

 Open

ng-book 2 总目录 #43



kittencup opened this issue
about 1 month ago



ng-book 2

该书是速翻，现版本为r16，还未进行校验，如果发现问题直接留言，angular2未正式发布，所以本书在不断修正和更新中。

- 编写你的第一个Angular2 Web应用


- 简单的Reddit克隆
- 快速入门
- 我们的第一个Typescript
- 运行应用
- 将数据添加到组件
- 使用数组
- 扩大我们的应用
- 渲染多行
- 添加新文章
- 收尾
- 完整代码
- 结束语
- 寻求帮助

- TypeScript

- Angular2是建立在TypeScript上
- TypeScript提供了什么特性?
- 类型
- 内置类型
- 类
- 工具
- 结束语

- Angular 2 如何运作
 - 应用
 - 产品模型
 - 组件
 - 组件装饰器
 - [ProductsList组件]
 - ProductRow组件
 - ProductImage组件
 - PriceDisplay组件
 - ProductDepartment组件
 - 完成的项目
 - 数据架构上的信息
- 内置组件
 - 介绍
 - NgIf
 - NgSwitch
 - NgStyle
 - NgClass
 - NgFor
 - NgNonBindable
 - 结论
- Angular 2 中的表单
 - 表单是重要的,表单是复杂的
 - Control和ControlGroup
 - 我们的第一个表单
 - 使用FormBuilder
 - 添加验证
 - 观察变化
 - ngModel
 - 结束语
- 基于Observables的数据架构 第一部分:服务
 - Observables和RxJS

- 聊天应用概述
- 实现模型
- 实现UserService
- MessagesService
- ThreadsService
- 数据模型的总结
- 基于Observables的数据架构 - 第2部分:视图组件
 - 建立我们的视图：顶层组件ChatApp
 - ChatThreads组件
 - 单个ChatThread组件
 - ChatWindow组件
 - ChatMessage组件
 - ChatNavBar组件
 - 总结
 - 下一步
- HTTP
 - 介绍
 - 使用 angular2/http
 - 一个基本的请求
 - 编写一个YouTubeSearchComponent
 - angular2 http API

 kittencup added the **ng-book 2** label about 1 month ago

kittencup **referenced** this issue from another issue

#42 HTTP

kittencup **referenced** this issue from another issue

#41 基于Observables的数据架构 - 第2部分:视图组件



Open

编写你的第一个Angular2 Web应用 #24



kittencup opened this issue
2 months ago



ng-book 2

该issue关闭讨论，如有问题请去 [#43](#) 提问

目录

- [简单的Reddit克隆](#)
- [快速入门](#)
- [我们的第一个Typescript](#)
- [运行应用](#)
- [将数据添加到组件](#)
- [使用数组](#)
- [扩大我们的应用](#)
- [渲染多行](#)
- [添加新文章](#)
- [收尾](#)
- [完整代码](#)
- [结束语](#)
- [寻求帮助](#)



kittencup added the [求解答](#) label 2 months ago



kittencup
2 months ago

简单的Reddit克隆

在本章中，我们要构建一个应用程序，允许用户发布了一篇文章（带有标题和URL）并且

可以给文章投票

你可以认为这个应用是一个站点的初期，像[Reddit](#)或[Product Hunt](#)，

在这个简单的应用中，我们将一起涉及到Angular 2大部分内容。包括：

- 构建自定义组件
- 从表单中接受用户输入
- 将对象列表渲染到视图
- 拦截用户点击并处理他们

当你完成这一章你会掌握如何构建基本的Angular 2应用程序。

我们的应用将会和下面的截图看起来差不多

image

首先，用户提交新的链接后，用户将能够对每篇内容进行upvote和downvote。每一个链接都会有一个分数，可以投票给我们发现有用的链接。

image

在这个项目和整本书中，我们使用TypeScript来编写，TypeScript是JavaScript的ES6的超集，增加了数据类型。在本章中我们不会深度讨论TypeScript，但如果你熟悉ES5/ES6,那应该没有任何问题。

我们会在下一章深度了解TypeScript。如果你遇到了一些新的语法无需太担心。



kittencup

2 months ago

快速入门

TypeScript

开始使用TypeScript前，你需要先安装Node.js。有很多不同的方法安装Node.js，请参阅Node.js的网站<https://nodejs.org/download>：

我必须用TypeScript吗？不，你可以不使用TypeScript来编写Angular 2,ng2有ES5

API,但是通常每个人都会使用TypeScript来编写anuglar2,

这本书中我们将使用Typescript, 因为他编写Angular2更为简单。也就是说, 它没有严格要求你必须使用TypeScript。

一旦你有了Node.js, 下一步就是安装TypeScript。确保您安装的版本至少是1.7或更高版本。运行下面的命令, 安装1.5版本:

```
$ npm install -g 'TypeScript@^1.7.3'
```

npm是node.js安装中的一部分, 如果您在系统上没有npm, 确保您使用Node.js的安装程序包含它。

window用户:在这本书我们在命令行中将使用Linux/Mac风格的命令, 我们强烈建议您安装cygwin2,因为它会让你可以运行这本书中的非window命令。

示例项目

现在, 你的环境已准备好了, 让我们开始写第一个Angular2应用!

打开随这本书下载的代码并解压。在你的命令行中, 通过cd进入 `first_app/angular2-reddit-base` 目录

```
$ cd first_app/angular2-reddit-base
```

如果你不熟悉cd命令, 它表示“更改目录”。你可以在mac上进行以下尝试:

1. 打开 /Applications/Utilities/Terminal.app
2. 输入cd
3. 在mac的finder中,将first_app/angular2-reddit-base文件夹拖拽到命令行窗口中
4. 按下回车, 你会切换到该目录下, 你可以继续下一步操作了

首先让我们先使用npm安装所有依赖

```
$ npm install
```

在项目的根目录下创建一个新的index.html文件, 并添加一些基本HTML结构:

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Simple Reddit</title>
  </head>
  <body>
  </body>
</html>
```

你的 angular2-reddit-base 目录看上去应该是这样子的

```
|-- README.md // A helpful readme
|-- index.html // Your index file
|-- index-full.html // Sample index file
|-- node_modules/ // installed dependencies
|-- package.json // npm configuration
|-- resources/ // images etc.
|-- styles.css // stylesheet
|-- tsconfig.json // compiler configuration
|-- tslint.json // code-style guidelines
```

Angular 2本身是一个JavaScript文件。所以我们需要一个script标签来引入它。并且还需引入一些Angular/TypeScript依赖的文件：

Angular的依赖

你并不需要为了使用Angular 2而严格地理解这些依赖，但你需要导入这些依赖，如果你对依赖并不感兴趣，请跳过本章节，但要确保你复制并粘贴这些脚本标签。

Angular 2依赖于这四个库：

- es6-shim - (为了旧浏览器)
- angular2-polyfills
- SystemJS
- RxJS

在你的 `<head>` 中添加这些标签

```
<script src="node_modules/es6-shim/es6-shim.js"></script>
<script src="node_modules/angular2/bundles/angular2-polyfills.js">
```

```
</script>  
<script src="node_modules/systemjs/dist/system.src.js"></script>  
<script src="node_modules/rxjs/bundles/Rx.js"></script>  
<script src="node_modules/angular2/bundles/angular2.dev.js"></script>
```

请注意，我们直接从 `node_modules` 目录中加载这些 `.js` 文件，`node_modules` 目录会在运行 `npm install` 时被创建，如果你没有 `node_modules` 目录，请确保你是在 `angular2-reddit-base` 目录下输入的 `npm install`

ES6 Shim

ES Shim 为旧版的Javascript引擎提供了ECMAScript 6的行为，该Shim对于较新新版本的Safari,Chrome等并不严格需要，但是对于旧版本的IE是需要的。

什么是Shim? 也许你听说过shims和polyfills,但你不知道它们是什么。
Shim是代码，它有助于适应跨浏览器之间的一种标准化的行为。

例如，看看这个[ES6兼容性表](#).不是每个浏览器的每一个功能都完全兼容。通过使用不同的shim，我们能够得到在不同浏览器（和环境）的标准的行为。

参见：[shim和polyfill之间的区别是什么？](#)

Angular 2 Polyfill

像ES6 Shim, angular2-polyfills提供跨浏览器的一些基本的标准化。

angular2-polyfills包含的代码专门用于zone,promise和reflection，如果你不知道这些东西是什么，你也不必担心。

SystemJS

SystemJS是一个模块加载器。它帮助我们创建模块和解决模块之间依赖，模块加载在浏览器端的JavaScript是出奇的复杂，SystemJS使得过程变得更加容易。

RxJS

RxJS是一个库用于在Javascript中进行反应式编程，一般来说,RxJS给了我们使用Observables的工具，用于发出的数据流。Angular 在许多地方使用了Observables，如在处理异步代码(例如: HTTP请求)

我们会在本书的RxJS这章讨论更多关于RxJS的内容，虽然在本章中它不是严格需要的， 8

但值得一提的，你会在项目中经常使用它。

加载所有依赖

现在我们已经添加了所有的依赖，我们的index.html看起来应该是这样的

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Simple Reddit</title>
    <!-- Libraries -->
    <script src="node_modules/es6-shim/es6-shim.js"></script>
    <script src="node_modules/angular2/bundles/angular2-polyfills.js">
</script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js">
</script>
  </head>
  <body>
</body>
</html>
```


添加CSS

我们也想添加一些CSS样式，使我们的应用不是完全无样式。让我们导入两个样式表：

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Simple Reddit</title>
    <!-- Libraries -->
    <script src="node_modules/es6-shim/es6-shim.js"></script>
    <script src="node_modules/angular2/bundles/angular2-polyfills.js">
</script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js">
</script>
    <!-- Stylesheet -->
    <link rel="stylesheet" type="text/css"
      href="resources/vendor/semantic.min.css">
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
```

```
</body>
</html>
```

对于这个项目，我们将要使用[Semantic-UI](#), Semantic-UI是一个CSS框架，类似于Foundation或Twitter Bootstrap,我们已经在示例代码中下载了因此你需要做的就是添加link标记。

 **kittencup** locked this issue 2 months ago



kittencup

2 months ago

我们的第一个TypeScript

现在创建我们第一个TypeScript文件，在同级目录下添加一个叫app.ts的文件，并添加些代码：

注意 TypeScript文件的后缀为.ts而不是.js，这里的问题是，我们的浏览器并不知道怎么读取ts文件，所以后面我们需要将ts文件编译成js文件

code/first_app/hello-world/app.ts

```
import { bootstrap } from "angular2/platform/browser";
import { Component } from "angular2/core";
@Component({
  selector: 'hello-world',
  template: `
    <div>
      Hello world
    </div>
  `
})
class HelloWorld { }
bootstrap(HelloWorld);
```

这段代码可能看起来完全看不懂，但别担心。我们会一步步解释。

`import` 语句定义了，在我们代码中需要用到的模块，在这里我们导入了2个模块，`Component` 和 `Bootstrap`。

我们从 `angular2/core` 模块中导入了 `Component` 模块, `angular/core` 这部分告诉我们程序在哪里可以找到我们正在寻找的依赖。

同样我们从模块 `angular2/platform/browser` 中导入 `bootstrap` 模块

注意, 这个import语句的结构格式是 `import { things } from wherever. { things }` 这部分是只需要导入的模块, 这个是一个ES6的特性, 我们将在下一章讨论更多。

import的想法是类似于java或者ruby的require,我们只是将这些依赖模块提供给这个文件。

创建一个组件

组件是Angular 2其背后一个很大的想法之一。

在我们的Angular应用中编写HTML标签来成为我们的交互式应用程序、但是浏览器只认识那些内置的标签, 如 `<select>`、`<form>` 或 `<video>` 等。但是, 如果我们想教浏览器认识新的标签呢, 如 `<weather>` 标签应该怎么显示天气, `<login>` 标签应该如果处理登录等。

这背后就是组件的想法。我们教浏览器来认识新功能的新标签。

如果你有用过Angular 1,组件就是新版本的指令

来创建我们第一个组件.当我们编写完这个组件后, 我们可以在HTML中这样使用它

```
<hello-world></hello-world>
```

那么, 我们如何定义一个新的组件? 一个基本组件有二个部分:

- 一个 `Component` 注解
- 一个定义组件的类

如果你已经有一段时间的JavaScript编程经验, 当看到下面的JavaScript会有点奇怪:

```
@Component({  
  // ...  
})
```

这里发生了什么事? 如果你有Java背景, 它看起来你会很熟悉: 他们是注解。

注解会作为元数据添加到您的代码里。当我们在 `HelloWorld` 类上使用 `@Component` 时, 11

我们会“decorating(装饰)” `HelloWorld` 类成为一个组件

我们希望用 `<hello-world>` 标签来使用我们的组件。要做到点，我们将 `@Component` 配置中的 `selector` 属性设置为 `hello-world`

```
@Component({
  selector: 'hello-world'
})
```

如果你熟悉CSS选择器，XPath，jQuery选择器等，你就知道有很多方法来配置一个 `selector`。Angular 2 向 `selector` 添加了自己的特殊混合酱料,在以后,我们将会讨论。现在，只需要知道，在本例中，我们只定义了一个新的标签。

这里的 `selector` 属性表示对应的DOM元素将要被组件使用。这样，在模板中的 `<hello-world></hello-world>` 标签，会使用这个组件类进行编译。

添加模板

我们可以通过 `@Component` 中 `template` 选项来添加模板：

```
@Component({
  selector: 'hello-world',
  template: `
    <div>
      Hello World
    </div>
  `
})
```

请注意，模板字符串定义在我们的反引号（``...``）之间。这是一个新的ES6的功能，可以让我们实现多行字符串。使用反引号里的多行字符串太棒了，使用它能更容易的来构建的模板代码。

我真的应该把模板放在我的代码文件里吗？答案是，这取决于你。长期以来普遍的理念是，你应该保持你的代码和模板分离。虽然这对一些团队来说可能更容易一些，但对于某些项目来说，它只是增加了很多开销。

当你需要在很多文件之间切换时，它会增加你的开发的开销。我个人而言，如果我的模板是小于一页，我更喜欢有模板和代码一起的。我可以看到逻辑和视图在一起，这很容易理解他们是如何相互作用的

把你的视图与代码内联最大的缺点是，许多IDE还不支持内部字符串语法高亮。我希望我们会看到更多的IDE支持语法高亮的HTML模板。

引导我们的应用

我们文件的最后行 `bootstrap(HelloWorld)` ,将启动我们的应用，第一个参数表明，我们的应用的“主(main)”组件是 `HelloWorld` 。

一旦被引导，在 `index.html` 文件中 `<hello-world></hello-world>` 会被我们的组件渲染，让我们尝试一下！

加载我们的应用

要运行我们的应用程序，我们需要做2件事情：

1. 需要告诉我们的HTML文件导入app文件
2. 需要在body中使用 `<hello-world>` 组件

将以下添加到body部分：

```
<!doctype html>
<html>
  <head>
    <title>First App - Hello world</title>
    <!-- Libraries -->
    <script src="node_modules/es6-shim/es6-shim.js"></script>
    <script src="node_modules/angular2/bundles/angular2-polyfills.js">
</script> <script src="node_modules/systemjs/dist/system.src.js">
</script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <!-- Stylesheet -->
    <link rel="stylesheet" type="text/css"
href="resources/vendor/semantic.min.css">
    <link rel="stylesheet" type="text/css" href="styles.css"> </head>
    <body>
      <script>
        System.config({
          packages: {
            app: {
              format: 'register',
              defaultExtension: 'js'
            }
          }
        });
```

```
System.import('app.js')
    .then(null, console.error.bind(console));
</script>
    <hello-world></hello-world>
</body>
</html>
```

在 `script` 标签中, 我们配置模块加载器 `System.js`, 在这里重要的是要了解这行:

```
System.import('app.js')
```

这行告诉 `System.js` 要加载 `app.js` 作为我们的主要入口。不过还有一个问题: 我们还没有一个 `app.js` 文件!(我们的文件是 `app.ts` TypeScript 文件。)



kittencup

2 months ago

运行应用

编译TypeScript代码为.js

我们使用TypeScript编写我们的应用, 我们有一个 `app.ts` 文件, 下一步是将这个文件编译成Javascript, 让我们的浏览器可以理解。

为了做到这一点, 让我们运行TypeScript编译器的命令行, 称为 `tsc`:

```
tsc
```

如果你得到一个没有错误信息的提示, 这意味着编译成功, 我们现在在同一目录下应该有 `app.js` 文件

```
ls app.js
# app.js should exist
```

故障排除:

也许你会收到以下消息: `tsc: command not found`, 这意味着, `tsc` 未安装或不在 `PATH`, 尝试使用路径在 `node_modules` 目录中的 `tsc` 二进

```
制: ./node_modules/.bin/tsc
```

在这种情况下你不需要指定，编译器tsc任何参数，因为它能编译当前目录中的所有ts文件。如果你没有得到一个app.js文件，用cd来更改目录,确保目录和你的app.ts文件在同一目录

当你运行TSC你也可能会得到一个错误。例如，它可能表示 app.ts(2,1): error TS2304: Cannot find name or app.ts(12,1): error TS1068: Unexpected token。

在本例中，当错误时编译器会给你一些提示，app.ts(12,1):表示错误在app.ts第12行。您还可以在网上搜索错误代码，可能会有如何解决这个错误的帮助。

使用npm

如果你在tsc命令上面工作，你也可以使用npm来编译文件，在package.json里包含了一些简单的代码，我们定义了一些快捷命令来帮助你编译

尝试运行:

```
npm run tsc // compiles TypeScript code once and exits
npm run tsc:w // watches for changes and compiles on change
```

应用的server

我们来测试应用还有一个步骤。我们需要一个webserver来运行测试应用。

如果你早期使用npm安装，你已经有了一个安装在本地的webserver,要运行它，只需运行下面命令

```
npm run serve
```

打开你得浏览器访问<http://localhost:8080>

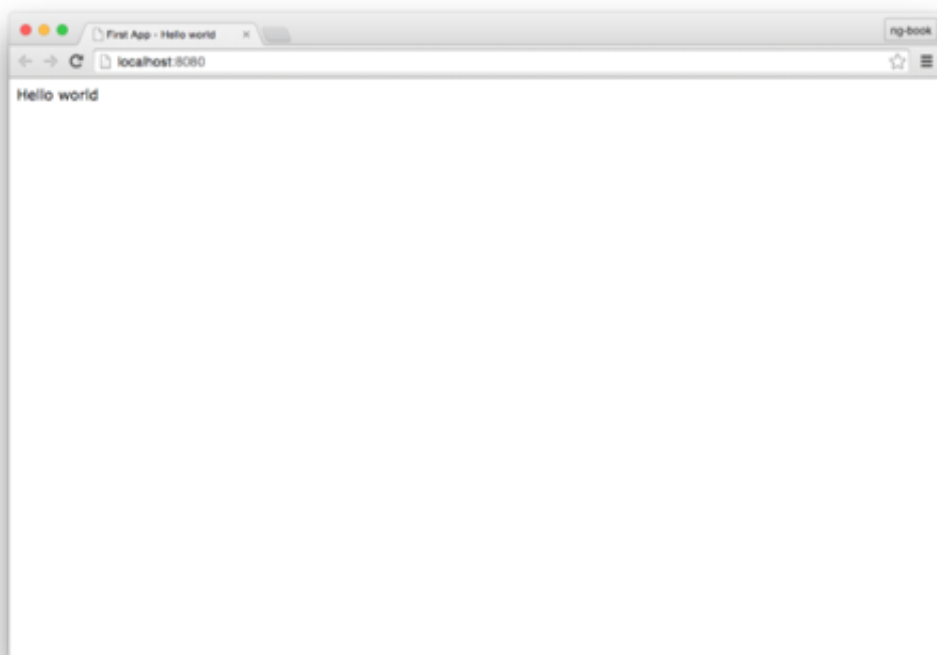
我为什么需要一个webserver? 如果你之前开发的JavaScript应用程序，您可能知道，有时你只需打开index.html文件，就能运行在你的浏览器。但这样对我们来说没有用，因为我们使用SystemJs。

当你直接打开index.html文件，你的浏览器将使用file:///URL。由于安全限制，当file:///protocol 时你的浏览器将不允许Ajax请求发生（这是一件好事，因为否则

JavaScript可以读任何在您的系统上的文件做一些恶意的（事）。

所以我们运行一个简单的本地网络服务器提供文件系统。这对于测试真的很方便，并不需要你知道如何部署生产应用。

如果一切运行正常，你应该看到以下内容：



如果无法运行此应用程序，你可以有几件事情尝试：

- 请确保您的app.js文件是从TypeScript编译器tsc创建的
- 请确保您的网络服务器开始和app.js文件在同一目录
- 请确保您的index.html文件符合我们上面的代码示例
- 尝试在Chrome中打开该网页，点击右键，并选择“检查元素”。然后点击“控制台”选项卡，并检查错误。
- 如果一切都失败了，加入[Gitter](#)聊天室来提问

任何改变自动编译

我们将对我们的应用程序代码进行大量的修改。我们可以利用--watch选项，不必每次都运行tsc生成新的js代码。

--watch选项将告诉tsc监视我们的TypeScript文件，文件有任何变化就自动重新编译新的JavaScript：


```
tsc --watch  
message TS6042: Compilation complete. Watching for file changes.
```

其实，这是很常见的，我们已经为其创建了一个快捷方式

- 1.文件改变后重新编译
- 2.重载你的开发服务器

```
npm run go
```

现在，您可以编辑你的代码,变化将会自动在浏览器中体现出来。



kittencup

2 months ago

将数据添加到组件

我们的组件现在不是很有趣。大多数组件将具有动态数据。

让我们把name作为组件部分的一个新属性。这样，我们可以重用相同的组件，用于不同的输入。

作出以下修改：

```
@Component({  
  selector: 'hello-world',  
  template: `<div>Hello {{ name }}</div>`  
})  
class HelloWorld {  
  name: string;  
  constructor() {  
    this.name = 'Felipe';  
  }  
}
```

在这里我们做了三个改变：

1.name 属性

在 `HelloWorld` 类中添加了一个属性，注意，语法是相对于ES5 JavaScript，在这里 `name: string`；这意味着`name`是属性的名字，`string`表示这个 `name` 是字符串类型的。

属性的类型是由TypeScript提供的特性。这将在我们的 `HelloWorld` 类中设置一个 `name` 属性，编译器可以确保这个 `name` 是一个字符串。

2. 一个 `constructor`

在`HelloWorld`类中我们定义了 `constructor` ,既，当我们实例化这个类时会调用该方法。

在我们的构造函数中，可以通过使用`this.name` 给`name`属性赋值。

当我们写成:

```
constructor() {  
    this.name = 'Felipe';  
}
```

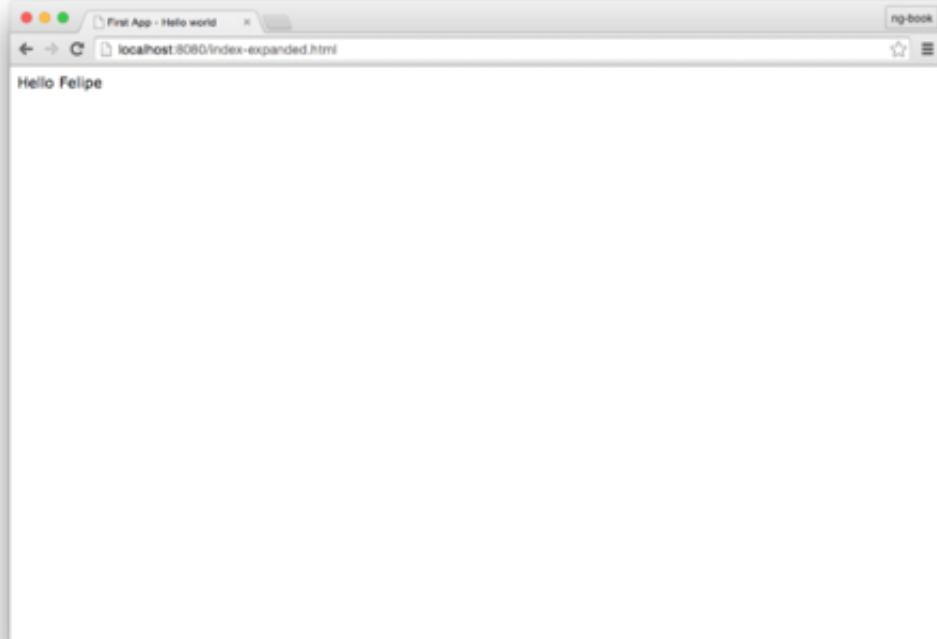
我们可以理解为,每当创建一个新的 `HelloWorld` 时将 `name` 设置为“Felipe”。

3. 模板变量

在视图上我们添加了一个新的语法: `{{ name }}`, 这对大括号叫做模板标记(template-tags), 模板标记之间的任何内容都将被扩展为表达式。在这里，因为组件绑定了我们的视图， `name` 会被渲染成Felipe

试试看

尝试这些更改后，重新加载页面。我们应该看到“Hello Felipe”



kittencup

2 months ago

使用数组

现在有一个 `name` 会说“Hello”，但如果我们有一系列 `name` 都想要说“Hello”呢？

如果之前你用过Angular 1,你会使用`ng-repeat`指令，在Angular 2中，这个相似的指令名为`NgFor`，它的语法有点不同，但它们有相同的目的：用于遍历集合对象。

让我们`app.ts`代码进行如下变化：

```
import { bootstrap } from "angular2/platform/browser";
import { Component } from "angular2/core";
import { NgFor } from "angular2/common";
@Component({
  selector: 'hello-world',
  template: `
    <ul>
      <li *ngFor="#name of names">Hello {{ name }}</li>
    </ul>
  `
})
```

```
class HelloWorld {
  names: string[];
  constructor() {
    this.names = ['Ari', 'Carlos', 'Felipe', 'Nate'];
  }
}
bootstrap(HelloWorld);
```

第一个指出的变化是在我们的 `HelloWorld` 类中又一个新的属性，类型为 `string[]`，这个语法意味着这个 `names` 是一个数组，数组中每一个元素是字符串类型

我们改变类中的 `this.names` 值为 `['Ari', 'Carlos', 'Felipe', 'Nate']`。

接下去改变的是我们的模板，我们现在有一个 `ul` 和一个 `li`，`li` 上有一个属性为 `*ngFor="#name of names"`。`*`和`#`字符可能有点混乱的,让我们将其分解:

`*ngFor` 语法说明我们想要在这个属性上使用`ngFor`指令，`NgFor`类似于一个`for`循环，我们的想法是为集合中的每项元素创建新的DOM元素。

值 `#name of names` 指出，`names`是我们在 `HelloWorld` 里定义的`names`数组，`#name`是`names`里每个元素的引用变量。

该`NgFor`指令会渲染`names`数组中的每一个元素产生一个新的`li`,每个`li`都会产生一个局部的`name`变量，这个变量会替换模板里的`{{ name }}`，渲染到页面

引用变量 `name` 非固定的，我们还可以写成

```
<li *ngFor="#foobar of names">Hello {{ foobar }}</li>
```

但如果相反呢，如果我们这样写会发生什么事：

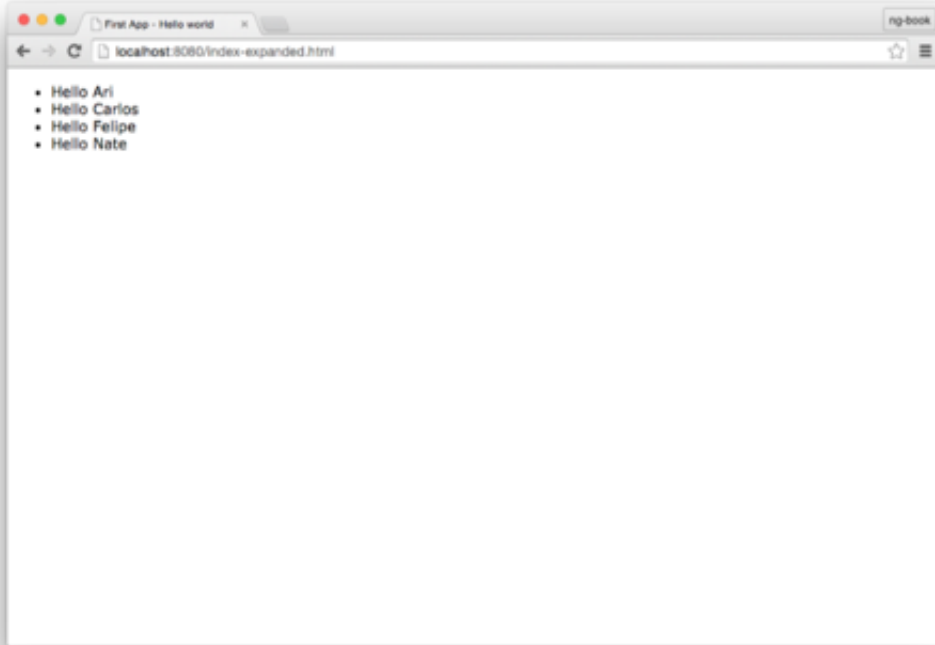
```
<li *ngFor="#name of foobar">Hello {{ name }}</li>
```

我们会得到一个错误，因为 `foobar` 不是我们组件的属性

`ngFor`会重复该`ngFor`附着的这个元素，也就是说，我们把它放在 `li` 标签上，而不是 `ul` 标签，因为我们想要重复列表的 `li` 元素，而不是这个列表 `ul` 本身

如果你感到很抽象，你可以通过直接阅读源代码来了解Angular 核心团队如何编写组件。例如，你可以在[这里找到NgFor指令的源码](#)

当你重新加载页面，你可以看到我们数组中的每一个字符串：



Application with Data



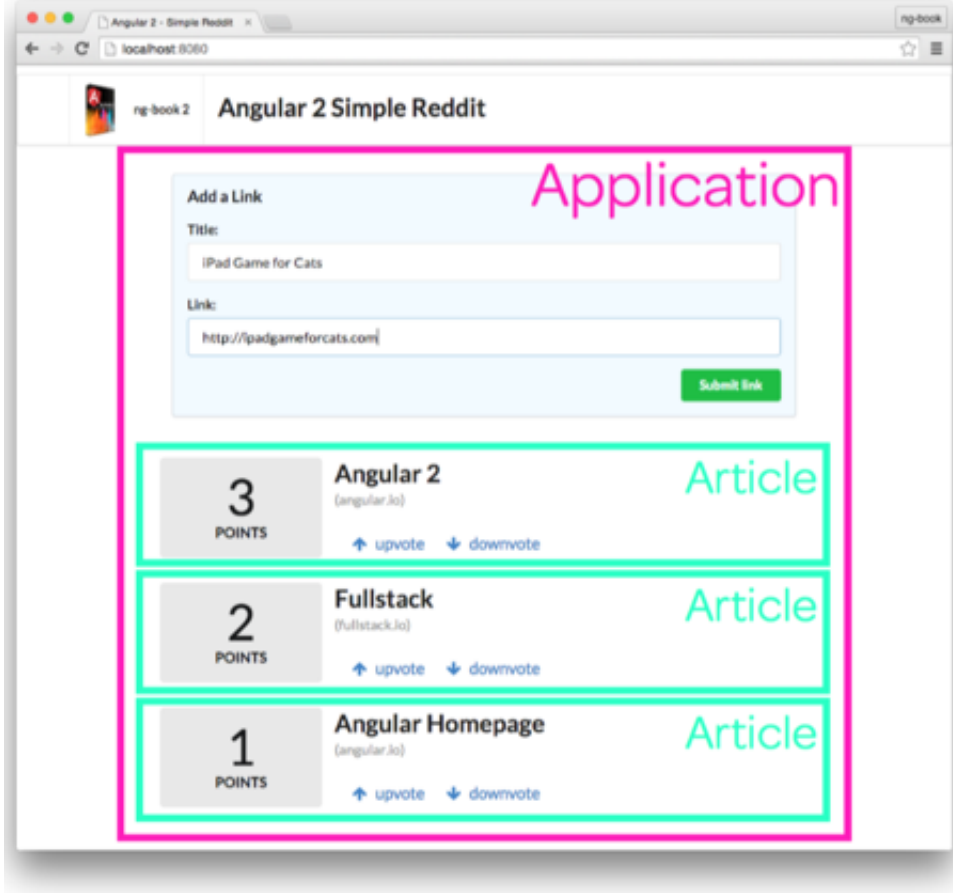
kittencup

2 months ago

扩大我们的应用

现在我们知道如何创建一个组件的基础部分，让我们重新审视我们的Reddit。在我们开始编码之前，先看看我们的应用，一个好的主意是将它分解成一个个单一逻辑组件。

我们将在这个应用程序中，使用两个组件：



Application with Data

- 用于提交新文章的表单将是一个组件（在图片中的红色标记）
- 每一篇文章（绿标记）

在一个大型应用中，用于提交文章的表单很可能会成为独立的组件，然而独立的组件使数据传递更为复杂，在本章中我们将其简化，只有2个组件。

现在，我们只做2个组件，但是在本书的后面章节，我们将学习如何处理更复杂的数据架构

应用组件

让我们开始构建顶层应用组件，这个组件将会

- 1.存储我们目前的文章列表
- 2.包含提交新文章的表单

我们要建立一个组件来代表我们整个应用：一个RedditApp组件。

为了做到这一点，我们将创建一个模板，一个新的组件：

在这个例子中，我们使用了 [Semantic UI CSS](#)，在我们下面的模板里当你看到属性上得class，类似于 `class="ui large form segment"` 这些样式都来自于Semantic。这让我们的应用看起来不错，没有太多额外的标记。

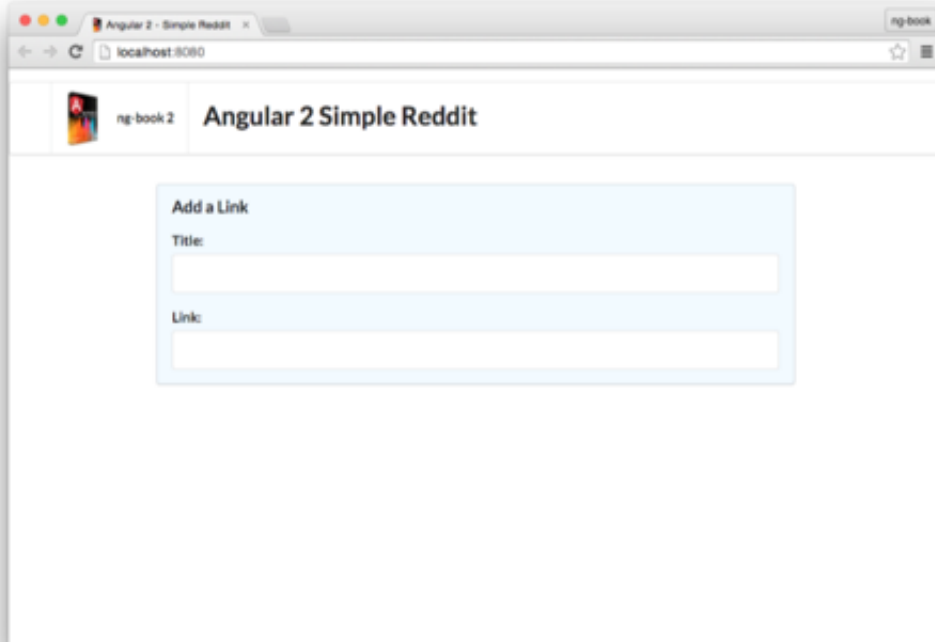
```
import { bootstrap } from 'angular2/platform/browser'; import {
Component } from 'angular2/core';
@Component({
  selector: 'reddit',
  template: `
    <form class="ui large form segment">
      <h3 class="ui header">Add a Link</h3>
<div class="field">
<label for="title">Title:</label> <input name="title">
      </div>
      <div class="field">
<label for="link">Link:</label>
        <input name="link">
      </div>
    </form> `
})
class RedditApp {
  constructor() {
  } }
bootstrap(RedditApp);
```

在这里我们申明了一个RedditApp组件，我们的selector是reddit,意味着这个组件会将 `<reddit></reddit>` 标签解析为组件

我们创建的模板定义了两个input，一个是文章的标题，一个是文章的链接地址

我们需要使用新的RedditApp组件，需要将index.html里的 `<hello-world></hello-world>` 标签来替换为 `<reddit></reddit>` 标签

当您重新加载浏览器，你应该可以看到表单被渲染：



Form

添加交互

现在我们在表单中有input标签了，但我们没有任何的方式来提交数据。让我们通过在表单中添加一个提交按钮来增加一些交互：

```
@Component({
  selector: 'reddit',
  template: `
<form class="ui large form segment"> <h3 class="ui header">Add a
Link</h3>
<div class="field">
<label for="title">Title:</label> <input name="title" #newtitle>
</div>
<div class="field">
<label for="link">Link:</label>
      <input name="link" #newlink>
    </div>
      <button (click)="addArticle(newtitle, newlink)"
        class="ui positive right floated button">
Submit link
      </button>
    </form>
  `
})
class RedditApp {
```



```

constructor() {
}
addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
  console.log(`Adding article title: ${title.value} and link:
${link.value}`);
}
}

```

注意我们已经做了4个变化

1. 创建了一个按钮标签，用于给用户点击
2. 我们创建了一个名为 `addArticle` 的函数，用来定义当我们点击按钮时我们想做的事情
3. 我们在 `<input>` 标签上添加了 `#newtitle` 和 `#newlink` 属性

让我们以相反的顺序来看看每一个步骤：

为input绑定一个局部变量

请注意下面是我们的第一个input

```
<input name="title" #newtitle>
```

`#newtitle` 是新引用语法，这个标记告诉angular将这个 `<input>` 绑定给 `newtitle` 变量，这使得在这个视图中可使用这个变量来访问这个 `input`

`newtitle` 是一个对象，表示该input的DOM元素(具体地说，其类型是 `HTMLInputElement`)，由于 `newtitle` 是一个对象，这意味着我们可以使用 `newtitle.value` 得到表单输入的值。

同样我们为另一个input标签添加一个 `#newlink`，这样我们就可以从中提取出值。

绑定事件

在我们的button按钮上添加了一个(click)属性来定义了点击事件，当button被点击时，会调用addArticle方法，addArticle方法有2个参数newtitle和newlink.这些东西是从哪里来的？

- `addArticle` 是在组件类 `RedditApp` 定义的方法
- `newtitle` 是 `name` 为title的 `<input>` 标签的引用
- `newlink` 是 `name` 为link的 `<input>` 标签的引用

所有在一起：

