架构概览

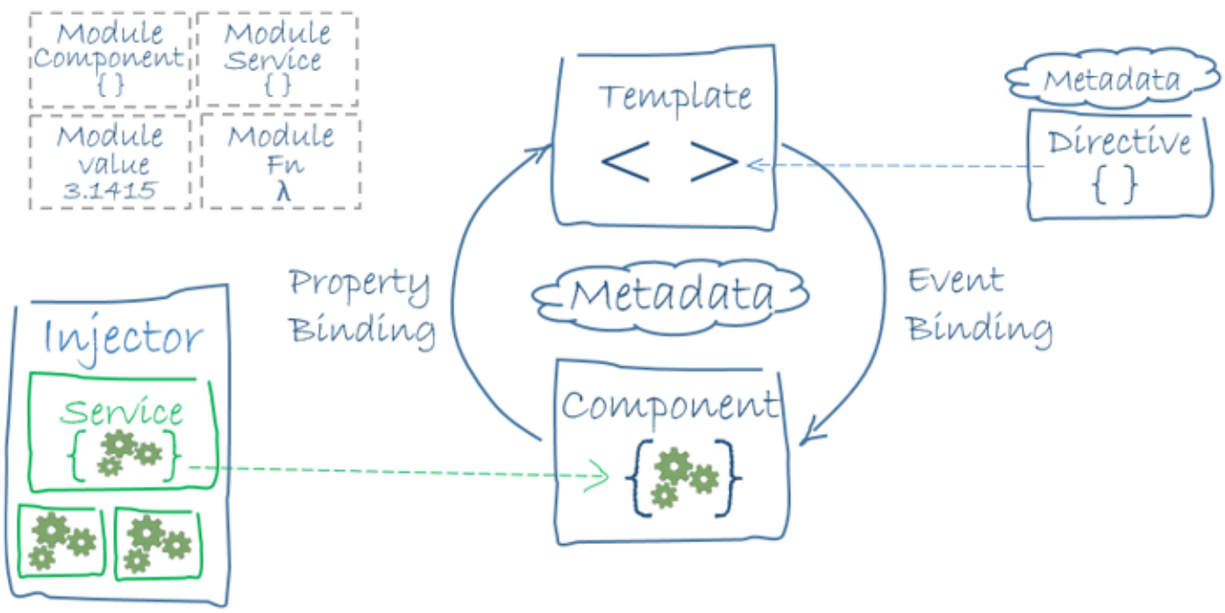
Angular 是一个用 HTML 和 JavaScript 或者一个可以编译成 JavaScript 的语言（例如 Dart 或者 TypeScript ），来构建客户端应用的框架。

该框架包括一系列库，有些是核心库，有些是可选库。

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

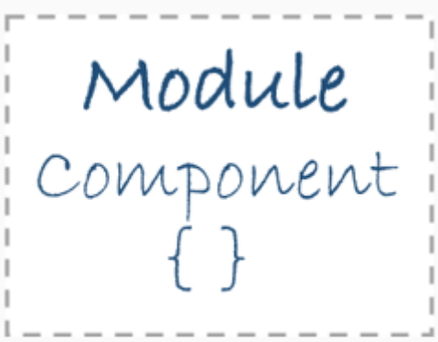
然后，我们通过*引导根模块*来启动该应用。 Angular 在浏览器中接管、展现应用的内容，并根据我们提供的操作指令响应用户的交互。

当然，这只是冰山一角。后面我们将学习更多的细节。不过，目前我们还是先关注全景图吧。



> 本章所引用的代码见在线例子 / 下载范例。

模块



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*。

NgModules 很重要。这里只是简单介绍，在 NgModules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，大多数应用会有很多*特性模块*，每个模块都是一个内聚的代码块专注于某个应用领域、工作流或紧密相关的功能。

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它们应如何工作。 关于装饰器的更多信息。

`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` - declarations 的子集，可用于其它模块的组件模板。
- `imports` - *本*模块声明的组件模板需要的类所在的其它模块。
- `providers` - 服务的创建者，并加入到全局服务列表中，可用于应用任何部分。
- `bootstrap` - 指定应用的主视图（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。

