# **［翻译］有关Angular 2.0的一切**

原文在[此](http://eisenbergeffect.bluespire.com/all-about-angular-2-0/)

是不是对Angular的战略有疑问？来这里就对了。在接下来的这篇文章里，我会解释Angular 2.0的主要特性区域，以及每个变化背后的动机。每个部分之后，我将提供自己在设计过程中的意见和见解，包括我认为仍然需要改进设计的重要部分。

注意：本文所反映是2014年11月6日的状态记录。如果你在较长时间之后读到此文，请检查一下我设计上是否有所变更。

## **AngularJS 1.3**

在开始讨论Angular的未来之前，我们先花点时间看看当前的版本。AngularJS 1.3是迄今为止最优的Angular版本，它是几周之前发布的。它提供了大量的bug修复，功能增强和性能提升。如果你正在使用Angular，会有升级的愿望。如果将要开始用Angular做新的项目，这也会是你想要使用的版本。这是一个强大而成熟的框架，已经摆在我们面前了。

#### **评注**

可能你现在会对AngularJS的未来有很多疑问。什么时候2.0会出来？1.x怎么办？会有从1.x到2.0的升级路径吗？AngularJS团队在回答这些问题上，可以做得更好一些，你应当鼓励他们这么做。我可以告诉你们的是，在Google内部，有1600个应用是基于Angular 1.2或者1.3构建的。所以，看起来Google对当前版本是会有很大兴趣的，也会需要支持它们一段时间。在[ngEurope的Q&A环节](https://www.youtube.com/watch?v=g-x1QKriY90&list=UUEGUP3TJJfMsEM_1y8iviSQ)中，Brad Green说在Angular 2.0的RTM版本发布之后，对Angular 1.3的支持会持续至少1.5-2年。我们也刚针对Angular 1.3的支持作了一些团队结构和领导的变更，即使是正在为Angular 2.0而努力，我们仍然保持了一个专有团队全职处理Angular 1.3。这个团队是由Pete Bacon Darwin领导的，我敢肯定你一定知道他在AngularJS上的丰富经验。我想要鼓励你们向Angular的引领者询问这些变化，并且一起设法完善官方的支持。

当2.0可用时，如果有人想要把Angular1.x的应用迁移到2.0，目前也没有什么可行的计划。我认为我们可以在这一块做些事。如果这对你来说也很重要，请大声说出来，当然要友善一点，但要让Angular团队知道这对你而言很重要，他们应当对此有所考虑，并且也有所规划。

## **Angular 2.0的动机**

那么，你可能会想知道，为什么要做Angular 2.0呢？为什么一步跨到2.0，并且作了这么多不兼容变更？这一切都是很随意的吗？我能够处理少量变更，但我所听到的消息，在2.0中有很多较大的变更，它们真的合理吗？值得吗？

在深入特性细节之前，我很乐意花点时间来探讨一些较高层次的动机，关于2.0所带来的变化。我希望这能够对后续细节建立一个基本的认识，在此基础上可以作一些有意义的批评（其中有些我打算自己提供）。

### **性能**

差不多五年前，当AngularJS刚创建出来的时候，它并不是给开发人员用的。它是一个工具，更倾向于给需要快速创建持久化HTML表单的设计人员用。随着时间推移，它作了改变以适应各种场景，开发人员也用它建造更多、更复杂的应用程序。Angular 1.x团队多年来一直努力增量化地改进设计，允许它适应现代Web应用程序需求的变更。然而，在所能做到的改进上，是有很大局限的，根源在于原始设计中的一些潜规则。很多这种限制，导致了当前的绑定与模板基础架构的性能问题。为了解决这些问题，需要新的策略。

### **变化的Web**

从最初设想Angular所开始的五年中，Web有了明显的改变。比如说，5年前没有jQuery之类框架的帮助，是基本不可能建立一个合适的跨浏览器网站的。但是，当今的浏览器DOM实现不仅更加一致，而且这些实现更快了，也提供了与应用程序框架相关的新特性。

而且web还在继续变化……

虽然在过去几年中，发生了巨大的变化，但与未来1-3年相比，这些变化还是显得微不足道。在几个月内，ES6规范将定稿。如果我们觉得在2015年就能看到完全实现此规范的浏览器，并非不可能。今天的浏览器已经支持其中一些特性了，并且正在实现其他剩余部分。这意味着浏览器支持像module、class、lambda、generator之类东西。这些特性从根本上改变JavaScript的编程体验。但是，大的变化并不是只体现在JavaScript上，Web Components也喷薄欲出。术语Web Components通常是指四个相关的W3C规范：

* 自定义元素，允许通过自定义标签来扩展HTML
* HTML Imports，允许对各种资源的打包（HTML，CSS，JS等等）
* 模板元素，允许在一个文档中包含inert HTML
* Shadow DOM，允许对DOM和CSS的封装

通过组合这四种能力，web开发人员可以创建声明式的组件（自定义元素），并且是完全封装的（Shadow DOM）。这些组件可以描述它们自己的视图（模板元素），并且能很容易打包发布给其他开发人员（HTML Imports）。当这些规范在所有主流浏览器都可用的时候，我们就可能会看到开发人员的创造力爆发，作很多努力来创建可复用的组件，以解决常见问题，或者是弥补标准的HTML工具集所存在的不足（摘自[Web Components与数据绑定](https://docs.google.com/document/d/1kpuR512G1b0D8egl9245OHaG0cFh0ST0ekhD_g8sxtI/edit?usp=sharing)）。今天，这已经变得可能，在Chrome和其他浏览器里，这些标准中有些已经实现，有些正在实现。未来显得很美好，对不对？只剩下一个问题了：当今的多数数据绑定框架尚未准备好应对这些。多数框架，包括Angular 1.x，包含一个数据绑定系统，它构建在一小部分已知的HTML元素和常用事件、行为的基础上。为了能让Angular开发人员享有Web Components，很需要有一个全新的数据绑定实现。

### **移动端**

想想5年前……噢，计算的情景已经有了多么大的改变！现在到处都是手机和平板了！虽然Angular可以被用于创建移动应用，但它的理念并非为它们设计的。这包括了所有的东西，从我刚提到过的基本的性能问题，到它的路由的能力缺失，以及不能缓存预编译视图，甚至是过于普通的触摸支持。其中有些东西可以借助Angular 1.3来实现（比如说路由），但其余的需要根本的变更来修复。