```
<button (click)="addArticle(newtitle, newlink)" class="ui positive  
right floated button">  
Submit Link  
</button>
```

定义action逻辑

在 `RedditApp` 类中我们定义一个新的函数为 `addArticle`，它接受2个参

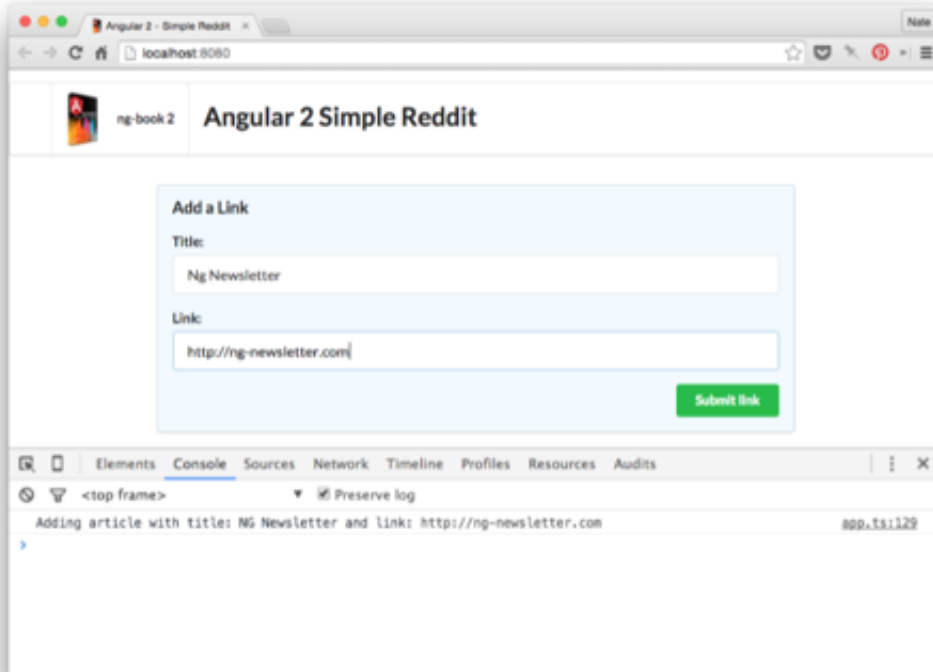
数，`newtitle` 和 `newlink`，

再一次,重要的是要意识到, `newtitle` 和 `newlink` 都是 `HTMLInputElement` 类型的对象,而不是直接输入值, 要从input中获取值, 我们需要调用 `title.value`，现在, 通过 `console.log`打印出这2个参数

```
addArticle(title: HTMLInputElement, link: HTMLInputElement):void {  
    console.log(`Adding article title: ${title.value} and link:  
    ${link.value}`);  
}
```

尝试执行它！

现在, 当你点击提交按钮, 你可以看到, 该消息在控制台上打印出来:



Clicking the Button

添加文章组件

现在有一个发布新文章的组件，但我们没有在任何地方展示新的文章。

因为每一篇文章提交将在页面上显示为一个列表，这是一个新的组件的最佳人选。

让我们创建一个新的组件来显示提交的文章。

为此，我们在同一文件内创建一个新的组件，将下面的代码在RedditApp组件中加上

```
@Component({
  selector: 'reddit-article',
  host: {
    class: 'row'
  },
  template: `
<div class="four wide column center aligned votes">
<div class="ui statistic">
  <div class="value"> {{ votes }}
</div>
  <div class="label">
    Points
  </div>
</div>
```

```

</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ link }}"> {{ title }}
</a>
  <ul class="ui big horizontal list voters">
    <li class="item">
      <a href (click)="voteUp()">
        <i class="arrow up icon"></i> upvote
      </a>
    </li>
    <li class="item">
      <a href (click)="voteDown()">
        <i class="arrow down icon"></i>
        downvote
      </a>
    </li>
  </ul>
</div>
~
}))
class ArticleComponent {
  votes:number;
  title:string;
  link:string;

  constructor() {
    this.votes = 10;
    this.title = 'Angular 2';
    this.link = 'http://angular.io';
  }

  voteUp() {
    this.votes += 1;
  }

  voteDown() {
    this.votes -= 1;
  }
}

```

请注意，我们三个部分来定义这个新组件：

- 通过**@Component**注解来描述组件属性
- 通过**@Component**注解的 `template` 来描述组件视图
- 创建一个组件类(ArticleComponent)来定义我们的组件逻辑

让我们来说说每个部分：

创建reddit-article组件

```
@Component({
  selector: 'reddit-article',
  host: {
    class: 'row'
  },
  },
```

首先，我们通过**@Component**注解来定义新的组件， `selector` 表示将使用 `<reddit-article>` 标签来使用组件(selector就是标签名)

因此，使用该组件的最重要的方法是将下列标记放置在标记中：

```
<reddit-article>
</reddit-article>
```

当页面渲染时，这些标记将留在我们的视图中。

我们希望每个 `reddit-article` 是独立的一行，我们使用Semantic UI,它提供了[CSS class for rows](#)

在Angular 2中，一个组件的host表示该组件元素，你会注意到，将 `host:{class:row}` 传递给我们的 `@Component`，这告诉Angular,我们要在host元素上设置class属性为row

使用host选项是一个比较好的选择,如果不是用host选项，我们需要在父视图中写上

```
<reddit-article class="row">
</reddit-article>
```

通过使用host选项，我们可以从组件中配置host元素。

创建reddit-artcile模板

然后我们通过 `template` 选项来定义模板

```
template: `
  <div class="four wide column center aligned votes">
  <div class="ui statistic">
    <div class="value"> {{ votes }}`
```

```

    </div>
    <div class="label">
      Points
    </div>
  </div>
</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ link }}"> {{ title }}
</a>
<ul class="ui big horizontal list voters">
  <li class="item">
    <a href (click)="voteUp()">
      <i class="arrow up icon"></i> upvote
    </a>
  </li>
  <li class="item">
    <a href (click)="voteDown()">
      <i class="arrow down icon"></i>
      downvote
    </a>
  </li>
</ul>
</div>
`

```

在这里有很多的标签，让我们把它分解：



我们有2列

- 1.投票数在左边
- 2.文章信息在右边

我们在模板中显示votes和title使用模板语法{{ votes }} 和 {{ title }}.这2个值将使用ArticleComponent类中的votes和title属性来渲染

我们也可以将模板语法使用在属性内。如a标签的 href="{{ link }}"，这个href的值会动态的从我们组件类中获取

我们的 `upvote/downvote` 链接也有一个行为，我们使用 `(click)` 事件来绑定 `voteUp()/voteDown()`，当点击该链接时，组件类 `ArticleComponent` 中的 `voteUp()/voteDown()` 方法将被调用

创建reddit-article的ArticleComponent定义类

最后，我们创建ArticleComponent定义类：

```
class ArticleComponent {
  votes:number;
  title:string;
  link:string;

  constructor() {
    this.votes = 10;
    this.title = 'Angular 2';
    this.link = 'http://angular.io';
  }

  voteUp() {
    this.votes += 1;
  }

  voteDown() {
    this.votes -= 1;
  }
}
```

该ArticleComponent类中有3个属性

- votes 一个number类型代表所有upvotes减去downvotes的总和
- title 文章里的标题 string类型
- link 文章的url string类型

在controlstor()我们设置下属性

```
constructor(){
  this.votes = 10;
  this.title = 'Angular 2';
  this.link = 'http://angular.io';
}
```

并且我们对于投票也定义了2个方法，`voteUp` 和 `voteDown`

```
voteUp() {  
    this.votes += 1;  
}  
voteDown() {  
    this.votes -= 1;  
}
```

`voteUp`中我们让 `this.votes` 自加1，在`voteDown`中我们让 `this.votes` 自减1

使用reddit-article组件

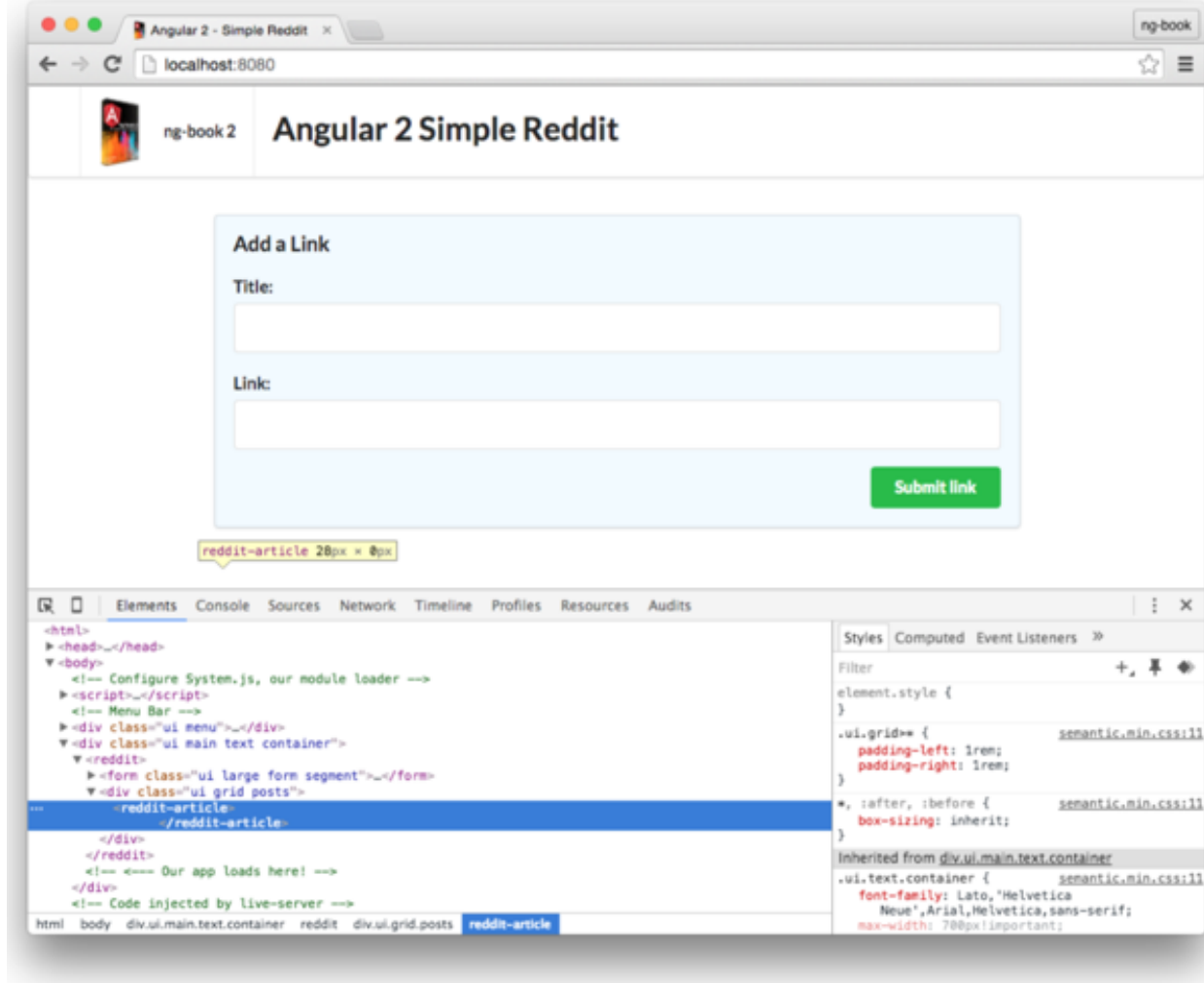
为了使用该组件，使数据可见，在需要使用地方添加上 `<reddit-article></reddit-article>` 标签

在本例中，我们想RedditApp组件中使用这个新组件，让我们改变该组件代码。首先我们需要在RedditApp模板的`..`标签后加上 `<reddit-article>` 标签

```
<button (click)="addArticle(newtitle, newlink)" class="ui positive  
right floated button">  
Submit link  
</button>  
</form>  
<div class="ui grid posts">  
  <reddit-article>  
  </reddit-article>  
</div>  
`
```

让我们重新加载下浏览器，我们会看到 `<reddit-article>` 没有被渲染

每当碰到这种问题，第一件事就是打开你的浏览器的开发者控制台。如果我们检查标签（见下面的截图），我们可以看到，`reddit-article` 标签在我们的网页中，但它并没有被编译成组件。为什么呢？



Unexpanded tag when inspecting the DOM

这标签未被渲染是因为RedditApp组件不知道ArticleComponent组件是什么。

Angular 1注意:如果你使用过Angular 1,你可能会觉得奇怪应用为什么不知道新的reddit-article组件,这是因为在Angular 1中,指令是全局的。然而在Angular 2你需要明确指定组件需要使用的组件是什么。

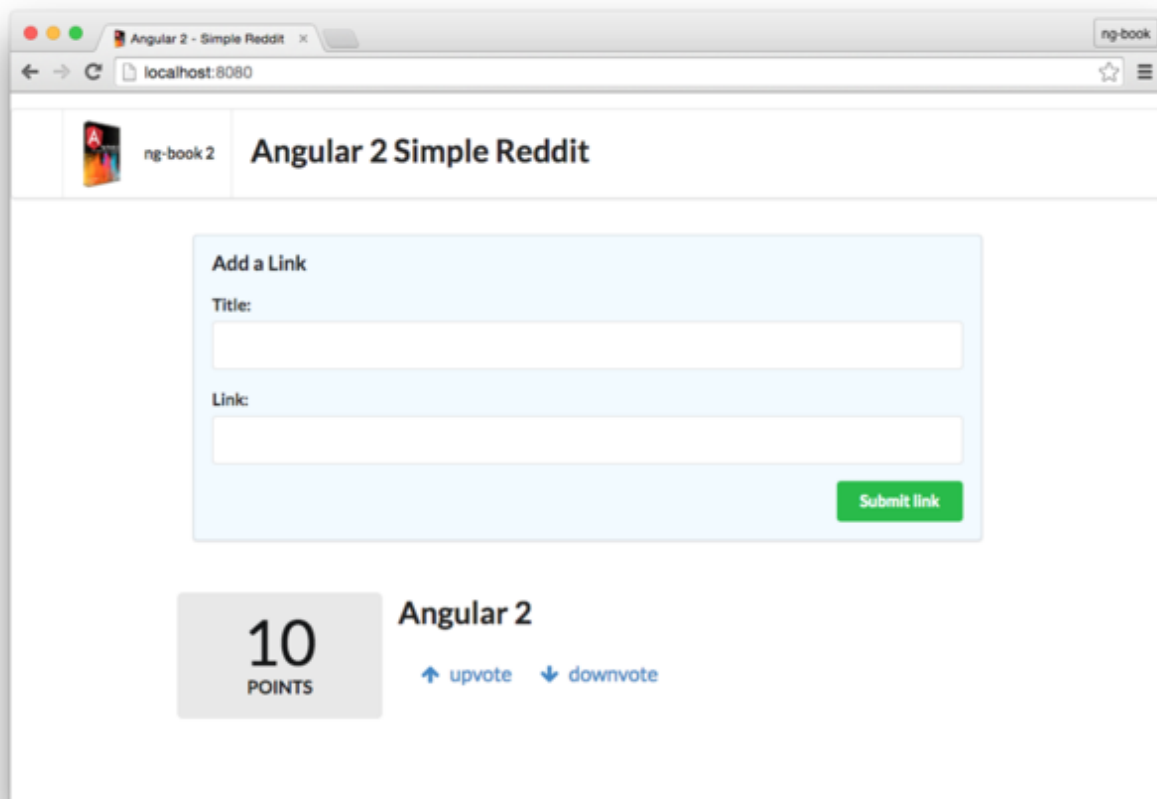
一方面,这需要更多的配置代码,但在另一方面,它非常适合构建可扩展的应用程序,因为这意味着你不必在一个全局命名空间中分享你得指令选择器。

为了告诉RedditApp关于新ArticleComponent组件,我们需要在RedditApp指令中添加属性

```
// for RedditApp
@Component({
  selector: 'reddit',
```

```
directives: [ArticleComponent],
template: `
// ...
```

现在，我们重新加载浏览器，我们应该看到文章被正确渲染了：



Rendered ArticleComponent component

不过，如果你现在点击voteUp 和 voteDown链接，你会看到页面竟然被重新加载

这是因为，javascript的默认情况下，点击事件会冒泡到所有的父组件上，因为点击事件被传播给父元素上，我们的浏览器试图访问空的链接。

要解决这个错误，我们只需使click事件处理程序中返回一个false。这将确保浏览器不会尝试刷新页面。来改变我们的代码：

```
voteUp() {
  this.votes += 1;
  return false;
}
voteDown() {
```

```
this.votes -= 1;
return false;
}
```

现在，如果你点击链接，你会看到投票的增加和减少。



kittencup

2 months ago

渲染多行

现在我们在这个页面只有一篇文章，没有办法渲染更多文章，除非我们创建新的 `<reddit-article>` 标签，即使我们这样做，所有的文章将会有相同的内容。

创建Article类

一个较好的做法是，当编写Angular2代码时试图从你的组件代码中独立出你的数据结构，为了实现这点，做任何进一步的更改组件之前，让我们创建将代表文章的数据结构，在 `ArticleComponent` 组件代码前添加下面代码

```
class Article {
  title: string;
  link: string;
  votes: number;
  constructor(title, link) {
    this.title = title;
    this.link = link;
    this.votes = 0;
  }
}
```

在这里我们创建一个表示Article的类，注意这只是一个普通的类，并不是一个组件，在MVC模式中它属于model

每篇文章有一个title，link和一个总的votes，当我们创建一个新的文章时，我们需要title和link,我们还假设默认的votes是0

现在在ArticleComponent代码中使用我们新的Article类，而不是直接在ArticleComponent组件存储的article的内容，

```

class ArticleComponent {
  article: Article;
  constructor() {
    this.article = new Article('Angular 2', 'http://angular.io', 10);
  }
  voteUp(): boolean {
    this.article.votes += 1;
    return false;
  }
  voteDown(): boolean {
    this.article.votes -= 1;
    return false;
  }
}

```

注意：现在在组件上已经不直接保存我们的title,link和votes，而是保存一个article的引用

当涉及到voteUp(和voteDown),我们不该增减组件上得vote属性，而是应该增减article上得vote属性

这个重构带来了另一个变化：我们需要更新我们的视图，从正确的位置获得模板变量。为了做到这一点，我们需要改变我们的模板标签。也就是说，在之前使用{{votes}}，我们需要将其更改为{{article.votes}}：

```

template: `
<div class="four wide column center aligned votes">
  <div class="ui statistic">
    <div class="value"> {{ article.votes }}
    </div>
    <div class="label">
      Points
    </div>
  </div>
</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ article.link }}"> {{
article.title }}
  </a>
  <ul class="ui big horizontal list voters">
    <li class="item">
      <a href (click)="voteUp()">
        <i class="arrow up icon"></i> upvote
      </a></li>
    <li class="item">
      <a href (click)="voteDown()">

```

```

        <i class="arrow down icon"></i>
        downvote
      </a></li>
    </ul>
  </div>
`

```

如果你重新加载浏览器，你会看到和前面一样的显示效果。

这是不错的，但在我们的代码的东西仍然是一个小问题：在我们的组件中我们的 `voteUp/voteDown` 方法会直接修改 `article` 内部的属性

问题是，我们的 `ArticleComponent` 组件知道太多关于 `Article` 类的内部。为了解决这个问题，让我们为 `Article` 也添加相应方法，`ArticleComponent` 也要做出相应修改：

```

class Article { title: string; link: string; votes: number;
  constructor(title: string, link: string, votes?: number) {
    this.title = title;
    this.link = link;
    this.votes = votes || 0;
  }
  voteUp(): void {
    this.votes += 1;
  }
  voteDown(): void {
    this.votes -= 1;
  }
}

```

然后我们改变 `ArticleComponent` 来调用这些方法：

```

class ArticleComponent {
  article: Article;
  voteUp(): boolean {
    this.article.voteUp();
    return false;
  }
  voteDown(): boolean {
    this.article.voteDown();
    return false;
  }
}

```

辑到我们的model内，这里的对应MVC的指导方针是[Fat Models, Skinny Controllers](#)，我们的想法是，我们要把大部分的逻辑转移到我们的model中，使我们的组件尽可能地完成最少的工作。

当我们重新加载浏览器后，你会注意到所有的显示都是一样的，但我们现在有更清晰的代码。

存储多个文章

让我们写一个允许我们有多篇文章的代码。

改变RedditApp的属性，创建一个articles集合

```
class RedditApp {
  articles: Article[];
  constructor() {
    this.articles = [
      new Article('Angular 2', 'http://angular.io', 3),
      new Article('Fullstack', 'http://fullstack.io', 2),
      new Article('Angular Homepage', 'http://angular.io', 1),
    ];
  }
  addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
```

注意我们RedditApp的这行

```
articles: Article[];
```

如果你不用TypeScript，`Article[]` 你可能看起来是有点奇怪。这种模式就是泛型，它的概念出现在Java,这个想法是，该集合中元素的类型必须是Article，该数组是一个集合，将只持有文章类型的对象。

我们可以在构造函数设置this.articles这个列表：

```
constructor() {
  this.articles = [
    new Article('Angular 2', 'http://angular.io', 3),
    new Article('Fullstack', 'http://fullstack.io', 2),
    new Article('Angular Homepage', 'http://angular.io', 1),
  ];
}
```

配置ArticleComponent组件的inputs属性

现在,我们已经创建了一个article model,我们怎样才能将他们交给ArticleComponent组件用呢?

在这里,我们引入了一个新的组件属性称为inputs。我们可以配置组件的inputs属性,它会接收从父传递进来数据。

以前我们的ArticleComponent组件类定义成这样的:

```
class ArticleComponent {
  article: Article;
  constructor() {
    this.article = new Article('Angular 2', 'http://angular.io');
  }
}
```

这里的问题是,我们已经将特定文章硬编码在构造函数中,制作组件的要点不仅是封装,而且还具有可重用性。

我们真正喜欢做的是配置我们想要显示的文章。如果,例如,我们两篇文章,文章1和文章2,我们希望能够通过article作为一个"参数"来传递给reddit-article组件:

```
<reddit-article [article]="article1"></reddit-article>
<reddit-article [article]="article2"></reddit-article>
```

Angular 允许我们这样做,通过使用 `Component` 的 `inputs` 选项

```
@Component({
  selector: 'reddit-article',
  inputs: ['article'],
  // ... same
})
class ArticleComponent {
  article: Article;
  //...
```

现在如果我们有一篇文章在变量myArticle里,我们可以在ArticleComponent的view里这样写

```
<reddit-article [article]="myArticle"></reddit-article>
```

请注意这里的语法：把在input的名称放在[]内，像这样：[article]属性的值就是我们要传递给该input的内容

然后，这是很重要的，在ArticleComponent实例上的 `this.article` 将会被设置为 `myArticle` ,你可以认为`myArticle`作为参数传递到你的组件内

注意 `inputs`是一个数组，这是因为你可以指定一个组件有许多`inputs`。

所以我们ArticleComponent完整的代码看起来像这样：

```
@Component({
  selector: 'reddit-article',
  inputs: ['article'],
  host: {
    class: 'row'
  },
  template: `
<div class="four wide column center aligned votes">
  <div class="ui statistic">
    <div class="value"> {{ article.votes }}
  </div>
    <div class="label"> Points
  </div>
  </div>
</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ article.link }}"> {{
article.title }}
  </a>
  <ul class="ui big horizontal list voters">
    <li class="item">
      <a href (click)="voteUp()">
        <i class="arrow up icon"></i>
        upvote
      </a></li>
    <li class="item">
      <a href (click)="voteDown()">
        <i class="arrow down icon"></i>
        downvote
      </a></li>
  </ul>
</div>
`
})
```



```

}))
class ArticleComponent {
    article:Article;

    voteUp():boolean {
        this.article.voteUp();
        return false;
    }

    voteDown():boolean {
        this.article.voteDown();
        return false;
    }
}

```

渲染文章列表

早些时候我们配置RedditApp来存储articles数组，现在来配置RedditApp来渲染所有的文章。要做到不只有一个 `<reddit-article>` 标签的话，将使用 `ngFor` 指令来遍历文章列表，并为每一篇文章渲染一个reddit-article

添加这些到RedditApp **@Component**的template内的 `</form>` 后，

```

Submit link
</button>
</form>
<!-- start adding here -->
<div class="ui grid posts">
    <reddit-article
        *ngFor="#article of articles"
        [article]="article">
    </reddit-article>
</div>
<!-- end adding here -->

```

还记得我们在前面章节使用 `ngFor` 指令来渲染name列表吗？嗯，这也适用于渲染多个组件。

`*ngFor="#article of articles"` 语法将通过迭代articles并创建局部article变量。

我们使用 `[inputName]="inputValue"` 表达式来指定组件需要的 `article input`。在这里，我们可以通过`ngFor`，将局部变量`article`设置给组件的 `article input`

我意识到，我们在前面的代码段中多次使用到article变量。如果我们重命名ngFor创建的临时变量名为foobar,你可能觉得更清楚。

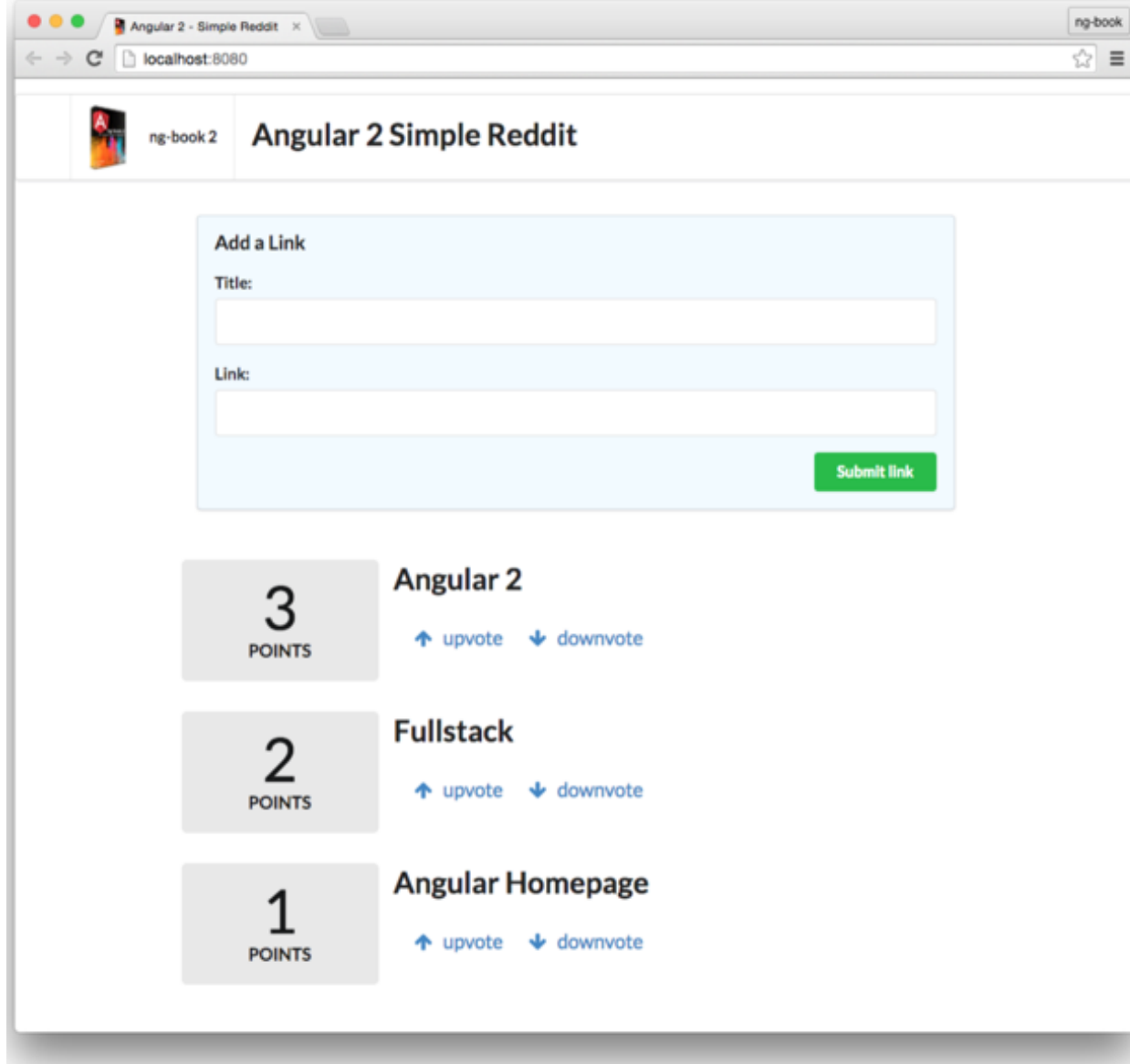
```
<reddit-article
  *ngFor="#foobar of articles"
  [article]="foobar">
</reddit-article>
```

在这里我们有3个变量

1. articles 是保存Articles的数组，定义在RedditApp组件里
2. foobar是articles中的每个元素，由NgFor定义
3. article是在ArticleComponent中定义的input字段名

基本上，NgFor产生一个临时变量foobar，然后我们传递它到reddit-article内

如果你现在重新加载你的浏览器，你可以看到所有文章将被渲染：



🏷 **kittencup** added the 解答中 label 2 months ago

🏷 **kittencup** removed the 求解答 label 2 months ago



kittencup

2 months ago

添加新文章

现在我们需要改变 `addArticle`，为了当按下按钮时候添加一篇新的文章。改变后的 `addArticle` 方法如下

```

addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
  console.log(`Adding article title: ${title.value} and link:
${link.value}`);
  this.articles.push(new Article(title.value, link.value, 0));
  title.value = '';
  link.value = '';
}

```

这将：

- 通过提交过来的title和value创建一个新的Article实例
- 将新的article添加到Articles中
- 清空input的值

我们如何清除input字段值？好吧，如果你还记得，title和link是HTMLInputElement对象。这意味着我们可以设置它们的属性。当我们改变属性值，input标签在我们页面上就更改了

如果你点击submit添加一篇新的article，你会在列表中看到这篇新加的文章



kittencup

2 months ago

收尾

让我们添加一个功能，显示用户点击该链接当会被重定向到的域名

添加domain方法到Article类：

```

domain(): string {
  try {
    const link: string = this.link.split('://')[1];
    return link.split('/')[0];
  } catch (err) {
    return null;
  }
}

```

将其添加到 ArticleComponent 模板里

```

<div class="twelve wide column">
  <a class="ui large header" href="{{ article.link }}">
    {{ article.title }}
  </a>
  <!-- right here -->
<div class="meta">({{ article.domain() }})</div> <ul class="ui big
horizontal list voters">
  <li class="item">
    <a href (click)="voteUp()">

```

现在当我们重新加载浏览器时，应该看到每个网址的域名。

基于评分的排序

如果你点击投票，你会发现有一些不太正确：我们的文章不按评分排序！我们绝对希望看到的最高评分的文章。

我们将文章存储在RedditApp的articles数组里，但数组是未排序的，最简单的方法是在RedditApp上创建一个新的方法sortedArticles

```

sortedArticles(): Article[] {
  return this.articles.sort((a: Article, b: Article) => b.votes -
a.votes);
}

```

现在我们可以使用ngFor来遍历我们的 sortedArticles() (而不是直接遍历articles)

```

<div class="ui grid posts">
  <reddit-article *ngFor="#article of sortedArticles()"
[article]="article">
  </reddit-article>
</div>

```



kittencup

2 months ago

完整的代码

```

import { bootstrap } from 'angular2/platform/browser';
import { Component } from 'angular2/core';

class Article {
  title: string;
  link: string;
  votes: number;

  constructor(title: string, link: string, votes?: number) {
    this.title = title;
    this.link = link;
    this.votes = votes || 0;
  }

  domain(): string {
    try {
      const link: string = this.link.split('/')[1];
      return link.split('/')[0];
    } catch (err) {
      return null;
    }
  }

  voteUp(): void {
    this.votes += 1;
  }

  voteDown(): void {
    this.votes -= 1;
  }
}

@Component({
  selector: 'reddit-article',
  inputs: ['article'],
  host: {
    class: 'row'
  },
  template: `
    <div class="four wide column center aligned votes">
      <div class="ui statistic">
        <div class="value">
          {{ article.votes }}
        </div>
        <div class="label">
          Points
        </div>
      </div>
    </div>
  `
})

```

```

</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ article.link }}">
    {{ article.title }}
  </a>
  <div class="meta">({{ article.domain() }})</div>
  <ul class="ui big horizontal list voters">
    <li class="item">
      <a href (click)="voteUp()">
        <i class="arrow up icon"></i>
        upvote
      </a>
    </li>
    <li class="item">
      <a href (click)="voteDown()">
        <i class="arrow down icon"></i>
        downvote
      </a>
    </li>
  </ul>
</div>
,
}))
class ArticleComponent {
  article: Article;

  voteUp(): boolean {
    this.article.voteUp();
    return false;
  }

  voteDown(): boolean {
    this.article.voteDown();
    return false;
  }
}

@Component({
  selector: 'reddit',
  directives: [ArticleComponent],
  template: `
    <form class="ui large form segment">
      <h3 class="ui header">Add a Link</h3>

      <div class="field">
        <label for="title">Title:</label>
        <input name="title" #newtitle>
      </div>
      <div class="field">

```

```

        <label for="link">Link:</label>
        <input name="link" #newlink>
    </div>

    <button (click)="addArticle(newtitle, newlink)"
        class="ui positive right floated button">
        Submit link
    </button>
</form>

<div class="ui grid posts">
    <reddit-article
        *ngFor="#article of sortedArticles()"
        [article]="article">
    </reddit-article>
</div>
-
}))
class RedditApp {
    articles: Article[];

    constructor() {
        this.articles = [
            new Article('Angular 2', 'http://angular.io', 3),
            new Article('Fullstack', 'http://fullstack.io', 2),
            new Article('Angular Homepage', 'http://angular.io', 1),
        ];
    }

    addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
        console.log(`Adding article title: ${title.value} and link:
${link.value}`);
        this.articles.push(new Article(title.value, link.value, 0));
        title.value = '';
        link.value = '';
    }

    sortedArticles(): Article[] {
        return this.articles.sort((a: Article, b: Article) => b.votes -
a.votes);
    }
}

bootstrap(RedditApp);

```




kittencup

2 months ago

结束语

我们成功了，我们创建了第一个Angular 2应用，我们在后面还会学习更多，理解数据流，使用ajax,组件的创建，路由，操纵DOM等。

但现在，请享受你的成功！大部分的Angular 2应用仅仅是象我们上面那样：

- 1.分解你得应用成为组件
- 2.创建视图
- 3.定义model
- 4.显示model
- 5.添加交互



kittencup

2 months ago

寻求帮助

这一章你有什么麻烦吗？你有没有发现一个bug或者有不能运行的代码？我们很乐意听到您的声音！

- 快来加入我们的（免费！）社区和[我们Gitter聊天室](#)
- 直接发邮件给我们us@fullstack.io



kittencup added the **ng-book 2** label 2 months ago



kittencup removed the **解答中** label 2 months ago



kittencup modified this issue 2 months ago

Comment on issue

Sign in to comment



Open

TypeScript #33



kittencup opened this issue
2 months ago



ng-book 2

该issue关闭讨论，如有问题请去 [#43](#) 提问

目录

- [Angular2是建立在TypeScript上](#)
- [TypeScript提供了什么特性?](#)
- [类型](#)
- [内置类型](#)
- [类](#)
- [工具](#)
- [结束语](#)



kittencup added the **ng-book 2** label 2 months ago



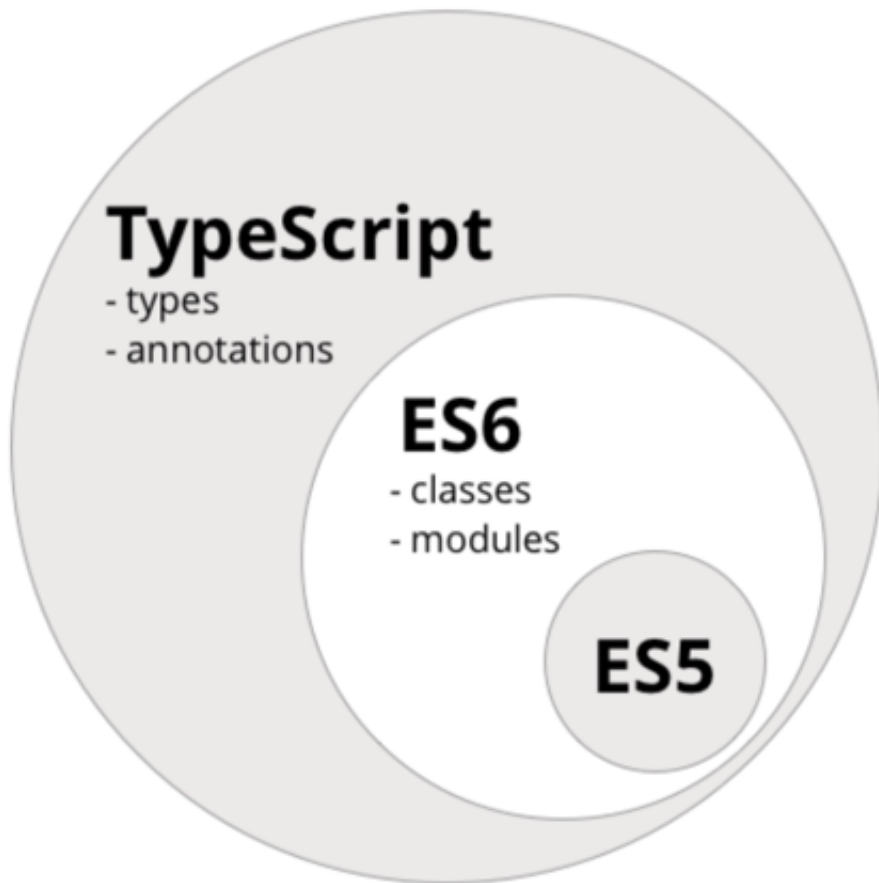
kittencup
2 months ago

Angular2是建立在TypeScript上

Angular 2是建立在一个Javascript般的语言，称为TypeScript。

你可能对使用新语言开发Angular 2进行怀疑，但事实证明，有很多理由使用TypeScript代替普通的Javascript来开发Angular2。

TypeScript并不是一个完全新的语言，它是ES6的一个超集。如果我们写ES6代码，它也是完全有效的，TypeScript也是可编译的。这里有一个图，显示了语言之间的关系：



ES5, ES6, and TypeScript

什么是ES5?什么是ES6。ES5是"ECMAScript 5"的缩写，也被成为"regular javascript", ES5是普通的javascript,他运行在大部分的浏览器上，ES6是javascript下一版本，我们在下面会更多的讨论

在这本书出版时，很少有浏览器支持ES6,更不用说TypeScript,为了解决该问题，我们有transpilers（有时称为transcompiler）。TypeScript的transpilers会将我们的TypeScript代码转换为ES5代码，这样几乎所有浏览器都能正常运行

转换TypeScript代码为ES5代码的 `transpiler` 是由TypeScript核心团队编写，然而，转换ES6代码到ES5代码有2个主要的transpilers：Google开发的[traceur](#)和JavaScript社区创建的[babel](#),我们不打算在这本书中使用直接，但他们都是伟大的项目值得我们了解。

我们在上一章安装了TypeScript，但如果你从本章开始学的，你可以这样安装：