下面是一个简单的根模块：

src/app/app.module.ts

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

> `AppComponent`的`export`语句只是用于演示如何导出的，它在这个例子中并不是必须的。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，你通常在一个`main.ts`文件中引导`AppModule`，就像这样：

src/main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

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

NgModules vs. JavaScript 模块

NgModule（一个带`@NgModule`装饰器的类）是 Angular 的基础特性之一。

JavaScript 也有自己的模块系统，用来管理一组 JavaScript 对象。 它与 Angular 的模块系统完全不同且完全无关。

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

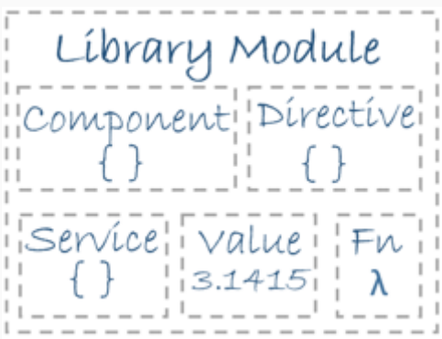
```
import { NgModule }     from '@angular/core';
import { AppComponent } from './app.component';
```

```
export class AppModule { }
```

> 学习更多关于 JavaScript 模块的知识。

这两个模块化系统是互补的，我们在写程序时都会用到。

Angular 模块库



Angular 提供了一组 JavaScript 模块。可以把它们看做库模块。

每个 Angular 库的名字都带有`@angular`前缀。

用 **npm** 包管理工具安装它们，用 JavaScript 的`import`语句导入其中某些部件。

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

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

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

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

在上面那个简单的根模块的例子中，应用模块需要`BrowserModule`的某些素材。要访问这些素材，就得把它加入`@NgModule`元数据的`imports`中，就像这样：

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

这种情况下，你同时使用了 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”。不过没关系，先放一放，随着时间和经验的增长，自然就清楚了。

> 更多信息，参见 NgModules。

组件

*组件*负责控制屏幕上的一小块区域，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，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，这个数组从一个服务获得。`HeroListComponent`还有一个当用户从列表中点选一个英雄时设置`selectedHero`属性的`selectHero()`方法。

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

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

  constructor(private service: HeroService) { }

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

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

当用户在这个应用中漫游时， Angular 会创建、更新和销毁组件。 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，例如上面声明的`ngOnInit()`。

