构造函数自身的。

使用“注入支持函数”，比如 `Attribute`、`Host`、`Optional`、`Self`、`SkipSelf`、`Query` 和 `ViewQuery` 等来调整依赖注入的行为。

使用嵌套数组来合并注入函数。

```

@Component({
  selector: 'hero-title',
  template: `
    <h1>{{titlePrefix}}
    {{title}}</h1>
    <button
      (click)="ok()">OK</button>
    <p>{{ msg }}</p>
  `
})
class TitleComponent {
  private msg: string = '';
  constructor(
    @Inject('titlePrefix')
    @Optional()
    private titlePrefix: string,
    @Attribute('title')
    private title: string) {
  }

  ok() {
    this.msg = 'OK!';
  }
}

```

```

var TitleComponent =
ng.core.Component({
  selector: 'hero-title',
  template:
    '<h1>{{titlePrefix}}
    {{title}}</h1>' +
    '<button
      (click)="ok()">OK</button>' +
    '<p>{{ msg }}</p>'
}).class({
  constructor: [
    [
      new ng.core.Optional(),
      new
        ng.core.Inject('titlePrefix')
    ],
    new
      ng.core.Attribute('title'),
      function(titlePrefix, title) {
        this.titlePrefix =
          titlePrefix;
        this.title = title;
        this.msg = '';
      }
  ],
  ok: function() {
    this.msg = 'OK!';
  }
});

```

我们可以用同样的方法来使用其它的额外参数装饰器，比如 `@Host` 和 `@SkipSelf`：往参数数组中添加 `new ng.core.Host()` 或 `ng.core.SkipSelf()`。

Host (宿主)与 Query (查询)元数据

TypeScript

ES5 JavaScript

Host (宿主) 装饰器

我们可以使用宿主属性装饰器来把宿主元素绑定到组件或指令。 @HostBinding 装饰器把宿主元素的属性绑定到组件的数据属性。

@HostListener 装饰器把宿主元素的事件绑定到组件的事件处理器。

```
@Component({
  selector: 'heroes-bindings',
  template: `<h1
[class.active]="active">
  Tour of Heroes
</h1>`
})
class HeroesComponent {
  @HostBinding() title = 'Tooltip
content';
  @HostBinding('class.heading')
  hclass = true;
  active: boolean;

  constructor() {}

  @HostListener('click')
  clicked() {
    this.active = !this.active;
  }

  @HostListener('dblclick',
  ['$event'])
  doubleClicked(evt: Event) {
    this.active = true;
  }
}
```

在 TypeScript 中，我们也可以用 host 元数据来代替 @HostBinding 和 @HostListener 属性装饰器。

Host 元数据

我们添加一个 host 属性到组件的元数据，以达到和 @HostBinding 、 @HostListener 一样的效果。

host 的值是一个对象，它的属性中包含了宿主属性和事件监听器的绑定。

- 每个键值 (key) 都遵循常规的 Angular 绑定语法：用 [property] 代表宿主属性绑定，用 (event) 代表数组事件绑定。
- 每个值 (value) 代表了对应的组件属性或方法。

```
var HeroesComponent =
ng.core.Component({
  selector: 'heroes-bindings',
  template: `<h1
[class.active]="active"> +
  'Tour of Heroes' +
</h1>`,
  host: {
    '[title]': 'title',
    '[class.heading]': 'hclass',
    '(click)': 'clicked()',
    '(dblclick)':
    'doubleClicked($event)'
  }
}).Class({
  constructor: function() {
    this.title = 'Tooltip content';
    this.hclass = true;
  },
  clicked: function() {
    this.active = !this.active;
  },
  doubleClicked: function(evt) {
    this.active = true;
  }
});
```