```
npm install -g 'typeScript'
```

TypeScript是微软和谷歌之间的官方合作。这是个好消息，因为它的背后两家科技巨头，我们知道它将会支持很长一段时间。这两个集团都致力于移动网络的发展。

一个关于transpilers的杰出的事情是，他们允许团队来改进语言增加功能，不要求每个人都在互联网上升级他们的浏览器。

有一点需要指出：Angular2应用不是只能用TypeScript开发。如果你想使用ES5（即“标准”的JavaScript），你绝对可以。有一个ES5 API，它提供了访问Angular2的所有功能。那么我们为什么要使用TypeScript呢？因为在TypeScript有一些强大的功能，使开发简单了很多。



kittencup

2 months ago

TypeScript提供了什么特性？

有迹象表明，TypeScript带来了ES5五大改进：

- 类型(types)
- 类(classes)
- 注解(annotations)
- 导入(imports)
- 语言工具(language utilities)



kittencup

2 months ago

类型

TypeScript 相对于ES6的主要改进是，对于变量，有一个类型系统

对于一些人来说，缺乏类型检查被认为是使用像JavaScript语言的好处之一。你可能会有点怀疑类型检查，但我会鼓励你给它一个机会。类型检查是一件好事。

- 它有助于编写代码，因为它可以在编译时防止错误
- 它可以帮助你阅读代码的时候，给出一个明确意图

另外值得一提的是，类型是TypeScript可选的。如果我们想要写一些简单的代码，我们可以省略类型，然后逐步增加他们，使代码变得更加成熟。

TypeScript的基本类型和我们现在使用的Javascript是相同的，字符串，数字，布尔值，等等。

直到ES5,我们都会使用var关键字定义变量，例如 `var name` 。

新的TypeScript语法也是从ES5的自然发展，我们仍然使用var，但现在我们可以选择性地提供变量的类型：

```
var name: string;
```

当声明function时，我们可以将类型用在参数和返回值上

```
function greetText(name: string): string {  
    return "Hello " + name;  
}
```

在上面的例子中，我们定义了一个名为greetText的新函数，它接受一个参数：name, 这个语法 `name: string` 说这个函数接受的参数name是一个字符串类型的，如果我们给函数提供给了一个非字符串类型的参数，我们的代码将无法编译，这是一个很好的事情，因为否则我们会引入一个错误。

注意这个greetText函数在括号前也有一个新语法 `:string {`，冒号表明我们将指定返回类型，返回的类型为字符串，这是非常有帮助的，因为如果当函数返回一个非字符串类型的返回值时，编译器将无法编译，其次，使用此功能也让开发者知道该函数返回什么类型的数据。

让我们来看看，如果我们试图编写不符合我们类型限定的代码会发生什么：

```
function hello(name: string): string {  
    return 12;  
}
```

如果我们尝试编译，我们会看到下面的错误：

```
$ tsc compile-error.ts
```

```
compile-error.ts(2,12): error TS2322: Type 'number' is not assignable  
to type 'string'.
```

这里发生了什么？我们试图返回12,这是一个数字，但我们的函数返回的限定类型是string

为了纠正这一点，我们需要更新函数声明为返回一个数字：

```
function hello(name: string): number {  
    return 12;  
}
```

这是一个小例子，但已经可以看出，通过使用类型可以为我们节省大量寻找bug的时间。

所以，现在我们知道了如何使用类型，那么我们如何才能知道什么类型都可以使用？让我们来看看内置类型的列表



kittencup

2 months ago

内置类型

String

一个保存字符串的文本，类型声明为string：

```
var name: string = 'Felipe';
```

Number

一个数字是任何类型的数值。在TypeScript，所有的数字都表示为浮点。对于数字的类型是number：

```
var age: integer = 36;
```

Boolean

boolean是true或false的值

```
var married: boolean = true;
```

Array

数组是Array类型。然而，因为数组是一个集合，我们还需要指定在数组中的元素的类型。

我们通过 `Array<type>` or `type[]` 语法为数组内的元素指定类型

```
var jobs: Array<string> = ['IBM', 'Microsoft', 'Google'];  
var jobs: string[] = ['Apple', 'Dell', 'HP'];
```

```
var jobs: Array<number> = [1, 2, 3];  
var jobs: number[] = [4, 5, 6];
```

Enums

Enums是列出所有可用值，例如，如果我们想角色为固定列表中的某个人，我们可以这样写：

```
enum Role {Employee, Manager, Admin};  
var role: Role = Role.Employee;
```

一个枚举的默认初始值是0。你可以调整一开始的范围：

```
enum Role {Employee = 3, Manager, Admin};  
var role: Role = Role.Employee;
```

在这里Employee是3，Manager是4,Admin是5，我们甚至可以设置单个值：

```
enum Role {Employee = 3, Manager = 5, Admin = 7};  
var role: Role = Role.Employee;
```

你也可以查找一个给定的枚举的名称，来获取它的值：

```
enum Role {Employee, Manager, Admin};  
console.log('Roles: ', Role[0], ',', Role[1], 'and', Role[2]);
```

Any

any是默认的类型，其类型的变量允许任何类型的值：

```
var something: any = 'as string';  
something = 1;  
something = [1, 2, 3];
```

Void

使用void是指没有预期的类型。这通常是在一个没有返回值的函数：

```
function setName(name: string): void {  
    this.name = name;  
}
```



kittencup

2 months ago

类

在ES5的JavaScript面向对象编程的采用基于原型的对象来实现的。此模型不使用类，而是依赖于原型。

然而，在ES6 JavaScript我们终于有内置类了。

要定义一个类，我们使用Class关键字，以及一个类名加一对大括号。

```
class Vehicle {}
```

类可以有一些属性，方法及constructor构造函数

属性

属性定义了一个类的实例数据。例如，一个类名为Person可能具有first_name, last_name和age属性。

在一类中的每个属性可任选具有一个类型。例如，我们可以说，first_name和last_name属性是字符串和age属性是一个数字。

一个Person类的声明，看起来像这样：

```
class Person{
    first_name:string;
    last_name: string;
    age:number;
}
```

方法

方法是在对象的上下文中运行的函数。要调用对象的方法，首先必须有该对象的一个实例。

实例化一个类，我们使用new关键字。使用new person() 创建人的新实例，例如。

如果我们想用上面类的方法来给一个人打招呼，我们会写点类似的东西：

```
class Person{
    first_name:string;
    last_name: string;
    age:number;
    greet(){
        console.log("Hello",this.first_name)
    }
}
```

请注意，我们可以在这个Person里通过使用this关键字来访问first_name。

如果方法不声明一个明确的返回类型和返回值，它假定他们可以返回任何东西（任何类型）。然而，在这种情况下，我们没有返回内容，因为没有明确的return语句。

需要注意的是一个空值也是一个有效的任意值。

要调用“greet”方法前，你需要首先有一个Person实例。下面是我们如何做到这：

```
// declare a variable of type Person
var p: Person;
// instantiate a new Person instance
p = new Person();
// give it a first_name
p.first_name = 'Felipe'; // call the greet method
p.greet();
```

你可以在同一行声明一个变量和实例化一个类：

```
var p:Person=newPerson();
```

假设我们在Person有一个返回值方法，例如，我们想得到1个人在多少年后的岁数

```
class Person{
  first_name:string;
  last_name: string;
  age:number;
  greet(){
    console.log("Hello",this.first_name)
  }
  ageInYears(years: number): number {
    return this.age + years;
  }
}
// instantiate a new Person instance
var p: Person = new Person();
// set initial age
p.age=6;
//how old will he be in 12 years?
p.ageInYears(12);
//->18
```

构造函数

构造函数是在创建类的新实例时执行的特殊方法。通常，在这里构造函数是为新对象执行初始设置的地方。

构造函数的方法名必须被命名为constructor。他们可以有参数，但他们不能返回任何值。

一个类也可以没有明确定义的构造函数：

```
class Vehicle {}  
var v = new Vehicle();
```

同等于

```
class Vehicle {  
    constructor(){}  
}  
var v = new Vehicle();
```

在TypeScript 每一个类只能有一个constructor

构造函数可以带参数，当我们要实例一个带有参数的类时。例如，我们可以为Person类使用构造函数初始化它的数据

```
class Person {  
    first_name: string;  
    last_name: string;  
    age: number;  
    constructor(first_name: string, last_name: string, age: number) {  
        this.first_name = first_name;  
        this.last_name = last_name;  
        this.age = age;  
    }  
    greet() {  
        console.log("Hello", this.first_name);  
    }  
    ageInYears(years: number): number {  
        return this.age + years;  
    }  
}
```

比前一个例子更容易使用：

```
var p: Person = new Person('Felipe', 'Coury', 36);  
p.greet();
```

我们创建对象时first_name, last_name, age都会被设置

继承

面向对象程序设计的另一个导入方面是继承。继承是表示一个类从父类接收行为的方法。然后我们可以重写，修改或增加新的类的行为。

如果你想更深的了解ES5如何实现继承，你可以看Mozilla Developer Network的文章 [Inheritance and the prototype chain](#)

TypeScript完全支持继承，不像ES5那样，它是核心语法。继承是通过extend关键字来实现的。

为了说明，我们创建一个Report类：

```
class Report{
    data:Array<string>;

    constructor(data:Array<string>){
        this.data = data;
    }

    run(){
        this.data.forEach(function(line) { console.log(line); });
    }
}
```

这个Report类有一个data属性，是一个数组，数组的每一个元素都是字符串类型的，当我们调用run方法，会将data的每个元素打印在控制台

forEach是数组的一个迭代方法，接受了一个函数作为参数，数组中的每个元素会调用函数的方法。

给Report添加一些数据，并调用run方法

```
var r: Report = new Report(['First line', 'Second line']);
r.run();
```

运行后应该显示：

```
First line
Second line
```

现在，我们希望第二个report，它需要一些headers和一些data，但我们仍然要重用的

report类的run方法 将数据打印出来。

要重用report类中的行为，我们可以使用extend关键字继承：

```
class TabbedReport extends Report {
  headers: Array<string>;
  constructor(headers: string[], values: string[]) {
    this.headers = headers;
    super(values)
  }
  run() {
    console.log(headers);
    super.run();
  }
}

var headers: string[] = ['Name'];
var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis
Blakenship'];
var r: TabbedReport = new TabbedReport(headers, data)
r.run();
```

super是父对象的引用，可以通过super访问父对象的方法



kittencup

2 months ago

工具

ES6和TypeScript提供了许多语法功能，使程序设计越来越轻松。两个重要的功能是：

- 箭头(=>)函数语法
- 模板字符串

箭头函数语法

=>符号是用于function的简洁语法

在ES5，每当我们想用一個函数作为参数，我们必须使用function关键字在加一对大括号像这样：

```
var data = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach(function(line) { console.log(line); });
```

然而，随着=>语法我们可以改写为

```
var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis
Blakenship']; data.forEach( (line) => console.log(line) );
```

=>语法可作为表达式：

```
var evens = [2,4,6,8];
var odds = evens.map(v => v + 1);
```

也可以是一个语句

```
data.forEach( line => {
    console.log(line.toUpperCase())
});
```

该=>语法的一个重要特点是，这个表达式代表的函数内的this和外部的this是一致的，这是非常重要的，当你创建一个function时，this是指向window，如果你想使用外部的this,需要先在外部创建一个变量指向那个this,这样会比较麻烦，当你用这种方式写的时，内部this则是指向外部的this。有时在JavaScript中我们看到这样的代码：

```
var nate={
    name: "Nate",
    guitars: ["Gibson", "Martin", "Taylor"],
    printGuitars:function(){
        var self = this;
        this.guitars.forEach(function(g) {
            // this.name is undefined so we have to use self.name
            console.log(self.name + " plays a " + g);
        });
    }
};
```

因为=>语法的this与外部this一致，所以我们可以写成：

```
var nate = {
  name: "Nate",
  guitars: ["Gibson", "Martin", "Taylor"],
  printGuitars: function() {

    this.guitars.forEach((g) => {
      console.log(this.name + " plays a " + g);
    });
  }
};
```

=>是处理内联函数的一个很好的方法。这使得它更容易在JavaScript中使用高阶函数。

模板字符串

模板字符串的两大特点

- 字符串中的变量（无需在使用+符号将字符串和变量连接）
- 多行字符串

字符串中的变量

此功能也被称为“字符串插值” 我们的想法是，你可以把变量插入字符串。具体方法如下：

```
var firstName = "Nate";
var lastName = "Murray";

// interpolate a string
var greeting = `Hello ${firstName} ${lastName}`;

console.log(greeting);
```

注意，要使用字符串插值，你必须用反引号不是单或双引号。

多行字符串

反引号字符串的另一大特点是多行字符串：

```
var template = `
  <div>
    <h1>Hello</h1>
    <p>This is a great website</p>
```

```
</div>
```

```
// do something with `template`
```

当我们想把字符串的代码写的有点长,像模板那样, 多行字符串对你来说是一个巨大的帮助。



kittencup

2 months ago

结束语

TypeScript/ES6 还有多种其他功能:

- Interfaces
- Generics
- 导入和导出模块
- Annotations
- Destructuring

我们将会讲到这些概念, 并在本书中使用它们,但现在这些基本知识够你开始使用

接下去让我们回到Angular



kittencup locked this issue 2 months ago

Comment on issue

Sign in to comment

or [sign up](#) to join this conversation on GitHub



Desktop version

 Open

Angular 2 如何运作 #34



kittencup opened this issue
2 months ago



ng-book 2

该issue关闭讨论，如有问题请去 [#43](#) 提问

目录

- [应用](#)
- [产品模型](#)
- [组件](#)
- [组件装饰器](#)
- [\[ProductsList组件\]](#)
- [ProductRow组件](#)
- [ProductImage组件](#)
- [PriceDisplay组件](#)
- [ProductDepartment组件](#)
- [完成的项目](#)
- [数据架构上的信息](#)

在本章，我们将讨论Angular 2中得高级概念，我的想法是，通过一步一步的方式，可以让你看到所有的组件是如何组合在一起的。


如果你有使用过Angular 1,您会注意到Angular 2以一个新的思维模式来构建应用程序。别慌!对于Angular 1用户来讲Angular 2既简单又熟悉。在本书的后面章节,我们将专门讨论如何从Angular 1转换应用到Angular 2

在接下来的章节中,我们将深入了解每个概念,但在这里我们只给出一个概述，并解释其基本思路。

第一个大的思想是一个Angular 2应用是由组件(Component)构成。组件的一种方式教浏览器来认识新标签，如果你有Angular 1的使用背景,组件是类似于Angular 1指令(事实证明,Angular 2

也有指令,但是以后我们会进一步讨论这个区别)。

然而 Angular 2组件相对于Angular 1指令有一些显著的优势。我们将在下面讨论。首先, 让我们从头开始: 应用程序。

 **kittencup** added the **ng-book 2** label 2 months ago



kittencup
2 months ago

应用

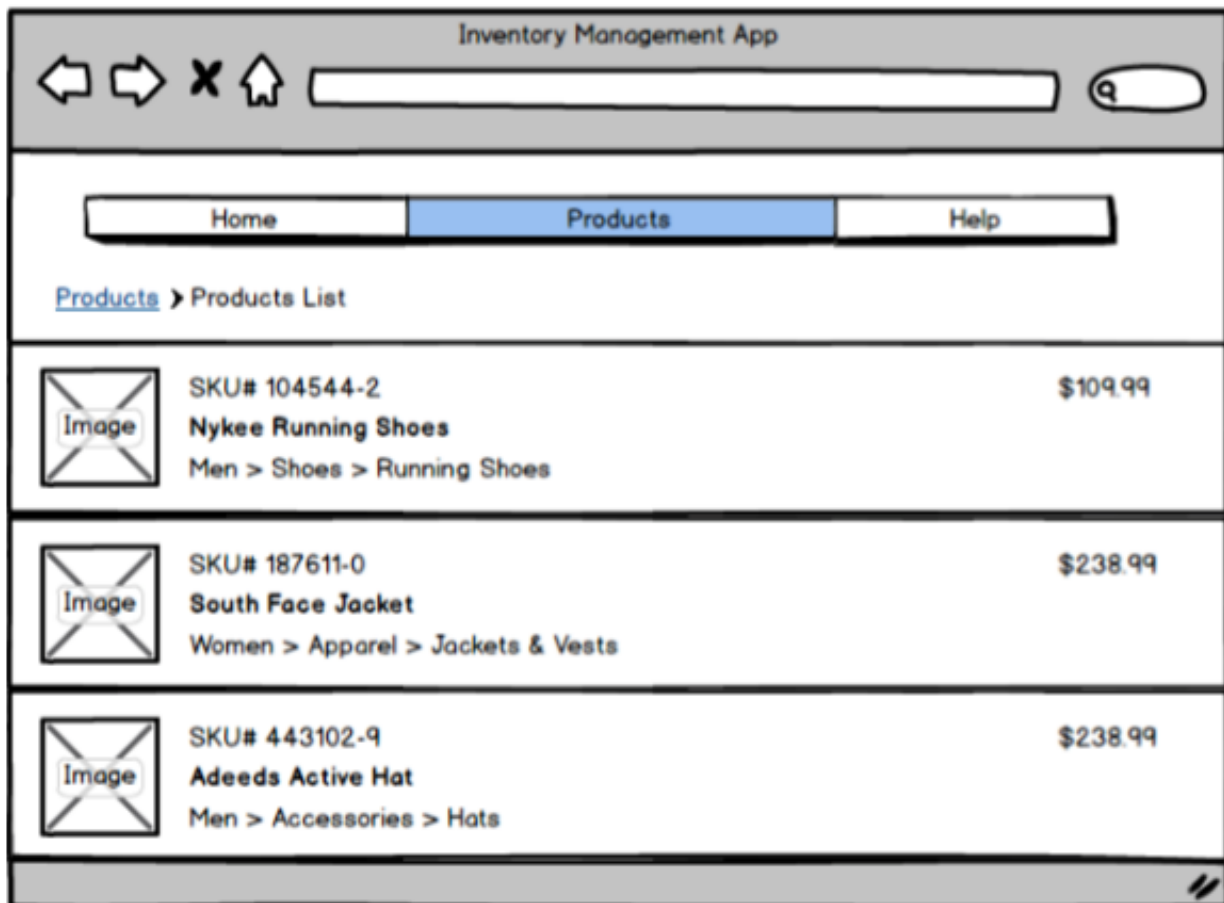
Angular 2应用无非就是一个是组件树。

在树的根部, 最顶层的组件就是应用的本身, 这就是浏览器将引导的(bootstrapping)的应用

其中一个有关组件强大的是, 他们是可组合。这意味着, 我们可以从较小组件来建立更大的组件。应用只是一个简单的渲染其他组件的组件。

因为组件是一个父/子树的结构,每个组件渲染时,将递归地渲染其子组件。

例如,让我们来讨论一个简单的库存管理应用程序, 下面是它的页面模型:



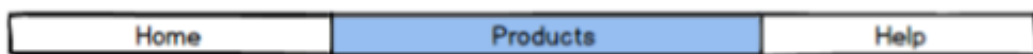
Inventory Management App

鉴于这种模型,编写这个应用程序我们要做的第一件事就是把它分成单个组件(组织成一个树)。在这个例子中,我们可以组页面分成三个相同级别的组件

- 导航组件
- 面包屑组件
- 产品列表组件

导航组件

该组件将渲染导航部分。这将允许用户访问应用的其他页面



Navigation Component

面包屑组件

这将渲染表示应用程序中的用户目前在哪里。

[Products](#) > Products List

Breadcrumbs Component

产品列表组件

这将是一个表示产品的集合。

	SKU# 104544-2 Nykee Running Shoes Men > Shoes > Running Shoes	\$109.99
	SKU# 187611-0 South Face Jacket Women > Apparel > Jackets & Vests	\$238.99
	SKU# 443102-9 Adeeds Active Hat Men > Accessories > Hats	\$238.99

Product List Component

把这个组件分解成更小的下一个级别组件,也就是说,产品列表组件是有多个产品列组成的

	SKU# 104544-2 Nykee Running Shoes Men > Shoes > Running Shoes	\$109.99
---	--	----------

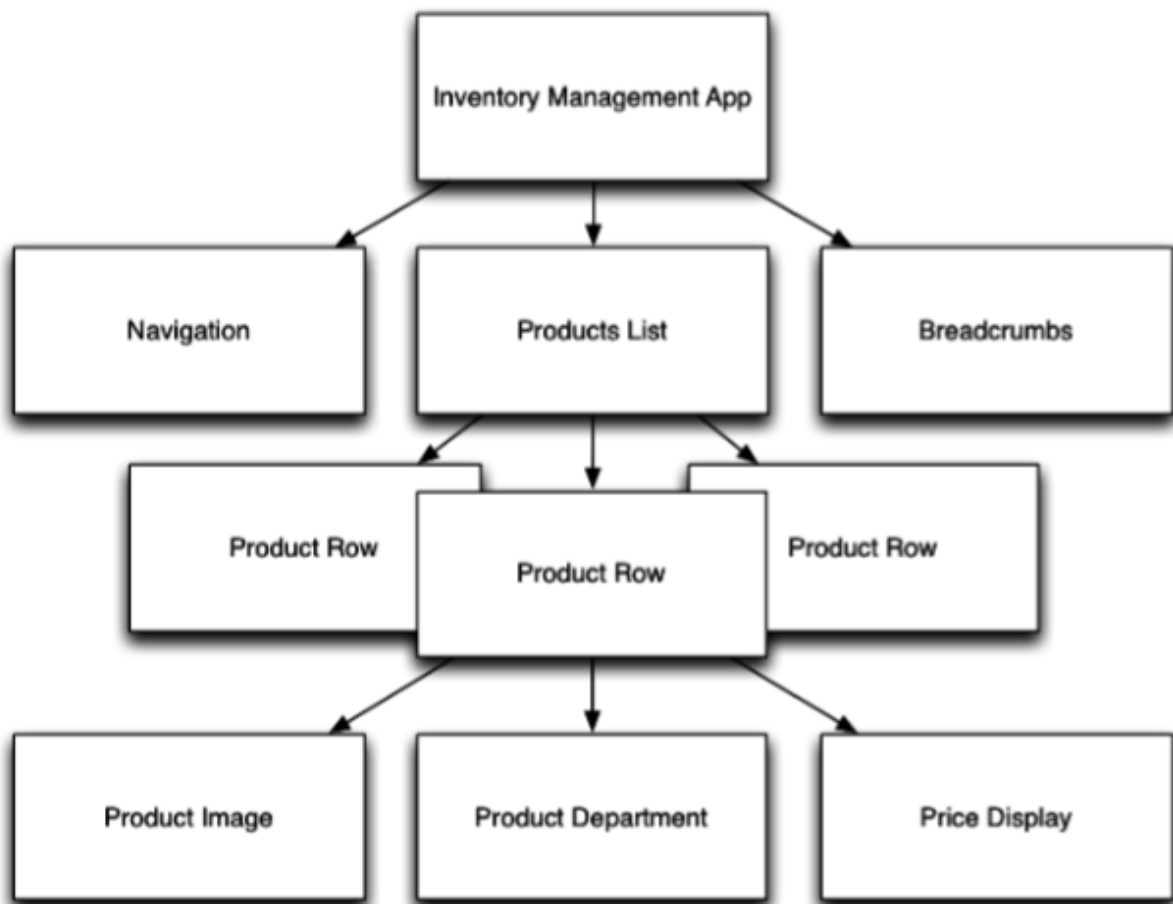
Product Row Component

当然,我们可以继续更进一步,将每个产品行分解成更小的片段:

- **产品图片(Product Image)组件**, 这将是负责渲染产品图片, 获取图片名字(例如,组件知道如何使用适当的Amazon S3的URL来获取图像)
- **产品分类(Product Department)组件**, 给定一个产品分类id, 来渲染分类树, 类似于 Men > Shoes > Running Shoes
- **显示价格(Price Display)** 将是一个更通用的组件,跨越应用时,可以再次利用,是用来正

确呈现一个价格。想象我们的实现定制的定价如果用户已经登录了则包括折扣或运费等。在这个组件我们可以实现所有这些行为。

最后，把它全部连成一个树表示，我们最终得到如下图：



App Tree Diagram

在顶部，我们看到**库存管理app**：这是我们的顶层应用

在应用下面，我们有导航，面包屑和产品列表的组件。产品列表每行为一个产品行组件。

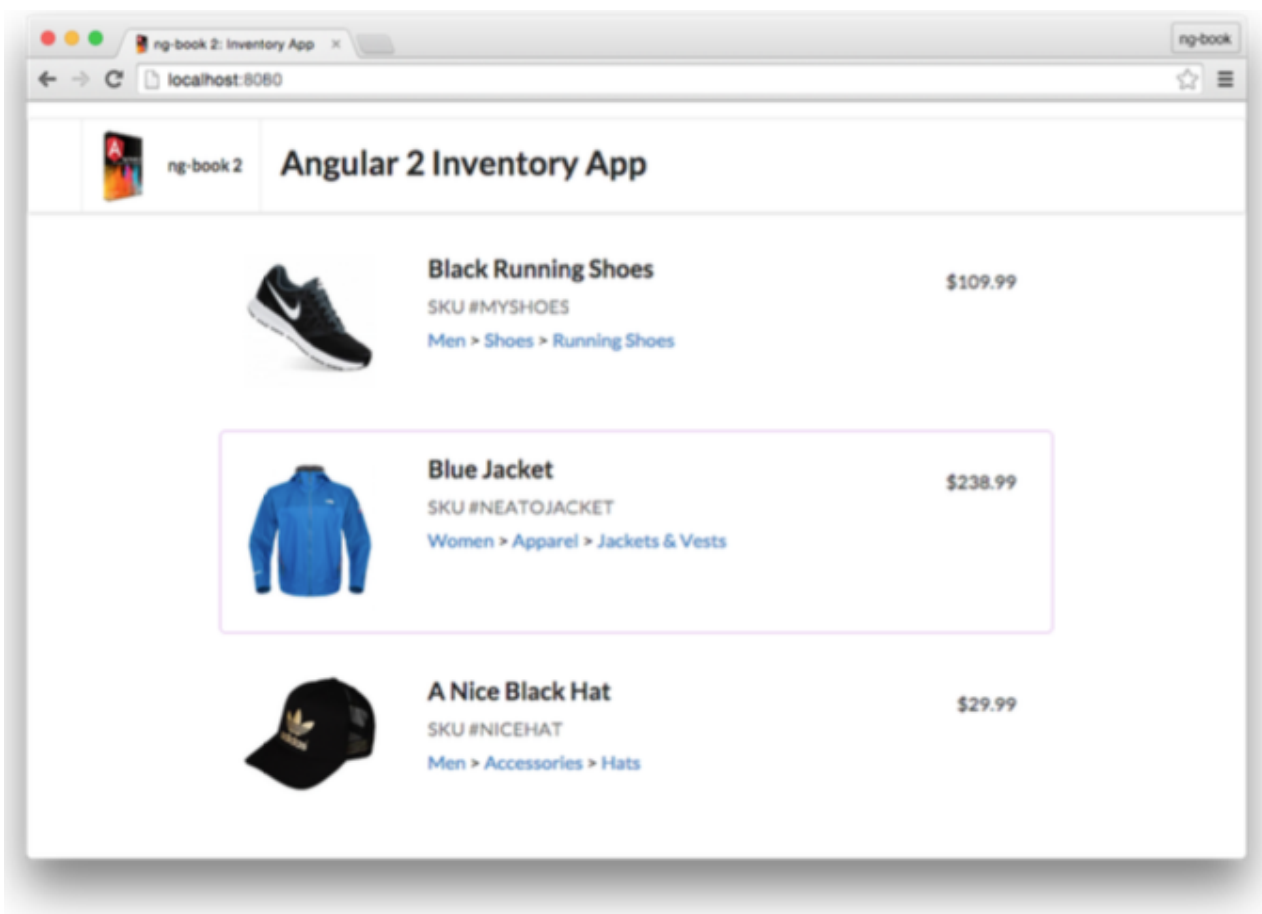
而这个产品行组件由分类，图片，价格组件组成的。

需要注意的重要一点是，每个应用只能有一个顶层组件。


让我们来共同构建这个应用。

你可以在 下载文件目录 `how_angular_works/inventory_app` 中找到该应用全部的代码

下面是当我们完成应用时的截图:



Completed Inventory App

 **kittencup** locked this issue 2 months ago



kittencup
2 months ago

产品模型

关于Angular 的一个关键的事情是，它没有规定一个特定的模型库。

Angular 是足够灵活的，可以支持许多不同类型的模型（和数据结构）。然而，这意味着选择是留给用户自己来确定如何实现这些东西。

在接下去的章节中，我们将有很多关于数据架构的讨论。现在，我们的模型则是简单的

JavaScript对象。

```
/**
 * Provides a `Product` object
 */
class Product {
  constructor(
    public sku: string,
    public name: string,
    public imageUrl: string,
    public department: string[],
    public price: number) {

  }
}
```

如果你还未接触过ES6/TypeScript, 这个语法可能会有点陌生。

我们创建一个新的 `Product` 类, `constructor` 有5个参数, 当我们写上 `public sku:string`, 它会做2件事

- 该类的实例会有一个public属性名为sku
- sku的类型是字符串

如果你已经熟悉JavaScript, 你可以在这里[learnxinyminutes](#)迅速赶上一些差异, 包括 `public constructor` 简写等

这个产品类在Angular没有任何依赖关系, 它只是一个模型, 将用在我们的应用上。



kittencup

2 months ago

组件

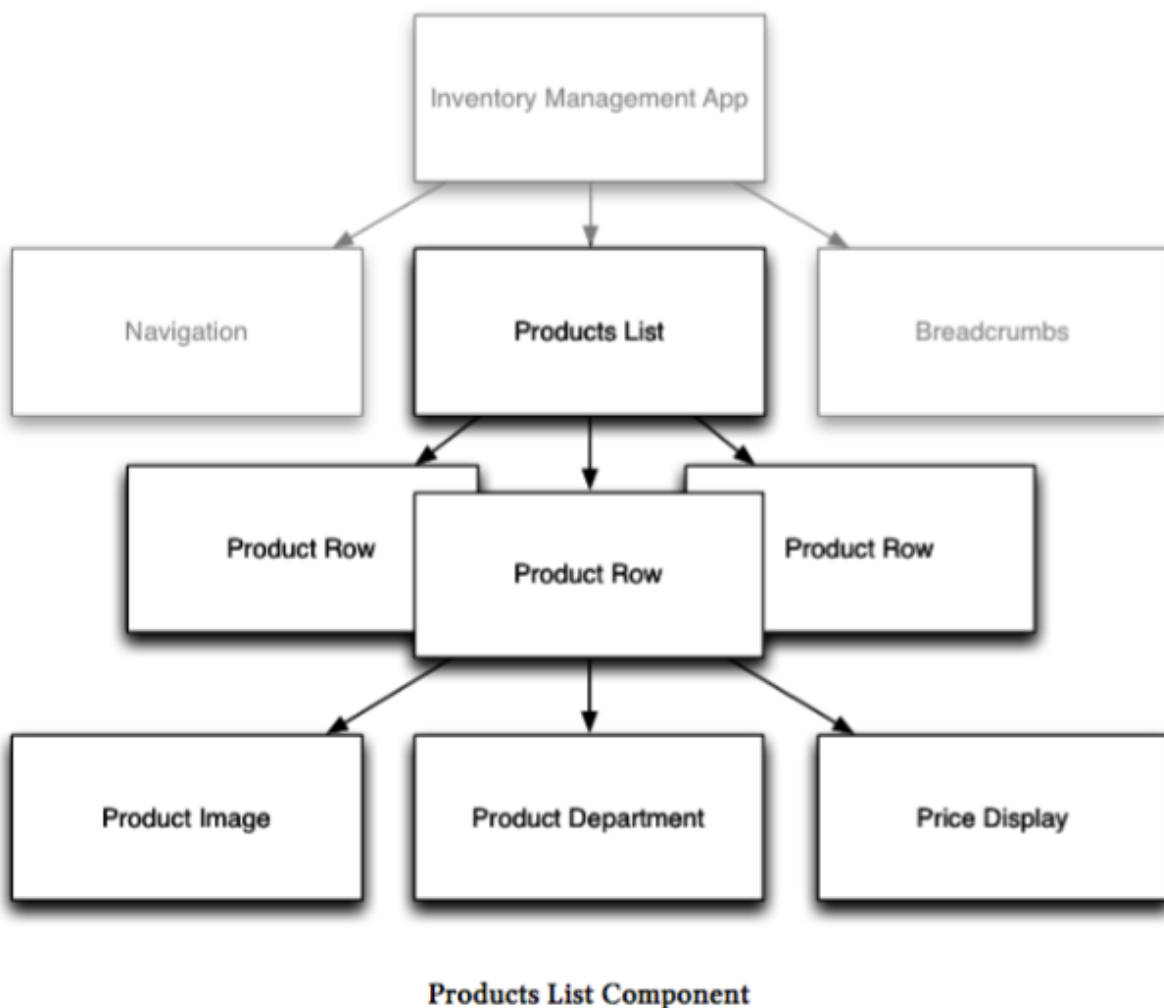
正如我们之前提到, 组件是Angular 2应用基本构建块。"应用"本身就是顶层组件。然后我们把应用划分为更细粒度的子组件。

提示: 当建立一个新的Angular应用, 先设计实体模型, 然后把它分解成组件。

我们将使用它们,所以值得更仔细地看看它们。每个组件都是由三部分组成:

- 组件装饰器
- 视图
- 控制器

为了说明我们需要了解有关组件的主要概念，让我们专注于产品列表中的子组件：



下面是一个最基础的顶层 `InventoryApp`：

```
@Component({
  selector: 'inventory-app',
  template: `
    <div class="inventory-app">
      (Products will go here soon)
    </div>
  `
})
class InventoryApp {
```



```
}  
bootstrap(InventoryApp);
```

如果你一直使用Angular 1的语法则看起来很奇怪!但思想很相似,所以让我们一步一步来看他们:

`@Component` 是一个装饰器, 它为紧随其后 (`InventoryApp`) 的类增加了元数据。

`@Component` 注解指定了

- `selector`, 告诉Angular 要怎么匹配元素
- `template`, 定义了视图

让我们更详细的看看每一部分



kittencup

2 months ago

组件装饰器

该 `@Component` 装饰器用于配置组件。 `@Component` 将配置外界如何与你的组件进行交互。

有许多可用于配置组件的选项, 我们将在组件这章具体讨论。在本章中, 我们只是要触及一些基本知识。

组件选择器(Selector)

`selector` 键, 表示当你渲染HTML模板时如果使你的组件被识别。这个想法是类似于CSS或XPath选择器。选择器将HTML元素匹配该组件。在 `selector: 'inventory- app'` 这种情况下, 是要匹配`inventory- app`标签, 也就是说每当我们使用这个组件, 那么这个新的标签就被定义了新的内置功能, 例如, 当我们HTML为

```
<inventory-app></inventory-app>
```

Angular 将使用 `InventoryApp` 组件来实现这功能。

另外, 我们也可以使用常规的div, 指定的组件将作为一个属性:

```
<div inventory-app></div>
```

组件模板

你可以把视图看作为组件的可视化的部分，在 `@Component` 中配置 `template` 选项，声明后该组件将具有HTML模板。

```
@Component({
  selector: 'inventory-app',
  template: `
    <div class="inventory-app">
      (Products will go here soon)
    </div>
  `
})
```

对于这个模板，请注意，我们使用的是TypeScript反引号“`”多行字符串语法。我们的模板到目前为止是相当简单的：只有一些占位文字。

添加一个产品

没有产品视图的应用不是很有趣。现在让我们添加一些。

我们可以创建一个新的产品：

```
let newProduct = new Product(
  'NICEHAT',
  'A Nice Black Hat',
  '/resources/images/products/black-hat.jpg',
  ['Men', 'Accessories', 'Hats'],
  29.99);
```

我们的 `Product constructor` 需要提供5个参数，通过使用 `new` 关键字来创建新的 `Product` 对象

通常我可能不会传递给函数超过5个参数，另一种选择是，通过传递一个Object给 `Product` 的 `Constructor`，那么我们就没有必要记住参数的顺序，也就是说，`Product` 可以改为这样：

```
new Product({sku: "MYHAT", name: "A green hat"})
```

我们希望能够展示这个 `Product` 的视图,为了使模板可以访问它们, 我们将它们添加到组件的实例变量中。

例如, 如果我們想在视图中访问 `newProduct` ,我们可能会这样写:

```
class InventoryApp { product: Product;
  constructor() {
    let newProduct = new Product(
      'NICEHAT', 'A Nice Black Hat',
      '/resources/images/products/black-hat.jpg',
      ['Men', 'Accessories', 'Hats'],
      29.99);
    this.product = newProduct;
  }
}
```

或更简洁:

```
class InventoryApp {
  product: Product;
  constructor() {
    this.product = new Product(
      'NICEHAT', 'A Nice Black Hat',
      '/resources/images/products/black-hat.jpg',
      ['Men', 'Accessories', 'Hats'],
      29.99);
  }
}
```

注意我们在这里做了三件事:

1. 添加了 `constructor` 方法 - 当Angular为该组件创建一个实例时, 会调用 `constructor` 函数, 在这里, 我们可以为初始化点数据。
2. 声明了一个实例变量 - 在 `InventoryApp` 上, 当写上 `product:Product` ,表示 `InventoryApp` 实例有一个 `product` 属性, 类型是 `Product` 对象
3. 为`product`赋值 - 在 `constructor` 中, 我们创建了`Product`实例, 并赋值给`product`属性

模板绑定

前面为`product`属性分配了值, 现在我们就可以在视图中使用该变量。改变我们的模板如

下:

```
@Component({
  selector: 'inventory-app',
  template: `
    <div class="inventory-app">
      <h1>{{ product.name }}</h1>
      <span>{{ product.sku }}</span>
    </div>
  `
})
```

使用 `{{...}}` 语法称为模板绑定。

所以在这种情况下，我们有两个绑定：

- `{{ product.name }}`
- `{{ product.sku }}`

`product` 变量来自于我们 `InventoryApp` 组件实例中的 `product` 属性

`{{}}` 中的代码是一个表达式，这意味着，你可以做这样的事情：

- `{{count+1}}`
- `{{ myFunction(myArguments) }}`

在第一种情况下，我们使用一个操作符来改变 `count` 的显示值。在第二种情况下，我们可以利用函数 `MyFunction(myArguments)` 的值替换表达式。使用模板绑定标签是你在应用中显示数据的主要方式。

添加更多产品

我们其实不想在我们的应用程序中显示一个单一的产品-我们实际上要显示一个完整的产品列表。让我们改变 `InventoryApp` 来存储产品数组而不是一个单一的产品：

```
class InventoryApp {
  products: Product[];
  constructor() {
    this.products = [];
  }
}
```

注意：我们把 `product` 重命名为 `products` ,并且将类型改变成 `Product[]` 。 `product[]` 字符表明我们想要的 `products` 是一个数组类型，里面的每一个元素都是 `Product` 对象，我们还可以写成 `Array<Product>` 。

现在，我们的 `InventoryApp` 拥有一个 `Products` 数组。让我们在构造函数中创建一些产品：