### **易用性**

老实说……AngularJS不是太容易学。是的，你选择了它，内心想着“这太美好，很简单，很魔幻！！！”然后开始建立自己的应用，发觉变成“TMD这什么啊！！？？我不懂！！！”这种事我听得多了，甚至还有个[直观的图](http://www.bennadel.com/blog/2439-my-experience-with-angularjs-the-super-heroic-javascript-mvw-framework.htm)用来描述它。如此种种，还是回头看看这个库最初的设计意图吧。比如，最开始是没有自定义指令的，它们都是硬编码的，然后，就有了一个用于添加指令的API。最开始是没有控制器的，然后……你懂的。这种绑死的特性，很多成为了现在的核心理念，导致了API的不优雅。如果Angular真想变的易学易用，那么，从一开始，它就必须对自己的核心特性有清晰的认识。对一个框架而言，如果把指令和控制器当作初始设计的一部分，肯定要比后面逐步拼凑起来要好几个数量级。

### **小结**

了解了Angular的设计起源，以及Web和通用计算情景的逐步变化，很明显需要作一些变更了。事实上，如果不开始解决这些问题，Angular很可能在一年内就有被淘汰的风险。一个框架，如果没法跟Web Components协作，在移动端上一塌糊涂，还继续推进自己的非标准的module和class API，离死也不远了。Angular团队对这些问题的回答是一个新版本：Angular 2.0。它本质上是为了现代Web而对AngularJS的重新想象，并且融合过去五年所得到的各种认识。

#### **评注**

尽管我尚未涉及详细的部分，你已经可以发现AngularJS 2.0是与1.x大为不同的了。可能有人会问，这还是不是同一个框架了？我觉得这是个好问题。我之前提到，我认为Angular团队需要提供对1.x支持的具体时间表，到2.0的迁移路径，以及给企业一些指引，供当前决策或者是想要升级为2.0的计划用。对于充满技术思维的Angular团队来说，这可能是很无趣的任务，但我认为它们对社区而言，是有必要的，有帮助的，也是一种尊重。

## **Angular 2.0的特性与设计**

现在，你已经对创建Angular 2.0的动机有了一点相关背景了，我们来看看一些关键的特性区域。

## **AtScript**

AtScript是一门语言，它是ES6的超集，被用来编写Angular 2.0。它使用TypeScript的类型语法来表达可选类型，这可以用来做运行时的类型推断，而不是编译时的检测。它也使用了元数据注解来扩展语言。这里有一个示例，有些AtScript代码就长这样：

import {Component} from 'angular';import {Server} from './server';

@Component({selector: 'foo'})export class MyComponent {

constructor(server:Server) {

this.server = server;

}

}

在这里，我们在基线ES6代码上添加了一些AtScript附属物。示例顶部的import语句和class语法是直接从ES6里来的，没什么特别的，但是，看一下构造函数，注意server参数指定了一个类型。在AtScript中，这个类型是用于生成运行时类型推断的，引用也会存在于已知的位置，这样，一个框架，比如说依赖注入框架，可以定位到类型信息，并且使用它。也注意一下class定义上面的**[@component](https://github.com/component)**语法，这是一个元数据注解。组件是一个普通的类，跟其他的一样。当你使用注解来装饰一个东西的时候，编译器会生成代码，初始化注解，并且储存在一个具体的位置，这样它可以被像Angular这样的框架访问到。考虑到这一点，这就是上面的代码转译成ES6语法之后的结果：

import \* as rtts from 'rtts';import {Component} from 'angular';import {Server} from './server';

export class MyComponent {

constructor(server) {

rtts.types(server, Server);

this.server = server;

}

}

MyComponent.parameters = [{is:Server}];MyComponent.annotate = [

new Component({selector: 'foo'})

];

RTTS的意思是运行时类型系统（RunTime Type System），这是一个小型的关于运行时类型检测的推断库。在此，编译器插入一些代码，以便把server变量推断为类型Server。你也可以编写自定义的类型推断以使用结构化类型，或者使用临时的类型规则。当部署到生产的时候，编译器可以省略这些推断以提高性能。

一个好的事情是，与类型推断相独立，类型注解和元数据注解是能够被跟踪到的。所有这些注解会被翻译成非常简单的ES5兼容数据结构，存储在MyComponent函数自身上。这使得任意框架或者库都能很容易发现这些元数据并且使用它们。多年来，这已经在像.NET和Java这样的平台上被证明是很方便的工具。它也和Ruby的元编程功能有一些相似之处。实际上，当跟一个库组合的时候，注解可以被用于做元编程，Angular 2.0就是借此简化指令的创建的。稍后将进行更详细的讨论。

#### **评注**

我个人是喜欢AtScript的，但我已经是一个TypeScript的爱好者了，所以你可能会说我是有前提条件的。我知道有些开发人员抵制在JavaScript中加入类型，我没法责怪他们。我自己已经在类型系统上有过相当广泛的经验了，有些是好的，有些不好。AtScript有个有意思的地方，你可以只把类型语法当成向其他库提供元数据的某种简单方式，而完全不用它来作类型检测。我觉得AtScript最强大的特性之一是，在运行时拥有类型和元数据信息，可供框架利用，或者是在自己的元编程中使用。如果[其他的解释型语言](http://blogs.msdn.com/b/typescript/archive/2014/10/22/typescript-and-the-road-to-2-0.aspx)也加上这种特性的话，我是不会感到惊讶的。

也就是说，我持有保留意见。

我很乐于看到AtScript变得更正式些，我意思是说，我认为它应当从Angular团队自身中释放出来，它应当有自己的发言权，Angular作为它的一个重要客户。应当至少有几个开发人员全职围绕AtScript工作，实现功能，修复bug，提升代码生成，构建工具等等，同时也应当有一个长期支持计划。当一个开发人员或者团队选择一种语言来编写他们应用的时候，他们所作出的是一种重大投资。我乐于看到Google能够为了未来，在AtScript上作出相当的投资。

关于AtScript，还有另外个问题，是跟Dart相关的。Dart是Google开发的另一种语言。它跟某种简单的解释性语言有所不同，是因为它有自己的运行时和基础类库。结果就是，Dart拥有自己的API，用于DOM处理，集合，事件，正则表达式等等。这些API在它们自己的领域中都很优秀，但跟已有的JavaScript代码不兼容。由于这种阻抗不匹配，Dart和外界的任何通讯都必须通过一个特殊的编组API来完成。所以，虽然从技术上可以调用现有的JavaScript库，一般来说不太实用。对AngularJS来说，性能上的损耗将是不可接受的。所以，Google创建了Angular Dart，一种用Dart重新思考过的AngularJS版本。