Query (查询) 装饰器

有好几个属性装饰器可以用来查询组件或指令的各级子节点。

Query (查询) 元数据

我们通过往组件元数据中添加 queries 属性来访问一个组件的各级子节点。它应该是一个对象，

@ViewChild 和 @ViewChildren 属性装饰器允许组件查询在自己模板里用到的其它组件实例。

```
@Component({
  selector: 'heroes-queries',
  template: `
    <a-hero *ngFor="let hero of
    heroData"
      [hero]="hero">
      <active-label></active-label>
    </a-hero>
    <button (click)="activate()">
      Activate
    </button>
  `

})
class HeroesQueriesComponent {
  heroData = [
    {id: 1, name: 'Windstorm'},
    {id: 2, name: 'Superman'}
  ];

  @viewChildren(HeroComponent)
  heroCmps: QueryList<HeroComponent>;

  activate() {
    this.heroCmps.forEach(
      (cmp) => cmp.activate()
    );
  }
}
```

@ContentChild 和 @ContentChildren 属性装饰器允许组件查询那些从其它地方投影到自己视图里的其它组件实例。

其中：

- 每个键值 (key) 都是组件中的一个属性名，该属性用来保存当前视图的子节点。
- 每个值都是一个 ViewChild 或者 ViewChildren 的实例。

```
var AppComponent =
  ng.core.Component({
    selector: 'heroes-queries',
    template:
      '<a-hero *ngFor="let hero of
      heroData" +'
      '[hero]="hero">' +
      '<active-label></active-
      label>' +
      '</a-hero>' +
      '<button (click)="activate()">' +
      'Activate' +
      '</button>',
    queries: {
      heroCmps: new
      ng.core.ViewChildren(
        HeroComponent)
    }
  }).class({
    constructor: function() {
      this.heroData = [
        {id: 1, name: 'Windstorm'},
        {id: 2, name: 'Superman'}
      ];
      activate: function() {
        this.heroCmps.forEach(function(cmp)
        {
          cmp.activate();
        });
      }
    }
  });
```

同样的，我们利用 ContentChild 和 ContentChildren 往 queries 属性中添加 投影内容 的子查询：

```

@Component({
  selector: 'a-hero',
  template: `<h2
[class.active]=active>
  {{hero.name}}
  <ng-content></ng-content>
</h2>`
})
class HeroComponent {
  @Input() hero: any;
  active: boolean;

  @ContentChild(ActiveLabelComponent)
  label: ActiveLabelComponent;

  activate() {
    this.active = true;
    this.label.activate();
  }
}

```

在 TypeScript 中，我们也可以使用 queries 元数据来代替 @ViewChild 和 @ContentChild 属性装饰器。

```

var HeroComponent =
ng.core.Component({
  selector: 'a-hero',
  template: '<h2
[class.active]=active>' +
  '{{hero.name}} ' +
  '<ng-content></ng-content>' +
  '</h2>',
  inputs: ['hero'],
  queries: {
    label: new
    ng.core.Contentchild(
      ActiveLabelComponent)
  }
}).class({
  constructor: [function() { }],
  activate: function() {
    this.active = true;
    this.label.activate();
  }
});
app.HeroQueriesComponent =
HeroComponent;

```

VISUAL STUDIO 2015快速起步

使用 Visual Studio 2015 快速起步

有些开发者喜欢用 Visual Studio 。

本烹饪宝典介绍了在 **Visual Studio 2015 的 ASP.NET 4.x 项目中**，用 Angular 实现“快速起步”所需的步骤。

本烹饪宝典中没有 **在线例子**，因为它介绍的是 Visual Studio，而不是应用程序。

ASP.NET 4.x 项目

本烹饪书解释了如何使用 Visual Studio 2015 在 **ASP.NET 4.x 项目** 中设置 **快速开始** 文件。

如果你希望使用 **ASP.NET Core** 并体验全新项目，参见 [预览版 ASP.NET Core + Angular 2 template for Visual Studio 2015](#)。注意，最终代码与本文不对应，请适当调节。