```
class InventoryApp {
    products:Product[];

    constructor() {
        this.products = [
            new Product(
                'MYSHOES', 'Black Running Shoes',
                '/resources/images/products/black-shoes.jpg', ['Men', 'Shoes', 'Running Shoes'],
                109.99),
            new Product(
                'NEATOJACKET', 'Blue Jacket',
                '/resources/images/products/blue-jacket.jpg', ['Women', 'Apparel', 'Jackets & Vests'], 238.99),
            new Product(
                'NICEHAT', 'A Nice Black Hat',
                '/resources/images/products/black-hat.jpg', ['Men', 'Accessories', 'Hats'],
                29.99)
        ];
    }
}
```

这段代码将会给我们一些产品为我们的应用工作。

选择一个产品

我们希望应用支持用户交互，例如，用户可以选择一个特定的产品来查看关于产品更多的信息，或把它添加到购物车等。

让我们来添加一些功能，当一个新产品被选中时在我们在 `nventoryApp` 中处理一些事情，要做到这一点，我们需要定义了一个新的函数， `productWasSelected`

```
productWasSelected(product: Product): void {
    console.log('Product clicked: ', product);
}
```

产品列表

现在，我们有顶层的InventoryApp组件，我们需要添加一个新组件用来渲染产品列表，在下一章节我们会创建并实现 ProductList 组件用来匹配 products-list 选择器，在我们深入实现细节之前，我们先看看如何使用这个新组件：

```
@Component({
  selector: 'inventory-app', directives: [ProductsList], template: `
  <div class="inventory-app">
    <products-list
      [productList]="products"
      (onProductSelected)="productWasSelected($event)"> </products-list>
  </div>
  ` })
```

在这里看到有一些新的语法和配置项，所以让我们来谈谈它们：

directives 选项

注意在我们 @Component 配置中添加了 directives 选项，这定义了我们在视图中想要使用的其他组件是什么。

不像Angular 1，所有的指令本质上是全局，在Angular 2，你必须明确说打算使用哪个指令。在这里，我们要使用的 ProductList 指令。

input和output

当使用的 products-list 时，我们要使用Angular的关键特性： input 和 output

```
<products-list
  [productList]="products" <!-- input -->
  (onProductSelected)="productWasSelected($event)"> <!-- output -->
</products-list>
```

[squareBrackets]传入输入,(parenthesis)处理输出

数据通过input流入组件，通过output流出组件

可以认为 input+ouput 是你组件定义的公开API

[squareBrackets] 传入输入

在Angular 可以使用input将数据传入到子组件

```
<products-list  
  [productList]="products"
```

这个属性分为2块

- `[productList]` (左侧)
- `"products"` (右侧)

左侧的 `[productList]` 表示我们在ProductsList组件接受的输入名为productList

右侧的products 表示要传递的值的表达式，也就是说InventoryApp类中的 `this.products` 数组

(parens) 处理输出

在Angular ,你可以使用outputs向组件外发送数据

```
<products-list  
  ...  
  (onProductSelected)="productWasSelected($event)">
```

我们从ProductsList组件监听output的onProductSelected

就是说:

- 左侧(onProductSelected),output我们想要监听的名字,
- 右侧productWasSelected,我们想要接受output的处理函数
- \$event是一个特殊的变量, 在这里表示输出的东西。

现在, 我们还没有谈到如何定义input或output, 但我们很快会在我们ProductsList组件中定义。

完整InventoryApp代码

```
@Component({
```

```

selector: 'inventory-app', directives: [ProductsList], template: `
  <div class="inventory-app">
    <products-list
      [productList]="products"
      (onProductSelected)="productWasSelected($event)"> </products-
list>
  </div>
`
}))
class InventoryApp {
  products:Product[];

  constructor() {
    this.products = [
      new Product(
        'MYSHOES', 'Black Running Shoes',
        '/resources/images/products/black-shoes.jpg',
        ['Men', 'Shoes', 'Running Shoes'],
        109.99),
      new Product(
        'NEATOJACKET', 'Blue Jacket',
        '/resources/images/products/blue-jacket.jpg',
        ['Women', 'Apparel', 'Jackets & Vests'],
        238.99),
      new Product(
        'NICEHAT', 'A Nice Black Hat',
        '/resources/images/products/black-hat.jpg',
        ['Men', 'Accessories', 'Hats'],
        29.99)
    ];
  }

  productWasSelected(product:Product):void {
    console.log('Product clicked: ', product);
  }
}
bootstrap(InventoryApp);

```



kattencup

2 months ago

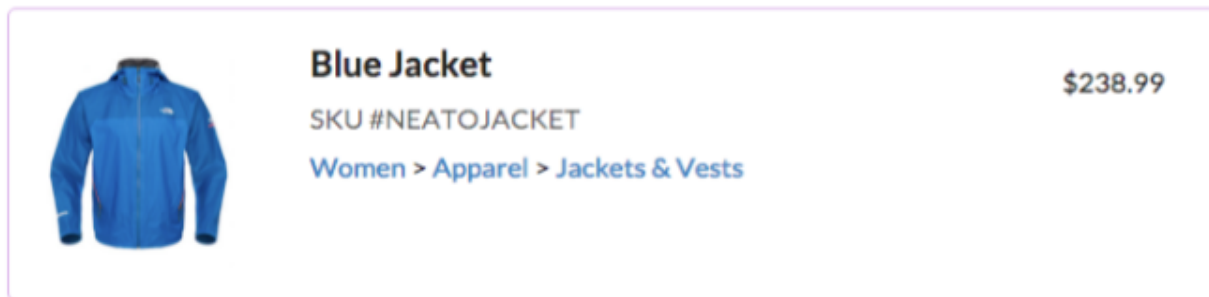
ProductsList组件



kittencup

2 months ago

ProductRow组件

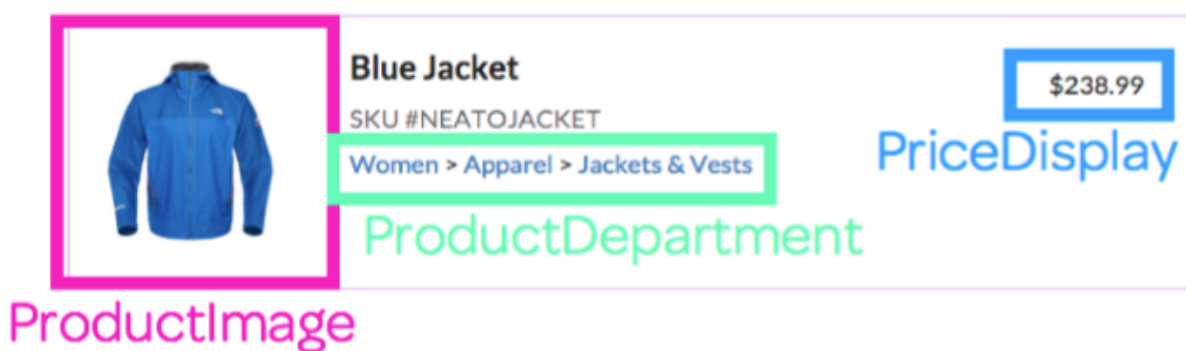


A Selected Product Row Component

我们的ProductRow用来展示 `Product.ProductRow` productrow将有它自己的模板，但也会被分割成三个更小的组件：

- ProductImage - 用于显示图片
- ProductDepartment - 用于分类的面包屑 • PriceDisplay - 用于显示产品价格

这里有一个更直观的图来显示 `ProductRow` 使用的三个组件：



ProductRow's Sub-components

ProductRow 组件的配置

```

* @ProductRow: A component for the view of single Product
*/
@Component({
  selector: 'product-row',
  inputs: ['product'],
  host: {'class': 'item'},
  directives: [ProductImage, ProductDepartment, PriceDisplay],

```

我们先定义 `selector` 为 `product-row`，这个 `selector` 我们已经看到过很多次了，这个组件将匹配的`product-row`标签。

接下去我们定义这个 `product-row` 需要一个 `product` 输入，这这个`product`将通过父组件传入进来

第3个 `host` 选项是一个新出现的。`host` 选项可以让我们在 `host element` 上设置属性，在这里，我们使用[Semantic Ui Item class](#),当我们设置 `host: {'class': 'item'}` 表示我们想要在 `host element` 上附加上class `item`

使用`host`是不错的，因为这意味着我。可以在组件中来配置我们的 `host element` .这是非常棒的，因为否则我们会要求 `host element` 来指定CSS标签，这是不好的，是因为在使用组件时还需要包含上所对应的CSS类部分

接下来，我们指定了我们将来要在模板中使用的指令。现在我们还没有定义这些指令，但我们将在稍后定义。

ProductRow 组件类

该ProductRow组件定义类很简单：

```

class ProductRow {
  product: Product;
}

```

这里我们指定了 `ProductRow` 将会有有一个 `product` 实例变量，因为我们指定了 `input` 为 `product` 。当Angular创建该组件实例时，会自动将 `input` 的 `product` 分配给 `ProductRow` 中的 `product` ,我们不需要手动来做，我们不需要一个构造函数。

ProductRow 模板

现在，让我们来看看模板：

```

template: `
<product-image [product]="product"></product-image> <div
class="content">
<div class="header">{{ product.name }}</div> <div class="meta">
<div class="product-sku">SKU #{{ product.sku }}</div> </div>
<div class="description">
<product-department [product]="product"></product-department>
    </div>
</div>
<price-display [price]="product.price"></price-display> `

```

我们的模板中没有什么新的概念。

在模板第一行，我们使用product-image指令，我们通过input将product传入到ProductImage组件里，我们以相同的方式使用product-department指令

我们使用price-display指令略有不同，我们通过product.price，而不是直接使用product.

模板的其余部分是标准的HTML元素的自定义CSS类和一些模板绑定

ProductRow完整代码

这里是ProductRow组件的所有代码

```

/**
 * @ProductRow: A component for the view of single Product
 */
@Component({
  selector: 'product-row',
  inputs: ['product'],
  host: {'class': 'item'},
  directives: [ProductImage, ProductDepartment, PriceDisplay],
  template: `
    <product-image [product]="product"></product-image>
    <div class="content">
    <div class="header">{{ product.name }}</div> <div class="meta">
    <div class="product-sku">SKU #{{ product.sku }}</div> </div>
    <div class="description">
    <product-department [product]="product"></product-department>
      </div>
    </div>
    <price-display [price]="product.price"></price-display>
  `
})
class ProductRow {

```

```
product: Product;
}
```

现在让我们谈谈我们使用的三个组件。他们很简单



kittencup

about 1 month ago

ProductImage组件

```
/**
 * @ProductImage: A component to show a single Product's image
 */
@Component({
  selector: 'product-image',
  host: {class: 'ui small image'},
  inputs: ['product'],
  template: `
    <img class="product-image" [src]="product.imageUrl">
  `
})
class ProductImage {
  product: Product;
}
```

这里要注意的是关于 `img` 标签的问题，我们使用`[src]`来给`img`输入值。我们本来可以这么写的：

```
<!-- 错误，不要这样做 -->

```

为什么是错误的？因为在这种情况下，浏览器在运行Angular前就已加载这个模板，浏览器将尝试加载`src`为`{{ product.imageUrl }}`的图像然后将得到一个404页面，它会在运行Angular前显示一个破碎的图像。

通过使用 `[src]` 属性，我们告诉Angular,我们想使用 `[src]` 为这个 `img` 标签来输入值，一旦表达式被运行，Angular会替换成`src`属性



kittencup

about 1 month ago

PriceDisplay组件

接下去，让我们来看看PriceDisplay

```
/**
 * @PriceDisplay: A component to show the price of a
 * Product
 */
@Component({
  selector: 'price-display',
  inputs: ['price'],
  template: `
<div class="price-display">\${{ price }}</div> `
})
class PriceDisplay {
  price: number;
}
```

这里简单的，但有一点要注意的是，我们现在要输出 `$`，而在es6多行字符串里，`${{}}` 有替换变量的作用，所以在这里我们在 `$` 之前添加了一个 `\` 用来转义



kittencup

about 1 month ago

ProductDepartment组件

```
/**
 * @ProductDepartment: A component to show the breadcrumbs to a
 * Product's department
 */
@Component({
  selector: 'product-department',
  inputs: ['product'],
  template: `
<div class="product-department">
  <span *ngFor="#name of product.department; #i=index">
    <a href="#">{{ name }}</a>
  </span>
</div>`
})
class ProductDepartment {
  product: string;
}
```

```

        <span>{{i < (product.department.length-1) ? '>' : ''}}</span>
      </span>
    </div>
  `
  })
  class ProductDepartment {
    product: Product;
  }

```

需要注意的是ProductDepartment组件的ngFor和span标签

ngFor循环 `product.department` ,并将每一个分类分配给 `name` ,新出现的部分是第二个表达式 `#i=index` ,这是从ngFor中获取当前迭代的索引值。

在 `span` 标签, 我们使用*i*来决定是否显示 `'>'` 符号

这里将显示每一个分类的字符串

Women > Apparel > Jackets & Vests

表达式 `{{i < (product.department.length-1) ? '>' : ''}}` 表示 如果不是最后一个分类, 就显示一个 `'>'` 符号, 如果是最后一个分类就显示空字符串

这种语法 `test ? valueIfTrue : valueIfFalse` 称为 三元运算符



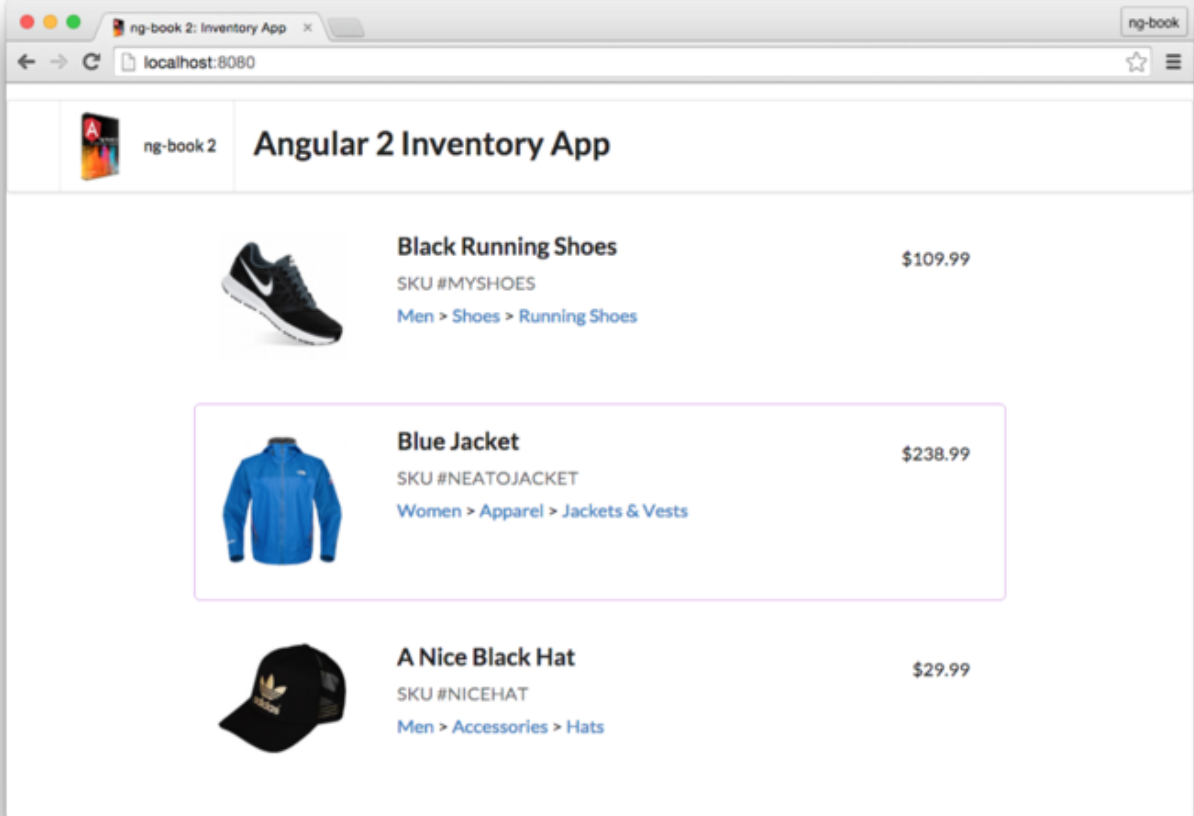
kittencup

about 1 month ago

完成的项目

现在有了项目所需要的所有部分。

当我们完成后它看起来像这样:



Completed Inventory App

你可以在目录 `how_angular_works/inventory_app` 下找到示例代码，详见README

现在，您可以通过点击来选择一个特定的产品，当选择后它会显示一个漂亮的紫色轮廓。如果您在代码中增加新的产品，你会看到他们被渲染。



kittencup

about 1 month ago

数据架构上的信息

如果我们开始添加更多的功能到该应用,你可能想知道我们如何管理数据流

例如，假设我们想增加一个购物车的视图，然后将项目添加到购物车。我们应该怎么实现它？

我们已经讨论过的唯一工具是发出output时间，当我们点击add-to-cart，我们简单的冒泡

addedToCart时间到根组件上进行处理？这感觉有点别扭。

数据结构是一个很大话题，有很多见解。值得庆幸的是，Angular是可以足够灵活处理各种各样的数据架构，这就意味着你必须自己决定使用哪一个。

在Angular 1，默认的选择是双向的数据绑定。双向数据绑定是超级容易上手：你的控制器有数据，表单直接操作数据，和视图显示数据。

双向数据绑定的问题是，随着你的项目增长，它往往会导致整个应用的级联效应，并很难跟踪你的数据流

双向数据绑定的另一个问题是，因为它常常迫使你的“数据布局树”与你的“DOM视图树”匹配。在实践中，这两者应该是分开的。

你可能处理这种情况,就是创建一个ShoppingCartService,这将是一个单例,保存当前购物车中商品的列表。此服务可以在购物车更改时通知任何感兴趣的对象。

这个想法是很容易的，但在实践中有很多细节需要解决。

在Angular2,并且很多现代的Web框架(如React)是采用单向数据绑定模式，也就是说，你的数据流只能向下流入组件，如果你需要进行数据变化，你可以发射导致变化的事件到顶部，然后在往下流入组件

单向数据绑定可能在开始的时候增加了一些性能开销,但它节省了大量的各地复杂的变化检测,它使您的系统更容易推算。

值得庆幸的是有两个主要的对手，用于管理您的数据架构：

1. 使用 Observables-based架构，像RxJs
2. 使用 Flux-based架构

在这本书的后面，我们将讨论如何为您的应用实现一个可扩展的数据架构。

Comment on issue

Sign in to comment

or [sign up](#) to join this conversation on GitHub





Open

内置组件 #38



kittencup opened this issue
about 1 month ago



ng-book 2

该issue关闭讨论，如有问题请去 [#43](#) 提问

目录

- [介绍](#)
- [NgIf](#)
- [NgSwitch](#)
- [NgStyle](#)
- [NgClass](#)
- [NgFor](#)
- [NgNonBindable](#)
- [结论](#)



kittencup added the **ng-book 2** label about 1 month ago



kittencup
about 1 month ago

介绍

Angular 2 提供了很多内置的组件。我们要覆盖每个内置组件，并通过例子告诉你如何使用它们。

内置的组件都是已经被Import，并自动提供给你的组件，所以你不需要将它作为一个指令，就像自己的组件。



kittencup

about 1 month ago

NgIf

ng-if指令是用于当你想要通过某个条件显示或者隐藏一个元素时，条件是由你传递给该指令的表达式结果来确定的

如果表达式结果是false的话，该元素将不会渲染DOM。

你可以使用任何计算结果为布尔值的表达式。一些例子：

```
<div *ngIf="false"></div> <!-- never displayed -->
<div *ngIf="a > b"></div> <!-- displayed if a is more than b -->
<div *ngIf="str === 'yes'"></div> <!-- displayed if str holds the
string "yes" -->
<div *ngIf="myFunc()"></div> <!-- displayed if myFunc returns a true
value -->
```

你过有Angular 1的经验，之前你可能使用ngIf指令，你可以认为Angular 2版本作为一个直接的替代品,另一方面

Angular 2 没有提供内置的ng-show，如果你的目标只是改变一个元素的css，你可以考虑使用ngStyle或class指令，这会在后面的章节中描述。



kittencup

about 1 month ago

NgSwitch

有时，你想根据一个给定的条件渲染不同的元素。

当你遇到这种情况，你可以用很多ngIf指令，像这样：

```
<div class="container">
  <div *ngIf="myVar == 'A'">Var is A</div>
  <div *ngIf="myVar == 'B'">Var is B</div>
  <div *ngIf="myVar != 'A' && myVar != 'B'">Var is something
else</div>
</div>
```

你可以看到条件既不是A又不是B的写法有点奇怪，当我们添加值，最后ngIf条件变得越来越复杂。

为了说明这一点,假设我们要处理新的c值。

为了做到这一点，我们就不仅要添加新的元素ngIf，而且还改变了最后一种情况：

```
<div class="container">
  <div *ngIf="myVar == 'A'">Var is A</div>
  <div *ngIf="myVar == 'B'">Var is B</div>
  <div *ngIf="myVar == 'C'">Var is C</div>
  <div *ngIf="myVar != 'A' && myVar != 'B' && myVar != 'C'">Var is
something else</div>
</div>
```

对于这样的情况，使用Angular2的ngSwitch指令。

如果你熟悉各种语言的switch，你会感到非常熟悉。

该指令背后的想法是一样的：允许判断一个表达式，然后显示内部嵌套的元素上于表达式结果一致的元素

一旦有了结果，我们就可以：

- 使用ngSwitchWhen来描述已知的结果
- 使用ngSwitchDefault来处理所有其他未知情况

让我们用这个新的指令集重写我们的例子吧：

```
<div class="container" [ngSwitch]="myVar">
  <div *ngSwitchWhen="A">Var is A</div>
  <div *ngSwitchWhen="B">Var is B</div>
  <div *ngSwitchDefault>Var is something else</div>
</div>
```

然后，如果我们要处理新的值，我们插入一行：

```
<div class="container" [ng-switch]="myVar">
  <div *ngSwitchWhen="A">Var is A</div>
```

```

<div *ngSwitchWhen="B">Var is B</div>
<div *ngSwitchWhen="C">Var is C</div>
<div *ngSwitchDefault>Var is something else</div>
</div>

```

并且我们不需要为default设置条件

`ngSwitchDefault` 是可选的，如果不使用，当以上条件都不满足时，就不会有元素被渲染

在不同元素上，你也可以声明相同的*ngSwitchWhen的值。下面是一个例子：

```

@Component({
  selector: 'switch-sample-app',
  template: `
    <h4 class="ui horizontal divider header"> Current choice is {{
choice }}
    </h4>
    <div class="ui raised segment">
      <ul [ngSwitch]="choice">
        <li *ngSwitchWhen="1">First choice</li>
        <li *ngSwitchWhen="2">Second choice</li>
        <li *ngSwitchWhen="3">Third choice</li>
        <li *ngSwitchWhen="4">Fourth choice</li>
        <li *ngSwitchWhen="2">Second choice, again</li>
        <li *ngSwitchDefault>Default choice</li>
      </ul>
    </div>
    <div style="margin-top: 20px;">
      <button class="ui primary button" (click)="nextChoice()">
        Next choice
      </button>
    </div>
  `
})

```

ngSwitchWhen的另一个特点是你不限于只匹配一个。例如,在上面的示例中choice是2时，第二和第五li将渲染。



kittencup

about 1 month ago

NgStyle

NgStyle 指令，可以通过angular表达式为DOM元素设置一个CSS属性。

使用该指令最简单的方法就是通过 `[style.<cssproperty>]="value"`

```
<div [style.background-color]='yellow'>
  Uses fixed yellow background
</div>
```

这段代码使用了NgStyle指令来设置元素的background-color为一个字符串yellow

使用NgStyle属性设置固定值的另一种方法是，使用{key:value}格式，比如这样：

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
  Uses fixed white text on blue background
</div>
```

注意ngStyle的规范中，我们给background-color括上单引号，而color没有，这是为什么？NgStyl的参数是一个Javascript对象，color是一个有效的key,不需要单引号，对于background-color, -字符不是一个有效的key,除非它是一个字符串，所以我们需要给它括上单引号

在这里我们设置的color和background-color属性。

ngStyle真正厉害之处是可以使用动态值

在我们的例子中，我们定义了2个输入框：

```
<div class="ui input">
  <input type="text" name="color" value="{{color}}" #colorinput>
</div>
<div class="ui input">
  <input type="text" name="fontSize" value="{{fontSize}}" #fontinput>
</div>
```

使用这2个输入框的值为三个元素设置CSS属性，在上面，我们根据输入值来设置字体大小

```
<div>
  <span [ngStyle]="{color: 'red'}" [style.font-
size.px]="fontSize">red text</span>
</div>
```

重要的是要注意，我们必须指定合适的单位，它是不是有效的CSS设置的12字体大小 - 我们必须指定一个单位，如12px或1.2em。

angular提供了一个方便的语法来指定单位：[style.font-size.px]。

该.px后缀表示，我们正在以像素为font-size属性值。你可以很容易地更换，通过[style.font-size.em]表示的字体大小使用em，甚至以百分比[style.font-size.%]。

另外2个元素使用#colorinput来设置文字和背景颜色

```
<h4 class="ui horizontal divider header">
  ngStyle with object property from variable
</h4>
<div>
  <span [ngStyle]="{color: colorinput.value}">{{ colorinput.value }}
text </span>
</div>
<h4 class="ui horizontal divider header">
  style from variable
</h4>
<div [style.background-color]="colorinput.value" style="color: white;">
  {{ colorinput.value }} background
</div>
```

这样，当我们点击Apply settings按钮时，我们调用了方法来设置新的值

```
apply(color, fontSize) {
  this.color = color;
  this.fontSize = fontSize;
}
```

这样,颜色和字体大小都将被应用到元素使用的NgStyle指令中。



kittencup

about 1 month ago

NgClass

NgClass指令，在你的HTML模板通过ngClass属性表示，允许您为一个给定的DOM元素动态地设置和改变CSS类。

如果你来自angular 1，该NgClass指令会感觉非常类似于angular 1中的ngClass指令。

第一种使用方式是通过对象字符串，对象用key作为类名称，value应该是一个true/false值，以指示该类是否应该应用。

假设我们有一个叫做bordered类，用来为元素增加了一个虚黑边框CSS

```
.bordered{
  border: 1px dashed black;
  border-color: 1px solid dashed;
}
```

让我们来添加2个div元素，一个总有bordered类(因此总有边框)，另一个则没有。

```
<div [ngClass]="{bordered: false}">This is never bordered</div>
<div [ngClass]="{bordered: true}">This is always bordered</div>
```

正如预期的那样，这两个div将这样渲染:

This is never bordered

This is always bordered

Simple class directive usage

当然，一个更为有有的是使用动态的赋值使NgClass指令创建class，为了使值动态，我们添加了一个变量作为对象的值，类似这样

```
<div [ngClass]="{bordered: isBordered}">
```



```
    This is a div with object literal. Border is {{ isBordered ? "ON"
: "OFF"}}}
</div>
```

或者，我们可以在组件中定义对象：

```
toggleBorder() {
    this.isBordered = !this.isBordered;
    this.classesObj = {
        bordered: this.isBordered
    };
}
```

直接使用对象：

```
<div [ngClass]="classesObj">
    Using object var. Border {{ classesObj.bordered ? "ON" : "OFF" }}
</div>
```

当你的类名包含破折号，类似于bordered-box，那你必须小心，JavaScript对象不允许键有破折号。如果你需要使用它们，你必须把键变为字符串像这样：

```
<div [ngClass]="{'bordered-box': false}">...</div>
```

```
<div class="base" [ngClass]="['blue', 'round']">
    This will always have a blue background and round corners
</div>
```

或在我们的组件中声明一个数组变量

```
this.classList = ['blue', 'round'];
```

并通过它：

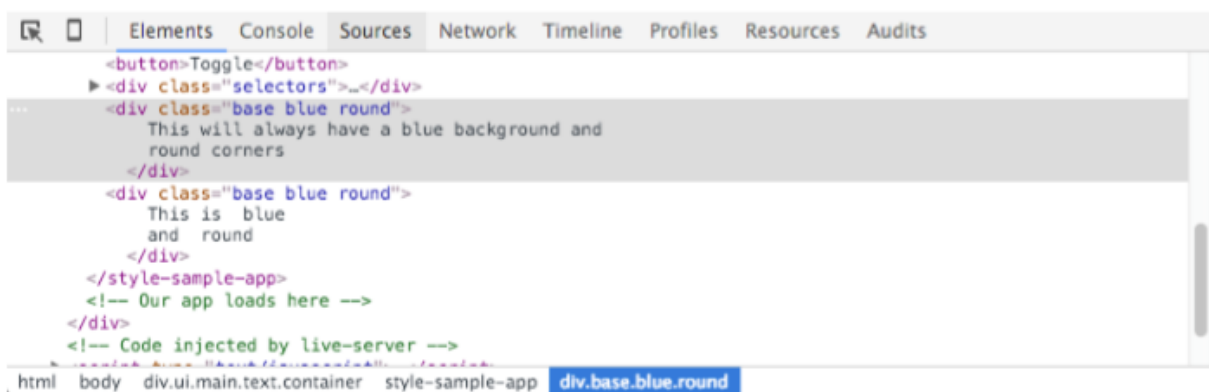
```
<div class="base" [ngClass]="classList">
This is {{ classList.indexOf('blue') > -1 ? "" : "NOT" }} blue
and {{ classList.indexOf('round') > -1 ? "" : "NOT" }} round
```

```
</div>
```

最后添加到元素的类，将会合并原先HTML中class属性设置的类及[ngClass]指令生成的类
在这个例子中：

```
<div class="base" [ngClass]="['blue', 'round']">
  This will always have a blue background and round corners
</div>
```

这个元素将会有3个类：HTML的class属性中设置base和[ngClass]指令中分配的blue和round



Classes from both the attribute and directive



kittencup

about 1 month ago

NgFor

该指令的作用是重复一个给定的DOM元素（或DOM元素的集合），通过一个数组为它遍历出不同的值。

该指令是 ng1 ng-repeat的继承者

语法是 `*ngFor="#item of items"`

- #item是模板变量，接受从items数组遍历出的每个元素

- 这个集合items来自于你的控制器中

为了说明这一点，我们可以看看代码示例。我们在我们组件控制器中声明一个cities数组控制器：

```
this.cities = ['Miami', 'Sao Paulo', 'New York'];
```

然后，在我们的模板中，我们可以有以下HTML代码：

```
<h4 class="ui horizontal divider header">
  Simple list of strings
</h4>
<div class="ui list" *ngFor="#c of cities">
  <div class="item">{{ c }}</div>
</div>
```

如你希望的那样，他将会在每个div中渲染city

Simple list of strings

Miami

Sao Paulo

New York

Result of the ng_for directive usage

我们也可以遍历这些对象数组：

```
this.people = [
  { name: 'Anderson', age: 35, city: 'Sao Paulo' },
  { name: 'John', age: 12, city: 'Miami' },
  { name: 'Peter', age: 22, city: 'New York' }
];
```

然后根据每一行的数据绘制一个table：

```
<table>
  <thead>
    <tr>
      <th>Name</th>
```

```

        <th>Age</th>
        <th>City</th>
    </tr>
</thead>
    <tr *ngFor="#p of people">
        <td>{{ p.name }}</td>
        <td>{{ p.age }}</td>
        <td>{{ p.city }}</td>
    </tr>
</table>

```

得到以下结果：

List of objects

Name	Age	City
Anderson	35	Sao Paulo
John	12	Miami
Peter	22	New York

Rendering array of objects

我们也可以使用嵌套的数组。如果我们想要上面的表一样,通过city分开数据,我们可以很容易地声明一个新数组的对象:

```

this.peopleByCity = [
  {
    city: 'Miami',
    people: [
      {name: 'John', age: 12},
      {name: 'Angel', age: 22}
    ]
  },
  {
    city: 'Sao Paulo',
    people: [
      {name: 'Anderson', age: 35},
      {name: 'Felipe', age: 36}
    ]
  }
];

```

然后我们可以使用NgFor为每个城市渲染1个div:

```
<div *ngFor="#item of peopleByCity">
  <h2 class="ui header">{{ item.city }}</h2>
```

并使用嵌套指令遍历了一个给定的城市的人们:

```
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
    </tr>
  </thead>
  <tr *ngFor="#p of item.people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
  </tr>
</table>
```

结果如下面的模板代码:

```
<h4 class="ui horizontal divider header">
  Nested data
</h4>
<div *ngFor="#item of peopleByCity">
  <div class="ui header">{{ item.city }}</div>

  <table class="ui celled table" >
    <thead class="ui header">
      <tr>
        <th>Name</th>
        <th>Age</th>
      </tr>
    </thead>
    <tr *ngFor="#p of item.people">
      <td>{{ p.name }}</td>
      <td>{{ p.age }}</td>
    </tr>
  </table>
</div>
```

它会为每个城市提供一个table：

Nested data

Miami

Name	Age
John	12
Angel	22

Sao Paulo

Name	Age
Anderson	35
Felipe	36

Rendering nested arrays

获取索引

有时候,当我们迭代数组时， 我们需要每一项的索引

我们可以通过从ngFor指令中附加语法#idx=index来获取索引， 当我们这样做， ng2将分配当前的索引给我们提供的变量（在此情况下， 是idx变量）。

注意， 如JavaScript， 索引始终是零开始。因此， 第一个元素的索引是0， 1等...

让我们改变我们的第一个例子， 添加 `#num = index` 代码：

```
<div *ngFor="#c of cities; #num = index">
  {{ num+1 }} - {{ c }}
</div>
```

它会在名字前加上城市的索引， 像这样：

1 - Miami

2 - Sao Paulo

3 - New York

Using an index



kittencup

about 1 month ago

NgNonBindable

当我们想告诉angular不编译或绑定我们的页面的特定部分,我们使用ngNonBindable

比方说, 我们要渲染在我们模板中的纯字符串 `{{content}}`。通常, 该文本将因为我们使用的 `{{ }}` 模板语法绑定到content变量的值。

那么怎样才能正确的渲染纯字符串 `{{content}}`? 我们使用ngNonBindable指令。

比方说, 我们希望有一个div渲染content变量和在右边的用来指出这一变量是通过`<- this is what {{content}} rendered`输出的

为此,模板中我们必须这样使用:

```
<div>
<span class="bordered">{{ content }}</span>
<span class="pre" ngNonBindable>
  <- This is what {{ content }} rendered </span>
</div>
```

使用ngNonBindable属性, ng2不会在该span范围内进行编译, 留下完好无损的原样:

```
Some text <- This is what {{ content }} rendered
```

Result of using ngNonBindable



kittencup

about 1 month ago

结论

angular 2只有几个核心指令，但我们可以结合这些简单的内容来创建动态应用程序

Comment on issue

Sign in to comment

or [sign up](#) to join this conversation on GitHub



Desktop version

 Open

Angular 2 中的表单 #39



kittencup opened this issue
about 1 month ago



ng-book 2

该issue关闭讨论，如有问题请去 [#43](#) 提问

目录

- 表单是重要的,表单是复杂的
- [Control](#)和[ControlGroup](#)
- 我们的第一个表单
- 使用FormBuilder
- 添加验证
- 观察变化
- [ngModel](#)
- 结束语



kittencup added the [ng-book 2](#) label about 1 month ago



kittencup locked this issue about 1 month ago



kittencup
about 1 month ago

表单是重要的,表单是复杂的

表单可能是web应用程序的最重要方面。虽然我们经常从点击链接或移动鼠标来获取事件，但是通过表单，我们可以得到来自用户的输入数据。

从表面上看，表单看起来很简单：你创建了一个input标签，用户填了它，点击提交。这有

多困难？

原来，表单也可以是非常复杂的。下面是几个原因：

- 表单输入意味页面和服务端都要修改数据
- 修改后经常需要在页面上的其他地方反映出修改的内容
- 用户在输入的过程中有很多问题，所以你需要验证用户输入
- 如果有期望和错误在用户界面需要清楚地显示
- 依赖字段可以有复杂的逻辑
- 我们希望能够测试我们的表单，而无需依赖于DOM选择器

值得庆幸的是，Angular 2可以帮助实现这些所有事情。

- **Controls** 在我们的form中封装了表单，提供给我们对象操作这些表单
- **Validators** 给我们希望的任何方式来验证表单的功能
- **Observers** 让我们看我们表单的变化，并相应地做出反应

在这一章中我们要一步一步构建表单，我们先从一些简单的表单，来建立更复杂的逻辑。



kittencup

about 1 month ago

Control和ControlGroup

在ng2 form中两个基本的对象是Control 和 ControlGroup.

