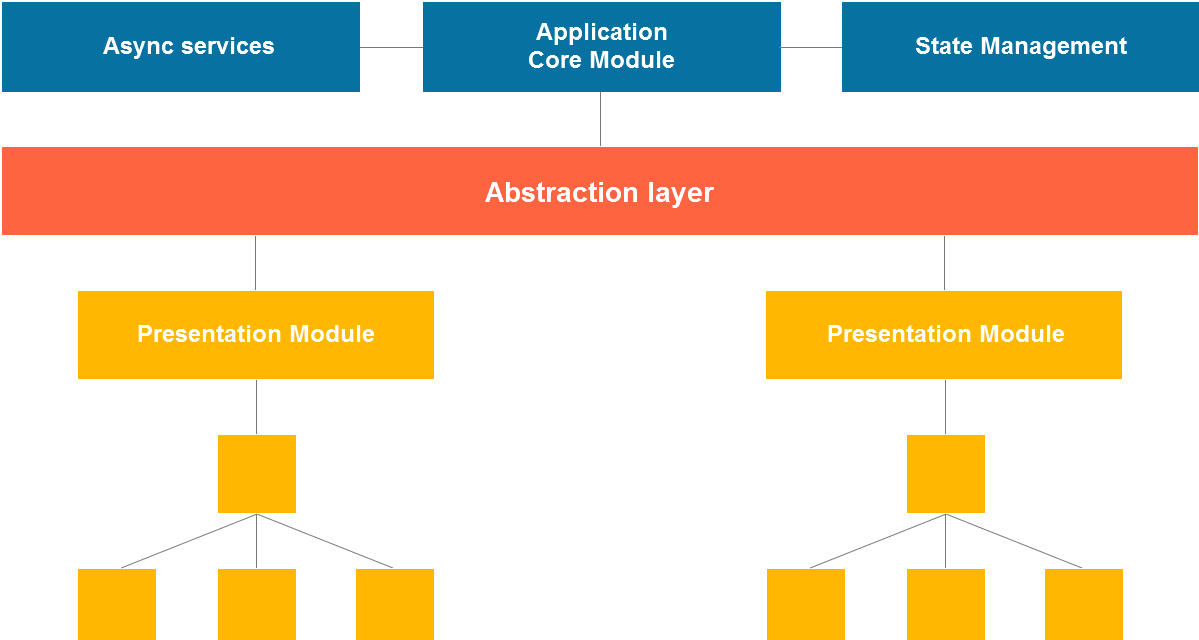
<https://netmedia.io/dev/angular-architecture-patterns-high-level-project-architecture_5589>

<https://github.com/omnitureSolution/AngularArchitecure> 对原程序库进行了升级

<https://github.com/vladeye/druo-dashboard-app/tree/1dc8f2fbc26ab5b441e8ca43b896285a8bc1a157> 原程序库升级到angular 6



这一系列的文章是在对Angular 2项目进行了数月的研究和工作后产生的。它基于经验和观察，试图创建一个可持续的项目架构，更重要的是，它代表了一套指导方针，以保持项目的组织性和可维护性。目标是将应用程序划分为功能层，并为每个功能层定义职责和限制。其背后的理论与框架无关，可以应用于任何框架，但代码示例是用Angular 4编写的。

我们将从更高的层次开始详细解释应用程序的每个部分。最后的结果将是一个示例项目，以说明如何解决一些架构问题。

演示应用程序的完整源代码在Github上发布。

**概述**

[Angular CLI](https://github.com/angular/angular-cli" \t "https://netmedia.io/dev/_blank)

设置空白项目  
定义高级项目架构图  
详细定义每一层  
添加额外的功能来完成项目

基本上，你需要做的就是在你的机器上安装Angular CLI，用内置命令创建一个新项目，然后在浏览器中运行它

npm install -g @angular/cli

ng new PROJECT\_NAME

cd PROJECT\_NAME

ng serve

**高级项目架构**

这是棘手的部分。这取决于您正在构建的应用程序的大小、参与项目的团队成员数量以及您的个人偏好。我们假设在项目往往生长和新的团队成员,与不同层次的专业知识,应该能够加入团队和开始工作在一个应用程序的一部分,即使他们不熟悉整个架构图。  
这些要求要求应用程序具有一些基本特性:

模块化设计  
单向数据流  
可预测的状态管理  
异步请求的通信层  
将表示层与核心层解耦

我们已经可以勾勒出到目前为止我们所拥有的需求的大致框图。下面的设计灵感来自于Nicholas Zakas和他在Aura框架中实施的沙箱( [sandbox principle](https://www.youtube.com/watch?v=b5pFv9NB9fs" \t "https://netmedia.io/dev/_blank)）原则。让我们分析一下每个特征，看看它给我们的生活带来了什么好处。

### **模块化设计（Modular design）** 这意味着我们应该能够在需要时轻松地插入/输出模块，使应用程序的每个部分都是可测试的，并使多个团队成员能够在项目中共同工作。这种方法将应用程序解耦为功能块，使其更容易维护和测试。

从上图中我们可以看到，应用程序被分为三个层——表示层、抽象层和核心层。表示层用于显示HTML和处理用户交互。抽象层处理表示层和核心层之间的通信。核心层包含应用的核心逻辑，如数据操作、与外部世界的通信等。每一层都被分成模块，根据特性进行组织。模块是可以插入/退出的独立代码块。通过查看图1，每个块都是独立的模块。特性是模块被划分的逻辑单元。这意味着我们将把异步服务和数据管理分离到它们自己的模块中，因为它们执行不同的任务。表示模块也是如此。我们可以将身份验证模块与仪表盘模块等分离开来。