步骤如下：

- 前提条件 : 安装 Node.js
- 前提条件 : 安装 Visual Studio 2015 Update 3
- 前提条件 : 配置 External Web tools
- 前提条件 : 安装 TypeScript 2 for Visual Studio 2015
- 第一步 : 下载“快速起步”的文件
- 第二步 : 创建 Visual Studio ASP.NET 项目
- 第三步 : 把“快速起步”中的文件拷贝到 ASP.NET 的项目目录中
- 第四步 : 恢复需要的包
- 第五步 : 构建和运行应用程序

前提条件 : Node.js

如果你的电脑里没有 Node.js® 和 npm , 请安装 [它们](#)。

在终端或者控制台中运行 `node -v` 和 `npm -v` , 请确认你的 Node 版本为 [4.6.x 或更高](#) , npm 的版本为 [3.x.x 或更高](#) 。老版本会引起错误。

前提条件 : Visual Studio 2015 Update 3

使用 Visual Studio 开发 Angular 2 应用程序的最低要求是 Update 3 。早期版本没有遵循使用 TypeScript 开发应用程序的最佳实践。要查看你的 Visual Studio 2015 版本号 , 到 [Help | About Visual Studio](#) 。

如果还没有 , 安装 [Visual Studio 2015 Update 3](#) 。或者使用 [Tools | Extensions and Updates](#) 来直接在 Visual Studio 2015 中更新到 Update 3 。

前提条件：配置 External Web tools

配置 Visual Studio 来使用全局 External Web Tools，而非 Visual Studio 默认的工具：

- 到 Tools | Options 打开 Options 对话框
- 在左边树型项目中，选择 Projects and Solutions | External Web Tools。
- 在右侧，将 \$(PATH) 移动到 \$(DevEnvDir) 上面。这样，Visual Studio 就会在使用自带的外部工具时，优先使用全局路径中的外部工具（比如 npm）。
- 点击 OK 关闭对话框。
- 重启 Visual Studio，以让设置变化生效。

Visual Studio 将优先在当前的工作区查找外部工具，如果没有找到，便查找全局路径，如果还没有找到，Visual Studio 就使用自带的工具版本。

前提条件：安装 TypeScript 2 for Visual Studio 2015

Visual Studio Update 3 自带 TypeScript 支持，但是它的 TypeScript 版本开发 Angular 2 应用需要的不是 2。

要安装 TypeScript 2：

- 下载并安装 [TypeScript 2.0 for Visual Studio 2015](#)
- 或者，通过 NPM 来安装：`npm install -g typescript@2.0`。

你可以在 [这儿](#) 查看更多 Visual Studio 中 TypeScript 2 的支持。

至此，Visual Studio 准备好了。重新启动 Visual Studio，这样我们可以有一个崭新的开始。

第一步：现在“快速起步”文件

从 [github 下载“快速起步”的源代码](#)。如果下载的是一个压缩的 zip 文件，解压它。

第二步：创建 Visual Studio ASP.net 项目

按照下列步骤创建 ASP.NET 4.x 项目：

- 在 Visual Studio 中，选择 File | New | Project 菜单。
- 在模板树中，选择 Templates | Visual C# (或 Visual Basic) | Web 菜单。
- 选择 ASP.NET Web Application 模板，输入项目名，点击“OK”按钮。
- 选择自己喜欢的 ASP.NET 4.5.2 模板，点击 OK。

本烹饪宝典选择了 Empty 模板，它没有添加过任何目录，没有身份验证，没有服务器托管。为你的项目选择合适的模板和选项。

第三步：拷贝“快速起步”的文件到 ASP.NET 项目所在的目录

拷贝从 github 下载的“快速起步”文件到包含 .csproj 文件的目录中。按照下面的步骤把它们加到 Visual Studio 中：

- 在 Solution Explorer 中点击 Show All Files 按钮，显示项目中所有隐藏文件。
- 右键点击每个目录和文件，选择 Include in Project。最少要添加下列文件：
 - app 目录（如果询问是否要搜索 TypeScript 类型，回答 No）
 - styles.css
 - index.html
 - package.json
 - tsconfig.json