####Control

一个Control代表着一个input字段，这是一个ng2表单的最小单位。

Control 封装了该字段的值，状态.. 例如，该字段是有效的，dirty（是否更改过），或有错误。

例如，我们可以在TypeScript里使用一个Control:

```
// create a new Control with the value "Nate"
var nameControl = new Control("Nate");
var name = nameControl.value; // -> Nate
// now we can query this control for certain values:
nameControl.errors // -> StringMap<string, any> of errors
```

```
nameControl.dirty // -> false
nameControl.valid // -> true
```

我们创建一系列Control并为他们附加元数据和逻辑来建立表单

在Angular中像这样的事情很多，我们有一个类(Control，在本例中)，我们附加到DOM属性上（ng-control,在本例中）。例如，在表单中可能如下：

```
<!-- part of some bigger form -->
<input type="text" ng-control="name" />
```

在我们的表单上下文中这将创建一个新的Control对象。我们将在下面进一步讨论如何工作。

####ControlGroup

大多数表单会有多个字段,所以我们需要一种方法来管理多个Control。

如果我们想要对我们的表单有效性进行检查，遍历一个数组并检查每个Control的有效性是繁琐的。

ControlGroups为Control集合提供一个封装接口来解决这个问题。

下面是如何创建一个ControlGroup:

```
var personInfo = new ControlGroup({
  firstName: new Control("Nate"),
  lastName: new Control("Murray"),
  zip: new Control("90210")
})
```

ControlGroup和Control有一个共同的父类([AbstractControl14](#))

这意味着我们检查status和personInfo的值就好像Control那样简单

```
personInfo.value; // -> {
  //  firstName: "Nate",
  //  lastName: "Murray",
  //  zip: "90210"
  // }
  // now we can query this control group for certain values, which
  have sensible
  // values depending on the children Control's values:
```

```
personInfo.errors // -> StringMap<string, any> of errors
personInfo.dirty // -> false
personInfo.valid // -> true
// etc.
```

注意当我们试图从ControlGroup获取value时，我们会收到了一个key-value结构的对象，这是一个很好的方式，从我们的表单获得全部的value无需在遍历每个单独Control。



kittencup

about 1 month ago

我们的第一个表单

有很多种方式来创建一个表单，有几个重要的，我们还没有谈到。让我们跳到一个完整的例子，我将解释每一块。

您可以在代码下载中找到该节的全部代码列表 forms

下面是我们要构建的第一个表单的截图：

image

在我们想象中的应用，我们正在创建一个e-commerce-type网站。在这个应用程序需要存储产品的SKU，让我们创建一个简单的表单，以SKU作为唯一的输入字段。

SKU是“库存单位缩写”。它的一个术语，一个要跟踪产品库存的唯一ID。当我们谈论一个SKU，我们谈论的是一个人类可读的id。

我们的表单超级简单:我们有一个SKU的输入框和一个提交按钮，让我们把这种形式转化为一个组件。如果你还记得,一个组件有三个部分:

- **@Component()** 注解
- 创建模板
- 在定义的组件类中实现自定义的功能

让我们来看看

简单的SKU表单:@Component() 注解

```
import { Component } from 'angular2/core';
import { FORM_DIRECTIVES } from 'angular2/common';
@Component({
  selector: 'demo-form-sku',
  directives: [FORM_DIRECTIVES],
```

在这里我们只定义了selector。selector会告诉Angular这个组件将绑定到哪个元素。在本例中，我们可以通过demo-form-sku标签来使用这个组件：

```
<demo-form-sku></demo-form-sku>
```

接下去我们定义在视图中要使用的directives,注意在这有一些关于注入有趣的东西，我们注入一种叫做FORM_DIRECTIVES。FORM_DIRECTIVES是一个常量，Angular为我们提供了一个几个指令的快捷常量，对于form都是非常有用的。FORM_DIRECTIVES包含了：

- ngControl
- ngControlGroup
- ngForm
- ngModel

等，[更多的指令](#)。

我们还没有讨论如何使用这些指令或他们做什么，但很快我们会知道。现在只需知道，注入form_directives，这意味着我们在视图中可以使用任何form_directives里包含的指令。

简单的SKU表单: 模板

让我们看一下模板：

```
@Component({
  selector: 'demo-form-sku',
  directives: [FORM_DIRECTIVES],
  template: `
    <div class="ui raised segment">
      <h2 class="ui header">Demo Form: Sku</h2>

      <form #f="ngForm"
        (ngSubmit)="onSubmit(f.value)"
        class="ui form">
        <div class="field">
          Forms in Angular 2 125
```

```

        <label
          <input
        </div>
        for="skuInput">SKU</label> type="text"
        id="skuInput" placeholder="SKU" ngControl="sku">
        <button type="submit" class="ui button">Submit</button>
      </form>
    </div>
  ` })

```

form & NgForm

现在事情变得有趣：因为我们注入FORM_DIRECTIVES指令，使ngForm可供我们的视图使用。

请记住，每当我们把指令提供给我们视图时，他们将连接到任何匹配他们选择器的元素。

NgForm做了一些不明显的事：自动附加在选择器元素中的form上(而不需要您在form标签上添加ngForm作为一个属性)，

这意味着如果你注入FORM_DIRECTIVES,在你的视图中NgForm会自动附加到任何

标签。这是非常有用的，但可能混淆，因为它发生在幕后。

NgForm提供给我们两个重要功能：

1. 一个名为ngForm的ControlGroup
2. 一个submit的事件

在我们的视图中的标签你可以看到这些

```
<form #f="ngForm" (submit)="onSubmit(f.value)">
```

首先我们有 `#f="ngForm"`，如果你还记得的 `#v=thing` 语法，我们要创建视图的一个局部变量。

所以我们所做的是为我们的视图中ngForm对象创建一个f变量的别名，这个form是从哪里来的呢？它是来自NgForm指令。

那这个ngForm对象是什么类型的？是ControlGroup类型,这意味着我们在视图中的ControlGroup可以使用f变量来使用，而这就被我们用在(submit)事件中

细心的读者可能会注意到，我在上面说到，NgForm会自动附加到标签上(因为NgForm指令selector是form,ngForm)，这意味着我们不需要添加NgForm属性来使用NgForm,但是在这里我们将ngForm添加到属性上。这是笔误？

不不是笔误，如果ngForm是该属性的键，那么会告诉Angular,我们想在这个属性上使用NgForm,在这里，ngForm是一个#f属性的值，我们只是将ngForm分配给局部模板变量f

我们在form中绑定submit事件使用的语法是： `(submit)="onSubmit(f.value)"` .

- (submit) - 来自NgForm
- onSubmit() - 将在我们的组件定义类里面实现
- f.value - f是我们上面指定的表单ControlGroup。value会返回这个ControlGroup的所有Control组成的key-value形式的对象

把这些结合来说“当我提交表单，将整个form中包含的值传给在组件中定义的onSubmit方法”。

input & NgControl

在我们讨论NgControl前在说下关于input标签的几件事

code/forms/adding_products_simple/forms/demo_form_sku.ts

```
<form #f="ngForm"
      (ngSubmit)="onSubmit(f.value)"
      class="ui form">

  <div class="field">
    <label for="skuInput">SKU</label>
    <input type="text"
           id="skuInput"
           placeholder="SKU"
           ngControl="sku">
  </div>
```

- `class="ui form"` 和 `class="field"` 这两个class是可选的，他们是CSS framework Semantic UI里的。我把他们加在我们的例子里，只是给他们一个更好的样式，但他们不是属于Angular的CSS部分。
- label "for" 属性和input "id" 属性互相匹配
- 我们设置了placeholder为"SKU" 这个只是当你input为空时给的输入提示

NgControl指令通过`ng-control`为其指定了一个选择器，这意味着，我们可以通过添加这类属性来连接我们的input标签：`ngControl="whatever"`，在本例中，我们说：`ngControl="sku"`。

NgControl 与 `ngControl`: 有什么区别？一般情况下，当我们使用帕斯卡(Pascal Case)，像NgControl，代码定义为我们指定的类，或指令，被指的对象。用小写（又名CamelCase），像`ngControl`，来自于指令中的选择器,它只是用于DOM /模板。

同样值得指出NgControl和Control是独立的对象,NgControl是视图上的指令，而Control是用于表示的数据和验证的对象。

值得指出的是,ngControl具体定义在类`NgControlName`(而不是在NgControl)。

NgControl创建一个新的Control自动添加到其父ControlGroup(在这里的代码，是Form)并且为这个Control绑定DOM元素。那就是说，它在视图中得input和Control之间建立了一种关联，这种关联通过一个名称相匹配，在这里，它叫sku

NgControl必须作为NgForm（或NgFormModel，我们下面会讨论到）的孩子。不要尝试使用没有NgForm的NgControl，这会出现问题。

####简单的 SKU 表单: 组件定义的类

现在来看下我们定义的类

```
export class DemoFormSku {
  onSubmit(value) {
    console.log('you submitted value: ', value);
  }
}
```

在这里我们在类中定义了一个方法:onSubmit,当我们提交整个Form时，这个方法会被调用，现在我们在这里通过console.log输出我们value

尝试它

把上面代码都放在一起，在这里我们的代码看起来像：

```
import { Component } from 'angular2/core';
import { FORM_DIRECTIVES } from 'angular2/common';
```



```

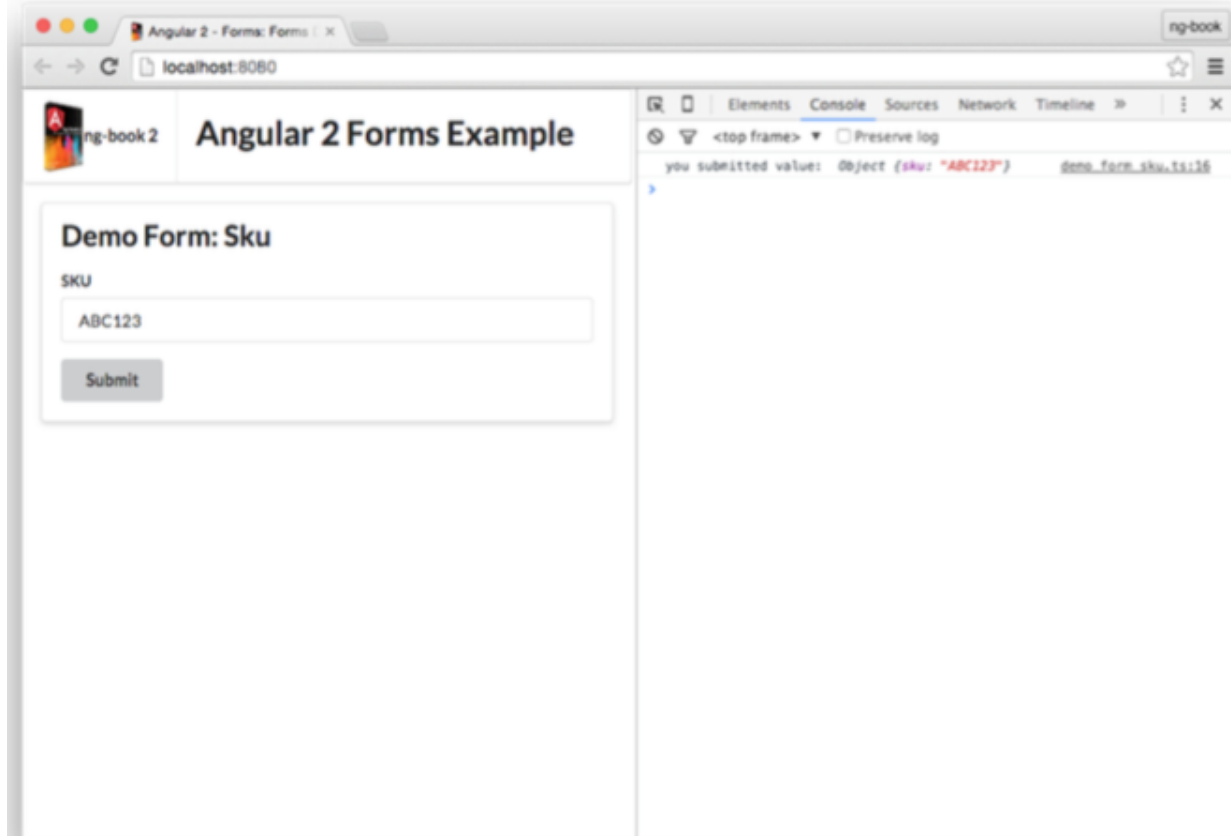
@Component({
  selector: 'demo-form-sku',
  directives: [FORM_DIRECTIVES],
  template: `
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: Sku</h2>
  <form #f="ngForm"
    (ngSubmit)="onSubmit(f.value)"
    class="ui form">

    <div class="field">
      <label for="skuInput">SKU</label>
      <input type="text"
        id="skuInput"
        placeholder="SKU"
        ngControl="sku">
    </div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
`
})
export class DemoFormSku {
  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}

```

如果我们尝试浏览器运行,这里运行的结果看起来应该是这样的:



kittencup

about 1 month ago

使用 FormBuilder

通过使用 `ngForm` 和 `ngControl` 方便的隐式的构建 `Control` 和 `ControlGroups`，但并没有给我们提供了更多定制选项。

使用 `FormBuilder` 是一种更灵活和通用的方式来配置表单。

`FormBuilder` 是一个名副其实的助手类，帮助我们构建表单。你应该记得，表单是 `Control` 和 `ControlGroup` 组成，`FormBuilder` 帮助我们创建他们（你可以认为它是一个“工厂”对象）。

让我们用先前的例子来添加一个 `FormBuilder`，让我们看看：

- 如何在我们的组件定义类的使用 `FormBuilder`
- 如何在视图表单中使用我们自定义的 `ControlGroup`

使用 FormBuilder

注入意味着什么？我们没有太多讨论过Di或di是如何去涉及到层次树的，所以上面的话可能没有太多的意义。如果你想深入学习更多,我们会在组件章节中讨论更多关于依赖关系的部分，

在更高层次上，依赖注入是一种方式来告诉这个angular组件正常运行需要依赖什么。

```
export class DemoFormSkuBuilder {
  myForm:ControlGroup;

  constructor(fb:FormBuilder) {
    this.myForm = fb.group({
      'sku': ['ABC123']
    });
  }

  onSubmit(value:string):void {
    console.log('you submitted value: ', value);
  }
}
```

FormBuilder的一个实例将被创建，我们把它分配给fb变量（在构造函数中）

还有，我们将在FormBuilder使用的两个主要功能：

- control - 创建一个新的Control
- group - 创建一个新的ControlGroup

注意，我们在该类中创建了一个myForm变量来保存ControlGroup实例(我们可能很容易地把它称为表单，但我想要区分ControlGroup和以前的表单)

myForm是一个ControlGroup类型的。我们通过fb.group()方法创建了一个ControlGroup，使用key-value形式的对象作为group的参数。

在这种情况下，我们设置了一个sku Control,它的值是['ABC123'], 这说明这个Control的默认value是"ABC123"(这里使用的时数组，那是因为我们后面会在后面添加更多的配置项)

现在我们需要在视图中使用我们的myForm(我们需要把它绑定到我们的表单元素上)

在视图上使用myForm

我想让

标签使用myForm, 如果你还记得,在上一节我们说当我们使用FORM_DIRECTIVES会自动将绑定ngForm。我们还提到ngForm会创建它自己的ControlGroup。

在这种情况下,我们不希望使用FORM_DIRECTIVES来创建ControlGroup。

相反, 我们希望用我们的FormBuilder创建的实例变量myForm的。我们怎样才能做到这一点?

Angular 提供了另一个指令时,可以让我们使用现有的ControlGroup:它叫NgFormModel, 我们这样使用它:

```
<form [ngFormModel]="myForm" (submit)="onSubmit(myForm.value)">
```

这里我们告诉Angular,我想使用myForm作为这个表单的ControlGroup。

还记得我们早些时候说,当使用FORM_DIRECTIVES NgForm将自动应用于元素。有一个例外:NgForm不会应用到有ngFormModel属性的元素。

如果你好奇, NgForm这个选择器

是: `form:not([ngNoForm]):not([ngFormModel]),ngForm,[ngForm]` 这意味着你可以以为form使用一个 `ngNoForm` 属性, 产生一个没有NgForm的表单

我们也需要将onSubmit中的f改变为myForm,因为现在myForm变量才保存着整个表单的配置和值

我们需要做的最后一件事是: 将我们的Control绑定在一个input标签上.记住NgControl会创建一个新的Control对象, 被附加到它的父ControlGroup中。但在这里, 我们使用FormBuilder去创建我们自己的Controls

当我们想将现有Control绑定到input,我们可以使用NgFormControl:

code/forms/adding_products_simple/forms/demo_form_sku_with_builder.ts

```
<input
  type="text"
  id="skuInput"
  placeholder="SKU"
  [ngFormControl]="myForm.controls['sku']">
```

在这里，我们使用NgFormControl指令来指示这个input标签使用myForm.controls中得sku Control

尝试运行它

将上面所有代码连在一起看起来是:

```
import { Component } from 'angular2/core';
import {
  FORM_DIRECTIVES,
  FormBuilder,
  ControlGroup
} from 'angular2/common';

@Component({
  selector: 'demo-form-sku-builder',
  directives: [FORM_DIRECTIVES],
  template: `
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: Sku with Builder</h2>
  <form [ngFormModel]="myForm"
    (ngSubmit)="onSubmit(myForm.value)"
    class="ui form">

    <div class="field">
      <label for="skuInput">SKU</label>
      <input type="text"
        id="skuInput"
        placeholder="SKU"
        [ngFormControl]="myForm.controls['sku']">
    </div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
`
})
export class DemoFormSkuBuilder {
  myForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['ABC123']
    });
  }

  onSubmit(value: string): void {
```

```
console.log('you submitted value: ', value);
```

```
}  
}
```

记住:

要隐式的创建一个新的ControlGroup和Controls,使用ng-form和ng-control

要绑定自己的ControlGroup和Controls使用ng-form-model和ng-form-control



kittencup

about 1 month ago

添加验证

我们的用户并不总是以正确的格式输入数据。如果有人输入错误的格式的数据，我们希望给他们反馈，而不允许提交表单。为此我们使用验证。

验证通过Validators模块，并提供最简单的验证是Validators.required，它只是说，指定的字段是必需的，否则Control将被视为无效。

使用验证器，我们需要做两件事情

- 分配给Control对象一个validator
- 在视图中检查验证器的状态,并相应地采取行动

分配给Control对象一个validator，我们可以简单的通过实例化Control时的第2个参数:

```
var control = new Control('sku', Validators.required);
```

在我们的例子中，因为我们使用的时 FormBuilder 我们将使用下面的语法:

```
this.myForm = fb.group({  
  "sku":  ["", Validators.required]  
});
```

现在我们需要在视图中使用我们的验证器，有两种方法使我们可以在视图中访问验证值:

- 我们可以在类中明确分配变量给sku Control,这样视图使用起来较为方便
- 我们可以在视图中在myForm中查找sku Control,这样在类中定义就少，但是在视图中，

为了使这种差异更加清晰，让我们来看看两种方式的例子：

明确设定sku Control作为一个实例变量

这里有一个截图表示验证我们的表单将是什么样子：

image

在您的视图中，最灵活的方式处理单个控件的方法是将每个控件设为您的组件定义类中的实例变量。在这里我们可以在类中设置sku：

```
export class DemoFormWithValidationsExplicit {
  myForm: ControlGroup;
  sku: Control;
  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      "sku": ["", Validators.required]
    });
    this.sku = this.myForm.controls['sku'];
  }
  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}
```

注意：

- 我们在类中创建sku:AbstractControl变量
- 通过FormBuilder创建的MyFrom变量中的sku Control分配给this.sku变量

这样的好处是因为这意味在我们的组件视图中任何地方都可以使用sku这个变量。

不利的一面是，通过做这种方式，我们不得不在我们的表单中的每个字段都分配一个实例变量。对于较大的表单来讲，这样代码显得相当冗长。

现在我们的sku可以被验证，在我们的视图时，我想通过四个不同方面来看看这个验证

- 检查我们的整个表单的有效性并显示一条消息
- 检查我们的单个字段的有效性并显示一条消息
- 检查我们的单个字段的有效性如果是无效的显示红色
- 检查我们的单个字段的有效性并显示特定的提示

表单信息

通过myForm.valid, 我们可以检查我们整个表单的有效性:

```
<div *ngIf="!myForm.valid" class="ui error message">Form is  
invalid</div>
```

记住, 如果myForm是一个ControlGroup, 如果ControlGroup有效, 那么他得所有子Control也肯定是有有效的

字段信息

如果字段Control是无效的, 我们可以显示一个特定的信息

```
<div *ngIf="!sku.valid" class="ui error message">SKU is invalid</div>
```

字段颜色

我使用Semantic UI CSS Framework's CSS 的.error类, 意味着如果我对.form-group类添加一个.error类的话, 这个表单会显示出一个红色的边框

要做到这一点,我们可以使用属性语法设置条件类:

```
<div class="field" [class.error]="!sku.valid && sku.touched">
```

注意在这里我们设置.error类有2个条件, 我们会检查!sku.valid 和 sku.touched.

如果我们有使用过表单并且表单无效, 则说明该表单是验证失败

特定的验证

一个表单字段可以有很多原因是无效的。我们经常想显示一个不同的消息, 这取决于一个失败的验证的原因。

要查找特定的验证失败信息, 我们用hasError方法:

```
<div *ngIf="sku.hasError('required')" class="ui error message">SKU is  
required</div>
```


主要注意的是在Control和ControlGroup中都有hasError方法。这意味着在ControlGroup中我们可以通过第2个参数来搜索特性字段的错误信息。

例如，在前面例子写成：

```
<div *ngIf="myForm.hasError('required', 'sku')" class="error">SKU is requi
```

###将他们连起来

下面是我们form与验证控件的完整代码：

```
/* tslint:disable:no-string-literal */
import { Component } from 'angular2/core';
import {
  FORM_DIRECTIVES,
  FormBuilder,
  ControlGroup,
  Validators,
  AbstractControl
} from 'angular2/common';

@Component({
  selector: 'demo-form-with-validations-explicit',
  directives: [FORM_DIRECTIVES],
  template: `
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: with validations (explicit)</h2>
  <form [ngFormModel]="myForm"
    (ngSubmit)="onSubmit(myForm.value)"
    class="ui form">

    <div class="field"
      [class.error]="!sku.valid && sku.touched">
      <label for="skuInput">SKU</label>
      <input type="text"
        id="skuInput"
        placeholder="SKU"
        [ngFormControl]="sku">
      <div *ngIf="!sku.valid"
        class="ui error message">SKU is invalid</div>
      <div *ngIf="sku.hasError('required')"
        class="ui error message">SKU is required</div>
      </div>

    <div *ngIf="!myForm.valid"
      class="ui error message">Form is invalid</div>
  </form>
</div>`
})
```

```

        <button type="submit" class="ui button">Submit</button>
    </form>
</div>
`
}))
export class DemoFormWithValidationsExplicit {
    myForm: ControlGroup;
    sku: AbstractControl;

    constructor(fb: FormBuilder) {
        this.myForm = fb.group({
            'sku': ['', Validators.required]
        });

        this.sku = this.myForm.controls['sku'];
    }

    onSubmit(value: string): void {
        console.log('you submitted value: ', value);
    }
}

```

不明确设定sku Control作为一个实例变量

正如我们在上一节中提到的,在表单中不必为每个输入标签创建一个实例变量

我们可以为每个Control不创建一个实例变量?我们可以,尽管有一些权衡。

首先,让我们再看看组件定义类:

```

export class DemoFormWithValidationsShorthand {
    myForm: ControlGroup;

    constructor(fb: FormBuilder) {
        this.myForm = fb.group({
            'sku': ['', Validators.required]
        });
    }

    onSubmit(value: string): void {
        console.log('you submitted value: ', value);
    }
}

```

请注意,我们已经删除了sku:Control。

让我们看看之前的三个验证,在这里发生了什么变化:

声明一个局部的sku引用

因为我们没有公开的sku Control作为一个实例变量,我们现在需要获得它的引用。我们可以有2种方法获得它:

- 通过myForm.find
- 通过ngFormControl指令

通过myForm.find设置字段颜色

ControlGroup有一个find方法,允许你根据名字找出所有的子Control,在这里我们可以找出sku Control,用来检查他得有效性或表单是否触碰过(对于input来说是否有获取过焦点)

```
<div class="field" [class.error]="!myForm.find('sku').valid &&  
myForm.find('sku').touched">
```

这是一个比之前更冗长的代码,但这并不太坏。

从NgFormControl输出form

有另一种方式,我们从Control得到一个引用,这个引用是通过NgFormControl指令来导出的。这是一个新的概念,我们至今没有涉及:

组件可以导出一个引用,这样你就可以在视图中使用它们了。

(我们会在组件如何使用exportAs这一章讨论,但是现在,只知道许多内置组件已经这样做了。)

在这种情况下, NgFormControl可以export它自己的ngForm,您可以使用使用#reference语法来获取该export。这就是它看起来像:

code/forms/adding_products_simple/forms/demo_form_with_validations_shorthand.ts

```
<input  
  type="text"  
  id="skuInput"  
  placeholder="SKU"
```

```
#sku="ngForm"
[ngFormControl]="myForm.controls['sku']">
```

这是使NgFormControl指令本身在视图中提供的sku变量，但请注意，这是指令，而不是Control。访问sku Control我们现在必须使用sku.control。

这个值得在说一遍，当我们#sku="form"引用NgFormControl指令时，sku指的是这个指令的实例，要从sku获取Control，必须通过sku.control

现在已经有sku提供给我们,我们可以检查的有效性和错误如下:

```
<div *ngIf="!sku.control.valid" class="ui error message">SKU is
invalid</div>
<div *ngIf="sku.control.hasError('required')" class="ui error
message">SKU is required</div>
```

本地引用的sku作用域

当我们使用#reference语法创建了一个本地引用，他只能用于相邻或者子元素，不能用于父元素

例如，我们可以尝试这样:

```
// this won't work
<div class="form-group" [class.has-error]="!sku.control.valid &&
sku.control.touched">
```

为什么不能用？这个.form-group是input元素的父元素

将他们连起来

```
import { Component } from 'angular2/core';
import {
  FORM_DIRECTIVES,
  FormBuilder,
  ControlGroup,
  Validators
} from 'angular2/common';

@Component({
```

```

selector: 'demo-form-with-validations-shorthand',
directives: [FORM_DIRECTIVES],
template: `
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: with validations (shorthand)</h2>
  <form [ngFormModel]="myForm"
    (ngSubmit)="onSubmit(myForm.value)"
    class="ui form">

    <div class="field"
      [class.error]="!myForm.find('sku').valid &&
myForm.find('sku').touched">
      <label for="skuInput">SKU</label>
      <input type="text"
        id="skuInput"
        placeholder="SKU"
        #sku="ngForm"
        [ngFormControl]="myForm.controls['sku']">
      <div *ngIf="!sku.control.valid"
        class="ui error message">SKU is invalid</div>
      <div *ngIf="sku.control.hasError('required')"
        class="ui error message">SKU is required</div>
    </div>

    <div *ngIf="!myForm.valid"
      class="ui error message">Form is invalid</div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
`
  })
}

export class DemoFormWithValidationsShorthand {
  myForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['', Validators.required]
    });
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}

```

我们经常会想写自己的自定义验证。让我们来看看如何做到这一点。

验证是如何实现的,让我们从Angular核心源码来看看Validators.required

```
export class Validators{
    static required(c: Control): StringMap<string, boolean> {
        return isBlank(c.value) || c.value == "" ? {"required": true} :
null;
    }
}
```

一个验证器 - 会有一个Control参数传入, 必须返回一个String类型, key为"error code" value为boolean值, 为真则表示无效的

编写一个Validators

比方说, 我们对SKU有个具体要求。例如, 假设我们的SKU需要已123开始.我们可以写一个验证器, 像这样:

```
function skuValidator(control) {
    if (!control.value.match(/^123/)){
        return {invalidSku: true};
    }
}
```

如果control.value不已123开始, 这个验证将返回错误代码invalidSku;

分配一个Validators给Control

现在, 我们需要添加验证到我们的Control。然而, 有一个小问题: 我们已经有一个sku验证。我们怎样才能添加多个验证到Control?

为此, 我们使用Validators.compose:

```
this.myForm = fb.group({
    "sku":  ["", Validators.compose([
        Validators.required, skuValidator])]
});
```

Validators.compose包装了我们两个验证器并分配给Control。所有的验证都是有效地

Control才是有效地

现在我们可以视图使用新的验证器:

```
<div *ng-if="sku.hasError('invalidSku')" class="bg-warning">SKU must  
begin with <tt>123</tt></div>
```

请注意，对于这一节，我使用“显式”为每个Control添加一个实例变量。这意味着，在这一节中，sku是指Control。

如果你尝试示例代码，一个绝妙的事情，你会注意到的是，如果你输入些内容到字段中，required验证将有效，但invalidSku验证可能是无效的。这太棒了 - 这意味着我们可以局部验证我们的字段，并显示相应的信息。



kittencup

about 1 month ago

观察变化

到目前为止我们只有在表单提交时通过调用onSubmit方法来提取form的值，但我们经常想看一个Control的变化。

ControlGroup和Control都提供了一个EventEmitter，我们可以用来观察变化

EventEmitter是一个Observable，这意味着它符合观察变化的定义规范。如果你有兴趣了解Observable规范，[你可以从这找到它](#)

我们在一个Control中作观察变化:

- 通过调用control.valueChanges 我们获得的EventEmitter
- 使用.observer()方法添加一个observer

下面是一个例子:

```
this.myForm = fb.group({  
  'sku': ['', Validators.required]  
});  
this.sku = this.myForm.controls['sku'];  
this.sku.valueChanges.subscribe((value:string) => {
```

```
        console.log('sku changed to: ', value);
    }
};
this.myForm.valueChanges.subscribe((value:string) => {
    console.log('form changed to: ', value);
});
```

在这里，我们观察了两个独立的事件：sku字段的变化及整个form的变化

我们通过给observable传递一个对象，在这里只有一个next键(还可以有其他键，在现在我们会去关心那些)，next对应的函数就是当我们观察的东西有一个新的变化时会调用的函数

如果我们输入'kj'到input表单中，我们可以在console中看到输出的内容

```
sku changed to:  k
form changed to:  Object {sku: "k"}
sku changed to:  kj
form changed to:  Object {sku: "kj"}
```

你会看到每一次按键使Control改变，，我们的observable会被触发，当我们观察单个字段时，你可以看到一个值('kj')

当我们观察整个表单时，我们可以获取一个key-value结构的对象({sku:'kj'})



kittencup

about 1 month ago

ngModel

ngModel是一个特殊的指令，它将一个model绑定到一个表单中。ngModel是特殊的，它自己实现了双向数据绑定。

双向数据绑定对于单向数据绑定比几乎总是更复杂和困难。Angular 2普遍地建立是数据单向流:自顶向下。然而,当谈到form,在一些情况下,更容易选择双向绑定。

不要因为你过去在Angular 1中使用ngModel，现在就急于马上使用ngModel。有很好的理由避免双向数据绑定。当然,ngModel可以非常方便,但就像我们在Angular 1中，我们不一定依赖于它

让我们换个form，我们要输入产品名称。我们将使用ngModel来保持组件实例变量与视图同步。

首先，我们定义组件类

```
export class DemoFormNgModel {
  myForm:ControlGroup;
  productName:string;

  constructor(fb:FormBuilder) {
    this.myForm = fb.group({
      'productName': ['', Validators.required]
    });
  }

  onSubmit(value:string):void {
    console.log('you submitted value: ', value);
  }
}
```

接下去，我们在input标签中使用ngModel

code/forms/adding_products_simple/forms/demo_form_ng_model.ts

```
<input class="form-control"
      id="productNameInput"
      placeholder="Product Name"
      [ng-form-control]="myForm.find('productName')"
      [(ngModel)]="productName">
```

现在注意到的东西:ngModel的语法是有趣的：我们在ngModel上使用括号和圆括号！实现的思想是该ngModel即是属性()又是事件[]。这是一个双向绑定的指示。

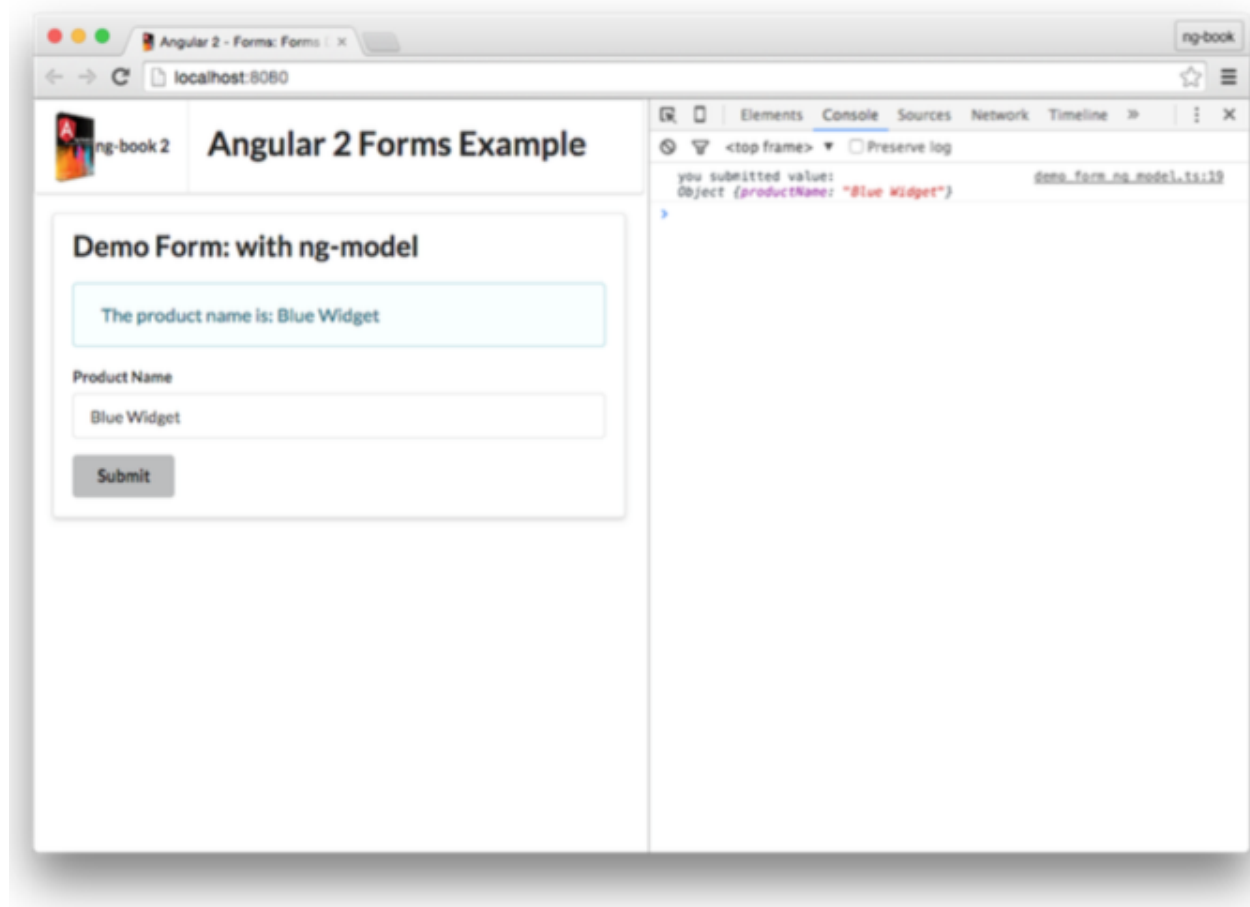
这里还有一些其他的東西：我們仍然在使用ngFormControl来指定这个input必须绑定到我们form上的Control。我们这样做，是因为ngModel只对实例变量的输入有约束，Control是完全分离的。但是，因为我们仍然要验证这个值，并将它作为表单的一部分提交，我们将保持ngFormControl指令。

最后，让我们在视图中显示我们的产品名称：

code/forms/adding_products_simple/forms/demo_form_ng_model.ts

```
<div class="ui info message">
  The product name is: {{productName}}
</div>
```

这里它看起来像:



Demo Form with ngModel



kittencup

about 1 month ago

结束语

Form有很多部件,但在Angular 2中是相当简单的。一旦你掌握如何使用ControlGroups,Control,和验证,它很容易使用!