问题解决了……好吧，可能没有。

现在，就有了两个Angular的版本，要在里面修改bug，实现新特性，发布，等等，使用不同的语言编写，由不同的团队维护。所以，解决了一个问题，却带来了更多问题。

现在你可能有疑问了：这跟AtScript有什么关系呢？

Angular 2.0的想法是把Angular和Angular Dart统一起来。一个团队在一个代码库上工作，要比两个团队在两个代码库上工作好多了。AtScript能在这个事情上起作用，因为它是在[Traceur](https://github.com/google/traceur-compiler)上面实现的，这个东西可扩展性很好。所以，Angular团队能够用AtScript编译出JavaScript和Dart两个版本。

太棒了！那么，问题在哪里呢？

记得我提到过Dart在DOM之类的东西上有不同的对象模型，这些东西就不是简单转译代码所能解决的了。因此，Angular 2.0的构建过程实际上就会复杂一些了。当开发Angular的时候，必须创建不同的门面（facade）以屏蔽JavaScript和Dart之间的API差异。然后编译器使用对应的门面来编译成每种指定的语言。这个事情在技术上肯定是令人印象深刻了，但是，却大大提高了想要转向Angular 2.0的准入门槛。值得注意的是，这方面的发展还处于试验阶段，这个问题可能会有其他的解决方案。我知道你们中的很多人已经转向了Angular，并且很珍惜这种经验。Angular团队也很珍惜它们，我们正在深度思考如何去改进这些，不过，到目前为止，还不是很理想。

注意：Angular 2.0是使用AtScript编写的，但这并不意味着你就需要用AtScript编写你的应用，或者为了使用Angular 2.0，要学AtScript的什么东西。你可以很轻松地使用TypeScript，ES6，ES5，CoffeeScript……随便什么喜欢的东西来写。目前来说，如果利用AtScript的话，能够获得最佳的Angular体验，因为它能够从语言原语自动生成元数据，不过，最终它还是会翻译成简单的ES5。最终的ES5在概念上某种程度类似于Angular 1.x里面的DDO对象，但在此情况下，它是被生成给任意JavaScript函数使用的，而不是某种指令相关的技术，需要用特殊的注册API来编写。

### **依赖注入**

Angular 1.x的核心特性之一是依赖注入（DI，Dependency Injection）。通过DI，你可以很容易地在软件开发过程中遵循“分而治之”的实践。复杂的问题可以根据其角色和职责进行概念化，然后表示成对象，共同协作以完成最终目标。使用这种方式解构的大型（或者小型）系统可以通过使用DI框架在运行时进行组装。这种系统通常是容易测试的，因为结果的设计更加模块化，也允许了更容易的组件隔离。当然，这一切在Angular 1.x中都是可以的，不过有一些问题。

困扰1.x DI实现的第一个问题是由压缩（minification）引起的。鉴于DI依赖于从函数解析参数名，本质上是把它们当作字符串令牌，而在压缩过程中，这些名称会被改变，就不再匹配于注册的服务、控制器和其他组件了。结果就是应用挂了。为了使得DI对压缩友好，添加了一个API，但它缺乏原始的优雅。在.NET和Java的世界中，先进的服务端DI框架里存在更多特性，1.x的实现主要就缺乏这些东西。欠缺的特性导致开发人员受到约束，两个大的例子是：生命周期/作用域的控制，以及子注射器。

### **注解**

通过AtScript，我们引入了一种广义的将元数据附加到任意函数的机制。同时，AtScript元数据格式是不怕压缩的，也容易使用ES5手工编写。这使得它能够出色地支撑一个DI库，提供其所需要用于构造对象实例的信息。不必见怪，这就是新DI的运作方式。

当DI需要实例化一个类（或者调用一个函数）的时候，会检测一下，看看它上面有没有带附属的元数据。回顾一下上面从AtScript转译出来的代码：

MyComponent.parameters = [{is:Server}];

如果新DI发现了parameters值，会用它来判断将要尝试调用的函数的依赖项。在本例中，它可以得知仅有一个类型为Server的参数。所以它会获取一个Server的实例，并且在调用这个函数之前传进去。你也可以显式提供一个特定的Inject注解给DI用，这会覆盖parameter数据。如果你在使用一种不能自动生成parameter元数据的语言，也很容易支持，下面就是用原生ES5代码写的样子：

MyComponent.annotate = [new Inject(Server)];

这个的运行时效果跟parameter数据是一样的。值得注意的是，你可以使用任意东西当作注入令牌，所以可以这样：

MyComponent.annotate = [new Inject('my-string-token')];

只要你在DI上配置过能映射到'my-string-token'的东西，它就能运行。也就是说，推荐的使用方式是通过构造函数的实例，正如我之前所有的例子所示。

#### **实例作用域**

在Angular 1.x中，DI容器中的所有实例都是单例。在Angular 2.0中，默认也是这样。为了获得不同的行为，你需要使用Service，Provider，Constant等等。那都挺容易让人迷惑的。幸运的是，新DI拥有一个新的，更通用，更强大的特性。它现在有实例作用域控制了。所以，如果你希望每次请求的时候，DI都创建一个类的新实例，可以这么做：

@TransientScopeexport class MyClass { ... }

当你组合子注射器来创建自己的作用域标识符的时候，这会更加强大……

#### **子注射器**

子注射器是一个主要的新特性。子注射器从其父项那里继承到所有父项的服务，但能够在子级别上覆盖它们。当它与作用域标识符组合使用的时候，你可以很轻松地在系统中调用到特定类型的对象，这些对象应当在不同作用域中被覆盖，这非常强大。新的路由有一个“子路由”的功能，就是使用它的一个例子。在内部，每个子路由创建自己的子注射器，这使得路由的每个部分能够从父路由继承服务，或者在不同的导航场景中覆盖这些服务。

注意：自定义作用域和子注射器会被认为是对注射器的中高级用法。我不希望太多的应用代码用到它。但是，既然它在Angular的内部被使用到了，如果你需要类似功能，也可以用。

#### **更多……**

在新DI中，还有一些其他特性，比如provider（自定义函数，用于提供一个注入值），懒注入（指定你所期望注入的东西，但又不立即需要，稍后才要），还有基于promise的异步注入（注入一个promise，可以从中获取异步的依赖项）。

#### **评注**

从个人角度，我非常喜欢新DI。我又有偏见了，因为我用DI好多年了，在我创建的其他UI框架中，它也是核心组件。新DI在Angular 2.0中扮演了很重要的角色，像子注射器等功能带来了巨大的变更。现在这个功能有了，它能够被模板引擎和路由利用，这两者都有创建作用域和隔离不同服务的需求。