- typings.json

第四步：恢复需要的包

按下面的步骤恢复 Angular 应用程序需要的包：

- 在 Solution Explorer 中右键点击 `package.json`，选择 `Restore Packages`。
这样，Visual Studio 会使用 `npm` 来安装在 `package.json` 中定义的所有包。这可能需要花一点时间。
- 如果愿意，打开 Output 窗口 (`View` | `Output`) 来监控 `npm` 命令的执行情况。
- 忽略所有警告。
- 当恢复完成后，将会出现一条消息：`npm command completed with exit code 0`。
- 在 Solution Explorer 里，点击 `Refresh` 图标。
- **不要** 将 `node_modules` 目录添加到项目中，让它隐藏。

第五步：构建和运行应用

首先，确认 `index.html` 已被设置为开始页面。在 Solution Explorer 中，右键点击 `index.html`，选择选项 `Set As Start Page`。

点击 **Run** 按钮或者按 `F5` 键，用调试器构建和启动应用。

按 `Ctrl-F5` 不带调试器的运行应用，速度会更快。

默认浏览器打开并显示快速开始例子应用。

尝试编辑任何项目文件，**保存** 并刷新浏览器来查看效果。

应用程序的路由

如果这个应用程序使用了 Angular 路由器，刷新浏览器时可能会返回一个 **404 - Page Not Found**。查看一下地址栏，它是否包含一个导航 url（“深链接”）... 以及任何除了 / 或 /index.html 以外的路径？

我们必须配置服务器，让它为这些请求直接返回 index.html 的内容。在配置完之前，请暂时删除导航路径，回到首页，再进行刷新。

官方中文版简介

ANGULAR官方中文版的简介、致辞等

Angular 开发组的致辞

这是 Angular 开发组的项目经理 (TPM) Naomi Black 和工程总监 (Engineering Director) Brad Green 代表官方开发组发来的致辞，这标志着 angular.cn 正式获得了官方身份。如文档中的其它地方一样，点击任意中文即可显示原文。

欢迎光临 angular.cn !

这份官方网站和《 Angular 2 开发指南》的中文版是中国的 Angular 开发者社区共同努力奉献的成果。它们来自深谙 Angular 设计理念的工程师，故此，这份中文版不仅是语言上的简单汉化。出于对“开源精神”和“共享精神”的执着与推崇，汪志成和叶志敏共同带领 angular.cn 团队 奋力工作，力争让全世界的中文开发者与 Angular 实现零距离的亲密接触。

除英文版之外，该中文版是第一个由官方正式发布的开发文档！

作为开源技术的倡导者， Google 已经并将继续投资于对 Angular 的完善和演进工作，造福并回馈中国区的用户以及全世界的用户。

看到中国的 Angular 开发社区已经如此繁荣，我们非常高兴。这不仅仅是出于开发者对技术的固有热情，更是因为我们希望“用 Angular 高效实现商业需求”的宏伟愿景。事实上，许多中国公司的业务早已基于 Angular 技术。我们希望通过官网内容的持续更新，鼓励并协助大家把 Angular 更多的应用于现实世界中的业务服务与应用程序。如果您的业务大厦已经构建在 Angular 的基础之上，我们欢迎您加入 中文社区 ，参与到我们持续增强和优化此项技术的努力中来。我们随时欢迎您的意见和建议，请发邮件到 Google 的技术推广部联系我们，我们的邮箱是 devrel-china-contact@google.com 。

Angular 的成功，是因为它由背景各异的开发者所创造，它为开发者而生。我们希望这些文档能帮助大家学习和理解 Angular，总有一天，大家还会为这个框架做出贡献。我们非常期待看到大家用各种方式应用 Angular 技术，更希望看到 Angular 在所有开发者的热情支持下继续演进。

通过 angular.cn , 我们希望广大的开发者们能继续发掘更多来自 Google 和开源世界的 Web 技术和移动技术 , 借助 Google 提供的平台登上业务成就的新高度。

非常高兴在这里遇见你们。 Google 欢迎你 !