Sign in to comment

or [sign up](#) to join this conversation on GitHub



Desktop version

 Open

基于Observables的数据架构 第一部分:服务 #40



kittencup opened this issue
about 1 month ago



ng-book 2

该issue关闭讨论，如有问题请去 [#43](#) 提问

目录

- [Observables和RxJS](#)
- [聊天应用概述](#)
- [实现模型](#)
- [实现UserService](#)
- [MessagesService](#)
- [ThreadsService](#)
- [数据模型的总结](#)



kittencup added the [ng-book 2](#) label about 1 month ago



kittencup locked this issue about 1 month ago



kittencup
about 1 month ago

Observables和RxJS

在Angular中，我们可以使用Observables作为骨干构建我们应用的数据架构，使用Observables来组织我们的数据被称为响应式编程(Reactive Programming)。

但是到底什么是Observables，及什么是响应式编程呢？响应式编程是一种使用异步数据流来工作的方法,Observables就是我们用来实现响应式编程的主要数据结构。但是我承认，

这些术语可能不是那么明确。我们在这一章来看看具体的例子,应该更有启发性的。

注意:一些RxJS所需知识

我想指出的这本书并不会重点介绍响应式编程。还有其他一些很好的资源,可以教你响应式编程的基础知识,你应该阅读。我们在下面列出了一些。

考虑到这一章是如何使用RxJS和Angular,而不是一个面面俱到的介绍RxJS和响应式编程的教程。

在这一章中,我将详细讲解RxJS概念和我们遇到的API解释。但是要知道,如果RxJS对你来任然是新的东西,你可能需要在这里补充内容和其他资源。

在本章中使用Underscore.js

[Underscore.js](#)是一个流行的库,它提供了对于Javascript数据结构,类似于Array和Object的操作函数,我们在本章和一堆RxJS使用它,如果你在代码中看到,类似于.map或_.sortBy,需要知道我们使用的是Underscore.js库,你可以在这找到[Underscore.js](#)文档

学习反应式编程和RxJS

如果你刚开始学习RxJS我建议你先看看这篇文章:

- [The introduction to Reactive Programming you've been missing](#) by Andr   Staltz
[中文版本](#)

当你熟悉RxJS背后的概念后,这里有一些更多的链接,可以帮助您:

- [Which static operators to use to create streams?](#)
- [Which instance operators to use on streams?](#)
- [RxMarbles](#) - 关于流的各种操作的交互图

在这一章我将提供RxJS的API文档的链接。RxJS文档有很多很好的例子代码,揭示不同的流是如何工作的。

我必须在Angular 2中使用RxJS?- 不,你肯定不喜欢。Observables只是一个模式,你可以在Angular 2中使用更多的数据模式。

我想给你一个合理的警告:学习RxJS有点令人费解的。但是请相信我, 你会得到它的窍门, 这是值得的。以下是您可能会发现有关于流有用的一些想法:

1. ***Promise只发射一个单一值, 而stream发出许多值*** - 在你的应用里Stream与Promise充当同样的角色, 如果你从callback转为promise, 你知道promise对于callback来说可读性和数据维护有了很大的进步, 同样, 而stream改善promise模式后, 我们可以在一个stream上持续对数据的变化作出反应(vs. 从一个promise的一次resolve)。
2. ***命令代码"pull", 而反应式streams "push" 数据*** - 在反应式编程中, 我们的代码订阅变化的通知, 而stream "push" 数据给这些订阅
3. ***函数式的RxJS*** - 如果你是个函数式操作迷, 例如map, reduce,和filter, RxJS因为是stream, 你会感到宾至如归,在某种意义上, 列表等功能强大的函数式操作都适用
4. ***流是可组合的*** - 想一想, 在您的数据里像一个管道的操作。你可以订阅任何一部分的数据流, 甚至可以结合它们来创建新的数据流

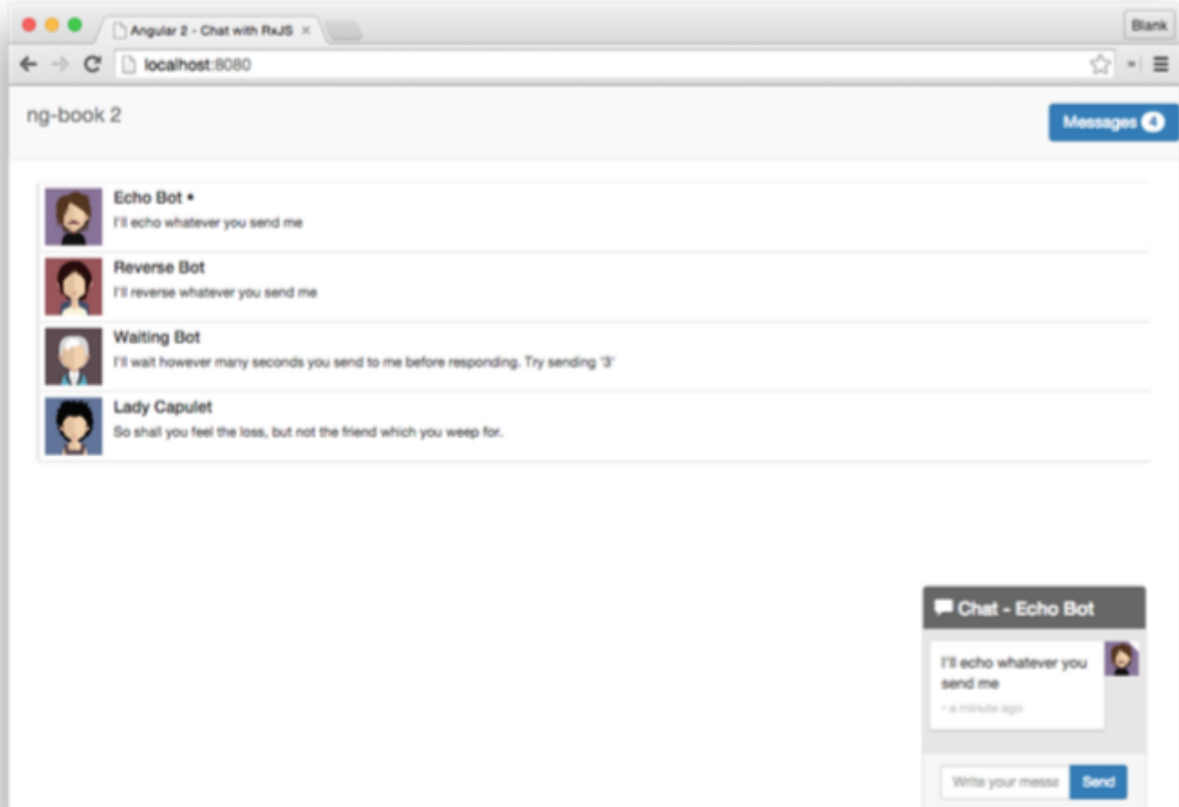


kittencup

about 1 month ago

聊天应用概述

在本章, 我们开始使用RxJS来构建一个聊天应用, 下面是它的截图



Completed Chat Application

通常我们试图在这本书内展示每一行代码。然而，这个聊天应用有许多组成部分，因此在本章中，我们不会有所有的代码。你可以在文件夹/rxjs/chat找到在本章中的聊天应用示例代码。

在这个应用中我们提供了一些机器人可以与你聊天。找到代码，并尝试它：

```
cd code/rxjs/chat
npm install
npm run go
```

现在打开浏览器访问<http://localhost:8080>。

如果这个URL出现问题，请尝试URL：<http://localhost:8080/webpack-dev-server/index.html>

请注意几件事情：

- 你可以点击主题帖与另一个人聊天
- 机器人会给你回短信，这取决于他们的个性
- 在右上角的未读邮件数与未读邮件的数量保持同步

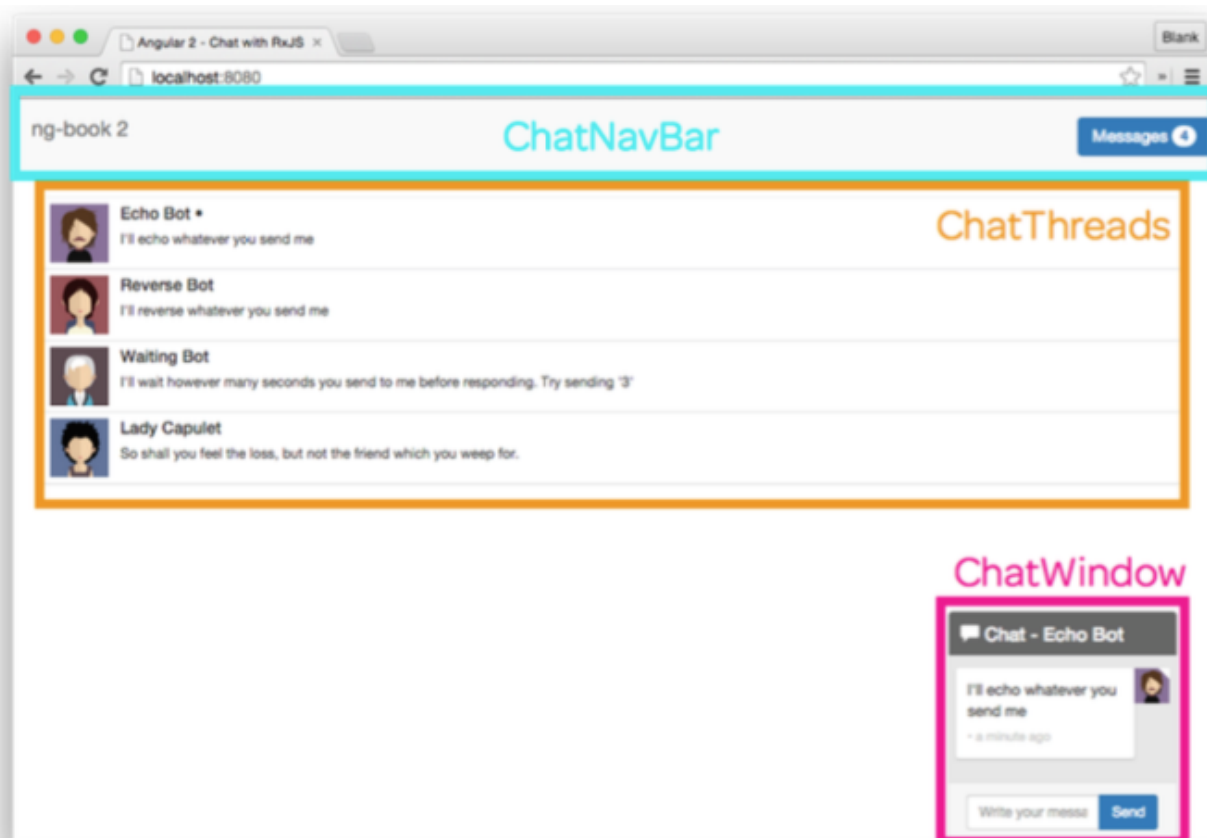
让我们来看看这个应用是如何构造的。我们有

- 3个顶级angular组件
- 3个model
- 3个service

让我们一起来看看它们:

组件

该页面被分解成三个顶级组件:



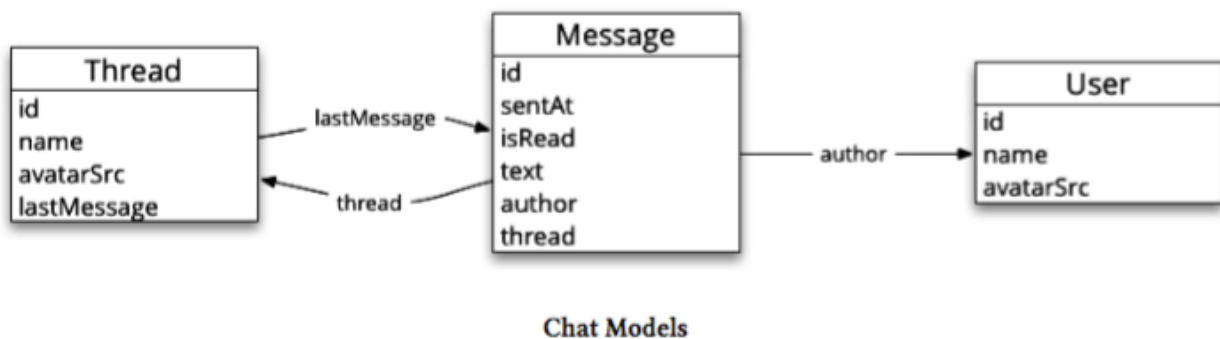
Chat Top-Level Components

- ChatNavBar - 包含未读邮件数
- ChatThreads - 显示可点击主题帖列表，以及最新的消息和聊天头像

- ChatWindow - 显示一个输入框，发送新消息

models

这个应用也有3个model



- User - 存储关于聊天参与者信息
- Message - 存储单条聊天信息
- Thread - 存储信息的集合以及关于对话中的一些数据

Services

在这个应用中，我们每一个model都有对应的service。service是一个单例对象，该对象扮演2个角色：

1. 提供我们的应用程序可以订阅的数据流
2. 提供添加或修改数据的操作

例如，UserService：

- 发布一个stream，用于发射当前用户
- 提供setCurrentUser功能，用来设置当前用户(也就是说，从currentUser流发射当前用户)

总结

在更高层次上，应用的数据结构是简单的：

- service保留流用来发射model(如 Messages)

- 组件订阅这些stream，根据最新的值进行渲染

例如，ChatThreads组件监听从ThreadService来的最新主题帖列表，ChatWindow订阅最新的消息列表

在本章的其余部分，我们要去深入了解我们如何实现这一点使用Angular 2和RxJS。我们将通过实现我们的model开始，然后看看我们是如何创建service来管理我们的流，然后最后实现了组件。



kittencup

about 1 month ago

实现model

让我们先从简单的东西开始，先来看看model

User

我们的User类很简单。我们有一个id，name，和avatarSrc

```
export class User {
  id: string;

  constructor(public name: string,
               public avatarSrc: string) {
    this.id = uuid();
  }
}
```

注意上面，我们在constructor中使用了一个TypeScript的简写语法，当我们声明 `public name: string`，我们会告诉TypeScript，1.我们要在这个类声明一个公开的name属性，2.当创建一个实例时，将参数分配给这个属性

Thread

同样，Thread 也是一个简单的TypeScript类：

```
export class Thread {
  id: string;
```

```

    lastMessage: Message;
    name: string;
    avatarSrc: string;

    constructor(id?: string,
                name?: string,
                avatarSrc?: string) {
        this.id = id || uuid();
        this.name = name;
        this.avatarSrc = avatarSrc;
    }
}

```

注意,在我们的Thread类里我们存储了lastMessage的对象。这让我们在主题帖列表中显示最新消息的预览。

Message

Message类也是个简单的TypeScript类,但是在这种情况下,我们使用一个稍微不同的构造函数形式:

```

export class Message {
    id: string;
    sentAt: Date;
    isRead: boolean;
    author: User;
    text: string;
    thread: Thread;

    constructor(obj?: any) {
        this.id = obj && obj.id || uuid();
        this.isRead = obj && obj.isRead || false;
        this.sentAt = obj && obj.sentAt || new Date();
        this.author = obj && obj.author || null;
        this.text = obj && obj.text || null;
        this.thread = obj && obj.thread || null;
    }
}

```

你看到这里的构造函数的模式允许我们使用关键字参数的构造函数来模拟。使用这种模式,我们可以使用任何可用的数据来创建一个新的消息,我们不必担心参数的顺序,例如,我们可以这样做:

```
let msg1 = new Message();

# or this

let msg2 = new Message({
  text: "Hello Nate Murray!"
})
```

现在，我们已经看到了我们的model，让我们来看看我们的第一个service：UserService。



kittencup

about 1 month ago

实现UserService

UserService的重点是提供了一个让我们应用了解当前的用户及如果当前用户有变更。则通知其余的应用。

我们需要做的第一件事是创建一个TypeScript类,使用**@Injectable**注解后，该类可被注入

```
# code/rxjs/chat/app/ts/services/UserService.ts

@Injectable()
export class UserService {
```

当我们创建一些injectable，则意味着在我们的应用中我们可以在其他组件中依赖它。简单的说，依赖注入的两大好处是 1.我们让angular来处理对象的生命周期,2.更容易测试注入的组件。

我们会在依赖注入这章更多的讨论**@Injectable**，现在在我们的组件中，我们可以作为一个依赖注入它，如下所示：

```
class MyComponent{
  constructor(public userService:UserService){
    // do something with `userService` here
  }
}
```

currentUser流

接下去 我们创建一个用来管理我们当前用户的流

```
# code/rxjs/chat/app/ts/services/UserService.ts

currentUser: Rx.Subject<User> = new Rx.BehaviorSubject<User>(null);
```

在这里有很多事情，让我们把它分解：

- 我们定义了一个实例变量currentUser，是一个Subject类型的流
- 具体而言,currentUser是一个BehaviorSubject，其中包含了User
- 然而，这个流的第一个值为null（构造函数参数）。

如果你还没有接触到过多RxJS，那么你可能不知道什么是Subject或BehaviorSubject。你可以把Subject认为是“读/写”流。

从技术上讲，一个Subject是从Observable和Observer继承的

流的一个相关问题是，由于当前用户是在订阅前被设置的，一个新的订阅会丢失在订阅前设置的内容。BehaviourSubject则弥补这一点。

BehaviourSubject有一个特殊的属性，它保存的最后一个值。这意味着任何订阅该流都将收到的上一次的值。这对我们很有用处，因为这意味着我们的应用任何地方都可以订阅UserService.currentUser流，并立即知道谁是当前用户。

设置新用户

我们需要一种方法用来只要当前用户的变化（如登录）就发布一个新用户到流。

有2种方法,我们可以公开一个接口来做这个

1. 直接将新用户添加到stream:

更新当前用户的最直接的方法是在客户端使用UserService只需简单发布新的用户到流,像这样：

```
userService.subscribe((newUser) => {
  console.log('New User is: ', newUser.name);
})
// => New User is: originalUserName
```

```
let u = new User('Nate', 'anImgSrc');
userService.currentUser.onNext(u);
// => New User is: Nate
```

注意，在这里，我们使用Subject上的onNext方法将一个新的值推到流

这里的好处是我们能够直接使用流现有的API，所以我们不会引入任何新代码或API

1. 创建setCurrentUser(newUser: User)方法

我们可以更新当前用户的另一种方法是在UserService创建一个辅助方法像这样：

```
public setCurrentUser(newUser: User): void {
    this.currentUser.onNext(newUser);
}
```

你会发现，我们仍然使用currentUser流上的onNext方法，为何这样做呢？

因为这个操作从流上实现解耦到了currentUser上实现，通过在setCurrentUser上包装onNext，我们可以在UserService中改变实现而不破坏我们客户端的代码

在这种情况下，我不太推荐第一种方式，但是，它可以在较大项目的可维护性有很大差异

第三种选择是更新自己暴露出的流（也就是说，我们把一个流放在改变当前用户的方法里）。我们会在下面的MessagesService探索这种模式。

UserService.ts

把它们连起来，UserService看起来是这样的

```
import {Injectable, provide} from 'angular2/core';
import {Subject, BehaviorSubject} from 'rxjs';
import {User} from '../models';
/**
 * UserService manages our current user
 */
@Injectable()
export class UserService {
    // `currentUser` contains the current user
    currentUser:Subject<User> = new BehaviorSubject<User>(null);
```

```

    public setCurrentUser(newUser: User): void {
        this.currentUser.next(newUser);
    }
}

export var userServiceInjectables: Array<any> = [
    provide(UserService).toClass(UserService)
];

```



kittencup

about 1 month ago

MessagesService

MessagesService 是这个应用的重要组成部分，在我们的应用中，所有的消息都会经过 MessagesService。

相比我们的 UserService，我们的 MessagesService 具有更复杂的 stream，有 5 个 stream 来构成我们的 MessagesService: 3 个 数据管理(data management) stream 和 2 个 行为(action) stream

这三个数据管理流是：

- newMessages - 发出每一个新的消息，但只发一次。
- messages - 发出当前所有消息的数组
- updates - 对消息进行操作

newMessages stream

newMessages 是一个 Subject，将发布每一个新的消息，但只发一次。

```

export class MessagesService {
    // a stream that publishes new messages only once
    newMessages: Rx.Subject<Message> = new Rx.Subject<Message>();
}

```

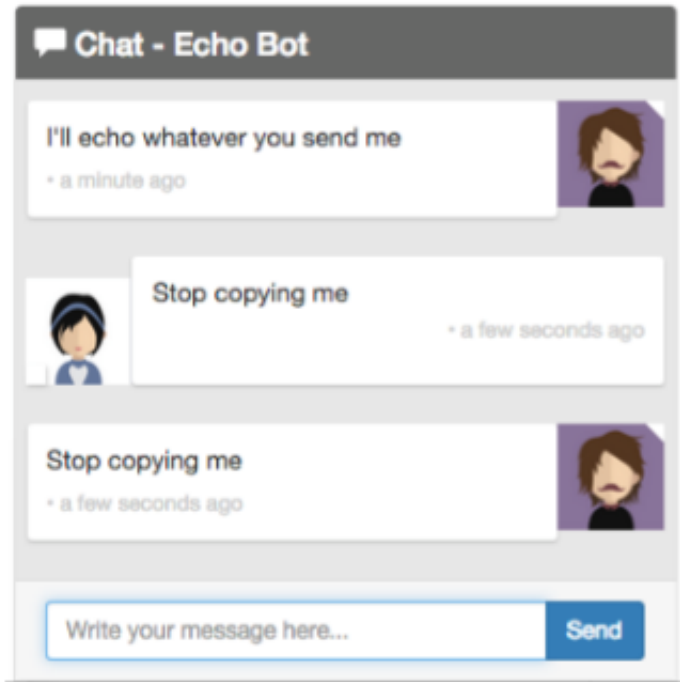
如果我们愿意，我们可以定义一个辅助方法，添加信息到该流中：

```

addMessage(message: Message): void {
    this.newMessages.onNext(message);
}

```

它也将有助于一个stream，从不是来自一个特定的用户的主题帖里获取所有的消息，例如，想一想 Echo Bot:



Real mature, Echo Bot

当我们正在实现Echo Bot，我们不希望进入一个无限循环和重复回到此bot的本身的消息。

要实现这一点，我们可以订阅newMessages流，并筛选出

- 1.属于这个主题帖，
- 2.不是此BOT写的 所有的消息

你可以这样认为，对于一个给定的主题帖我想要当前用户的消息流

```
messagesForThreadUser(thread: Thread, user: User):  
Rx.Observable<Message> {  
    return this.newMessages  
        .filter((message: Message) => {  
            // belongs to this thread  
            return (message.thread.id === thread.id) &&  
                // and isn't authored by this user  
                (message.author.id !== user.id);  
        });  
}
```


messagesForThreadUser接受一个Thread和一个User,并返回一个新的消息流,这个流过滤掉了所有非当前主题帖和用户的消息,也就是说,是这个主题帖中其他人的消息流

messages流

鉴于newMessages发出单个信息,messages则是一个保存所有消息的流。

```
messages: Rx.Observable<Message[]>;
```

该 Message[] 类型 同Array是一样的, 另一种一样的写法

是, Rx.Observable<Array<Message>>, 当我们定义消息类型为

Rx.Observable<Message[]> 时, 我们的意思是这个流包含了所有发出的组数(消息), 而不是单个消息。

那么, 消息是如何被填充的呢? 我们需要讨论的update流和一个新的模式: 操作stream。

操作流模式

这里的想法:

- 我们在messages里保持所有最新消息数组的状态
- 我们使用一个updates流, 这个updates流会通过操作函数应用到messages

你可以认为它是这样的: 这个操作函数是指, 被放在updates流将改变当前的消息列表。一个放在updates数据流上的函数应该接受一个消息列表, 然后返回一个消息列表。让我们通过在代码中创建一个接口来正式化这个想法:

```
interface IMessagesOperation extends Function {  
    (messages: Message[]): Message[];  
}
```

让我们定义我们的updates流

```
updates: Rx.Subject<any> = new Rx.Subject<any>();
```

记住, updates接收操作将应用于我们的消息列表。但是, 我们如何作出这样的连接? 我们是这样的 (在我们的MessagesService的构造函数):

```

constructor() {
  this.messages = this.updates
  // watch the updates and accumulate operations on the messages
  .scan(initialMessages, (messages: Message[],
    operation: IMessagesOperation) => {
return operation(messages); })

```

这段代码引入了一个新的流功能：[scan](#)。如果你熟悉函数式编程，scan很像reduce,为输入流中的每个元素运行该函数，并积累值。scan的特殊之处在于它将为中间每一个结果发出一个值。也就是说，它不会等待流完成后，发射出一个结果，这正是我们想要的。当我们调用this.updates.scan时，我们创建一个新的流，它将订阅updates流，在循环每一个时，我们将：

- 我们累积message
- 申请新操作函数

并且我们返回一个新的Message[]

共享流

有一点需要知道的是流默认不是可共享的，也就是说，如果一个用户从一个流中读取值，这个值就可以被永远地消失。在我们的信息的情况下，我们需要1，在许多订阅之间共享同一个流，2.对于那些迟到的订阅，重播最后个值。

要做到这一点，我们使用[shareReplay](#)方法,如下所示：

```

constructor() {
  this.messages = this.updates
  // watch the updates and accumulate operations on the messages
  .scan(initialMessages, (messages: Message[],
    operation: IMessagesOperation) => {
return operation(messages); })
// make sure we can share the most recent list of messages across
anyone // who's interested in subscribing and cache the last known list
of
// messages
.shareReplay(1);

```

添加一些消息到message流中

现在，我们可以添加一条消息到message流如下所示：

```
var myMessage = new Message(/* params here... */);
updates.onNext( (messages: Message[]): Message[] => { return
messages.concat(myMessage);
})
```

上面，在update流中我们增加一个操作。消息订阅该流并因此将应用该操作，这会将我们的消息合并到原先的累加的消息列表里。

如果这需要几分钟的时间来考虑它的好。如果你不习惯这种编程方式，它可以感觉到一点点外国话。

上述方法的一个问题是，它的使用有点啰嗦。不需要每次都写内部函数就好了。我们可以做类似的事情：

```
addMessage(newMessage: Message) {
    updates.onNext( (messages: Message[]): Message[] => {
        return messages.concat(newMessage);
    })
}

// somewhere else

var myMessage = new Message(/* params here... */);
MessagesService.addMessage(myMessage);
```

这是一个更好的方式，但它不是“反应式的方式”。在某种程度上，因为创建消息的这个动作是不可与其他流组合的。（另外这个方法绕过我们的newMessages流。更在它之后。）

创建一个新的消息的反应式的方式是，有一个接受消息添加到该列表中的流。同样，这可能是一个有点新的，如果你不习惯这样的想法。这里是实现它：

首先，我们创建一个“操作流”称为create。（术语“操作流”这个词只是用来描述，流本身仍然是一个常规的Subject）：

```
// action streams
create: Rx.Subject<Message> = new Rx.Subject<Message>();
```

接下来，在我们的构造函数中，我们配置了create流：

```
this.create
.map( function(message: Message): IMessagesOperation {
    return (messages: Message[]) => { return messages.concat(message);
}; })
```

该`map`运算符是很像javascript内置的`Array.map`功能，但这里它只适用于流。

在这种情况下，我们说“因为我们接收每一个输入的消息，返回`IMessagesOperation`类型的操作，它将此消息添加到消息列表中”。换句话说，这个流将发出一个函数，它接受消息列表，并将新的消息添加到该消息列表

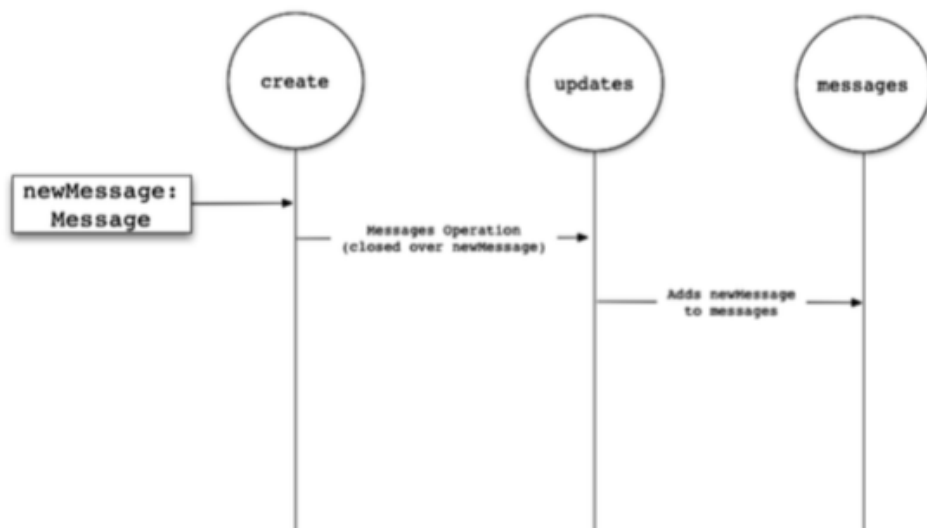
现在，我们有`create`流，我们还有一件事没有做：我们需要真正地把它挂到更新流上。我们通过使用`subscribe`。

```
this.create
.map( function(message: Message): IMessagesOperation {
    return (messages: Message[]) => { return messages.concat(message);
}; })
.subscribe(this.updates);
```

我们在这里所做的是`update`流订阅`create`流。这意味着，如果`create`收到消息时，它会发出`IMessagesOperation`将由`update`接收执行，然后信息会被添加到消息列表中的。

下面是一个图，说明我们目前的情况：

Here's a diagram that shows our current situation:



Creating a new message, starting with the create stream

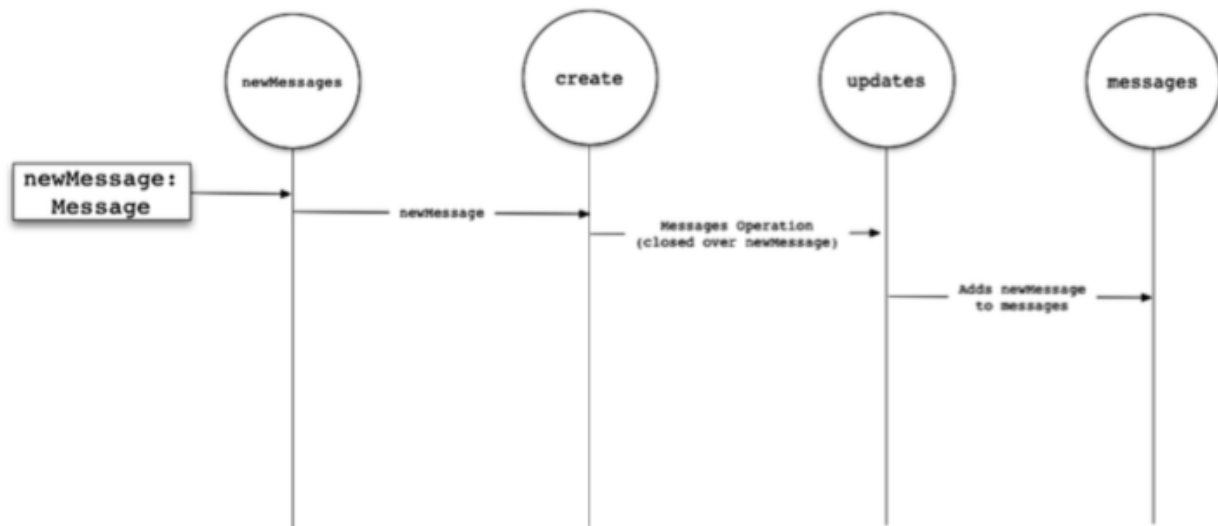
这是伟大的，因为这意味着我们得到了几件事情：

- 1.当前消息列表来自于message
- 2.一种处理当前消息列表的方法(通过update)
- 3.这是用在我们update流上的一个易于使用的流（通过create）

在我们代码的任何地方，如果我想要获得当消息的最新列表，我们只需要进入messages流,但我们还有一个问题，我们还没有将流流向连接上newMessages流

```
this.newMessages.subscribe(this.create);
```

现在我们的图看起来是这样的：



Creating a new message, starting with the newMessages stream

现在我们的流向就完成了！这是两全其美的：我们能够通过订阅newMessages流获得独立的消息流，但是如果只是想要最新消息列表，我们可以订阅message流。

值得指出的是这个设计的一些影响:如果你直接订阅newMessages，你必须要小心下游(downstream)可能发生的变化。这里有三件事情要考虑：

首先,你显然不会得到任何下游应用于信息的更新。

第二，在这种情况下，我们有可变Message对象。所以，如果你订阅newMessages，并存储一个Message引用，该Message的属性可能会改变。

第三,在这种情况下您可能无法利用我们Message可变性的优势，考虑一下这种情况:我们可以在这个update队列上使用一个操作，这个操作会复制每一个Message,然后去改变这个复制的Message

(这可能是比我们在这里做的更好的设计)，在这种情况下，你不能让任何直接从newMessages发出的消息处于“最终”状态。

也就是说,只要你记住这些注意事项,你不应该有太多的麻烦。

我们完成后的MessagesService

下面是完整的MessagesService的样子：

```
# code/rxjs/chat/app/ts/services/MessagesService.ts

import {Injectable, bind} from 'angular2/core';
import {Subject, Observable} from 'rxjs';
import {User, Thread, Message} from '../models';
```

```

let initialMessages: Message[] = [];

interface IMessagesOperation extends Function {
  (messages: Message[]): Message[];
}

@Injectable()
export class MessagesService {
  // a stream that publishes new messages only once
  newMessages: Subject<Message> = new Subject<Message>();

  // `messages` is a stream that emits an array of the most up to date
  messages: Observable<Message[]>;

  // `updates` receives _operations_ to be applied to our `messages`
  // it's a way we can perform changes on *all* messages (that are
  currently
  // stored in `messages`)
  updates: Subject<any> = new Subject<any>();

  // action streams
  create: Subject<Message> = new Subject<Message>();
  markThreadAsRead: Subject<any> = new Subject<any>();

  constructor() {
    this.messages = this.updates
      // watch the updates and accumulate operations on the messages
      .scan((messages: Message[],
        operation: IMessagesOperation) => {
        return operation(messages);
      },
        initialMessages)
      // make sure we can share the most recent list of messages across
  anyone
      // who's interested in subscribing and cache the last known list
  of
      // messages
      .publishReplay(1)
      .refCount();

    // `create` takes a Message and then puts an operation (the inner
  function)
    // on the `updates` stream to add the Message to the list of
  messages.
    //
    // That is, for each item that gets added to `create` (by using
  `next`)

```

```

// this stream emits a concat operation function.
//
// Next we subscribe `this.updates` to listen to this stream, which
means
// that it will receive each operation that is created
//
// Note that it would be perfectly acceptable to simply modify the
// "addMessage" function below to simply add the inner operation
function to
// the update stream directly and get rid of this extra action
stream
// entirely. The pros are that it is potentially clearer. The cons
are that
// the stream is no longer composable.
this.create
  .map( function(message: Message): IMessagesOperation {
    return (messages: Message[]) => {
      return messages.concat(message);
    };
  })
  .subscribe(this.updates);

this.newMessages
  .subscribe(this.create);

// similarly, `markThreadAsRead` takes a Thread and then puts an
operation
// on the `updates` stream to mark the Messages as read
this.markThreadAsRead
  .map( (thread: Thread) => {
    return (messages: Message[]) => {
      return messages.map( (message: Message) => {
        // note that we're manipulating `message` directly here.
Mutability
        // can be confusing and there are lots of reasons why you
might want
        // to, say, copy the Message object or some other
'immutable' here
        if (message.thread.id === thread.id) {
          message.isRead = true;
        }
        return message;
      });
    });
  })
  .subscribe(this.updates);
}

```



```
// an imperative function call to this action stream
addMessage(message: Message): void {
  this.newMessages.next(message);
}

messagesForThreadUser(thread: Thread, user: User):
Observable<Message> {
  return this.newMessages
    .filter((message: Message) => {
      // belongs to this thread
      return (message.thread.id === thread.id) &&
        // and isn't authored by this user
        (message.author.id !== user.id);
    });
}

export var messagesServiceInjectables: Array<any> = [
  bind(MessagesService).toClass(MessagesService)
];
```

试试运行 MessagesService

如果你还没有准备好，这将是一个好时机去开拓代码和感受下MessagesService它是如何工作的。我们有一个例子，你可以在test/services/MessagesService.spec.ts启动。

要运行这个项目的测试，那么打开你的终端：

```
cd/path/to/code/rxjs/chat//<--your path will vary
npm install
karma start
```

让我们开始为我们的模型创建一些实例：

```
# code/rxjs/chat/test/services/MessagesService.spec.ts

let user: User = new User('Nate', '');
let thread: Thread = new Thread('t1', 'Nate', ''); let m1: Message =
new Message({
  author: user, text: 'Hi!', thread: thread
});
let m2: Message = new Message({ author: user,
  text: 'Bye!',
  thread: thread
```

```
})
```

接下来，让我订阅我们的几个流：

```
# code/rxjs/chat/test/services/MessageService.spec.ts

let messageService = new MessageService();

// listen to each message individually as it comes in
messageService.newMessages
  .subscribe( (message: Message) => {
    console.log("=> newMessages: " + message.text);
  });

// listen to the stream of most current messages
messageService.messages
  .subscribe( (messages: Message[]) => {
    console.log("=> messages: " + messages.length);
  });

messageService.addMessage(m1);
messageService.addMessage(m2);

// => messages: 0
// => messages: 1
// => newMessages: Hi!
// => messages: 2
// => newMessages: Bye!
```

这里要注意两件事情：

- 1.我们的消息订阅立即收到一个长度为0的数组，这是因为我们在流上使用了shareReplay，正如你还记得，这意味着订阅会给出了一个最后的结果。
- 2.尽管我们先订阅newMessages，newMessages是由addMessage方法直接调用，我们的messages订阅会先打印出日志，这样做的原因是，因为messages订阅newMessages早于我们在这个测试中的订阅(当MessageService实例化时)(你不应该依靠在你的代码独立的流排序，但为什么是这样工作是值得我们思考的。)

测试下MessageService及感受一下那里的流。我们在下一节中建立了ThreadService，来使用它们。



ThreadsService

在我们的ThreadsService上定义4个流用于发射，分别为：

- 1.当前帖子的一个map(在threads)
- 2.按时间顺序排列的帖子列表，最新的（在orderedthreads）
- 3.当前选定的帖子（在currentThread）
- 4.当前选定的帖子消息列表（在currentThreadMessages）

让我们通过如何建立这些流，我们将沿途学习更多关于RxJS。

当前帖子的一个map(在threads)

让我们先来定义我们的ThreadsService类和实例变量，将发出Threads：

```
import {Injectable, bind} from 'angular2/core';
import {Subject, BehaviorSubject, Observable} from 'rxjs';
import {Thread, Message} from '../models';
import {MessagesService} from '../MessagesService';
import * as _ from 'underscore';

@Injectable()
export class ThreadsService {

  // `threads` is a observable that contains the most up to date list
  of threads
  threads: Observable<{ [key: string]: Thread }>;
```

注意：这个流将发射一个map(object),map的key是Thread的字符串id,value则是这个Thread本身

要创建一个流保持的当前主题列表，我们首先附着在messagesService.messages流：

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

this.threads = messagesService.messages
```

回想一下，一个新的消息被添加到流，message将发射当前消息数组。我们要看看每个消息

息，并我们想返回帖子的唯一列表。

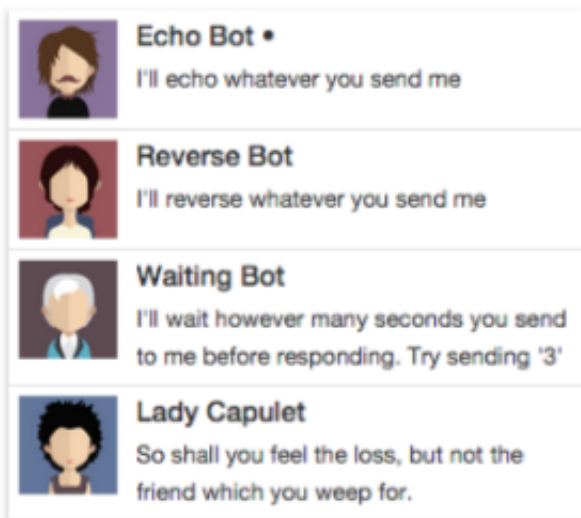
```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

this.threads = messagesService.messages
    .map( (messages: Message[]) => {
let threads: {[key: string]: Thread} = {};
// Store the message's thread it in our accumulator `threads`
messages.map((message: Message) => {
threads[message.thread.id] = threads[message.thread.id] ||
message.thread;

```

注意上面，每次我们将创建帖子的一个新的列表。这样做的原因是因为我们可能会删除一些消息（如离开谈话）。因为我们每一次重新计算帖子列表，如果它没有消息，我们自然会“删除”帖子。

在帖子列表中，我们希望通过在该帖子显示聊天的文字预览使用最新的消息。



List of Threads with Chat Preview

为了做到这一点，我们为每个帖子将存储最新的消息。我们通过比较sentAt时间知道这消息是最新的：

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

// Cache the most recent message for each thread
let messagesThread: Thread = threads[message.thread.id];
```

```

    if (!messagesThread.lastMessage ||
        messagesThread.lastMessage.sentAt < message.sentAt) {
        messagesThread.lastMessage = message;
    });
    return threads;
})

```

全部放在一起，threads是这样的：

```

# code/rxjs/chat/app/ts/services/ThreadsService.ts

this.threads = messagesService.messages
    .map( (messages: Message[]) => {
        let threads: {[key: string]: Thread} = {};
        // Store the message's thread in our accumulator `threads`
        messages.map((message: Message) => {
            threads[message.thread.id] = threads[message.thread.id] ||
                message.thread;

            // Cache the most recent message for each thread
            let messagesThread: Thread = threads[message.thread.id];
            if (!messagesThread.lastMessage ||
                messagesThread.lastMessage.sentAt < message.sentAt) {
                messagesThread.lastMessage = message;
            }
        });
        return threads;
    });

```

尝试ThreadsService

让我们尝试我们的ThreadsService。首先，我们将创建一个模型来工作：

```

# code/rxjs/chat/test/services/ThreadsService.spec.ts

let nate:User = new User('Nate Murray', '');
let felipe:User = new User('Felipe Coury', '');
let t1:Thread = new Thread('t1', 'Thread 1', '');
let t2:Thread = new Thread('t2', 'Thread 2', '');
let m1:Message = new Message({
    author: nate,
    text: 'Hi!',
    thread: t1
});

```

```

let m2:Message = new Message({
  author: felipe,
  text: 'Where did you get that hat?', thread: t1
});
let m3:Message = new Message({
  author: nate,
  text: 'Did you bring the briefcase?', thread: t2
});

```

现在让我们创建一个服务实例：

```

# code/rxjs/chat/test/services/ThreadsService.spec.ts

let messagesService: MessagesService = new MessagesService();
let threadsService: ThreadsService = new
ThreadsService(messagesService);

```

注意这里我们通过messagesService作为参数传递给我们的threadsService构造函数。一般来说，我们让依赖注入系统来处理这个问题。但在我们的测试中，我们可以自己提供依赖。

让我们订阅threads，然后通过：

```

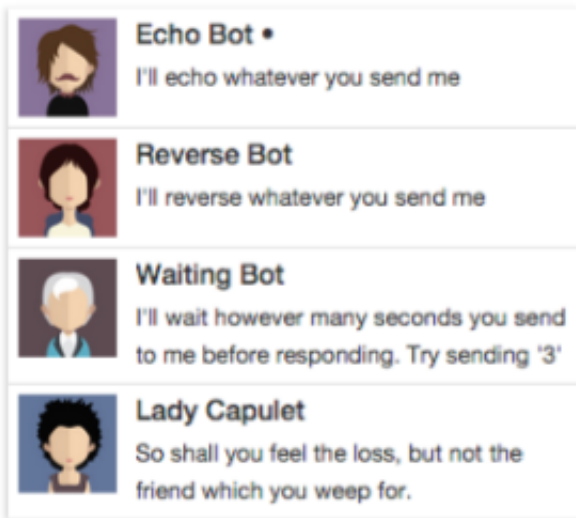
# code/rxjs/chat/test/services/ThreadsService.spec.ts

threadsService.threads
  .subscribe( (threadIdx: { [key: string]: Thread }) => {
    let threads: Thread[] = _.values(threadIdx);
    let threadNames: string = _.map(threads, (t: Thread) => t.name)
      .join(', ');
    console.log(`=> threads (${threads.length}): ${threadNames} `);
  });
messagesService.addMessage(m1);
messagesService.addMessage(m2);
messagesService.addMessage(m3);
// => threads (1): Thread 1
// => threads (1): Thread 1
// => threads (2): Thread 1, Thread 2
}); });