然后我们就来到了一个从Angular中移除的重要特性：$scope。不过，虽然$scope自身被移除了，它的有些特性还在。这些特性被作为此设计的一个部分，重新换了个位置，也有所提升。你可能会被$scope的缺失搞得措手不及，但新的设计既简化了Angular内部的东西，也简化了提供给你，开发人员的东西。我提到这些，是因为DI的有些新功能，比如说子注射器，与$scope中之前的一些功能重叠了。在这个情况下，我认为新DI系统拿出来的是一个更好的解决方案。它更加通用，所以不但能解决Angular的内部需求，还给你们开放了很多种可能性。

不幸的是，玫瑰带着刺。我们来讨论下一些其他问题。在Angular 1.x中，还有个相关功能我没有提到：module，你可能想知道它的位置在哪。Angular 2.0的方案是吸收ES6中关于module的标准。在Angular的之前版本中，处理模块的方式是Angular特有的方式。五年前，当Angular刚开始构思时，并没有用于完成此事的标准方式。今天，事情不同了，已经有了一个明确路径。这当然是一种不兼容升级，如果有人要作迁移的话，需要对代码重新作点调整。要作这么一种不兼容变更，是很恶心，但这就是Web的变化影响框架的一个实例，如果这事现在不解决，2.0将面临被边缘化的风险。

关于DI，还有另外一个坑，特别是如果你在用ES5写代码的话。Angular 2.0依托作为元数据的注解，支持了基于类的设计。类和注解的语法在ES5中并不太好，事实上，ES5压根就没这些语法。你可以使用原型之类来表示一切，但就没有AtScript甚至ES6或者TypeScript那么清晰了，它们可是有类和静态的类成员的。我想知道能不能为不准备迁移到ES6的开发人员做点什么，也许是一个简单的可选降级库，给出一种简单的方式来创建带元数据的类？可能会类似于Angular 1.x中的DDO对象，但是更通用，这样能创建任意的类和元数据。我很想听听你对这个想法的意见，或者其他可能会解决ES5开发问题或对迁移能有所提升的主意。

### **模板与数据绑定**

如果你已经看了这么多，一定属于对Angular 2.0非常好奇的，感谢花了这么多时间。我们还有一条路要走，现在我们要进入真正有意思的地方了：模板和绑定。我打算把它们放在一起讨论，虽然从技术上看，数据绑定系统是与模板系统分离的，你在编写应用的时候却会感觉它们是一个整体。所以，我觉得把它们拿到一起来说会比较好。

我们先从理解视图到屏幕的显示过程，然后一点一点地看。本质上，你是从一段HTML片段开始的，这会存在于一个<template>元素中。这个HTML片段被传递给模板编译器，编译器遍历模板，辨识任意的指令，绑定表达式，事件处理函数等等。所有这些数据从DOM自身中提取，放到最终用于初始化模板的数据结构里。作为这个阶段的一部分，在数据上作了一些处理工作，比如说解析绑定表达式。每个包含上面这种特殊指令节点会被打上一个特殊的class。这一过程的结果会被缓存，这样才不至于需要重复这些工作。我们把这种结果称为一个ProtoView。一旦我们有了ProtoView，就可以用它来创建View。当一个ProtoView生成了View，所有刚才辨识出的指令就会初始化，并且附加到它们的DOM节点上，绑定表达式上建立了监控，事件处理器也配置好了。明白了吧。在编译阶段，之前处理过的数据结构能够让我们很快地做这些事。一旦你得到一个View了，就可以把它添加到一个ViewPort中，并且显示出来。一个ViewPort表达了屏幕的一个区域，可以在其中显示View。作为一名开发人员，大部分东西你是看不到的，你写模板就好了，它会运行的。可我还是希望在深入细节之前，在一个较高层次上把这些过程罗列出来。

#### **动态加载**

Angular 1.x所缺乏的重大功能之一是代码的动态加载。如果你想在运行中添加新的指令或者控制器，非常困难，或者就做不到。它没有被支持。在2.0中，我们从开始设计东西的时候，就把异步放在心里。所以，当你开始编译一个模板的时候，实际上它是个异步过程。

现在我需要详细讨论上面一笔带过的模板编译了。当你编译一个模板的时候，你并不仅仅为编译器提供了一个模板，也同时提供了一个Component的定义。我们稍微深入一点。在模板中使用的时候，Component的定义就包含了什么指令啊，过滤器啊之类的元数据。这确保了在模板被编译器处理之前，所有必要的依赖项都已加载。由于我们的代码架设在ES6 module规范的基础上，只需简单地在Component定义中引用依赖项，如果他们尚未加载，module加载器就会加载它们。因此，通过这种结合ES6 module的方式，我们不费事就得到了各种东西的动态加载。

#### **指令**

在我们深入模板的语法之前，需要先看一看指令——Angular用于扩展HTML自身的方式。在Angular 1.x中，使用指令定义对象（DDO，Directive Definition Object）来创建指令。这好像是很多Angular开发人员巨大痛苦的来源之一。

如果我们能把指令弄简单点，会怎样呢？

我们已经讨论过模块、类和注解了，如果我们能用这些核心建筑来构建指令会怎样呢？好吧，我们当然就是这么干的。

在Angular 2.0中，有三种指令类型。

* 组件指令（Component Directive）创建一个组合了View和Controller的自定义组件，你可以把它当成一个自定义HTML元素。路由也可以映射到组件。
* 装饰指令（Decorator Directive）使用附加行为来装饰一个已有的HTML元素，一个经典的例子是ng-show。
* 模板指令（Template Directive）把HTML转换成一个可复用的模板。指令的作者可以控制模板何时、怎样初始化，并且插入DOM中。示例包括ng-if和ng-repeat。

你可能听说过在Angular 2.0里面，Controller没了。好吧，不完全正确。其实，Controller成为了我们称之为Component的一部分。Component拥有一个View和一个Controller。View就是你的HTML模板，Controller就是你的JavaScript行为。不像在1.x中那样，要用显式的或者非标准的API来注册控制器，在2.0中，只需创建一个普通的带一些注解的类。这里是选项卡容器组件的控制器的一个部分（稍后会看到它的视图）：

@ComponentDirective({

selector:'tab-container',

directives:[NgRepeat]

})export class TabContainer {

constructor(panes:Query<Pane>) {

this.panes = panes;