Naomi Black 与 Brad Green

奕跃代表 Google 开发技术推广部的致辞

这是开发技术推广部的主管奕跃发来的致辞。该部门也会安排专人来全职运营 Angular 中文社区 , 包括组织线上线下的活动、对外合作等。希望大家多多捧场。

angular.cn 社区的开发者朋友们 , 大家好 !

我谨代表谷歌公司开发技术推广部 (Developer Relations) 中国团队热烈欢迎你 ! 我们非常高兴和中国的 Angular 开发者社区在这里的合作 , 也希望通过这个合作和 Angular.CN 社区的开发者朋友们扩大未来更多的合作和分享。

这个 Angular 中文网站的大量资源 , 体现了一个很好的合作模式和精神。它不仅有谷歌公司投资所推动的 Angular 技术的资源和更新 , 更是有众多中国的技术爱好者和开发者们通过自己的无私义务劳动为广大开发者们所做的技术资料的中文本地化翻译工作的成果。这些志愿者们辛勤劳动所带来的中文资源 , 将帮助更多的中国开发者们能够方便地学习和采纳 Angular 技术。我们非常赞赏这样的奉献精神和分享实践 , 正是这样的精神和实践才造成了互联网今天的兴旺发展。我们向所有的志愿者们致敬 , 谷歌公司也将一如既往地继续支持开源代码技术和社区的发展。

让我们大家通过更多的分享和合作 , 一起快乐创新 !

谷歌开发技术推广部大中华区主管 奕跃

关于中文版

这是一份跟官方网站保持 **同步更新** 的中文版。虽然保持同步非常耗费精力 , 不过为了防止过时的内容误导读者 , 这份额外的付出还是很值得的。

我们全文采用意译的方式 , 在确保理解作者意思的前提下用中文重新表述 , 力求做到“信雅达” , 当原文难以直译时更是如此。在必要时 , 我们会加“译注”来辅助读者阅读。当然 , 即便如此 , 我们理解错误的可能性也还是有的 , 所以我们的译稿都提供了中英文对照 , 如果你对某些语句有疑问 , 只要点一下它 , 就可以显示对应的英文版内容 , 读者可自行对照理解。

对于英文词汇，我们尽量采用业内成熟的译法，以利于口头交流。对少部分在 Angular 之外比较罕见的专有名词，我们会在译文中写成中英双语。这部分做了初步的梳理，但是仍可能有遗漏，如果你发现哪里有问题，请到我们的[github](#)上提出 issue 或 Pull Request。

同时，要注意：我们只翻译“TypeScript”版的文档，其它语言的版本大同小异，可以在看懂 TypeScript 版之后再对照看 JS 和 Dart 的英文版。不过我们还是建议你试用一下 TypeScript。

授权方式

本文档遵循“[保持署名—非商用](#)”创意共享4.0许可证 (CC BY-NC 4.0) 授权方式，你不用知会我们就可以转载，但必须保持署名（特别是：链接到 <https://angular.cn>，并且不得去掉本页入口链接，也不得修改本页内容），并且不得用于商业目的。如果需要进行任何商业推广，请接洽 Google 开发技术推广部 devrel-china-contact@google.com，我们将给出积极的回应。

本文档首发于 [Angular 中文网](#)。如果你要进行转载，请自行同步，不过小心别 DDoS 了我们。

关于我们

两位译者汪志成和叶志敏的简介请参见[这里](#)。

Angular 中文社区的简介和更多资源请参见[这里](#)。

工作预告

我们已经组织了一个九人专家组，开始了对 [ng-book2](#) 的翻译工作，出版社也将加快流程，争取能尽早与各位见面。以我们一向的质量标准，我敢保证这会是一份精品。敬请期待！

将来，Google 技术推广部还会有一系列线上和线下推广工作，如果您有意为这些活动贡献力量，请接洽Google 开发技术推广部 devrel-china-contact@google.com。