表示模块是独立于其他模块的独立单元。它们包括来自应用程序核心的必要依赖(例如配置、实用模块……)，并且它们通过抽象层来实现。抽象层提供了对应用程序核心的访问，这样表示层和核心层就可以解耦。表示模块也包括它们自己的路由。这不是必须的，但我们可以将此职责交给表示层，因为它实现了一个功能模块的全部功能，因为所有相关的部分都在一个地方。这种方法的另一个好处是，路由有变得更复杂的趋势，因此当它们被分离时，维护它们要容易得多。更多关于路由配置的信息可以在Angular路由文档中找到。

同样的东西也代表其他应用程序层。我们可以将服务添加为模块。它们可以在内部开发，也可以作为第三方依赖项添加。例如，我们可以添加工具模块，它包含所有的帮助函数和逻辑，在整个应用程序中可重用。我们还可以为共享的UI组件(智能的和哑的)使用模块，并将它们包含在其他表示模块中。

**单向数据流（Unidirectional data flow)**

状态是一个保存应用程序数据结构的javascript对象。这里我们可以存储显示给用户所需的数据，比如产品列表、登录用户信息等。状态是整个应用程序体系结构中最重要的事情之一(我敢说是最重要的事情)。为什么?因为现代单页应用程序有多个状态突变源。用户交互，https请求，实时服务推送通知，点对点连接，SQLite数据库存储，所有这些都可以改变状态。可预测的应用程序状态对于避免混淆非常重要，并且应用程序之间的状态版本不同，数据也不同。

我们需要一个集中的地方来保存我们的数据。为此，我们选择了ngrx store，它基本上是Angular实现了可观察对象的redux版本。您可以选择其他的库或者自己开发它，但是您应该注意的一件基本的事情是使那些改变状态的函数保持纯净，而不是改变现有的对象。这意味着如果我们有javascript对象，例如产品

var product = {

id: 8,

name: "Television"

};

如果我们想改变name属性的值，我们需要用改变的值创建一个全新的product对象。我们为什么要这样做?

这一点很重要，因为突变会导致不容易调试的错误，如果数据不是不可变的，Angular变化检测就不能像预期的那样工作。

为了更清楚地解释不变性，我们假设有一个UI组件，它有一个输入参数Product。组件等待Product对象传入，以便在模板中显示产品数据。只有当组件的输入参数的引用(在内存中)发生了变化，并且只有当数据用纯函数创建时，引用才会发生变化，Angular才会对组件执行变化检测(并传入新的产品数据)。只有在完全重新创建现有对象时，引用才会改变。旧的将被垃圾回收，因为它在内存中不再有引用。通过这种方式，Angular消除了输入参数没有变化的组件，从而加速了变化检测。因此，可以忽略组件树的整个分支。

想了解更多关于变化检测的信息，这里有一篇由Pascal Precht撰写的很棒的文章和谈话。(https://blog.thoughtram.io/angular/2016/02/22/angular-2-change-detection-explained.html)

**异步请求的通信层**

几乎每一个改变状态的动作都是异步的。我们肯定会有HTTP服务，但很有可能在一段时间后需求改变，我们需要引入一些本地存储，例如SQLite数据库(用于存储用户偏好，PWA移动应用程序等)。或者我们可能需要web套接字，或者点对点通信。所有这些通信类型都是异步的。

在异步调用中，我们需要处理错误、显示通用通知消息、拦截请求、解析响应等。为每个请求单独处理所有这些操作是非常不方便的。我们最终会得到大量重复的代码，这将破坏DRY原则，我们会浪费大量时间重新编写相同的内容。为了简单起见，我们需要一个地方来处理通信逻辑——通信层。

我们必须遵循一条规则:不要让异步服务知道状态管理逻辑。可能很自然地将响应数据保存到服务中的状态，但在这种情况下，我们将通信层与数据管理层紧密耦合。你还记得我们在模块设计课上讲过什么吗

异步服务应该只执行异步请求和返回数据。我们还可以在这里做一些额外的逻辑，比如设置默认报头、解析响应、处理常见错误或插入数据适配器。稍后我们将更详细地讨论执行异步请求后的数据处理。目前，我们应该将这些关注点分开。

**将表示层与核心层解耦**

将表示逻辑与核心分离意味着我们不应该将诸如异步或数据管理服务之类的核心依赖注入UI组件中。这意味着你的组件不应该是这样的:

@Component({

selector: 'product',

template: `Product details`

})