}

select(selectedPane:Pane) { ... }

}

这里有几个特性值得注意。

首先，组件的控制器只是一个类。它的构造函数会被自动注入其依赖项。因为使用了子注射器，它可以获得沿DOM树向上所有服务的访问，还包括从属于自己元素的本地服务。比如说，这里它就被注入了一个Query，这是一个特殊的集合，会自动跟子Pane元素保持同步，让你获知何时出现新增或者移除。同时，你也被注入了Element自身，这能让你处理与Angular 1.x中$link回调相同的逻辑，但却是通过类构造函数，用一种更一致的方式来处理的。

现在，看一看@componentdirective注解吧。它把类标识为一个Component，并且提供了编译器所需用于挂接的元数据。比如说，selector:'tab-container'是一个CSS选择器，会被用于匹配HTML。任何匹配这个选择器的元素都会被转换成一个TabContainer。同时，directives:[NgRepeat]表明了这个组件的模板的依赖项。到现在还没给你们看过，马上讲语法的时候就会看到了。

一个重要的需要注意的细节是，模板将会直接绑定到这个类上，意味着类的任何属性和方法都能直接在模板上访问。这根Angular 1.2中的“controller as”语法很相似。在类和模板之间，不再有$scope了。结果就是Angular内部得到了简化，开发人员也得到了更简单的语法，那种在$scope对象上搞来搞去的事情变少了。

接下来，我们来看看Decorator Directive。一个简单的NgShow是怎样的呢？

@DecoratorDirective({

selector:'[ng-show]',

bind: { 'ngShow': 'ngShow' },

observe: {'ngShow': 'ngShowChanged'}

})export class NgShow {

constructor(element:Element) {

this.element = element;

}

ngShowChanged(newValue){

if(newValue){

this.element.style.display = 'block';

}else{

this.element.style.display = 'none';

}

}

}

这里，我们可以看到指令的更多方面。我们又写了个带注解的类，构造函数注入了装饰器要附加到的HTML元素。因为有DecoratorDirective，编译器知道这是一个装饰器，也知道把它添加到任意匹配于selector:'[ng-show]' CSS选择器的元素上。

在这个注解上，还有其他一些奇怪的属性。

bind: { 'ngShow': 'ngShow' }用于把类属性映射到HTML attribute。不是所有的类属性都直接暴露成HTML的attribute的，如果你想要让属性在HTML中可绑定，需要在bind元数据中指定它。observe: {'ngShow': 'ngShowChanged'}告诉绑定系统，你想要在每次ngShow属性变更的时候得到通知，并且使用ngShowChanged方法作为回调。注意ngShowChanged回调响应变更的方式是，改变附加到的HTML元素的display。（注意，这只是个非常幼稚的实现，仅作演示之用）。

好了，那Template Directive长什么样呢？为什么不看看NgIf呢？

@TemplateDirective({

selector: '[ng-if]',

bind: {'ngIf': 'ngIf'},

observe: {'ngIf': 'ngIfChanged'}

})export class NgIf {

constructor(viewFactory:BoundViewFactory, viewPort:ViewPort) {

this.viewFactory = viewFactory;

this.viewPort = viewPort;

this.view = null;

}

ngIfChanged(value) {

if (!value && this.view) {

this.view.remove();

this.view = null;

}

if (value) {

this.view = this.viewFactory.createView();

this.view.appendTo(this.viewPort);

}

}

}

希望你能理解TemplateDirective注解。它注册了这个指令，并且提供了一些必要的元数据用于设置属性和观测，就像NgShow示例那样。这就是一个TemplateDirective，它能够访问一些特殊的服务，这些服务可以被注入它的构造函数。第一个是ViewFactory，之前我提到，Template Directive把它附加到的HTML转换为模板，模板被自动编译，然后你就能在模板指令中访问视图工厂了。调用工厂的createView API会初始化模板自身。你也可以访问ViewPort，这代表了模板从DOM中提取的位置，你可以用它在DOM上添加或者移除模板的实例。注意ngIfChanged回调是怎样响应变更，初始化模板，添加到viewport，或者从viewport上移除的。如果你在实现类似于NgRepeat的东西，可以把模板实例化很多次，甚至给createView API提供一个指定的数据项，然后可以把多个实例添加到viewport上。基本就是这样。

现在，你已经看到三种类型指令的一些典型例子了。我希望这能够大致说明了如何使用新的行为来扩展HTML编译器。

不过，还有一个重要的东西我尚未充分解释：Controllers。

怎样为应用创建一个控制器呢？设想你要建立一个路由，导航到一个控制器，然后显示它的视图。怎么做到这个呢？简单的回答就是使用一个Component Directive来做。

在Angular 1.x中，Directive和Controller是两种不同的东西，API不同，功能也不同。在Angular 2.0中，既然我们已经去掉了DDO，把Directive变成了基于类的，我们可以把Directive和Controller统一成Component模型。所以，现在可以一箭双雕，当建立路由的时候，只要把路由映射到一个ComponentDirective（本质上由一个视图和控制器构成，就像之前一样）。

所以呢，如果你创建一个假想的客户编辑控制器，可能会是这样：

@ComponentDirectiveexport class CustomerEditController {

constructor(server:Server) {

this.server = server;

this.customer = null;

}

activate(customerId) {

return this.server.loadCustomer(customerId)

.then(response => this.customer = response.customer);

}

}

真没什么新东西，我们就是在注入假想的服务端服务，当被路由激活的时候，使用它加载客户。有意思的是，你不需要使用选择器或者是其他任何元数据，因为这个组件不是被当作自定义元素来使用的。它是被路由动态创建，然后动态渲染到DOM中的。总之，不要太在意细节了。

那么，如果你明白了怎样创建ComponentDirective，你就明白了怎样创建等同于Angular 1.x中使用路由创建的控制器。在Angular 1.x中，很难把这些统一起来，但鉴于我们在Angular 2.0中有了这么帅的类和元数据驱动系统，指令就可以很显著地简化了，用这种方式创建你的“控制器”也变得很容易了。

注意：我想指出，上面的指令代码示例基于早期的原型代码和较新的设计文档规范，它们应当被解读为一种解释的工具，而不是指令的准确语法，那东西还在不断变化。模板编译器和绑定语言现在是Angular 2.0中最不稳定的部分，设计的变更非常频繁。

#### **模板语法**

至此，你已经对编译过程有了一个概要的认识了：知道它可以异步加载代码，如何编写指令，它们是怎样装配的，控制器怎么适应这些东西的。但我们尚未看一下真正的模板。我们现在来看看刚才假想的TabContainer的模板吧。为了方便起见，把指令代码再贴一遍：

@ComponentDirective({

selector:'tab-container',

directives:[NgRepeat]

})export class TabContainer {

constructor(panes:Query<Pane>) {

this.panes = panes;