模板

我们通过组件的自带的**模板**来定义组件视图。模板以 HTML 形式存在，告诉 Angular 如何渲染组件。

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

src/app/hero-list.component.html

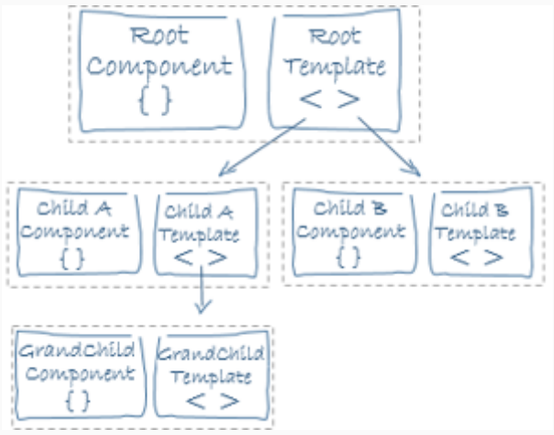
```
<h2>Hero List</h2>

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

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

模板除了可以使用像`<h2>`和`<p>`这样的典型的 HTML 元素，还能使用其它元素。 例如，像`*ngFor`、`{{hero.name}}`、`(click)`、`[hero]`和`<app-hero-detail>`这样的代码使用了 Angular 的模板语法。

在模板的最后一行，`<app-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`（代码未显示）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*。



注意到了吗？`<app-hero-detail>`舒适地躺在原生 HTML 元素之间。 自定义组件和原生 HTML 在同一布局中融合得天衣无缝。

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* 它是一个组件。

要告诉 Angular `HeroListComponent`是个组件，只要把**元数据**附加到这个类。

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。

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

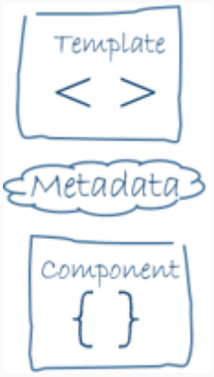
```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
/* . . . */
}
```

这里看到`@Component`装饰器，它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。

`@Component`的配置项包括：

- `selector`： CSS 选择器，它告诉 Angular 在*父级* HTML 中查找`<app-hero-list>`标签，创建并插入该组件。 例如，如果应用的 HTML 包含`<app-hero-list></app-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，如前所示。
- `providers` - 组件所需服务的*依赖注入提供商*数组。 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英雄数据。



`@Component`里面的元数据会告诉 Angular 从哪里获取你为组件指定的主要的构建块。

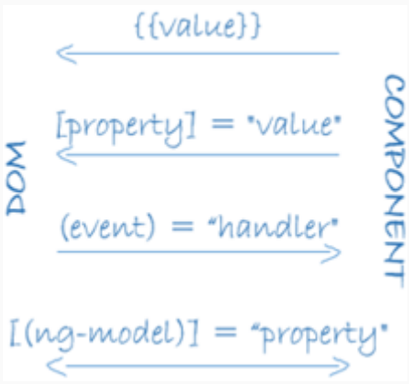
模板、元数据和组件共同描绘出这个视图。

其它元数据装饰器用类似的方式来指导 Angular 的行为。 例如`@Injectable`、`@Input`和`@Output`等是一些最常用的装饰器。

这种架构处理方式是：你向代码中添加元数据，以便 Angular 知道该怎么做。

数据绑定 (data binding)

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。



Angular 支持**数据绑定**，一种让模板的各部分与组件的各部分相互合作的机制。 我们往模板 HTML 中添加绑定标记，来告诉 Angular 如何把二者联系起来。

如图所示，数据绑定的语法有四种形式。每种形式都有一个方向 —— 绑定到 DOM 、绑定自 DOM 以及双向绑定。

`HeroListComponent`示例模板中有三种形式：

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

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

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

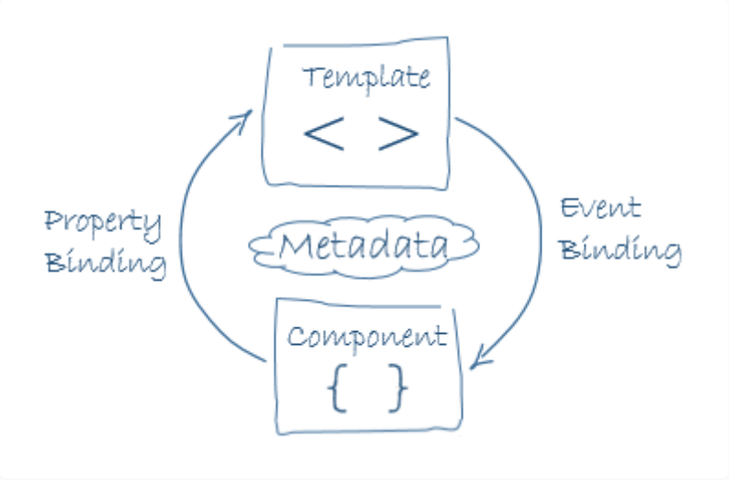
双向数据绑定是重要的第四种绑定形式，它使用`ngModel`指令组合了属性绑定和事件绑定的功能。 下面是`HeroDetailComponent`模板的范例：

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

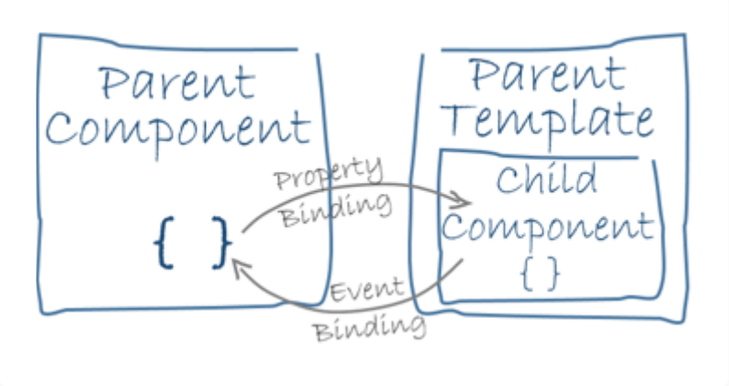
```
<input [(ngModel)]="hero.name">
```

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

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



数据绑定在模板与对应组件的交互中扮演了重要的角色。



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**提供的操作对 DOM 进行转换。

组件是一个*带模板的指令*；`@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特性。

> 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。

还有两种*其它*类型的指令：*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。

下面的范例模板中用到了两个内置的结构型指令：

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

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

- `*ngFor`告诉 Angular 为`heroes`列表中的每个英雄生成一个`<li>`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。 `ngModel`修改现有元素（一般是`<input>`）的行为：设置其显示属性值，并响应 change 事件。