export class ProductComponent{

constructor(private store: Store<AppStore>, private httpService: HttpService){}

}

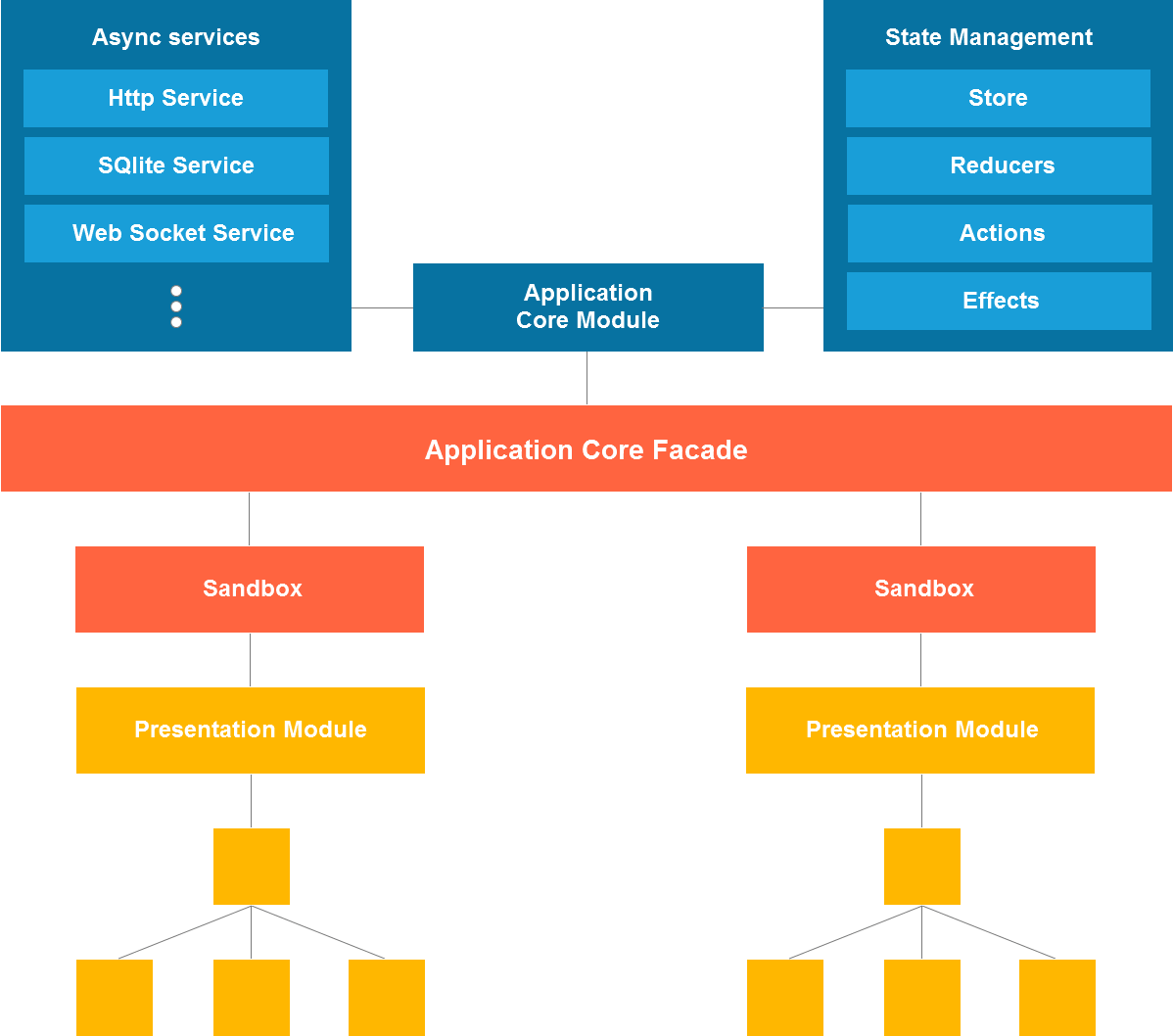
也许您已经习惯处理状态更新或管理来自智能组件(容器)的HTTP请求。这种架构方法主张应该避免这种耦合。为什么这对我们很重要?我们的模块应该有单一的职责，并且只做一件事，这是由关注点分离原则定义的。这意味着表示层应该只关心表示，而不是把手放在核心逻辑上。我们必须在某个地方划清界限，说异步服务和数据管理是我们应用程序的核心部分。这种设计的其他好处是:

组件将更容易测试，因为我们不需要在测试中注入存储和异步服务  
组件将更容易使用，因为它们很小  
更容易分解成多个开发人员任务

因此，智能组件会知道应用程序的状态，但不知道如何管理它。其他核心内容也是如此。但是，它们可以访问抽象层来与应用程序的其余部分通信，并通过向它们传递数据和处理它们的事件来管理它们的子组件。抽象层将充当中介和表示层的外观。这意味着它将公开一个API，并协调多个表示组件和应用程序核心之间的通信。

在这篇博客文章中，我们建立了我们将要构建的项目的基本想法。我们对项目关注点进行了解耦，并定义了一个基础，我们将在其中添加额外的特性。在下一篇博客文章中，我们将更详细地描述每个功能模块，并开始在代码中构建它们。请继续关注。

在前一章中，我们已经定义了主要的项目需求和项目应该如何运行的大致图景。现在我们需要分解每个功能块，我们在图1中绘制了第一章，看看我们需要做些什么来让每个功能块都活着。除了重新定义每个块之外，我们还将添加两个额外的模块来完成我们的项目。扩展的框图现在看起来是这样的:



我们将关注于应用程序的关键构建块，并给出一个示例将其放入代码的何处。异步服务是一组模块，每个模块负责处理与外部世界的不同类型的通信。状态管理由与ngrx库和状态操作相关的部分组成。应用程序核心facade是一个抽象类，它包含应用程序核心API的公共逻辑。它包括每个沙箱将继承的函数，例如获取应用程序状态的特定部分等。Sandbox是一个服务，它扩展了应用程序核心外观，并公开了状态流和到异步服务的连接。

让我们快速回顾一下这里发生了什么，以及通信是如何通过表示模块和应用程序核心进行的。

1 - 每个表示模块通过自己的沙箱订阅由应用程序核心发布的事件  
2 - UI模块调用一个沙箱方法，该方法在异步服务中触发相应的进程

3 - 异步服务将消息转换成合适的格式并将请求发送到外部世界(例如服务器请求)  
4 - 来自服务器的异步响应由应用程序核心转换为javascript对象，并进一步以数据流的形式转发到沙箱。  
5 - 每个订阅该事件的表示模块都通过沙箱得到通知

**目录结构(Folder structure)**

现在我们对总体设计有了一个更清晰的了解，我们可以看到如何在代码中组织所有这些。我们可以从创建一个文件夹结构开始。它将帮助我们可视化问题，并使开始开发每个模块变得更容易。在实践中，表示层和核心层之间没有明确的界限，由于实际原因，我们经常需要将它们混合在一起。我们将把代码分成两组:

1 - Presentational features(表示特性)——逻辑单元，代表圆形的、独立的和可重用的代码片段(例如认证、仪表板、用户管理……)。我们的目标是采用，例如，身份验证模块，并简单地将其插入我们开始工作的下一个项目中。

2 - Shared features(共享功能)——通过整个应用程序使用的模块(例如异步服务、数据管理、实用服务、配置……)。这些部分代表了我们应用程序的主干。它们中的大多数将在接下来的项目中保持不变，而其他的将以最小的努力进行调整。