```

按时间顺序排列的帖子列表，最新的（在orderedthreads）

threads提供了一个map，它作为我们的帖子列表的“索引”。但我们希望帖子视图以最新消息为排列顺序的。



Time Ordered List of Threads

让我们创建一个新的流来返回按最近消息时间排序的threads数组：我们将开始定义orderedThreads实例属性：

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

// `orderedThreads` contains a newest-first chronological list of
threads
orderedThreads: Rx.Observable<Thread[]>;
```

其次，在构造函数中我们定义orderedThreads来通过订阅threads对消息按照时间进行排序：

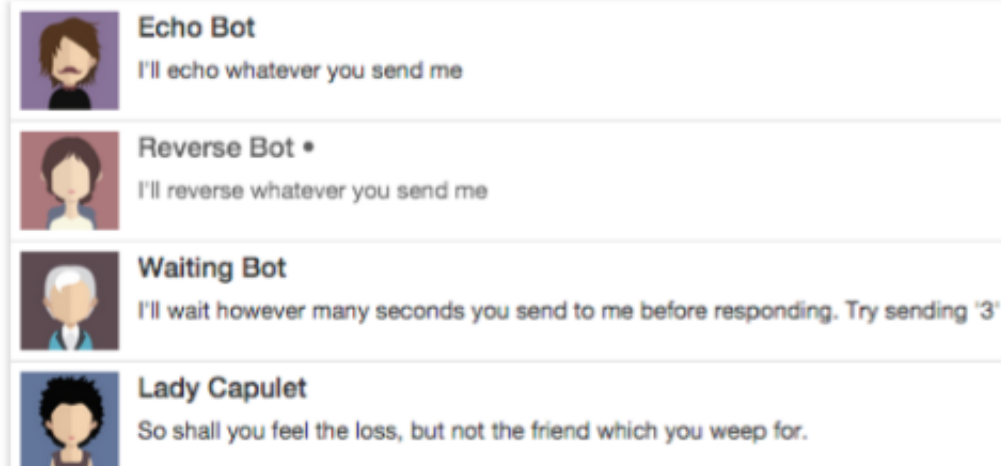
```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

this.orderedThreads = this.threads
    .map((threadGroups: { [key: string]: Thread }) => {
        let threads: Thread[] = _.values(threadGroups);
        return _.sortBy(threads, (t: Thread) =>
            t.lastMessage.sentAt).reverse();
    })
    .shareReplay(1);
```

当前选定的帖子（在currentThread）

我们的应用程序需要知道哪些thread是当前选定的thread。这让我们知道：

- 1.在消息窗口中应该显示哪个thread
- 2.在thread列表中，thread应该被标记为当前thread



The current thread is marked by a 'â€¢' symbol

让我们创建BehaviorSubject将存储currentThread：

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

// `currentThread` contains the currently selected thread
currentThread: Rx.Subject<Thread> =
  new Rx.BehaviorSubject<Thread>(new Thread());
```

注意，我们正在发布一个空Thread作为默认值。我们并不需要任何进一步配置currentThread。

设置currentThread

要设置当前thread，我们可以在客户端或者1.通过onNext直接提交新的thread或2.添加一个辅助的方法来做到这一点。

让我们定义一个setCurrentThread辅助方法，我们可以使用它来设置下一个thread：


```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

setCurrentThread(newThread: Thread): void {
    this.currentThread.onNext(newThread);
}
```

标记当前主题为已读

我们要跟踪未读邮件的数量。如果我们切换到一个新的thread，那我们要标记所有在该thread的消息的为已读。我们需要做的部分：

- 1.messagesService.makeThreadAsRead接受一个Thread用来标记着该Thread所有消息为已读
- 2.我们的currentThread发射单个Thread，它代表这当前的Thread

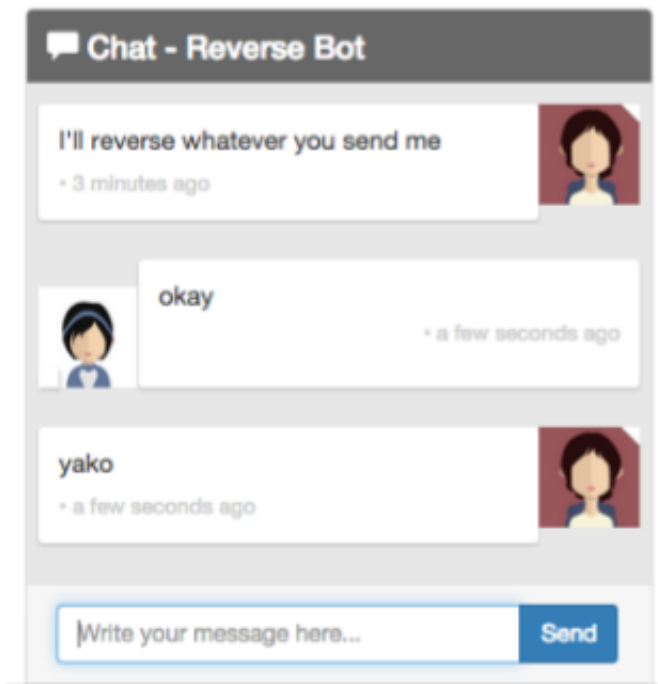
所以，我们需要做的把它们勾在一起：

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

this.currentThread.subscribe(this.messagesService.markThreadAsRead);
```

当前选定的帖子消息列表（在currentThreadMessages）

现在，我们有了当前选定的Thread，我们需要确保我们可以显示该thread中的消息列表。



The current list of messages is for the Reverse Bot

实现这是可能看起来比表面的有点复杂得多。假设我们实现这样的：

```
var theCurrentThread: Thread;
this.currentThread.subscribe((thread: Thread) => { theCurrentThread =
thread; })

this.currentThreadMessages.map((messages: Message[]) => {
  return _.filter(messages, (message: Message) => {
    return message.thread.id == theCurrentThread.id; })
})
```

这种方法有什么不好？恩，如果currentThread有变化，currentThreadMessages不会知道这件事，所以在currentThreadMessages我们将有一个过时的名单！

如果我们推翻它，存储当前消息列表在一个变量中，并订阅currentThread的变化？我们会有同样的问题，只有这个时候，我们知道Thread的变化，而不是一个新的消息进来。

我们怎样才能解决这个问题？

原来，RxJS拥有一套可以结合多个流的操作符。在这种情况下，我们想说“如果currentThread或messagesService.messages改变，我们要发射的某些内容”。在这里我们使用combineLatest操作。

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts
```

```
this.currentThreadMessages = this.currentThread
    .combineLatest(messagesService.messages,
        (currentThread: Thread, messages: Message[]) => {
```

当我们将两者相结合的时候，一个或另一个会先到达，并没有保证我们将有一个值在两个流，所以我们需要检查，以确保什么是我们需要，否则我们将返回一个空列表。

现在，我们有两个当前thread和消息，我们可以过滤掉只是我们不感兴趣的消息：

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts
```

```
this.currentThreadMessages = this.currentThread
    .combineLatest(messagesService.messages,
        (currentThread: Thread, messages: Message[]) => {
            if (currentThread && messages.length > 0) {
                return _.chain(messages)
                    .filter((message: Message) =>
                        (message.thread.id === currentThread.id))
```

另外一个细节，因为我们已经在寻找对于当前Thread的消息，这是一个方便的地方为这些邮件标记已读

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts
```

```
.filter((message: Message) =>
    (message.thread.id === currentThread.id))
.map((message: Message) => {
    message.isRead = true;
    return message; })
.value();
```

我们是否应该在这里为已读消息标志是有争议的。最大的缺点是，我们改变的对象是什么，从本质上讲，一个“读”的thread。即这是一个具有副作用的读操作，这通常是一个坏主意

这就是说，在本应用的currentThreadMessages只适用于currentThread，该currentThread应该始终将它的邮件标记为已读。这就是说，我建议一般“读有副作用的”不是一个模式。

将其组合在一起，这里是currentThreadMessages样子：

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

this.currentThreadMessages = this.currentThread
    .combineLatest(messagesService.messages,
        (currentThread: Thread, messages: Message[]) => {
            if (currentThread && messages.length > 0) {
                return _
                    .chain(messages)
                    .filter((message: Message) =>
                        (message.thread.id === currentThread.id))
                    .map((message: Message) => {
                        message.isRead = true;
                        return message; })
                    .value();
            } else {
                return [];
            }
        })
```

我们完成的ThreadsService

下面是我们的ThreadService的样子:

```
# code/rxjs/chat/app/ts/services/ThreadsService.ts

import {Injectable, provide} from 'angular2/core';
import {Subject, BehaviorSubject, Observable} from 'rxjs';
import {Thread, Message} from '../models';
import {MessagesService} from './MessagesService';
import * as _ from 'underscore';

@Injectable()
export class ThreadsService {

    // `threads` is a observable that contains the most up to date list
    of threads
    threads: Observable<{ [key: string]: Thread }>;

    // `orderedThreads` contains a newest-first chronological list of
    threads
    orderedThreads: Observable<Thread[]>;

    // `currentThread` contains the currently selected thread
    currentThread: Subject<Thread> =
        new BehaviorSubject<Thread>(new Thread());
```

```

// `currentThreadMessages` contains the set of messages for the
currently
// selected thread
currentThreadMessages: Observable<Message[]>;

constructor(public messagesService: MessagesService) {

    this.threads = messagesService.messages
    .map( (messages: Message[]) => {
        let threads: {[key: string]: Thread} = {};
        // Store the message's thread in our accumulator `threads`
        messages.map((message: Message) => {
            threads[message.thread.id] = threads[message.thread.id] ||
                message.thread;

            // Cache the most recent message for each thread
            let messagesThread: Thread = threads[message.thread.id];
            if (!messagesThread.lastMessage ||
                messagesThread.lastMessage.sentAt < message.sentAt) {
                messagesThread.lastMessage = message;
            }
        });
        return threads;
    });

    this.orderedThreads = this.threads
    .map((threadGroups: { [key: string]: Thread }) => {
        let threads: Thread[] = _.values(threadGroups);
        return _.sortBy(threads, (t: Thread) =>
t.lastMessage.sentAt).reverse();
    });

    this.currentThreadMessages = this.currentThread
    .combineLatest(messagesService.messages,
        (currentThread: Thread, messages: Message[]) => {
            if (currentThread && messages.length > 0) {
                return _.chain(messages)
                    .filter((message: Message) =>
                        (message.thread.id === currentThread.id))
                    .map((message: Message) => {
                        message.isRead = true;
                        return message; })
                    .value();
            } else {
                return [];
            }
        });
}

```

```
this.currentThread.subscribe(this.messagesService.markThreadAsRead);  
}  
  
setCurrentThread(newThread: Thread): void {  
    this.currentThread.next(newThread);  
}  
  
}  
  
export var threadsServiceInjectables: Array<any> = [  
    provide(ThreadsService).toClass(ThreadsService)  
];
```



kittencup

about 1 month ago

数据模型的总结

我们的数据模型和服务都完成了！现在，我们现在需要将这一切绑在我们的视图组件！在接下来的章节中，我们将建立我们的3个主要组件来渲染，并与这些流交互。



kittencup modified this issue about 1 month ago

Comment on issue

Sign in to comment

or [sign up](#) to join this conversation on GitHub



Desktop version

 Open

基于Observables的数据架构 - 第2部分:视图组件 #41



kittencup opened this issue
about 1 month ago



ng-book 2

该issue关闭讨论，如有问题请去 [#43](#) 提问

目录

- [建立我们的视图：顶层组件ChatApp](#)
- [ChatThreads组件](#)
- [单个ChatThread组件](#)
- [ChatWindow组件](#)
- [ChatMessage组件](#)
- [ChatNavBar组件](#)
- [总结](#)
- [下一步](#)



kittencup added the [ng-book 2](#) label about 1 month ago



kittencup locked this issue about 1 month ago



kittencup
about 1 month ago

建立我们的视图:顶层组件ChatApp

让我们将注意力转移到我们的应用程序，并实现我们的视图组件。

为了清晰和空间，在下面的章节中，我将要离开1。import语句，CSS，和其他一些不重要的东西。如果您对这些每一行细节好奇的话，打开示例代码，因为它包含了我们应用需要

运行的所有代码。

我们要做的第一件事就是创建我们的顶层组件chat-app

正如我们前面谈到的，页面被分为三个顶层组件：

image

- ChatNavBar 包含了消息未读数
- ChatThreads 显示可点击thread列表，以及最新的消息和聊天头像
- ChatWindow 显示了一个输入框的消息在当前线程中发送新消息

下面是我们的组件代码的样子：

```
# code/rxjs/chat/app/ts/app.ts

@Component({
  selector: "chat-app"
}) @View({
  directives: [ChatNavBar,
               ChatThreads,
               ChatWindow],
  template: `
<div> <nav-bar></nav-bar> <div class="container">
  <chat-threads></chat-threads>
  <chat-window></chat-window>
</div>
</div>
` })
class ChatApp {
  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService,
              public userService: UserService) {
    ChatExampleData.init(messagesService, threadsService, userService);
  }
}
bootstrap(ChatApp, [ servicesInjectables, utilInjectables ]);
```

看一看构造函数，在这里我们注入了3个服务，MessagesService, ThreadsService, 和 UserService 我们使用这些服务来初始化我们的示例数据。

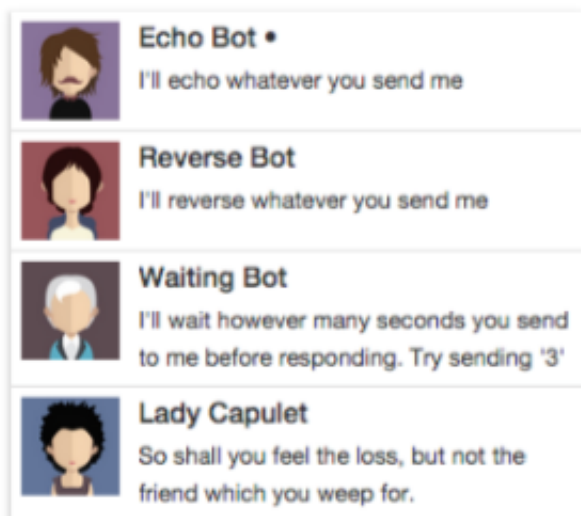
如果你对例子数据感兴趣的，你可以在这找到

它 `code/rxjs/chat/app/ts/ChatExampleData.ts`



ChatThreads组件

接下来让我们在chatThreads组件建立我们的Thread列表。



Time Ordered List of Threads

我们的selector很简单，我们想要匹配chat-threads的选择器

```
# code/rxjs/chat/app/ts/components/ChatThreads.ts

@Component({
  selector: "chat-threads"
})
```

ChatThreads控制器

看看我们的组件控制器ChatThreads:

```
# code/rxjs/chat/app/ts/components/ChatThreads.ts

export class ChatThreads {
  threads: Observable<any>;

  constructor(public threadsService: ThreadsService) {
```

```
    this.threads = threadsService.orderedThreads;
  }
}
```

在这里，我们注入ThreadsService然后我们保持一个orderedThreads的引用。

ChatThreads 模板

最后，让我们来看看模板配置

```
# code/rxjs/chat/app/ts/components/ChatThreads.ts

@Component({
  selector: 'chat-threads',
  directives: [ChatThread],
  changeDetection: ChangeDetectionStrategy.OnPushObserve,
  template: `
    <!-- conversations -->
    <div class="row">
    <div class="conversation-wrap">
      <chat-thread
        *ngFor="#thread of threads | async"
        [thread]="thread"> </chat-thread>
      </div>
    </div>
  `
})
```

来看看这里东西：带有async管道的NgFor，ChatThread和ChangeDetectionStrategy。

该ChatThread指令组件（相匹配的chat-thread标记）将显示该Threads的视图。这个组件是我们在上面定义的。

NgFor迭代我们的threads，并传入属性[thread]给我们的ChatThread指令。

但是，你可能会注意到在我们的*ngFor里的新东西:一个叫async管道

RxPipe是定义这种行为的类。RxPipe是一个自定义，管道,它让我们在视图使用RxJS的Observable.我们打算花太多时间谈论RxPipe的实现细节，但你可以找到在/rxjs/app/ts/util/RxPipe.t找到定义代码

async是通过AsyncPipe实现的，它可以让我们在模板中使用RxJS的Observable，这是非常棒的，因为Angular2让我们使用异步Observable就好像是同步集合。这是超级方便，真0

的很酷。

在这部分，我们指定一个自定义changeDetection。Angular2具有灵活，高效的变化检测系统。其中一个好处是，如果我们组件有不可变的或可观察的绑定，那么我们能够得到变化检测系统的提示，这将使我们的应用程序的运行效率更高。

In this case, instead of watching for changes on an array of Threads, Angular will subscribe for changes to the threads observable - and trigger an update when a new event is emitted.

=====这里不翻译了，.OnPushObserve已经弃用了=====

下面是我们的总的ChatThreads组件的样子：

```
# code/rxjs/chat/app/ts/components/ChatThreads.ts

@Component({
  selector: 'chat-threads',
  directives: [ChatThread],
  changeDetection: ChangeDetectionStrategy.OnPushObserve,
  template: `
    <!-- conversations -->
    <div class="row">
      <div class="conversation-wrap">

        <chat-thread
          *ngFor="#thread of threads | async"
          [thread]="thread">
        </chat-thread>

      </div>
    </div>
  `
})
export class ChatThreads {
  threads: Observable<any>;

  constructor(public threadsService: ThreadsService) {
    this.threads = threadsService.orderedThreads;
  }
}
```



kittencup

about 1 month ago

单个ChatThread组件

让我们看一下我们的ChatThread组件。这将用于显示单个Thread。首先是@Component:

```
# code/rxjs/chat/app/ts/components/ChatThreads.ts

@Component({
  inputs: ['thread'],
  selector: 'chat-thread',
```

ChatThread控制器和ngOnInit

在这里，当我们的组件在第一次被检查变化后会调用onInit.

让我们来看看组件控制器:

```
# code/rxjs/chat/app/ts/components/ChatThreads.ts

class ChatThread implements OnInit {
  thread:Thread;
  selected:boolean = false;

  constructor(public threadsService:ThreadsService) {
  }

  ngOnInit():void {
    this.threadsService.currentThread
      .subscribe((currentThread:Thread) => {
        this.selected = currentThread &&
          this.thread &&
          (currentThread.id === this.thread.id);
      });
  }

  clicked(event:any):void {
    this.threadsService.setCurrentThread(this.thread);
    event.preventDefault();
  }
}
```

注意在这里我们实现了一个接口:OnInit,Angular 组件有很多生命周期事件，我们会在组件

在这里，因为我们实现了OnInit接口，ngOnInit方法将会在组件在第一次被变化检测后调用。

我们使用ngOnInit的一个重要原因是因为我们的thread属性将不提供在构造函数中使用

上面可以看到，在我们的ngOnInit订阅threadsService.currentThread如果currentThread与此组件的thread id相匹配，我们设置选择为true（相反，如果thread不匹配，我们设置选择为false）。

我们还要设置一个点击事件处理，这就是我们如何处理选择当前thread的方式。在我们**@View**下面，我们会在thread视图上绑定一个click()。如果我们接受到clicked()，那么我们告诉threadsService来设置当前Thread为此Thread

ChatThread 模板

下面是我们的模板

```
# code/rxjs/chat/app/ts/components/ChatThreads.ts

@Component({
  inputs: ['thread'],
  selector: 'chat-thread',
  template: `
    <div class="media conversation">
      <div class="pull-left">
        </div>
        <div class="media-body">
          <h5 class="media-heading contact-name">{{thread.name}}
            <span *ngIf="selected">&bull;</span></h5>
          <small class="message-preview">
            {{thread.lastMessage.text}}</small>
          </div>
          <a (click)="clicked($event)" class="div-link">Select</a>
        </div>
      </div>
    `
})
```

请注意，我们已经有了像一些直接的绑定{{thread.avatarSrc}}， {{thread.name}}和 {{thread.lastMessage.text}}。

我们有一个*ng-if, 如果选择了这个threads, 将会显示一个 `&bull`

最好, 我们绑定了(click)事件到我们的clicked()处理程序上, 注意, 当我们调用clicked时我们会传递一个\$event参数, 这是Angular提供的一个特殊变量, 它描述了event,我们在我们的clicked处理程序中调用了event.preventDefault().这可以确保我们不会导航到不同的页面。

ChatThread完整的代码

这里是整个ChatThread组件代码

```
import {
  Component,
  OnInit,
  ChangeDetectionStrategy
} from 'angular2/core';
import {ThreadsService} from '../services/services';
import {Observable} from 'rxjs';
import {Thread} from '../models';

@Component({
  inputs: ['thread'],
  selector: 'chat-thread',
  template: `
<div class="media conversation">
  <div class="pull-left">
    
  </div>
  <div class="media-body">
    <h5 class="media-heading contact-name">{{thread.name}}
      <span *ngIf="selected">&bull;</span>
    </h5>
    <small class="message-preview">{{thread.lastMessage.text}}
  </small>
  </div>
  <a (click)="clicked($event)" class="div-link">Select</a>
</div>
`
})
class ChatThread implements OnInit {
  thread: Thread;
  selected: boolean = false;

  constructor(public threadsService: ThreadsService) {
  }
```

```

ngOnInit(): void {
  this.threadsService.currentThread
    .subscribe( (currentThread: Thread) => {
      this.selected = currentThread &&
        this.thread &&
        (currentThread.id === this.thread.id);
    });
}

clicked(event: any): void {
  this.threadsService.setCurrentThread(this.thread);
  event.preventDefault();
}
}

```

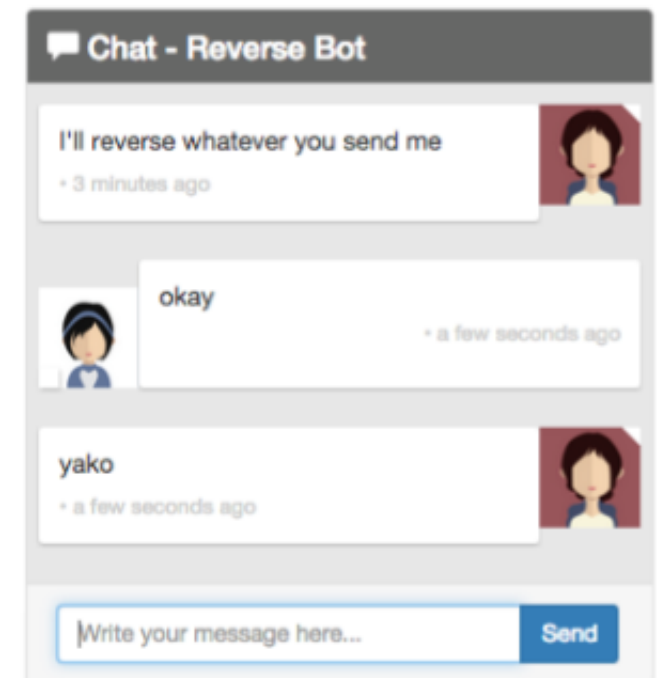


kittencup

about 1 month ago

ChatWindow组件

该ChatWindow是我们的应用程序中最复杂的部分。让我们逐一看这部分：



The Chat Window

首先来定义我们的@Component

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

@Component({
  selector: 'chat-window',
  directives: [ChatMessage,
               FORM_DIRECTIVES],
  changeDetection: ChangeDetectionStrategy.OnPushObserve,
```

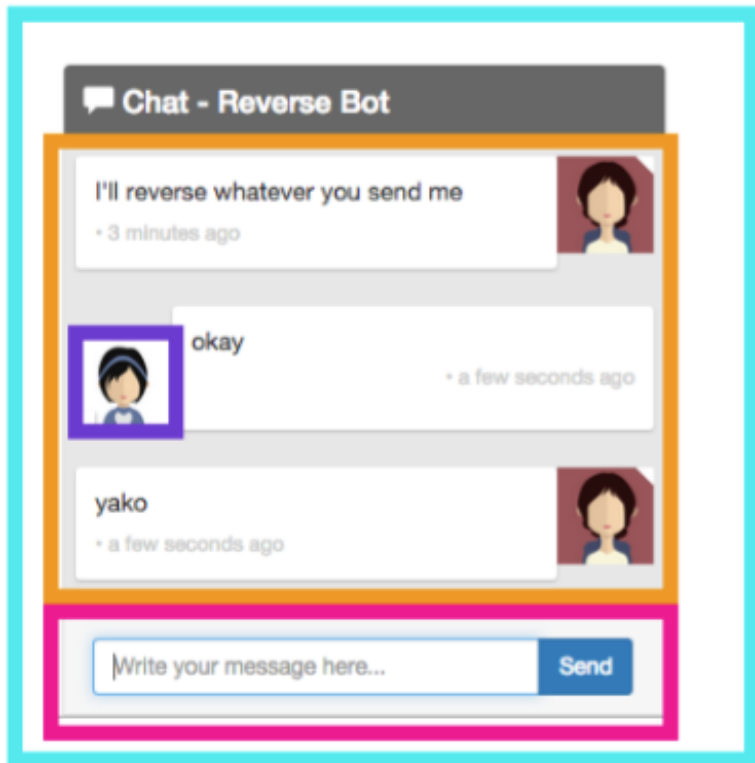
ChatWindow属性

我们的ChatWindow有4个属性

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

export class ChatWindow {
  messages: Rx.Observable<any>;
  currentThread: Thread;
  draftMessage: Message;
  currentUser: User;
```

以下图是其中每一个组件组成部分：



currentThread

messages

currentUser

draftMessage

Chat Window Properties

在我们的构造函数中注入4样东西

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

constructor(public messagesService: MessagesService,
             public threadsService: ThreadsService,
             public userService: UserService,
             public el: ElementRef) {

}
```

前3个是我们服务。第4个 `el` 是一个 `ElementRef`，我们可以用它来获得host DOM元素。当我们创建和接受新消息时我们使用它滚动到聊天窗口的底部

记住: 在构造函数中使用 `public messagesService: MessagesService`，我们不仅注入 `MessagesService` 并建立一个实例变量，我们也可以在后面在类中通过 `this.messagesService` 使用

ChatWindow onInit

我们要把这部分在 `NgOnInit` 初始化.我们将在这里主要是设置订阅我们的 `Observable` 来改变

我们的组件属性。

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

NgOnInit(): void {
  this.messages = this.threadsService.currentThreadMessages;
  this.draftMessage = new Message();
}
```

首先，我们将currentThreadMessages保存到messages属性，接下去我们创建一个空的Message来初始化默认的draftMessage

当我们发送一个新的消息时，我们需要确保Messages保存一个发送的Thread的引用，发送的thread将一直是当前thread,所以让我们存储一个引用到当前选定的thread：

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

this.threadsService.currentThread.subscribe( (thread: Thread) => {
  this.currentThread = thread;
});
```

我们也希望从当前用户发送新邮件，让我们用currentUser做同样的事：

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

this.userService.currentUser
  .subscribe(
    (user: User) => {
      this.currentUser = user;
    });
```

ChatWindow sendMessage

既然我们在讨论它，那么让我们来实现一个SendMessage函数用来发送一个新的信息：

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

sendMessage(): void {
  let m: Message = this.draftMessage;
  m.author = this.currentUser;
  m.thread = this.currentThread;
  m.isRead = true;
}
```

```
    this.messagesService.addMessage(m);  
    this.draftMessage = new Message();  
}
```

sendMessage上面会产生draftMessage,使用我们的组件属性设置author和thread,每一个消息我们发送并"已阅读"(我们编写的), 所以我们给他们打上已读标记

注意在这里我们没有更新draftMessage保存的消息文字, 这是因为在稍后我们会在视图中为消息绑定值

之后, 我们已经更新了draftMessage属性, 我们把它交给messageService,并且我们为draftMessage创建了一个Message, 我们这样做是为了确保我们不会改变已经发送的消息。

ChatWindow onEnter

在我们的视图中, 在2种情况下我们要发送消息

- 1.用户点击"发送"按钮
- 2.用户敲击Enter(或Return)按键

让我们来定义一个处理这个事件的函数

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts  
  
onEnter(event: any): void {  
    this.sendMessage();  
    event.preventDefault();  
}
```

ChatWindow scrollToBottom

当我们发送一个消息, 或当有一个新的消息近来, 我们在这个聊天窗口中滚动到底部, 要做到这点, 在我们的host元素上设置scrollTop属性

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts  
  
scrollToBottom(): void {  
    let scrollPane: any = this.el  
        .nativeElement.querySelector(".msg-container-base");  
    scrollPane.scrollTop = scrollPane.scrollHeight;  
}
```

现在有一个函数用来滚动到底部，我们必须保证在正确的时候调用该函数，早在NgOnInit，让我们订阅currentThreadMessages的列表，任何时候我们得到一个新消息就滚动到底部

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

this.messages
  .subscribe(
    (messages: Array<Message>) => {
      setTimeout(() => {
        this.scrollToBottom();
      });
    });
```

为什么会有setTimeout?

当我们获取一个新的消息，如果我们立即调用scrollToBottom,那么会滚动到底部会发生在新消息渲染中之前，使用setTimeout是告诉javascript我们想要在这次执行队列完成后运行该函数，这发生在组件的渲染后，所以它是我们想要的。

ChatWindow 模板

template应该看起来很熟悉,首先我们定义了一些标记和header面板:

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

@Component({
  selector: 'chat-window',
  directives: [ChatMessage,
    FORM_DIRECTIVES],
  changeDetection: ChangeDetectionStrategy.OnPushObserve,
  template: `
<div class="chat-window-container">
  <div class="chat-window">
    <div class="panel-container">
      <div class="panel panel-default">
        <div class="panel-heading top-bar">
          <div class="panel-title-container">
            <h3 class="panel-title">
              <span class="glyphicon glyphicon-comment">
</span> Chat - {{currentThread.name}}
            </h3></div>
```

```

        <div class="panel-buttons-container">
            <!-- you could put minimize or close buttons
here -->

        </div>
    </div>

```

接下去 我们显示消息列表，在这里我们使用了ngFor与async管道来迭代我们的消息列表，我们将描述下个人chat-message组件

```

# code/rxjs/chat/app/ts/components/ChatWindow.ts

<div class="panel-body msg-container-base">
    <chat-message
        *ng-for="#message of messages | async"
        [message]="message">
    </chat-message>
</div>

```

最后，我们有消息输入框和关闭标签：

```

# code/rxjs/chat/app/ts/components/ChatWindow.ts

<div class="panel-footer">
    <div class="input-group">
        <input type="text"
            class="chat-input"
            placeholder="Write your message here..."
            (keydown.enter)="onEnter($event)"
            [(ngModel)]="draftMessage.text" />
        <span class="input-group-btn">
            <button class="btn-chat"
                (click)="onEnter($event)"
                >Send</button>
        </span>
    </div>
</div>

</div>
</div>
</div>
</div>

```

消息输入框是这个视图中最有趣的部分，所以让我们来谈谈几件事情.

我们的input标签有2个有趣的属性: 1.(keydown.enter) 和 2.[(ngModel)]

处理按键

Angular 2提供了一个简单的方式来处理键盘操作:我们绑定事件在一个元素上, 在这个例子中, 我们绑定了Keydown.enter, 这表示如果敲击"Enter", 则调用表达式里的函数, 这种情况下是onEnter(\$event)函数

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

<input type="text"

      class="chat-input"
      placeholder="Write your message here..."
      (keydown.enter)="onEnter($event)"
      [(ngModel)]="draftMessage.text" />
```

使用ngModel

正如我们之前所说的, Angular没有一个通用的双向绑定模型, 然后, 它可以在组件和视图之间有一个非常有用双向绑定, 它可以是一个非常方便的方式来保持一个组件属性与视图同步。

在这种情况下, 我们在input标签的值和draftMessage.text之间建立了双向绑定, 就是说, 如果在input标签中输入, draftMessage.text会自动设置这个input的值, 同样, 如果在我们代码中我们更新draftMessage.text, 在视图中的input标签的值也会改变。

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

<input type="text"

      class="chat-input"
      placeholder="Write your message here..."
      (keydown.enter)="onEnter($event)"
      [(ngModel)]="draftMessage.text" />
```

点击 "发送"

在我们的"Send"按钮, 我们绑定(click)属性到我们组件中的onEnter函数

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts
```

```
<span class="input-group-btn">
  <button class="btn-chat" (click)="onEnter($event)">Send</button>
</span>
```

完整的chatwindow组件

下面是整个ChatWindow组件的完整的代码：

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

@Component({
  selector: 'chat-window',
  directives: [ChatMessage,
    FORM_DIRECTIVES],
  changeDetection: ChangeDetectionStrategy.OnPushObserve,
  template: `
    <div class="chat-window-container">
      <div class="chat-window">
        <div class="panel-container">
          <div class="panel panel-default">

            <div class="panel-heading top-bar">
              <div class="panel-title-container">
                <h3 class="panel-title">
                  <span class="glyphicon glyphicon-comment"></span>
                  Chat - {{currentThread.name}}
                </h3>
              </div>
              <div class="panel-buttons-container">
                <!-- you could put minimize or close buttons here -->
              </div>
            </div>

            <div class="panel-body msg-container-base">
              <chat-message
                *ngFor="#message of messages | async"
                [message]="message">
              </chat-message>
            </div>

            <div class="panel-footer">
              <div class="input-group">
                <input type="text"
                  class="chat-input"
                  placeholder="Write your message here..."
```



```

    });
  });
}

onEnter(event: any): void {
  this.sendMessage();
  event.preventDefault();
}

sendMessage(): void {
  let m: Message = this.draftMessage;
  m.author = this.currentUser;
  m.thread = this.currentThread;
  m.isRead = true;
  this.messagesService.addMessage(m);
  this.draftMessage = new Message();
}

scrollToBottom(): void {
  let scrollPane: any = this.el
    .nativeElement.querySelector('.msg-container-base');
  scrollPane.scrollTop = scrollPane.scrollHeight;
}
}

```



kittencup

about 1 month ago

ChatMessage组件

image

这个组件相对简单，这里的主要是逻辑是如果消息是由当前用户编写的渲染一个稍微不同的视图。如果消息不是当前用户编写的，我们要考虑消息的传入

我们开始定义@Compoent:

```

# code/rxjs/chat/app/ts/components/ChatWindow.ts

@Component({
  inputs: ['message'],
  selector: 'chat-message',

```

```
    pipes: [FromNowPipe],  
  })  
}
```

设置incoming

记住每个ChatMessage属于一个Message.所以在onInit里我们订阅currentUser流，并取决于该消息是否是当前用户编写的来设置incoming

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts  
  
export class ChatMessage implements OnInit {  
  message:Message;  
  currentUser:User;  
  incoming:boolean;  
  
  constructor(public userService:UserService) {  
  }  
  
  ngOnInit():void {  
    this.userService.currentUser  
      .subscribe(  
        (user:User) => {  
          this.currentUser = user;  
          if (this.message.author && user) {  
            this.incoming = this.message.author.id !==  
user.id;  
          }  
        });  
  }  
}
```

ChatMessage 模板

在我们的模板有2个有趣的东西:

1. FromNowPipe
2. [ngClass]

首先，这里的代码：

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

@Component({
  inputs: ['message'],
  selector: 'chat-message',
  pipes: [FromNowPipe],
  template: `
    <div class="msg-container"
      [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">

      <div class="avatar"
        *ngIf="!incoming">
        
      </div>

      <div class="messages"
        [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
        <p>{{message.text}}</p>
        <time>{{message.sender}} • {{message.sentAt | fromNow}}</time>
      </div>

      <div class="avatar"
        *ngIf="incoming">
        
      </div>
    </div>
  `
})
```

该FromNowPipe是一个管道，它将我们消息的发送的时间转化为人类可读的"x seconds ago"。你可以看到我们这样使用它 `{{message.sentAt | fromNow}}`

FromNowPipe使用的是优秀的 [moment.js](#) 库。如果你想学习关于如何创建自定义的 pipes, 可以查看[Pipes](#)这章。你也可以在code/rxjs/chat/app/ts/util/FromNowPipe.ts 阅读FromNowPipe源码

我们也在视图中大量的使用了ngClass, 我们的想法是，当我们说：

```
[ng-class]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

我们告诉angular如果incoming是true(msg-sent类是incoming为false时)则接受msg-receive类

当使用incoming属性，我们已不同的方式来显示传入和传出消息。

完整ChatMessage代码

在这是我们完成的ChatMessage组件

```
# code/rxjs/chat/app/ts/components/ChatWindow.ts

import {
  Component,
  OnInit,
  ElementRef,
  ChangeDetectionStrategy
} from 'angular2/core';
import {FORM_DIRECTIVES} from 'angular2/common';
import {
  MessagesService,
  ThreadsService,
  UserService
} from '../services/services';
import {FromNowPipe} from '../util/FromNowPipe';
import {Observable} from 'rxjs';
import {User, Thread, Message} from '../models';

@Component({
  inputs: ['message'],
  selector: 'chat-message',
  pipes: [FromNowPipe],
  template: `
<div class="msg-container"
  [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">

  <div class="avatar"
    *ngIf="!incoming">
    
  </div>

  <div class="messages"
    [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
    <p>{{message.text}}</p>
    <time>{{message.sender}} • {{message.sentAt | fromNow}}</time>
  </div>

  <div class="avatar"
    *ngIf="incoming">
    
  </div>
  `
})
```

```

</div>
-
}))
export class ChatMessage implements OnInit {
  message: Message;
  currentUser: User;
  incoming: boolean;

  constructor(public userService: UserService) {
  }

  ngOnInit(): void {
    this.userService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
          if (this.message.author && user) {
            this.incoming = this.message.author.id !== user.id;
          }
        }
      );
  }
}
}

```

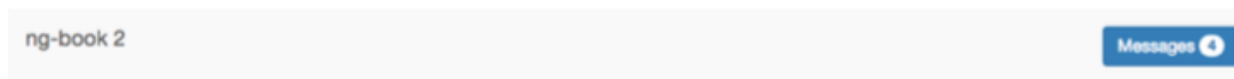


kittencup

about 1 month ago

ChatNavBar组件

我们要讨论的最后一个组件是ChatNavBar，在nav-bar上我们要显示用户未读消息数量



 Echo Bot •

The Unread Count in the ChatNavBar Component

尝试的未读邮件数的最好方法是使用"Waiting Bot"。如果你还没有未读邮件，尝试发送“3”到Waiting BOT然后切换到另一个窗口,Waiting BOT将等待3秒之后送你一个消息，你会看到未读邮件计数器增加。

ChatNavBar @Component

首先，我们定义一个相当简单的@Component