}

select(selectedPane:Pane) { ... }

}

<template>

<div class="border">

<div class="tabs">

<div [ng-repeat|pane]="panes" class="tab" (^click)="select(pane)">

<img [src]="pane.icon"><span>${pane.name}</span>

</div>

</div>

<content></content>

</div>

</template>

当你看到这个语法的时候，不要害怕。是啊，这是符合[规范](http://www.w3.org/TR/html5/syntax.html" \l "attributes-0)的HTML，但不是我们最后的绑定语法。不过，还是用它作例子吧，这样我们能有个比较丰富的讨论的起点。

理解数据绑定语法的关键是属性定义的左侧，考虑到这点，我们先来看一下image标签。

<img [src]="pane.icon"><span>${pane.name}</span>

当你看到一个属性名称被[]包围的时候，它意思是右侧的属性值有一个绑定表达式。

当你看到一个表达式被${}包围的时候，它意思是这是一个表达式，应当被当作字符串插入到内容中（这跟ES6用来做字符串插值的语法相同）。

从模型/控制器到视图的绑定都是单向的。

现在让我们看看这可怕的div：

<div [ng-repeat|pane]="panes" class="tab" (^click)="select(pane)">

ng-repeat是一个TemplateDirective，你可以看出我们正在使用一个表达式来绑定它，因为它外面有[]。不过，它里面还有个 | ，还有一个单词“pane”，这表明在模板中使用的局部变量名称为“pane”。

现在看看(^click)，使用括号表明我们把这个表达式作为一个事件处理函数。如果在括号里还有个 ^ ，就意味着不把处理函数直接附加到DOM节点上，而是让它冒泡，在文档的级别处理它。

在这个和模板部分的其他东西上，我暂不表达自己的意见，到下面的评注章节再说。现在先不管你我第一次看到这个的想法，先讨论为什么会选择这样的语法。

Web Components改变了所有东西。这是另外一个Web的变化影响框架的例子。多数基于数据绑定的框架假定了HTML元素的一个固定集合，并且已经预知了一些特定元素的行为，比如说input等等。可是，在Web Components的世界里，没有什么可以假设。一个开发人员，不针对Angular，可以编写一个自定义的元素，带有任意数量的属性，高兴加什么事件就加什么事件。不幸的是，没有办法检测Web Component来收集有关这些元数据，驱动绑定系统需要这些数据。比如说，没有办法知道实际触发了什么事件。看这个例子：

<x-foo bar="..." baz="..."></x-foo>

看看bar和baz，你能知道哪个是事件，哪个是属性？不……不幸的是，Angular也不知道，因为Web Components规范没有包含自描述组件的概念。这很不幸，因为它意味着一个数据绑定系统没法知道：它是不是需要连接一个绑定表达式，或者是不是需要添加一个事件处理函数来调用表达式。为了解决这个问题，我们需要一个通用的数据绑定系统，语法能够让开发人员区分哪个是事件，哪个是属性绑定。

这还不是唯一的困难。此外，所提供的信息还必须以这么一种不打破Web Component自身的方式。我这话的意思是，不能让Web Component看到这些表达式，那会破坏这个组件，它只应当看到表达式执行后的结果。实际上这不仅仅影响到Web Components，也会影响原生元素。考虑一下这个：

<img src="{{some.expression}}">

这个代码会产生一个错误的http请求，企图寻找“some.expression”这个图。这压根就不是我们想要的，我们根本不想img看到这个表达式，只希望它看到值。AngularJS 1.x解决这问题的方式是使用ng-src，一个自定义指令。现在，我们回到Web Components……如果你要给任意Web Compoents的每个属性都创建一个自定义指令，会是一场灾难，是不是？我觉得不能这样，所以需要在绑定系统中更普遍地解决这个问题。

要完成这事，你有两个选择。第一个是在模板编译期间，从DOM上移除属性。这能够阻止Web Component碰到这个表达式的文本。可是，这么做就意味着检测DOM的话，跟踪不到属性上绑定表达式的执行。这会让调试更加困难。另外一个选择是把属性名编码，这样Web Component就“认不出”它。这样可以让Angular看到这些表达式，但是Web Components却看不到。我们也可以在编译后把属性留在元素上，这样检测DOM的时候可以看得到。在调试的时候，当然就好得多了。

从以上可以看到，Angular团队目前支持的是编码属性的方法，这种编码也需要区分属性和事件。上面所示的语法是完成此事的多种可选项之一。

Angular团队已经在这一块严重争论了几个月。上述语法并未获得一致同意，但被多数人认同。当制订绑定语法的时候，也有过大量的考虑。如果你对这些感兴趣，我建议你读一下关于此主题的相当广泛【丰富】的[文档](https://docs.google.com/document/d/1kpuR512G1b0D8egl9245OHaG0cFh0ST0ekhD_g8sxtI/edit?usp=sharing)。

好吧，现在我们终于介绍完了模板、绑定和指令是怎么混到一起的……

#### **评注**

关于刚才这些，我有太多话要说了……真不知道从何说起……

先从编译器自身说起吧。

我们是从一个较高层次看待模板的编译过程的。虽然在这一块，还有大量的实现要做，我对编译器的设计非常满意了。在这里面有一些挺好的东西，保持小的内存占用，减少了垃圾，并且使得模板的实例化超快。这些都很伟大，当然还需要改进，但已经很稳定了。虽然我们尚未谈及脏检测（数据绑定表达式更新的机制），它的实现也有一些新的不错的想法，可能会让模板实例化和脏检测自身的性能都有所提升。当然，能够动态加载任意东西太可怕了，这是Angular 1.x非常缺乏的一个特性，对大型应用却很关键。所以，它能作为核心需求来设计，我很高兴。

现在我们来讨论指令吧。

使用类、更好的依赖注入和注解来创建指令的新机制非常棒，它比Angular 1.x所需要的要简单多了。不幸的是，对于1.x开发人员而言，这是一个相当大的不兼容变更，如果你限于使用ES5，而不能或者不想使用ES6，TypeScript或者AtScript的话，写起来也会有些困难。本文的前面部分，我提到过提供小型库用于在ES5中更方便地创建注解类，这样的API可能会搞得像DDO对象那样，也许这能让从1.x到2.0的移植过程简单点。或许我们现在就应当开始构建它了，这样你可以在1.3里面使用，然后为2.0提供一个不同的实现……一种迁移抽象层。我也不确定，我想听听你们关于这块的想法，我知道很多人很关心这个。

关于指令，还有另外一件让我很困扰的事：注解有些冗长。回顾一下NgShow指令，你看到文本‘ngShow’或者它的某种变体重复了多少次？这对我来说显得有些傻。再看看CustomerEditController，我们要ComponentDirective干什么啊？既然路由都知道它是什么了，我们只写个普通类不行吗？

在内部我说了很多约定优于配置的想法，这也是Rails流行并影响很多现代框架的原因，我认为这是一种积极的方式。我想看到一些用于创建指令的约定能把样板消除。没有它们的话，我会认为新的指令系统并未把指令简化到应有的程度，感觉就像是把DDO的一些复杂性放到另外一个地方，也就是注解中去了。你会怎么想呢？喜欢约定吗？觉得这能让指令简单吗（假设你一直选择明确 这个地方怎么翻译啊）？

(assuming you always had the option to be explicit and override the conventions with annotations)?