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

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

Angular 还有少量指令，它们或者修改结构布局（例如 ngSwitch）， 或者修改 DOM 元素和组件的各个方面（例如 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*是一个广义范畴，包括：值、函数，或应用所需的特性。

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、明确的用途。它应该做一件特定的事情，并把它做好。

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，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。

下面是一个服务类的范例，用于把日志记录到浏览器的控制台：

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

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

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`服务和另一个用于处理服务器通讯的`BackendService`服务。

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

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

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

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

服务无处不在。

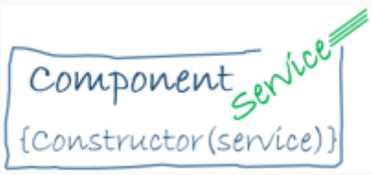
组件类应保持精简。组件本身不从服务器获得数据、不进行验证输入，也不直接往控制台写日志。 它们把这些任务委托给服务。

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*的某些概念）之间。 设计良好的组件为数据绑定提供属性和方法，把其它琐事都委托给服务。

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*遵循*这些原则 —— 它让我们能轻易地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（dependency injection）



“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 使用依赖注入来提供新组件以及组件所需的服务。

Angular 通过查看构造函数的参数类型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`服务：

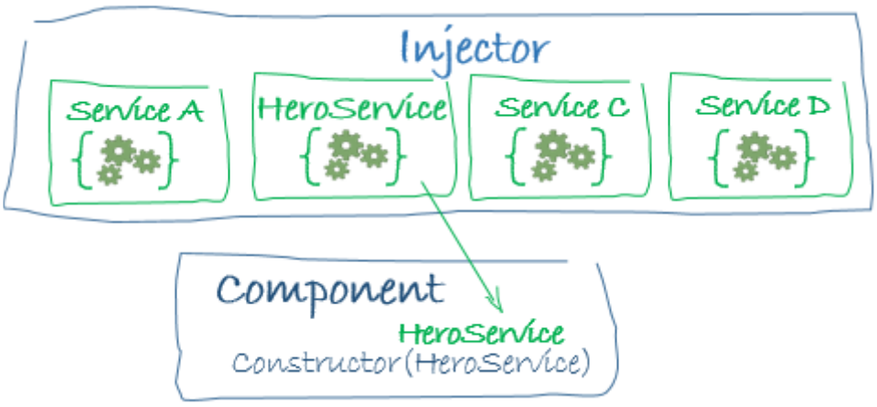
src/app/hero-list.component.ts (constructor)

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

当 Angular 创建组件时，会首先为组件所需的服务请求一个**注入器 (injector)**。

注入器维护了一个服务实例的容器，存放着以前创建的实例。 如果所请求的服务实例不在容器中，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。 当所有请求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`注入的过程差不多是这样的：



如果注入器还没有`HeroService`，它怎么知道该如何创建一个呢？

简单点说，我们必须先用注入器（injector）为`HeroService`注册一个**提供商（provider）**。 提供商用来创建或返回服务，通常就是这个服务类本身（相当于`new HeroService()`）。

我们可以在模块中或组件中注册提供商。

但通常会把提供商添加到根模块上，以便在任何地方都使用服务的同一个实例。

src/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：

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

```
@Component({
  selector:    'app-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，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，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，什么时候该更新到屏幕， 以及它是如何利用**区域 (zone)** 来拦截异步活动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 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。

Pipes: Use pipes in your templates to improve the user experience by transforming values for display. Consider this `currency` pipe expression:

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

管道：在模板中使用管道转换成用于显示的值，以增强用户体验。例如，`currency`管道表达式：

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

它把价格“42.33”显示为`$42.33`。

路由器：在应用程序客户端的页面间导航，并且不离开浏览器。

测试：使用 *Angular 测试平台*，在你的应用部件与 Angular 框架交互时进行单元测试。