正如你所看到的，每个演示模块都有一个单独的文件夹，还有一个共享文件夹保存所有的应用核心逻辑。这里我们需要注意两件事:

1 - 将特性组织在合理命名的文件夹中

2 - 尽量保持结构平坦

对于那些想了解更多关于约定和应用结构的人，Angular有它非常详细的风格指南。这通常很难实现，因为在复杂的应用程序中，很难同时拥有分组和平面文件。你需要找到最适合你的方法。让我们分析图中的每个块，看看在代码中的什么地方放置它。

**Application core module**

我们也可以叫它根模块它位于app/app.module.ts文件。它描述了应用程序各部分如何组合在一起，它也是用于启动应用程序的入口点。根模块的主要任务是:

1 Imports 导入我们想要插入应用程序的所有其他模块  
2 Provides提供我们想要在应用程序内部全局公开的服务，并且只实例化一次  
3 Declares 声明应用程序的根组件  
4 Bootstraps引导Angular创建的根组件，并将其插入宿主web页面index.html中

**Application core facade**

应用程序核心facade表示为沙箱。它是一个抽象类，包含应用程序核心API的公共逻辑。我们把它放在app/shared/sandbox/base.sandbox.ts文件。

每个表示模块的沙箱将扩展作为接口的基沙箱类和它们将继承的基类。在这里，我们可以定义每个沙箱实例需要拥有哪些方法和属性。它将代表一个契约，以及可以被覆盖的方法实现。

**Sandbox**

Sandbox是一个服务，它扩展了应用程序核心外观，并公开了状态流和到异步服务的连接。它充当每个表示模块的中介和外观，带有一些额外的逻辑，如从存储中提供所需的状态块、向UI组件提供必要的异步服务、调度事件……

正如我们所说的，表示逻辑和核心逻辑之间没有明确的界限，所以定义每个沙箱的位置是很棘手的。我们可以把它放在app/shared/sandbox文件夹中，按特性分组，或者把它放在相应的表示模块文件夹中。我们将使用第二个选项，因为沙箱逻辑显式地与我们为其构建的表示模块相关。这样我们就可以把所有相关的逻辑放在一个地方。

下面的示例演示了用于身份验证模块的沙箱。它处理登录和密码恢复操作。沙箱从状态管理层导入操作(因为它知道app core包含它)，并定义公共变量，这些变量代表一个可观察的状态。例如，每当用户按下登录按钮并等待服务器响应时，loginLoading$ observable都会发出一个事件。我们的组件可以订阅事件并切换spinner来指示用户发生了一些事情。所有这些神奇的事情都发生在状态管理层，通过操作和简化器，表示组件不需要了解它。Sandbox将选择所需的状态块，它将根据分派的操作进行更改，组件将使用它。在这个组织中，我们的应用程序是由事件驱动的，我们可以说它是反应性的，因为它对可观察到的事件作出反应。

*注意:在订阅许多事件时要小心，因为我们需要取消订阅它们，以避免用户再次访问同一页面时进行多次订阅。这可能导致内存泄漏。*

Angular提供了非常方便的 [async pipe](https://angular.io/docs/ts/latest/api/common/index/AsyncPipe-pipe.html" \t "https://netmedia.io/dev/_blank)(异步管道)，可以在模板中使用，它会自动完成这项工作。

**State management**

我们不会对状态管理层的每个部分进行太深入的讨论，因为这不是本文的主题。如果您是新手，请参阅ngrx存储文档。尽管如此，在这一章我们将集中关注两件事:

1 如何组织store  
2 使用ngrx/effects处理异步操作

我们将把存储看作一个数据库，其中每个reducer都是一个表，它表示我们想要跟踪的状态片。存储类似于关系数据库，我们可以在其中使用高级选择器合并状态的不同部分。假设我们的应用程序中有一个带有产品列表和过滤条的页面，可以根据名称搜索产品并根据类别过滤它们。在一边我们有一个产品列表(数组)的状态，而在另一边我们有一个对象，它持有一个选定过滤器的值。为了根据选择的类别和名称筛选产品，我们将合并两种状态流(产品和筛选器)并显示结果。与实际数据库中的逻辑几乎相同，只是使用了联接选择器。

我们把store放在app/shared/store/index.ts文件。它将拥有一个接口，用于描述存储的每个部分，并表示来自每个 reducer的State(状态)。这个接口只是到内部状态类型的键的映射。除了总体状态之外，存储还包含选择器函数来获取状态的每个小块，而子缩减器不知道总体状态树。

以下是这store的简图:

export interface State {

products: fromProducts.State;

login: fromAuth.State;

}

const reducers = {

products: fromProducts.reducer,

login: fromAuth.reducer

};

export function store(state: any, action: any) {

const store: ActionReducer<State> = compose(combineReducers)(reducers);

return store(state, action);

}

export const getLoginState = (state: State) => state.login;

export const getLoginLoaded = createSelector(getLoginState, fromAuth.getLoaded);

export const getLoginLoading = createSelector(getLoginState, fromAuth.getLoading);

export const getLoginFailed = createSelector(getLoginState, fromAuth.getFailed);

export const getLoggedUser =

createSelector(getLoginState, fromAuth.getLoggedUser);

另一方面，我们使用ngrx/effects。他们是什么?效应与“副作用”一词有关。它是在调用ngrx操作之后需要执行的一段代码。它基本上是一个返回可观察对象的函数。

假设我们需要执行一个异步调用，并用给定响应更改一个状态。我们需要在某个地方触发异步调用，分派一个动作来指示加载，等待异步响应分派另一个动作来存储数据，并指示成功或错误的响应。所有这些行动都将在我们的沙箱中结束。

这就是effects产生的地方。effects用于处理我们的动作的异步调用，并在异步调用结束时链接其他动作。这样，我们就不需要为同步操作和异步调用而烦恼了。要操作应用程序状态，我们应该只处理操作。通过这种方式，我们让一个不同步的动作看起来更同步，这很自然，因为这是我们大脑工作的方式。我们将从组件发送一个动作来与app core通信，http请求、web socket请求等将在后台触发。效果实现示例如下:

@Effect()

doLogin$: Observable<Action> = this.actions$

.ofType(actions.ActionTypes.DO\_LOGIN)

.map((action: actions.DoLoginAction) => action.payload)

.switchMap(state => {

return this.authApiClient.login(state)

.map(user => new actions.DoLoginSuccessAction(new User(user)))

.catch(error => of(new actions.DoLoginFailAction()));

});

这并不意味着我们不会直接处理异步服务。我们可以调用asnc服务的任何时候，我们需要得到一些数据，而这些数据没有进入状态。

**Async services**

异步服务是负责不同类型通信的模块的集合。他们的职责是准备相应格式的数据，建立与相关通信协议的通信，并将响应转换为应用程序友好的格式。由于http服务是最常见的服务，我们来解释如何实现它。

除了上面提到的，http层的目标是添加头部、管理请求方法、拦截请求、接收响应、解析它们和处理各种类型的错误，而无需在应用程序中重新编写。

还有一个要求。当使用http层时，拥有类似rest的接口将是非常好的。这是非常有用的，因为我们通常习惯于在服务器上rest api服务，另一个原因是它们非常自我描述。我们的目标是拥有一个非常小的http客户端，它具有类似rest的方法，我们可以从沙箱中调用这些方法。最终的解决方案应该如下所示，它的灵感来自于Angular2 -rest客户机(使用RESTful服务的Angular2 HTTP客户机)。

@Injectable()

@DefaultHeaders({

'Accept': 'application/json',

'Content-Type': 'application/json'

})

export class ProductsApiClient extends HttpService {

@PUT("/products/{id}")

public updateProductById(

@Path("id") id: number, @Body product: Product): Observable<any>

{ return null; };

}

通过两行代码，我们可以编写一个http方法，同时读取我们需要知道的关于该方法的所有信息。我们可以像这样从代码中调用这个方法:

updateProductById(2， {title: "Book"， stock: 5});

让我们回顾一下我们在这里所做的。

@DefaultHeaders装饰器为类中的所有方法设置默认的标题。@PUT decorator将请求方法设置为带有目标api端点“/products/{id}”的PUT类型。@Path装饰器设置url中的给定id参数。@Body装饰器指定要发送到服务器的数据，类型为Product。方法返回一个可观察的，因此结果也可以在调用方方法中处理。正如您所看到的，我们仅用两行代码就完成了很多工作。还有很多很酷的东西我们可以添加到这个方法中。

让我们看另一个例子。我们希望发送表单数据类型到服务器，而不是json。我们还希望覆盖默认的标头并应用自定义适配器来响应，以便将数据转换为适合我们自定义方法的格式。以下是我们如何通过rest服务实现的。

const formDataMediaType = {

'Accept': 'application/json, text/plain, \*/\*',

'content-Type': 'application/x-www-form-urlencoded;charset=UTF-8'

};

@POST("/products")

@Produces(MediaType.FORM\_DATA)

@Headers(formDataMediaType)

@Adapter(CustomService.customAdapter)

我们的http服务负责所有这些很酷的语法。场景背后的逻辑我们不能在这里完全展示，但通常我们有一个函数，它读取所有这些装饰器，并基于它构建一个发送到服务器的请求。它还创建钩子来拦截和修改请求数据或头，捕获错误并操作响应数据。

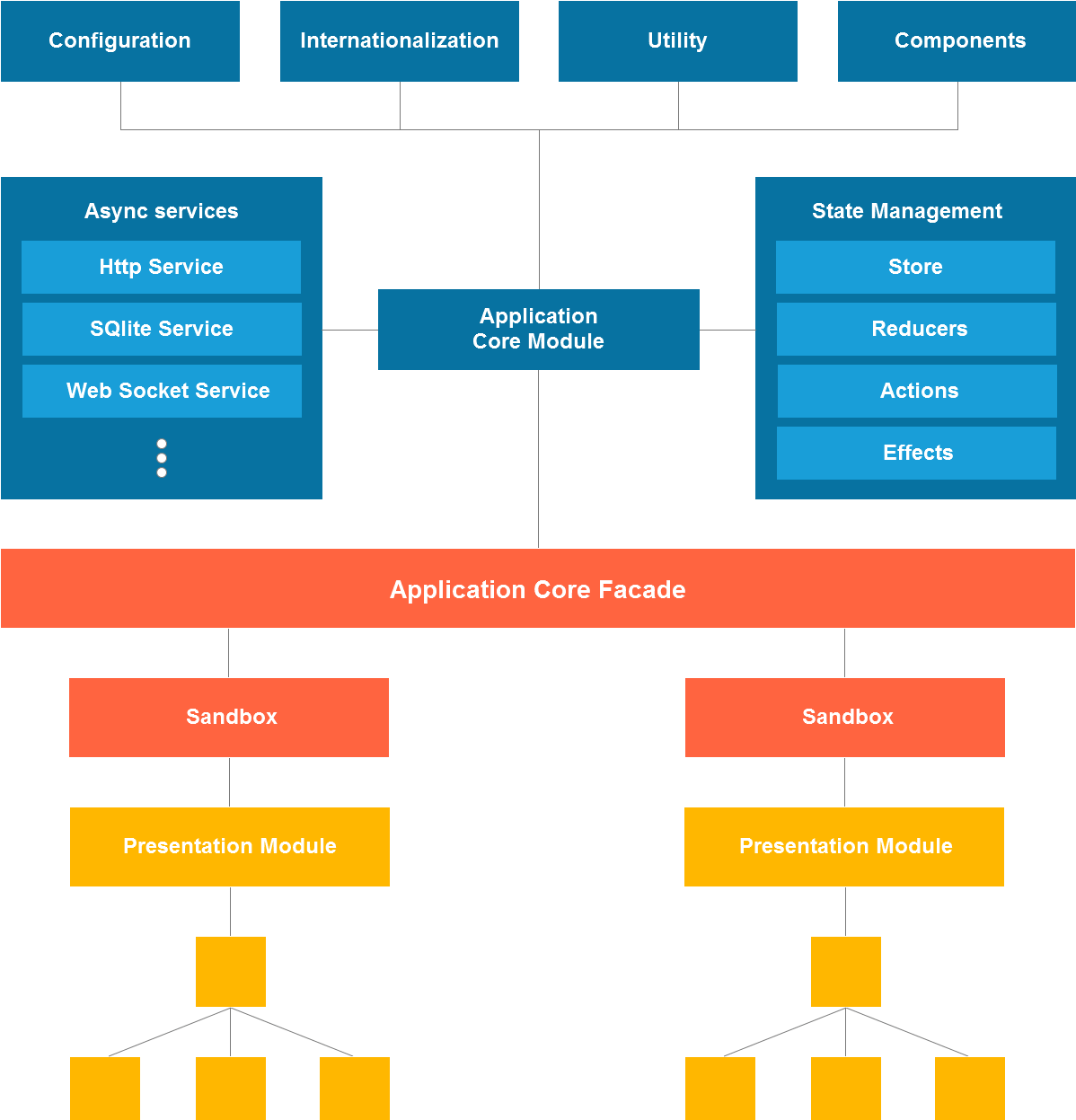
我们需要提到的最后一件事是适配器。适配器是用于将从服务器接收的数据转换为格式友好的，以便由UI组件显示的功能。我们有基本的http适配器，它可以将http json响应对象解析为javascript对象文本。它还检查定制适配器是否已被@Adapter装饰器应用，并调用适配器函数。如果自定义适配器函数与表示层相关，则可以位于表示模块中。