不幸的是，这些都还不是真正的大问题。在Component Directive中有一个严重的问题，希望你已经看到了。注意到它们破坏了展现分离（Separated Presentation）原则吗？再看看我的TabContainer示例：

@ComponentDirective({

selector:'tab-container',

directives:[NgRepeat]

})export class TabContainer {

constructor(panes:Query<Pane>) {

this.panes = panes;

}

select(selectedPane:Pane) { ... }

}

有没有看到TabContainer必须在其元数据中，列出其模板使用到的所有指令？这是TabContainer（控制器）到其视图实现细节的直接耦合。之前我提到过，对编译器而言，这是有必要的，因为它在编译模板之前，需要知道要加载什么，但是，这抵消了使用MVC，MVVM或者其他展现分离模式所带来的主要优势。为了避免你觉得我在纸上谈兵，我来指出一些后果吧：

* 没办法实现ng-include了。为了编译HTML，编译器需要ComponentDirective，因此你就没法编译它自己上面的HTML，并且将其包含到一个View里。
* 如果你想要为同一个组件创建多个\_潜在的\_视图，会很痛苦。想象一下，你有一个组件，但是想要在手机和电脑上显示不同的视图，需要聚合所有的指令、过滤器之类，这些东西你在所有视图都要用，还要确保在单个组件的元数据里把它们都列出来。这个维护起来太可怕了。如果不检测所有的视图，从依赖列表中删任何东西都是不可靠的，也有可能会忘记加东西。
* 对同一个组件而言，是不可能拥有多种\_运行时\_视图的。想象一下，你配置了一些路由，某些路由使用了相同的“控制器”，但是你需要不同的视图。不好意思，真做不到。
* 完全不可能实现展现层的临时组合，最起码这会让数据驱动的UI构建更加复杂。你没法简单地组合视图和控制器（视图模型）来渲染，这抑制了UI的组合方式，也限制了可复用性，它迫使你为了得到不同的视图，把控制器拆分成子类。

幸好，设计还是会有不少变化的，我提了个建议来解决这个问题，非常简单：通过让模板指定自己的imports，使得它能够完全自包含。把元数据移出指令，放到HTML模板中。可以这样使用一个自定义元素：

<ng-import src="ngRepeat"></ng-import>

编译器可以很容易找到他们，并且确保编译模板内容的时候，所有东西都加载完成了。就这样，刚才我提到的所有问题都解决了。

好了，现在我们说完编译器和指令了……

我感觉接下来是不是该说模板语法了？

老实说，很多人看到这个模板语法的时候可能会吐了，不是所有人，但有不少。我个人是不太喜欢这种语法的，但这是多数人投票的结果（为了理解为什么它还是个草案，你需要看看[这篇文档](https://docs.google.com/document/d/1kpuR512G1b0D8egl9245OHaG0cFh0ST0ekhD_g8sxtI/edit?usp=sharing)）。别怕！已经有一些技术性的问题让这种语法变不成事实，更不用说社区的反对之声了。Angular团队已经回头继续讨论最佳语法了，很多社区成员也加入了，并且提出了[自己的见解](https://github.com/angular/angular/issues/133)，很棒。我会把我自己的推荐放在这里，这样每个人都可以评论。

这是我提议的基本语法：

property="{{expression}}" - 从模型到元素属性的单向绑定，使用{{}}标识  
on-event="{{expression}}" - 给事件添加处理函数，执行表达式，使用on-前缀标识  
${expression} - HTML内容和属性中的字符串插值（基于ES6语法）

就这样。然后，在实现的时候，我们需要把表达式从DOM移除，以避免各种Web Component的问题之类。仅在调试模式，我们可以通过一个前缀，比如bind-，把它们加回来，这样可以在不影响Web Components的情况下，通过检测DOM的方式看到它们。这使得你所写的和在DOM检测器中看到的东西不太对称，但我觉得为了清晰、更加标准的绑定语法起见，这是一种合理的权衡。不是所有人都同意我，你觉得呢？

这个提议解决了绑定的技术问题，也对向后的兼容性有所帮助。可能我们能够对向后兼容性做很多的事情。我们能够允许在HTML内容中使用{{expression}}来做字符串插值，但是你可能会对此有选择余地。它可能会计划在20xx年被淘汰。推荐的方式可能会是${expression}，但这可以对模板提供一个更平缓的升级路径。也有可能创建一套可选的指令用于支持ng-click之类，同样将于20xx年废弃。此外，我们可以提供文档，对照显示新旧的差异，帮助人们在“截断日期”之前，逐步地转换模板。

这就是我提案的基本内容，也有其他的提案，当然我是有倾向的，我也想知道你们的看法。向后兼容对你来说重要吗？你是更倾向于使用{{}}这样的语法，还是用某种方式把属性名进行编码？有太多选择了。

嗯，现在我们的问题都解决了。没，还有个超大的。

看看双向绑定！

我不知道你注意到没有，整篇文章连一个双向绑定的例子都没有。其实，我上面解释过的所有语法都不能用于指定各种绑定选项，如：方向性，触发器，防反跳等等。那，怎样绑定一个input元素，把数据推送到模型中呢？怎样绑定一个需要更新模型的自定义Web Component呢？

在Angular 2.0是否需要双向的数据绑定，Angular团队中产生了激烈的辩论。如果你读过公开的设计文档（包括[这篇文档](https://docs.google.com/document/d/1kpuR512G1b0D8egl9245OHaG0cFh0ST0ekhD_g8sxtI/edit?usp=sharing)），或者看过[ngEurope关于Angular 2.0核心的演讲](https://www.youtube.com/watch?v=gNmWybAyBHI&list=UUEGUP3TJJfMsEM_1y8iviSQ)或[Q&A](https://www.youtube.com/watch?v=g-x1QKriY90&list=UUEGUP3TJJfMsEM_1y8iviSQ)，你可能会发现这一点。我强烈支持保留双向数据绑定，在我看来，这是Angular灵魂的一部分。我尚未看到哪个建议能提供一个优雅的替代，在我能提出之前，还是会认同支持保留双向绑定。