```
# code/rxjs/chat/app/ts/components/ChatNavBar.ts

@Component({
  selector: "nav-bar"
```

ChatNavBar控制器

ChatNavBar控制器需要做的唯一一件事是跟踪unreadMessagesCount。这比表面看起来要复杂得多。

最直接的方法是简单地监听messagesService.messages，并计算Message里属性isRead为false的总数，这对于当前Thread之外的所有消息都可以正常工作，然而在当前Thread里的新消息不能保证在messages发射出新的值时打上read标记。

处理这种情况最安全的方式是结合messages和currentThread流，确保我们不计算当前Thread的任何消息。

我们使用的combineLatest操作，我们已经在之前使用过了：

```
# code/rxjs/chat/app/ts/components/ChatNavBar.ts

export class ChatNavBar {
  unreadMessagesCount: number;

  constructor(public messagesService: MessagesService,
               public threadsService: ThreadsService) {
  }

  onInit(): void {
    this.messagesService.messages
      .combineLatest(
        this.threadsService.currentThread,
        (messages: Message[], currentThread: Thread) =>
          [currentThread, messages] )

      .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
        this.unreadMessagesCount =
          _
            .reduce(
              messages,
              (sum: number, m: Message) => {
                let messageIsInCurrentThread: boolean = m.thread &&
```

```

        currentThread &&
        (currentThread.id === m.thread.id);
        if (m && !m.isRead && !messageIsInCurrentThread) {
            sum = sum + 1;
        }
        return sum;
    },
    0);
});
}
}

```

如果你不是TypeScript专家，你可能会发现上面的语法有点难解析，在combineLatest我们返回有currentThread和messages的2个元素数组素。

然后我们订阅该流，我们调用函数解构这些对象。接下去我们对message使用reduce来计算未读和不在当前线程的消息数。

ChatNavBar 模板

在我们的视图，我们剩下的唯一要做的是显示我们的unreadMessagesCount：

```

# code/rxjs/chat/app/ts/components/ChatNavBar.ts

@Component({
  selector: 'nav-bar',
  template: `
    <nav class="navbar navbar-default">
      <div class="container-fluid">
        <div class="navbar-header">
          <a class="navbar-brand" href="https://ng-book.com/2">
            
              ng-book 2
          </a></div>
          <p class="navbar-text navbar-right">
            <button class="btn btn-primary" type="button">
              Messages <span class="badge">
                {{unreadMessagesCount}}</span></button>
            </p>
          </div>
        </nav>
      `
})

```

完成的ChatNavBar代码

下面是完整的ChatNavBar代码：

```
# code/rxjs/chat/app/ts/components/ChatNavBar.ts

import {Component, OnInit} from 'angular2/core';
import {MessagesService, ThreadsService} from '../services/services';
import {Message, Thread} from '../models';
import * as _ from 'underscore';

@Component({
  selector: 'nav-bar',
  template: `
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <a class="navbar-brand" href="https://ng-book.com/2">
        
        ng-book 2
      </a>
    </div>
    <p class="navbar-text navbar-right">
      <button class="btn btn-primary" type="button">
        Messages <span class="badge">{{unreadMessagesCount}}</span>
      </button>
    </p>
  </div>
</nav>
`
})
export class ChatNavBar implements OnInit {
  unreadMessagesCount: number;

  constructor(public messagesService: MessagesService,
               public threadsService: ThreadsService) {
  }

  ngOnInit(): void {
    this.messagesService.messages
      .combineLatest(
        this.threadsService.currentThread,
        (messages: Message[], currentThread: Thread) =>
          [currentThread, messages] )
  }
}
```



```

    .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
      this.unreadMessagesCount =
        _.reduce(
          messages,
          (sum: number, m: Message) => {
            let messageIsInCurrentThread: boolean = m.thread &&
              currentThread &&
              (currentThread.id === m.thread.id);
            if (m && !m.isRead && !messageIsInCurrentThread) {
              sum = sum + 1;
            }
            return sum;
          },
          0);
    });
  }
}

```

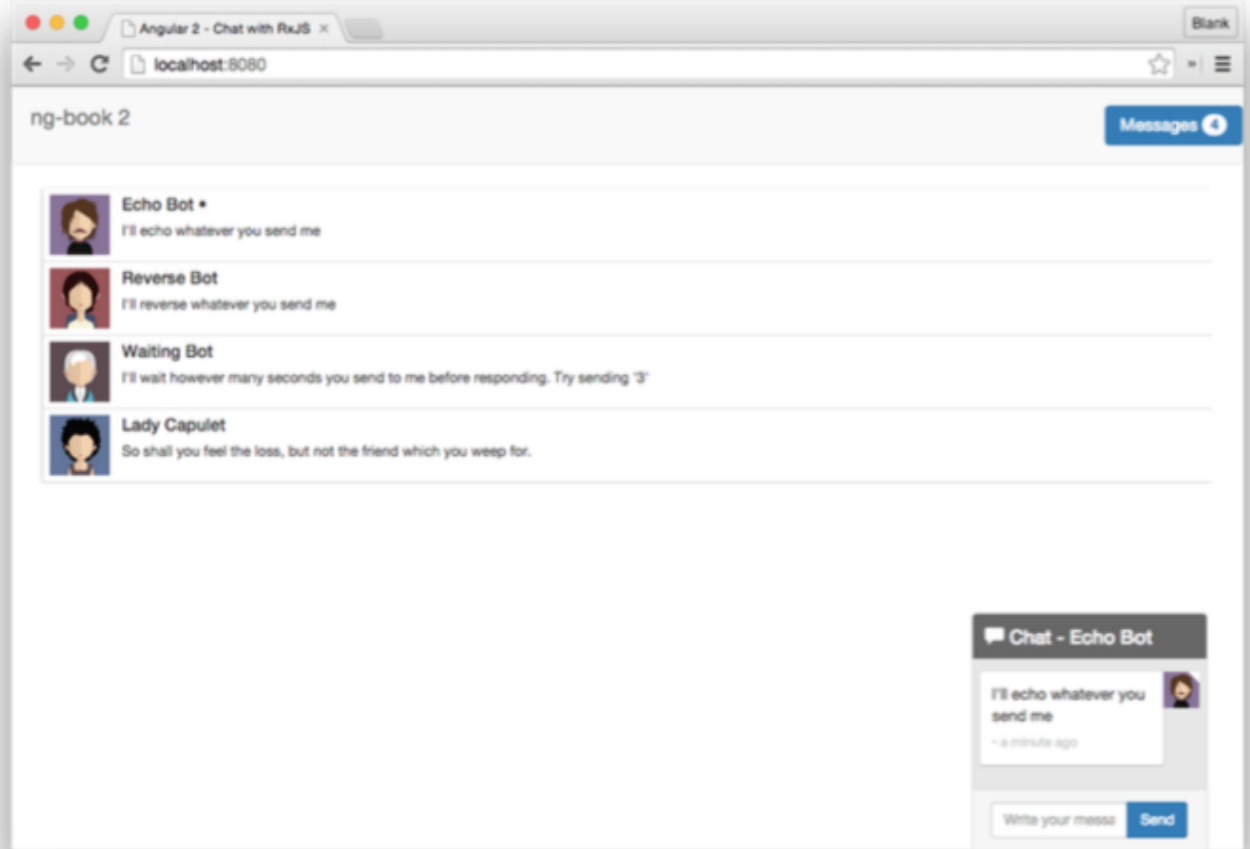


kittencup

about 1 month ago

总结

如果我们把他们放在一起我们就有一个功能齐全的聊天应用程序！



Completed Chat Application

如果你签出 `/rxjs/chat/app/ts/ChatExampleData.ts` 代码，你会看到，我们已经为你写了几个机器人，可以用来聊天的。下面是Reverse Bot的代码摘录：

```
let rev: User = new User("Reverse Bot", require("images/avatars/female-  
avatar-4.png"));  
let tRev: Thread = new Thread("tRev", rev.name, rev.avatarSrc);
```

```
# code/rxjs/chat/app/ts/ChatExampleData.ts
```

```
messagesService.messagesForThreadUser(tRev, rev)  
  .forEach( (message: Message) => {  
    messagesService.addMessage(  
      new Message({  
        author: rev,  
        text: message.text.split("").reverse().join(""),  
        thread: tRev  
      })  
    );  
  });
```

```
});
```

上面可以看到，我们已经通过messages- ForThreadUser订阅了“Reverse Bot”的消息。试着写你自己的一些机器人。



kittencup

about 1 month ago

下一步

在RxJS一些方法来改善这个聊天应用程序将会变得更强，然后把它挂到一个实际的API。我们将在HTTP这章讨论如何使用API请求。现在，享受您的喜欢聊天应用程序！

Comment on issue

Sign in to comment

or [sign up](#) to join this conversation on GitHub



Desktop version



Open

HTTP #42



kittencup opened this issue
about 1 month ago



ng-book 2

该issue关闭讨论，如有问题请去 [#43](#) 提问

目录

- [介绍](#)
- [使用 angular2/http](#)
- [一个基本的请求](#)
- [编写一个YouTubeSearchComponent](#)
- [angular2 http API](#)



kittencup added the **ng-book 2** label about 1 month ago



kittencup
about 1 month ago

介绍

Angular 2 有它自己的HTTP库，我们可以用它来调用外部API。

当我们调用外部服务器时，我们希望我们的用户仍然能够与页面进行交互。也就是说，我们不希望我们的页面冻结直到从外部服务器的HTTP请求返回。为了达到这种效果，我们的HTTP请求是异步的。

异步处理从历史上看，比处理同步码更麻烦。在Javascript中，一般有三种方法来处理异步代码：

- 1.Callbacks

- 2.Promises
- 3.Observables

在Angular 2，处理异步代码的首选方法是使用Observables，所以我们将在本章覆盖。

有一整章的RxJS和Observable：在这一章中我们将使用Observables并没有太多解释。如果你是刚开始读这本书的这一章，你应该知道，在上一章有更详细的介绍。

在本章中，我们将要：

- 1.显示1个基本的http例子
- 2.创建一个YouTube search-as-you-type 组件
- 3.讨论关于HTTP库API的细节

本章中完整示例代码可以在示例代码中的HTTP文件夹中找到。该文件夹包含README.md，是构建和运行项目的说明。

在阅读本章的时候，试着在代码中运行代码，并自由发挥，以获得关于它的所有作品的更深入的了解。



kittencup

about 1 month ago

使用angular2 http

HTTP在Angular 2中已经分解成一个独立的模块。这意味着要使用它，你需要导入angular2/http模块，例如

```
import { Http, Response, RequestOptions, Headers } from
'angular2/http';
```

从 angular2/http 导入

在我们的app.ts中，我们导入HTTP_BINDINGS,这是一个模块的便利集合。

```
# code/http/app/ts/app.ts
```

```
import {Component, bootstrap, View} from "angular2/angular2";
```

```
import {HTTP_BINDINGS} from "angular2/http";
```

当我们引导我们的应用程序，我们将添加HTTP_PROVIDERS作为一个依赖。其效果是，我们将能够注入HTTP（和其他几个模块）到我们的组件。

```
bootstrap(HttpApp, [HTTP_PROVIDERS]);
```

现在 我们就可以注入Http服务到我们的组件里(在任何地方，实际上也可以使用DI)

```
class MyFooComponent {  
  constructor(public http: Http) {}  
  makeRequest(): void {  
    // do something with this.http ...  
  }  
}
```



kittencup

about 1 month ago

一个基本的请求

我们要做的第一件事是向[jsonplaceholder API](#)创建一个简单的GET请求的

我们要做的是：

- 1.有一个按钮触发makeRequest
- 2.makeRequest将会调用http库对我们的API进行GET请求
- 3.当请求返回，我们将更新this.data数据结果，这将在视图中渲染。

下面是我们的例子中的截图：

Basic Request

Make Request

```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehende
rit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"
}
```

Basic Request

创建SimpleHTTPComponent @Component

我们要做的第一件事就是导入几个模块，然后为我们@Component指定一个selector

```
# code/http/app/ts/components/SimpleHTTPComponent.ts

import {Component, View, NgIf} from "angular2/angular2";
import {Http, Response} from "angular2/http";

@Component({
  selector: "simple-http"
})
```

创建SimpleHTTPComponent 模板

下一步创建我们的视图

```
# code/http/app/ts/components/SimpleHTTPComponent.ts

@Component({
  selector: 'simple-http',
  template: `
    <h2>Basic Request</h2>
    <button type="button" (click)="makeRequest()">Make Request</button>
    <div *ngIf="loading">loading...</div>
    <pre>{{data | json}}</pre>
  `
})
```

首先我们指定一个我们用到的NgIf指令

我们的模板有三个有趣的部分：

- 1.button
- 2.loading指示器
- 3.data

在button上 我们绑定一个(click)来调用在我们控制器里的makeRequest函数，我们将立即定义这个函数

我们要告诉用户我们的请求在加载中，所以使用ng-if如果实例变量loading是true的话显示loading

data是一个对象，比较好的调试对象的方式是我们在这里使用的JSON pipe。我们在这里使用pre标签，易于我们阅读的格式。

创建SimpleHTTPComponent控制器

我们开始为我们的SimpleHTTPComponent定义一个新的类

```
# code/http/app/ts/components/SimpleHTTPComponent.ts

export class SimpleHTTPComponent {
  data: Object;
  loading: boolean;
```

我们有2个实例变量:data和loading, 这将被分别用于我们的API返回值和loading指示器。

接下去我们定义construct:

```
# code/http/app/ts/components/SimpleHTTPComponent.ts

constructor(public http: Http) {}
```

constructor内容为空，但我们注入一个模块:Http

记住当我们在 `public http: Http` 使用public关键字,TypeScript将会将http分配给this.http，这是一个缩写：

```
// other instance variables here
http: Http;
```



```

constructor(http:Http){
    this.http = http;
}

```

现在，让我们实现makeRequest函数，来创建我们的第一个HTTP请求：

```

# code/http/app/ts/components/SimpleHTTPComponent.ts

makeRequest(): void {
    this.loading = true;
    this.http.request("http://jsonplaceholder.typicode.com/posts/1")
        .toRx()
        .subscribe((res: Response) => {
            this.data = res.json();
            this.loading = false;
        });
}

```

当我们调用makeRequest,第一件事我们会设置this.loading = true,这将打开我们的视图中的loading指示器。

创建一个HTTP请求很简单：我们向创建一个GET请求。我们调用this.http.request并传入URL

http.request 返回Observable。我们可以使用subscribe来订阅这个变化

```

# code/http/app/ts/components/SimpleHTTPComponent.ts

.subscribe((res: Response) => {
    this.data = res.json();
    this.loading = false;
});

```

当我们http.request返回，这个流会发色一个Response对象，我们通过使用json提取响应的主体对象，然后我们设置this.data为该对象。

因为我们有一个响应，我们不需要在加载了，所以我们设置this.loading = false

.subscribe还可以处理failures，分别通过传递函数到流的第二和第三个参数。在生产应用程序，这将是一个好主意来处理这些情况了。也就是说，如果请求失败（即流发

出错误) this.loading也应设置为false。

完整的SimpleHTTPComponent

下面是我们的SimpleHTTPComponent完全样子：

```
# code/http/app/ts/components/SimpleHTTPComponent.ts

/*
 * Angular
 */
import {Component} from 'angular2/core';
import {Http, Response} from 'angular2/http';

@Component({
  selector: 'simple-http',
  template: `
    <h2>Basic Request</h2>
    <button type="button" (click)="makeRequest()">Make Request</button>
    <div *ngIf="loading">loading...</div>
    <pre>{{data | json}}</pre>
  `
})
export class SimpleHTTPComponent {
  data: Object;
  loading: boolean;

  constructor(public http: Http) {
  }

  makeRequest(): void {
    this.loading = true;
    this.http.request('http://jsonplaceholder.typicode.com/posts/1')
      .subscribe((res: Response) => {
        this.data = res.json();
        this.loading = false;
      });
  }
}
```



kittencup

about 1 month ago

编写一个YouTubeSearchComponent

最后一个例子是一个简单的方式来获得一个API Serve的数据到你的代码。现在，让我们尝试建立一个更复杂的例子。

在本节中，我们要建立一个方法来搜索YouTube作为你的类型。当搜索返回时，我们将显示一个视频缩略图的结果列表，以及每个视频的描述和链接。

下面是一个截图当我搜索“cats playing ipads”：



在这个例子中，我们打算写几件事情：

- 1.一个SearchResult对象，我们每个结果所需的保存的数据
- 2.YouTubeService将管理YouTube API请求，并转化为结果为SearchResult[]流
- 3.一个SearchBox组件，它会呼出到YouTube的服务为用户类型
- 4.一个SearchResultComponent组件，将会渲染一个指定的SearchResult
- 5.一个YouTubeSearchComponent，这将封装我们整个YouTube的搜索应用程序，和渲染结果列表

让我们一次来处理每一个部分

Patrick Stapleton 有一个很棒的名为[angular2-webpack-starter](#)资源。次资源有一个RxJS例子，用来自动完成Github上的库。在这一节一些想法是从这个例子启发的。这是有很多奇妙的例子项目你应该看看。

编写SearchResult

首先，让我们开始写一个基本的SearchResult类。这个类只是一个方便用来存储从我们的搜索结果中感兴趣的特定字段

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

class SearchResult {
```

```

id: string;
title: string;
description: string;
thumbnailUrl: string;
videoUrl: string;

constructor(obj?: any) {
  this.id = obj && obj.id || null;
  this.title = obj && obj.title || null;
  this.description = obj && obj.description || null;
  this.thumbnailUrl = obj && obj.thumbnailUrl || null;
  this.videoUrl = obj && obj.videoUrl ||
`https://www.youtube.com/watch?v=${this.id}`;
}
}

```

采取obj?:any这种模式，让我们模拟关键字参数。我们的想法是，我们可以创建一个新的SearchResult，并只是传递一个包含我们要指定键的对象。

唯一在这里指出的是，我们用了一个硬编码的URL格式构建videoUrl。欢迎使用更多参数的函数来改变这个功能，如果你需要的话，可以直接在视图中使用id来构建这个URL。

编写 YouTubeService

API

这个例子中我们将要使用[YouTube V3搜索API](#)。

为了使用这个API，你需要有一个API key. 在示例代码中我已包含了API key给你使用，然而，当你读到本书时，你可能会发现它的速度限制。如果发生这种情况，你需要发行你自己的key。

要发行你自己的key 请查看这个[文档](#),为简单起见，我已经注册了一个server key，如果你打算把你的javascript代码在线使用，你可能应该使用一个browser key。

我们建立2个常量为我们的YouTubeService映射我们API key和API URL

```

let YOUTUBE_API_KEY: string = "XXX_YOUR_KEY_HERE_XXX";
let YOUTUBE_API_URL: string =
"https://www.googleapis.com/youtube/v3/search";

```

最终 我们要测试我们的应用程序。当我们测试时，我们发现的一个问题是，我们并不总是

想要测试生产，我们通常要测试的分期或开发的API。

为了帮助这种环境的配置，我们可以做的事情之一是让这些常量注入。

为什么我们要注入这些常量，而不是让它们以正常方式使用？因为如果我们让他们注入，我们可以

- 1.在部署时给定的环境下注入正确的常量
- 2.在测试时更换注入的值

通过注入这些值，对于这些值我们有很多灵活性

为了使这些值注射，我们使用 `provide(...,{useValue:....})` 语法，是这样的：

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

export var youtubeServiceInjectables: Array<any> = [
  provide(YouTubeService,{useClass:YouTubeService}),
  provide(YOUTUBE_API_KEY,{useValue:YOUTUBE_API_KEY}),
  provide(YOUTUBE_API_URL,{useValue:YOUTUBE_API_URL})
];
```

在这里我们指定我们要绑定的 YOUTUBE_API_KEY injectably 到YOUTUBE_API_KEY的值(对于YOUTUBE_API_URL，和我们接下去要定义的YouTubeService是一样的)

如果你还记得，在我们的应用程序中提供一些可供选择的東西，我们需要使它成为bootstrap的依赖,因此我们在这export youtubeServiceInjectables,在我们的app.ts我们就可以import它

```
// http/app.ts
import {youtubeServiceInjectables} from
"components/YouTubeSearchComponent"; // ....
// further down
bootstrap(HttpApp, [HTTP_PROVIDERS, youtubeServiceInjectables]);
```

现在我们可以注入YOUTUBE_API_KEY代替直接使用变量。

YouTubeService Constructor

我们创建我们的YouTubeService，并使用注解 `@Injectable`：

```
# code/http/app/ts/components/YouTubeSearchComponent.ts
@Injectable()
export class YouTubeService {
  constructor(public http: Http,
    @Inject(YOUTUBE_API_KEY) private apiKey: string,
    @Inject(YOUTUBE_API_URL) private apiUrl: string) {
  }
}
```

在construct我们注入3个东西:

- 1.http
- 2.YOUTUBE_API_KEY
- 3.YOUTUBE_API_URL

注意我们也为3个参数创建了3个实例变量, 这意味着我们可以从this.http,this.apiKey和this.apiUrl来分别访问它们

注意, 我们使用@Inject(YOUTUBE_API_KEY)明确地注入。

YouTubeService 搜索

接下去 来实现search函数, search需要一个query字符串, 并返回一个Observable,它发射出一个 SearchResult[]的流, 也就是说发射每一个SearchResults数组

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

search(query: string): Rx.Observable<SearchResult[]> {
  let params: string = [
    `q=${query}`,
    `key=${this.apiKey}`,
    `part=snippet`,
    `type=video`,
    `maxResults=10`
  ].join("&");
  let queryUrl: string = `${this.apiUrl}?${params}`;
}
```

我们手动的方式建立queryUrl。首先我们简单地将每个query参数放到params变量内(你可以通过[阅读API文档](#)搜索找到每个参数的意义)

然后, 我们通过连接apiUrl和params建立了queryUrl。

现在有一个queryUrl, 我们可以创建一个请求:

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

return this.http.get(queryUrl)
    .map((response: Response) => {
        return (<any>response.json()).items.map(item => {
            // console.log("raw item", item); // uncomment if you want to
debug
            return new SearchResult({
                id: item.id.videoId,
                title: item.snippet.title,
                description: item.snippet.description,
                thumbnailUrl: item.snippet.thumbnails.high.url
            });
        });
    });
```

在这里我们返回http.get的值，我们使用map从请求中获得响应，我们使用json()将响应主体转换为对象，通过map遍历它们为每个item，并将其转换为SearchResult

如果你想看看原item的样子，就取消该console.log的注释，在你的浏览器开发者控制台进行检查。

注意我们调用 (<any>response.json()).items. ,在这里发生了什么？。在这里我要告诉TypeScript在这里我没有兴趣做严格的类型检查。

当使用JSON API的工作中，我们通常不会给API响应定义类型，所以TypeScript不会知道返回的对象有一个items键,所以编辑器会报错，我们可以称 response.json() ["items"] 强制转换为一个数组。 but in this case (and in creating the SearchResult, it's just cleaner to use an any type, at the expense of strict type checking

YouTubeService 完整代码

下面是我们YouTubeService的完整代码：

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
```

```

export class YouTubeService {
  constructor(public http: Http,
    @Inject(YOUTUBE_API_KEY) private apiKey: string,
    @Inject(YOUTUBE_API_URL) private apiUrl: string) {

  }

  search(query: string): Observable<SearchResult[]> {
    let params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    let queryUrl: string = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl)
      .map((response: Response) => {
        return (<any>response.json()).items.map(item => {
          // console.log("raw item", item); // uncomment if you want to
          debug

          return new SearchResult({
            id: item.id.videoId,
            title: item.snippet.title,
            description: item.snippet.description,
            thumbnailUrl: item.snippet.thumbnails.high.url
          });
        });
      });
  }
}

```

编写SearchBox

SearchBox在应用中起着关键的作用：它是我们的UI和youtubeservice之间的协调者。

SearchBox将:

- 1.在input上监听keyup，给YouTubeService提交一个search
- 2.当我们loading时，发射一个loading事件(或者没有loading时)
- 3.当我们有新的搜索结果时，发射一个results事件

定义SearchBox @Component

来定义我们的 SearchBox @Component:


```
# code/http/app/ts/components/YouTubeSearchComponent.ts

@Component({
  outputs: ["loading", "results"],
  selector: "search-box"
})
```

selector, 我们已经看到很多次: 这允许我们能够创建一个标记。

outputs键指定了从这个组件将要被发射的事件列表。也就是说, 我们可以在我们视图中使用 `(event)="callback()"` 语法来监听这个组件的事件, 例如, 这里我们将如何在我们的视图中使用搜索框标记:

```
<search-box
  (results)="updateResults($event)"
  (loading)="loading = $event"
></search-box>
```

在这个例子中, 当SearchBox组件发射一个loading事件, 我们将在其父组件上设置loading变量, 同样的, 当SearchBox发射一个results事件, 我们将会将值传递给父组件上的updateResults函数

在@Component配置中, 我们只是简单地指定了字符串“loading”和“results”的事件的名称。在这个事件中, 每一个事件在控制器类中都会有一个对应的EventEmitter实体变量, 等我们会实现它们。

现在, 请记住, @Component就像我们组件的公共API, 所以在这里我们只是指定事件的名称, 并且之后我们会考虑EventEmitters的实现。

定义SearchBox 模板

我们的视图很简单。就一个input标签:

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

@Component({
  outputs: ['loading', 'results'],
  selector: 'search-box',
  template: `
```

```
<input type="text" class="form-control" placeholder="Search"
autofocus>
`
})
```

定义SearchBox控制器

我们的SearchBox控制器是一个新的类:

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

class SearchBox implements OnInit {
  loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  results: EventEmitter<SearchResult[]> = new
  EventEmitter<SearchResult[]>();
}
```

我们说这个类实现了Oninit, 因为我们要使用ngOnInit生命周期事件回调, 如果一个类实现了OnInit, 那么这个oninit方法将会在第一次变化检测检查后调用。

onInit是一个做初始化(和构造函数比较)的好地方, 因为在组件上设置一个属性在构造函数上是不能被接受的

定义SearchBox控制器 constructor

但首先我们讨论构造函数:

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

constructor(public youtube: YouTubeService,
             private el: ElementRef) {
}
```

在我们的构造函数上我们注入:

- 1.YouTubeService
- 2.ElementRef是连接了angular包装了该组件的原始元素

我们同时设置注射为实例变量

定义SearchBox控制器 ngOnInit

在这个Input框上，我想要监听keyup事件，事情是这样的，如果我们只是简单的对每个keyup进行搜索，就不是十分理想。有三件事我们可以做，以提高用户体验：

- 1.过滤掉任何空或短查询
- 2."debounce"输入，即，不搜索的每一个字符，但只有当用户停止打字后很短的时间内
- 3.如果用户已作出新的搜索,丢弃任何旧的搜索

我们可以手动绑定keyup事件，为每一个keyup事件调用函数，并过滤和debouncing，然而，有一个更好的办法，将keyup事件转换为一个Observable流

RxJS提供一种方式来监听元素的事件，使用Rx.Observable.fromEvent，我们可以这样使用它：

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

ngOnInit(): void {
  // convert the `keyup` event into an observable stream
  Observable.fromEvent(this.el.nativeElement, 'keyup')
```

注意在fromEvent

- 第一个参数是this.el.nativeElement(是连接到组件原始的DOM元素)
- 第二个参数是一个字符串的"keyup",这是我想变成一个流的事件名字

现在，我们在这个流上可以进行一些RxJS魔术，把它变成SearchResult。让我们一步一步来。

鉴于keyup事件流我们可以链接更多的方法，在接下来的几段中，我们将给我们的数据流使用多种函数，这将改变数据流。然后，我们会一起展示整个例子。

首先，让我们提取输入标签的值：

```
.map((e: any) => e.target.value) // extract the value of the input
```

以上表达的是，map每一个keyup事件，找出事件目标(e.target表示我们的input元素)，并提取这个元素的值，这意味着我们的流现在是一个字符串流

```
.filter((text: string) => text.length > 1)
```

filter意味着流将不会发射那些搜索字符串长度小于1的，如果你想忽略短搜索，你可以把这个设置为一个更高的数字。

```
.debounce(250)
```

debounce意味不发射那些快于250ms的请求，那就是，我们不会搜索每一次按键，而是在用户暂停输入250MS后进行搜索

```
.do(() => this.loading.next(true))
```

使用在流中的方式是每个事件执行一个函数,但它并没有改变流中的任何东西。这里的想法是，我们要去搜索，它有足够的字符，用户也暂停输入了，所以我们要显示loading指示

this.loading是一个EventEmitter，我们要显示loading,通过发射一个true,我们通过调用EventEmitter的next来发射一些东西，this.loading.next(true)意味着，在loading EventEmitter发射一个true，当我们在这个组件上监听loading事件,\$event值现在是true(我们在下面会更密切地关注使用\$event)

```
.flatMapLatest((query: string) => this.youtube.search(query))
```

从本质上讲，通过flatMapLatest，表示我们要忽略所有的搜索时间，但只保留最新的那次。也就是说，如果一个新的搜索进来，我们要使用最新的和放弃的其余部分。

对于每一个进来的查询，我们将执行我们的YouTubeService搜索。把它们链在一起，我们有这样的：

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

Observable.fromEvent(this.el.nativeElement, 'keyup')
  .map((e: any) => e.target.value) // extract the value of the
input
  .filter((text: string) => text.length > 1) // filter out if empty
  .debounceTime(250) // only once every
250ms
  .do(() => this.loading.next(true)) // enable loading
  // search, discarding old events if new input comes in
  .map((query: string) => this.youtube.search(query))
```

```
.switch()
```

RxJS的API可以是一个有点吓人，因为API功能强大。这就是说，我们的代码非常少行实现了复杂的事件处理流！

因为我们调用YouTubeService我们的流现在已经是SearchResult[]流了。我们可以订阅这个流，并执行相应的动作。

subscribe需要3个参数onSuccess, onError, onCompletion.

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

.subscribe(
  (results: SearchResult[]) => { // on success
    this.loading.next(false);
    this.results.next(results);
  },
  (err: any) => { // on error
    console.log(err);
    this.loading.next(false);
  },
  () => { // on completion
    this.loading.next(false);
  }
);
```

第一个参数指定了，当流发射一个正常的事件我们需要做什么，在这我们发射2个EventEmitters事件

- 1.我们调用this.loading.next(false),说明我们已经停止loading
- 2.我们调用this.results.next(results),这将发出一个包含结果列表的事件

第二个参数指定了当我们的流发生错误时，应该做什么，在这里我们设置this.loading.next(false)并且打印error日志

第3个参数指定了当流完成时应该做什么，在这里我们也是发射loading完成

SearchBox组件 全部代码

连在一起，这是我们的搜索框组件的完整代码：

```

# code/http/app/ts/components/YouTubeSearchComponent.ts

@Component({
  outputs: ['loading', 'results'],
  selector: 'search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search"
autofocus>
  `
})
class SearchBox implements OnInit {
  loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  results: EventEmitter<SearchResult[]> = new
EventEmitter<SearchResult[]>();

  constructor(public youtube: YouTubeService,
               private el: ElementRef) {

  }

  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    Observable.fromEvent(this.el.nativeElement, 'keyup')
      .map((e: any) => e.target.value) // extract the value of the
input
      .filter((text: string) => text.length > 1) // filter out if empty
      .debounceTime(250) // only once every
250ms
      .do(() => this.loading.next(true)) // enable loading
      // search, discarding old events if new input comes in
      .map((query: string) => this.youtube.search(query))
      .switch()
      // act on the return of the search
      .subscribe(
        (results: SearchResult[]) => { // on success
          this.loading.next(false);
          this.results.next(results);
        },
        (err: any) => { // on error
          console.log(err);
          this.loading.next(false);
        },
        () => { // on completion
          this.loading.next(false);
        }
      );
  }
}

```

编写SearchResultComponent

SearchBox是相当复杂的。现在让我们来处理一个更容易的组件：

SearchResultComponent。该SearchResultComponent的工作就是使一个单一的SearchResult。

这里真的没有什么新的想法，所以让我们把它全部一次写出来：

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

@Component({
  inputs: ['result'],
  selector: 'search-result',
  template: `
<div class="col-sm-6 col-md-3"> <div class="thumbnail">
 <div class="caption">
<h3>{{result.title}}</h3> <p>{{result.description}}</p> <p><a href="
{{result.videoUrl}}"
class="btn btn-default" role="button">Watch</a></p> </div>
</div>
</div>
` })
export class SearchResultComponent {
  result: SearchResult;
}
```

几件事情

@Component需要有一个属性result，我们将要把SearchResult分配给该组件

模板 显示一个标题，描述和视频的缩略图，然后通过一个按钮链接到视频。

该SearchResultComponent简单地存储SearchResult中实例变量的result。

编写YouTubeSearchComponent

最后一个组件我们要实现YouTubeSearchComponent.这是将一切组成部分联系在一起。

YouTubeSearchComponent @Component

```
# code/http/app/ts/components/YouTubeSearchComponent.ts
```

```
@Component({
  selector: 'youtube-search',
  directives: [SearchBox, SearchResultComponent],
```

我们的@**Component**注解是非常简单的，设置selector为youtube-search

YouTubeSearchComponent控制器

我们在看模板之前，先看下YouTubeSearchComponent控制器

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

export class YouTubeSearchComponent {
  results: SearchResult[];
  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

这个组件持有一个实体变量:results是一个SearchResults[]

我们也定义一个函数updateResults，updateResults只是需要任何新的SearchResult[]给他，并设置this.results为新值。

我们会在模板中使用Results和updateResults

YouTubeSearchComponent 模板

我们视图需要做3件事

- 1.如果我们在加载中 显示loading指示器
- 2.监听search-box事件
- 3.显示搜索结果

接下去来看看我们的模板，我们在header中建立了一些基本的结构和显示loading gif

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

template: `
```



```

<div class='container'>
  <div class="page-header">
    <h1>YouTube Search
    <img
      style="float: right;"
      *ng-if="loading"
      src=${loadingGif} />
    </h1>
  </div>

```

注意 我们的img的src为\${loadingGif} 这loadingGif变量来自先前在程序中的声明，在这里，我们采取的是WebPack的图像加载功能

如果您想了解更多有关如何工作，可查看本章示例代码中的webpack配置或者 checkout [image-webpack-loader](#)

如果loading是true，我们只想显示这个loading图片，所以我们使用ng-if来实现这个功能

接下去让我们看看search-box标签

```

# code/http/app/ts/components/YouTubeSearchComponent.ts

<div class="row">
  <div class="input-group input-group-lg col-md-12">
    <search-box
      (loading)="loading = $event"
      (results)="updateResults($event)"
    ></search-box>
  </div>
</div>

```

这里有趣的是，我们如何绑定到loading和results事件。请注意，我们在这里使用了 (event)="action()"

对于loading事件，我们运行表达式 loading = \$event. \$event的值将会被从EventEmitter发射过来的event值给取代，也就是说，在我们searchBox组件，当我们调用 this.loading.next(true)，那么这个\$event会变 true

同样，都与results事件，当一个新设置的results被发射我们调用updateResults()函数，这有更新我们的组件results实例变量的效果。

最后，我们要在这个组件中循环results渲染每一个result为search-result

```
# code/http/app/ts/components/YouTubeSearchComponent.ts
<div class="row">
  <search-result
    *ng-for="#result of results"
    [result]="result">
  </search-result>
</div>
</div>
`
```

YouTubeSearchComponent 完整代码

下面是该YouTubeSearchComponent的完整代码：

```
# code/http/app/ts/components/YouTubeSearchComponent.ts

@Component({
  selector: 'youtube-search',
  directives: [SearchBox, SearchResultComponent],
  template: `
<div class='container'>
  <div class="page-header">
    <h1>YouTube Search
      <img
        style="float: right;"
        *ngIf="loading"
        src='${loadingGif}' />
      </h1>
    </div>

  <div class="row">
    <div class="input-group input-group-lg col-md-12">
      <search-box
        (loading)="loading = $event"
        (results)="updateResults($event)"
      ></search-box>
    </div>
  </div>

  <div class="row">
    <search-result
      *ngFor="#result of results"
      [result]="result">
    </search-result>
  </div>
`
```

```

</div>
-
}))
export class YouTubeSearchComponent {
  results: SearchResult[];

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a
look
  }
}
}

```

就讲到这里，一个search-as-you-type YouTube 视频的功能实现，如果你还没有代码，尝试运行示例代码



kittencup

about 1 month ago

angular2 http API

当然，我们创建的所有HTTP请求都是简单的GET请求。重要的是我们要知道们如何创建其他的请求。

创建一个POST请求

使用angular/http创建POST请求非常像GET请求 所不同的是，我们有一个附加参数：body。

[jsonplaceholder API](#)还提供了一个URL来测试我们的POST请求,让我们来创建一个POST

```

# code/http/app/ts/components/MoreHttpRequests.ts

makePost(): void {
  this.loading = true;
  this.http.post(
    "http://jsonplaceholder.typicode.com/posts",
    JSON.stringify({
      body: "bar",
      title: "foo",
      userId: 1
    })
  )
}

```

```

.toRx()
.subscribe((res: Response) => {
  this.data = res.json();
  this.loading = false;
});
}

```

注意在第2个参数，我们使用JSON.stringify将一个object转换为JSON字符串

PUT / PATCH / DELETE / HEAD

还有其他一些很常见的HTTP请求，我们以大致相同的方式调用他们。

- http.put和http.patch 分别映射为PUT和PATCH,它们都需要一个URL和一个body
- http.delete和http.head分别映射了DELETE和HEAD，它们需要一个URL(不需要body)

以下是我们可以创建一个DELETE请求：

```

# code/http/app/ts/components/MoreHTTPRequests.ts

makeDelete(): void {
  this.loading = true;
  this.http.delete("http://jsonplaceholder.typicode.com/posts/1")
    .subscribe((res: Response) => {
      this.data = res.json();
      this.loading = false;
    });
}

```

RequestOptions

所有的http方法最后一个参数都是可选的:RequestOptions,该RequestOptions对象封装：

- method
- headers
- body
- mode
- credentials
- cache
- url

- search

比方说，我们要精心创建一个使用特殊的X-API-TOKEN头的GET请求。我们可以创建一个像这样的header的请求：

```
# code/http/app/ts/components/MoreHttpRequests.ts

makeHeaders(): void {
  let headers: Headers = new Headers();
  headers.append('X-API-TOKEN', 'ng-book');
  let opts: RequestOptions = new RequestOptions();
  opts.headers = headers;
  this.http.get('http://jsonplaceholder.typicode.com/posts/1', opts)
    .subscribe((res: Response) => {
      this.data = res.json();
    });
}
```

总结

Angular/http 还很年轻，但已经有足够的各种各样的功能API。

一个关于Angular/HTTP的伟大的事情是，它的支持mock backend，测试是非常有用。可以到测试那章，了解更多关于测试的内容。

Comment on issue

Sign in to comment

or [sign up](#) to join this conversation on GitHub



Desktop version