**结论**  
让我们重述一下这一章的内容:  
 1 定义更详细的项目架构，并决定将代码的哪一部分放入每个应用层  
 2 创建一个基本的文件夹结构  
 3 解释了应用程序核心facade和沙箱是如何工作的  
 4 解释了状态管理和效果逻辑  
 5 演示了异步服务实现

现在我们已经更好地理解了如何根据第一章中的理论来解耦一个应用程序。在实践中实现这些想法总是比较困难的，重要的是实际的实现与项目的框图并不完全相同。有时很难决定将某段代码放在何处，并在各层之间划清界限。

在下一章，也就是最后一章，我们将讨论这个项目应该包含的其他特性，我们将提供一个包含所有特性的完整的项目示例。

应用程序架构将有以下结构:



**Configuration**

配置几乎是每个应用程序都需要的。我们需要一个地方来存储广泛使用的常量(字符串、数字、对象文字……)，并且能够为开发、生产和测试环境提供不同的配置设置。配置中的内容包括api基础url、各种文件夹的路径、全局通知设置等。

为了在项目中设置配置，我们将有三个json文件和一个Angular服务，它将读取这些文件并根据当前的环境变量加载配置设置。第一个json文件名为env。它只包含一个属性env，它保存当前设置的环境。这个变量将被设置为一个预定义的值(开发、生产),因为我们将在整个项目中使用它,在CLI构建应用程序,所以我们将做一个脚本,将作为一个钩子,使用node . js内建函数,将环境变量设置为所需的值。

{ "env": "development" }

myApp/config/env.json

function setEnvironment(configPath, environment) {

fs.writeJson(configPath, {env: environment},

function (res) {

console.log('Environment variable set to ' + environment)

}

);

}

// Set environment variable to "production"

setEnvironment('./src/config/env.json', 'production');

myApp/hooks/pre-build.js

setEnvironment函数读取evn.json文件，修改env变量并保存它。在我们的package.json 。我们将定义自定义脚本，它将在开始和构建应用程序之前调用setEnvironment函数。

"scripts": {

"start": "npm run pre-start & ng serve",

"pre-start": "node hooks/pre-start.js",

"pre-build": "node hooks/pre-build.js",

"my-custom-build": "npm run pre-build & ng build --prod --aot"

}

package.json

因此，env变量(在我们的示例中)只能设置为开发或生产(通常可以是任意值，但我们将坚持使用这些值，因为它们是最具描述性的)。我们将根据这个值加载相应的配置文件—development.json或production.json。两者都包含配置内容。

{

"api": {

"baseUrl": "/api"

},

"debugging": true

}Copy

myApp/config/development.json

为了将json文件中的值加载到我们的应用程序中，我们需要一个读取env的服务。查看使用的环境，然后加载配置文件并将其转换为javascript对象文字，以便在我们的代码中读取。

@Injectable()

export class ConfigService {

private config: Object

private env: Object

constructor(private http: Http) {}

/\*\*

\* 首先加载环境配置文件。读取环境\*变量从文件中，并基于此加载适当的配置文件-开发或生产

\*/

load() {

return new Promise((resolve, reject) => {

this.http.get('/config/env.json')

.map(res => res.json())

.subscribe((env\_data) => {

this.env = env\_data;

this.http.get('/config/' + env\_data.env + '.json')

.map(res => res.json())

.catch((error: any) => {

return Observable.throw(error.json().error || 'Server error');

})

.subscribe((data) => {

this.config = data;

resolve(true);

});

});

});

}

/\*\*

\* Returns environment variable based on given key

\*

\* @param key

\*/

getEnv(key: any) {

return this.env[key];

}

/\*\*

\* Returns configuration value based on given key

\*

\* @param key

\*/

get(key: any) {

return this.config[key];

}

}Copy

myApp/src/app/app-config.service.ts

现在让我们看看如何实际使用该服务:

let baseUrl: string = this.configService.get(api).baseUrl;

我们可以尽可能多地嵌套属性，并以非常优雅的方式将配置拉到应用程序的任何部分。

还有一件事我们需要设置。我们想要在任何组件初始化之前加载配置文件，包括我们的主app.component.ts文件。这一点很重要，因为一旦应用程序启动，配置就需要可用并可以使用。

在我们app.module。我们将定义一种特殊类型的提供者—APP\_INITIALIZER来在应用程序启动前执行config.load()方法，这就是我们在myApp/src/app/app-config.service.ts中定义的load()方法。

mport { APP\_INITIALIZER } from '@angular/core';

export function configServiceFactory (config: ConfigService) {

return () => config.load()

}

...