你可能想知道这到底为了什么。

我听到过一些有关为数据流执行[DAG](http://en.wikipedia.org/wiki/Directed_acyclic_graph)的解释。这个思路是最近被[ReactJS](http://facebook.github.io/react/)搞得火起来的。但是坦率地说，你不能完全执行它。我只要用一个[事件聚合器](http://martinfowler.com/eaaDev/EventAggregator.html)就足以把它搞挂，这是一个在复合应用中非常常见的模式。我觉得你应当教给人们有关DAG的事情，帮助他们在合适的情况下使用，但不能强求。这会使他们的工作变得困难。

我听说过另外一个论点，主要围绕校验能力的不足，但这不是一个移除双向绑定的理由。你可以很容易在底层放双向绑定功能，把校验系统放在它的上层。

我认为最大问题来自Angular用于实现绑定的脏检测。因为脏检测，你每做一次检测，其实是做了两次。原因在于，如果第一次检测导致了变化，作为一种副作用，它可能导致其他变化。所以，为了确认，你一定还要再检测一次。然后，如果第二次检测之后，又变化了，还得检测第三次……等等。这个事情就称为模型的稳定化。是啊，这是脏检测系统的痛苦，但是移除双向绑定并不能解决这个问题。你还需要移除所有的监控器，这样一个表达式的变化不会导致它们中的任意一个产生变更。很明显，这也就是也需要考虑移除监控器的原因。可是这样也还是不能解决问题，因为一个事件聚合器就能绕过它……坦白地说，有时候你是需要这样的。数据绑定是一个很强大的工具，人也是会犯错的，但我认为我们能解决它。我知道你们中的很多人都可以的。

可能你不同意我的观点，你觉得“good riddance to two-way binding.”，持这种观点的人肯定很多。不过，我怀疑多数Angular，Durandal，Knockout，Ember等框架的用户会认同我。所幸的是，Angular团队在此事上尚未下定决心，他们在尝试考虑所有的可能性。所以，没必要担心。不过，如果你爱双向绑定的话，要来帮我，我觉得，如果Angular团队的其他成员能听到你们有多爱双向绑定的话，就太好了。

另一方面，如果你认为双向绑定是个坏主意，请你帮我们调查替代品。到目前为止，我尚未见到一个差不多好的替代方式，但也许你有比较好的想法呢。如果是这样的话，我请你来跟我们分享一下。如果我们能一起想出一些更好的东西……那就太棒了。

## **路由**

啊！你对Angular 2.0真够感兴趣的。我真不相信你一直看到这里了。多谢！现在我们来讨论路由……

注意：如果你看累了，想要看一段有关路由的视频的话，可以找到[我在ngEurope上关于这个主题的25分钟演讲](https://www.youtube.com/watch?v=h1P_Vh4gSQY&list=UUEGUP3TJJfMsEM_1y8iviSQ)。

#### **基础**

为了让Angular 2.0成为一个能干的框架，它需要有一个强大的路由解决方案。今年早些时候，Brian Ford开始围绕Angular社区内外已有的路由解决方案，进行了大量信息的整合。我们看了已有方案的很大一部分，并且把这些案例的研究与我们从社区收到的请求整合起来。把这些放在一起之后，社区就此文档提供了反馈  
，然后我尝试实现一些东西。几个短的迭代之后，我们觉得我们做了个挺酷的东西。

自然，所有你期望路由处理的基本场景，新的路由处理了……

* 简单的基于JSON的路由配置
* 可选的约定优于配置
* 静态的，参数化的，splat（这个什么意思？）的路由模式
* 查询字符串的支持
* 使用Push State或者Hashchange
* 导航模型（用于生成导航UI）
* 文档标题更新
* 404路由的处理
* 历史的操作
* 更多

#### **子路由**

你可能习惯了有一个路由，并且不得不提前为整个应用配置所有的路由。但是，基于我们的新路由，你拥有更多的灵活性。事实上，每个你导航到的组件都可以有一个路由。我们称之为子路由，它允许你把应用的整个功能区域进行封装。如果你有一个拥有多个团队的大型项目，或者你是个“个体户”，这能够把你的代码库良好地分割，你会喜欢这个特性的。现在你可以把应用的每个部分当做一个小型应用来构建，它们有自己的路由。然后，你只要把他们挂接到主应用上，给组件映射一个相对路径，它就能运行了。如果你想看点有意思的，看看[我演讲](https://www.youtube.com/watch?v=h1P_Vh4gSQY&list=UUEGUP3TJJfMsEM_1y8iviSQ)里面的递归子路由示例。

#### **屏幕激活**

有时候，在导航中你需要对过程有所控制。也许你是从一个带有未保存数据的数据入口屏幕离开，然后需要跟用户确认一下这样行不行。也许你在实现一个向导，在显示到第三步之前，需要确保数据存在于前两步，然后做相应的重定向。为了处理这类场景，我们实现了一个显式的导航生命周期，你的控制器可以选择对导航过程作控制。这里是一个生命周期的钩子列表：

* canActivate，允许/阻止导航到新控制器
* activate，对成功导航到新控制器的响应
* canDeactivate，允许/阻止从旧控制器离开
* deactivate，对成功离开旧控制器的响应

can\*回调通过返回布尔值的方式，让你控制导航。你也可以为这个值返回一个Promise，这可以让你进行异步的操作，作为过程的一部分。此外，你可以返回一个特定的NavigationCommand（比如Redirect），它能让你对过程作底层控制。

如你所愿，所有这些都可以无缝与子路由协作。

#### **设计**

我们努力让设计尽可能可插拔。所有处理导航请求的逻辑都基于管道架构来建立，这意味着你可以向管道中加入自己的步骤，甚至移除一些我们默认的步骤。例如，如果你不喜欢屏幕激活行为，你可以把它干掉。在管道中，建模了四个步骤，每个代表一个生命周期阶段。管道的另外一个重要特性是，每一步都是异步的。所以，如果你需要发起一个服务端请求来对一个用户进行身份验证，或者为一个控制器加载数据，你可以在管道中做这个，并且把这个代码从控制器中移除。

#### **评注**

我很难对路由表示中立，因为是我做的实现。我认为这对于一个新路由而言，是良好的起步。肯定还有缺失的功能，但我认为高层次的设计是非常强大的。

作为奖励，我们也将把它移植回Angular 1.3上。