@NgModule({

...

providers: [

...

ConfigService,

{

provide: APP\_INITIALIZER,

useFactory: configServiceFactory,

deps: [ConfigService],

multi: true

}

],

...

});Copy

myApp/src/app/app-module.ts

configServiceFactory函数用作导出函数，因为在使用AoT时，元数据(提供者定义)不支持调用函数或调用new。这种限制的原因是AoT编译器需要生成调用工厂的代码，而且无法从模块中导入lambda，只能导入导出的符号。之所以使用“multi: true”，是因为应用程序可以有更多APP\_INITIALIZER提供程序。

这样，配置设置就完成了，可以使用了。

**国际化**  
国际化不仅对于多语言的目的很重要，而且对于在一个地方定义标签值也很重要。这意味着我们可以在一个地方定义转换后的值，然后在项目的多个地方使用它们。它使得整个模板的维护和更改更加容易。

为此，我们使用了ng2-translate npm模块。基本的安装非常简单。我们只需要在我们的主应用模块中导入TranslateModule。

import { TranslateModule } from 'ng2-translate';

@NgModule({

imports: [

TranslateModule.forRoot()

],

});

有关安装的更多选项可以参考模块文档。默认情况下，模块在myApp/i18n文件夹中查找json文件。我们需要为每一种我们想要支持的语言提供一个json文件，例如en。json,人力资源。json等。一旦我们创建了这些文件，我们就可以使用TranslatePipe和TranslateService来翻译应用中的值。TranslatePipe是一个Angular管道，可以用来翻译静态值，比如模板中的标签。

<label>{{ 'HeaderTitle' | translate }}</label>

他可以使用TranslateService来翻译JavaScript值(也包括可观察值)，并更改应用程序中使用的当前语言。

translateService.get('HeaderTitle').subscribe(

value => {

let alertTitle = value;

}

)

let title = translateService.instant('HeaderTitle');

在开始使用翻译后的值之前，我们需要设置一个默认语言。这是应用程序根组件的任务，因为它用于引导应用程序。实际上根组件会把这个任务委托给它的沙箱，因为沙箱知道配置和其他智能应用程序的核心内容。可用语言是我们可以存储在配置中的另一个东西。下面的代码展示了如何设置默认语言并在存储中存储可用语言。

private setupLanguage(): void {

// Load localization object from the confguration

let localization: any = this.configService.get('localization');

// Save language codes as an array

let languages: Array<string> = localization.languages.map(lang => lang.code);

// Get browser’s default language

let browserLang: string = this.translate.getBrowserLang();

// Tell the translate service for the available languages and set a

// default one

this.translate.addLangs(languages);

this.translate.setDefaultLang(localization.defaultLanguage);

// The default language will be a default language from the configuration

// or selected browser language if it matches one of our predefined values

let browserMatch = browserLang.match(/en|hr/);

let selectedLang = browserMatch ? browserLang : localization.defaultLanguage;

let selectedCulture = localization.languages.filter(lang => {

return lang.code === selectedLang;

})[0].culture;

// Tell the translate service to use selected language

this.translate.use(selectedLang);

// Save selected language and culture in the store

this.appState$.dispatch(new settingsActions.SetLanguageAction(selectedLang));

this.appState$.dispatch(new settingsActions.SetCultureAction(selectedCulture));

}

在本例中，我们使用浏览器的默认语言作为应用程序的默认语言。如果我们有这样的需求，我们也可以在配置文件中硬编码它。

最后要考虑的是，我们可能需要将json文件分割成更小的部分，因为随着应用程序的不断增长，我们的翻译文件将会越来越大。因此，我们将按照特性对它们进行分离，就像分离表示模块一样。与此同时，我们可以为我们在整个应用程序中经常使用的东西提供通用的json文件，例如服务器响应消息、通知标题……

**实用程序**  
实用模块部分解决了干燥(不要重复自己)的问题。它用于保存所有常用的逻辑和辅助函数。我们可以把它分为两个部分，使用其他应用程序依赖的函数(例如，一个服务需要翻译服务作为依赖)和独立的函数(例如，大写单词)。

第一个是通过依赖注入包含其他模块的服务。例如，我们可以定义一个通用的函数来显示不同类型的toast通知(成功、错误、信息…)为此，我们需要通知和翻译模块的一个实例

public displayNotification(msgCode: string, type: string = 'info', titleCode?: string) {

let message: string = this.translateService.instant(msgCode);

let notificationOpts: any = this.configService.get('notifications').options;

let title: string = titleCode ? this.translateService.instant(titleCode) : null;

switch (type) {

case "error":

title = this.translateService.instant('ErrorTitle');

break;

case "success":

title = this.translateService.instant('SuccessTitle');

Break;

default:

title = this.translateService.instant('InfoNotificationTitle');

break;

}

this.notificationService[type](title, message, notificationOpts);

}

另一种不依赖于其他模块的实用函数是用于一般低级JavaScript任务的函数，如将句子中的第一个字母大写、扁平化对象字面量、日期到字符串的转换(和反向)等。它们是独立的，使用JavaScript浏览器API。

export function capitalize(s: string) {

if (!s || typeof s !== 'string') return s;

return s && s[0].toUpperCase() + s.slice(1);

}

**共享组件**共享组件是存储在应用程序核心层中的表示元素。这听起来奇怪吗?好的，应用程序核心可以保存在项目中反复使用的表示元素，并将它们提供给表示模块，以便作为模板中的片段包括进来。每个组件都属于声明和导出它的模块。此模块将包括组件正常工作所需的其他依赖模块。假设我们需要用TranslatePipe翻译一些文本，我们必须包含翻译模块。

我们这里也可以有智能组件。现在，把它们放在哪里是由架构决定的。我们可以为容器创建单独的模块，这样就可以很容易地区分它们之间的区别。在代码中也更容易找到它们。另一种方法是将它们放在表示层，但在这种情况下，我们将打破按特性组织表示模块的规则。如果您还记得本系列第二篇文章中的文件夹结构，可用的选项是myApp/src/app/shared/components或myApp/src/app/components文件夹。

假设我们正在构建一个包含仪表盘和登录页面的商业应用程序。登录页面将有一些简单的背景登录表单。一旦用户登录，他/她将被重定向到仪表板。仪表盘页面将包含一个侧边栏导航菜单，标题与用户的形象，链接注销和最新添加的产品列表。要求是每隔一个页面都包含相同的布局，包括侧边栏和标题。

我们已经可以看到，我们需要一个地方来处理所有这些常用的逻辑。我们可以用几种方法来处理这个问题，其中一种是创建一个智能布局组件，它将把它的子事件(从导航栏和标题)委托给它自己的沙箱。另一种方法是让根应用程序组件完成这项工作。我们将使用创建一个单独的容器模块的方法，它将在应用程序中保存所有可重用的容器。这样，每个表示根组件(登录组件除外)都将被包装到布局容器中，并自动继承侧边栏和头部。这样登录组件就可以在不使用公共布局的情况下单独设置样式。

**性能优化**  
在我们完成了代码组织和项目架构之后，我们可以考虑提高性能以加速应用程序的初始启动。当项目还很小的时候，这在一开始并不是很重要，但是当它变大时，性能优化就变得至关重要了。

*Service Workers*

我们可以做的第一件事是介绍服务工作者。服务工作者是我们的应用程序和web服务器之间的中间件。它是浏览器在后台运行的JavaScript程序，与我们的web应用程序在一个单独的进程中运行。我们可以使用它来控制网络请求和处理浏览器缓存。换句话说，我们可以告诉浏览器在第一次下载应用程序资产后缓存它们。这样，我们就可以从缓存中提取资产，而不发出网络请求。有很多选项可以控制服务工作人员和处理后台同步。我们可以指定谁拥有优先级、网络或缓存，并控制如果没有网络连接会发生什么。我们最感兴趣的是并行地从缓存和网络请求资源，然后用最先返回的那个来响应。通常这将是缓存版本(如果有的话)。一方面，该策略将始终发出网络请求，即使资源已被缓存。另一方面，如果/当网络请求完成时，缓存会被更新，以便将来的缓存读取会更及时。

选择哪个服务工作者策略的决定很大程度上取决于我们正在构建的应用程序的类型，并且不同的情况下会有所不同。如果我们考虑我们有一个商业应用，这个解决方案，我们请求资源从缓存和网络满足我们的需求最好。

关于实现，实际上不需要做太多工作，这要感谢谷歌的sw-precache插件，它负责为我们生成服务工作者，并配置好一切。首先，我们需要安装插件并在根项目文件夹中创建sw-precache-config.js文件。

module.exports = {

navigateFallback: '/index.html',

stripPrefix: './dist',

root: '../root/',

staticFileGlobs: [

'./dist/index.html',

'./dist/\*\*.js',

'./dist/\*\*.css',

'./dist/\*\*.ttf',

'./dist/assets/images/\*',

'./dist/config/\*',

'./dist/i18n/en.json',

'./dist/i18n/hr.json'

],

runtimeCaching: [{

urlPattern: /\/api\/lookup/,

handler: 'fastest'

}]

};

我们定义了要缓存的所有静态文件。我们还可以通过指定urlPattern来缓存http请求，并通过指定handler属性来定义要使用的缓存机制类型。通过运行:sw-precache -root =。console中的/dist -config =sw-precache-config.js命令将为我们生成服务worker，我们需要将其包含在index.html文件中，以便在浏览器中注册服务worker。

<script>

if ('serviceWorker' in navigator) {

navigator.serviceWorker

.register('/service-worker.js')

.then(function(registration) {

console.log('Service Worker registered');

}).catch(function(err) {

console.log('Service Worker registration failed: ', err);

});

}

</script>

我们可以通过在package中添加上面的命令来自动化构建服务工作者的过程。在构建用于生产的应用程序之前使用json。对于service worker的实现来说，差不多就是这样了。想了解更多信息，你可以参考科里·瑞恩的博客文章，在那里你可以了解更多关于服务人员的细节。

关于性能改进，我们可以做的另一件事是延迟加载一些表示特性模块。

*Lazy loading*

延迟加载是允许按需加载Angular模块的一个特性。这在包含许多表示模块的大型应用程序中非常有用。再想象一下，我们有一个具有登录屏幕和仪表板的应用程序。我们可以在初始JavaScript包中包含这两个模块。一旦用户登录，他/她将被自动重定向到仪表盘。如果我们添加更多的表示模块，例如一个带有产品的CRUD，我们可以在用户从仪表板导航到产品页面后加载产品模块，甚至可以在用户仍在仪表板页面时通过后台线程通过service worker加载。这样，我们将把bundle文件分解成多个文件，加快初始应用程序的引导时间，并按需加载额外的模块。更多关于延迟加载的信息可以在Angular文档中找到。

**结论**在最后一章中，我们用几个额外的核心特性丰富了应用程序，使我们能够构建更加健壮和可伸缩的应用程序。我们还添加了性能优化，以加快启动时的引导速度。当启动应用程序时，这些小的增强会对用户产生很大的影响。

我们来回顾一下我们在这一系列文章中所做的事情:

1 项目被划分为三个主要层(核心层、立面层和展示层)，它们有各自独立的关注点  
2 状态管理和异步服务是应用程序核心的主要构建块，它们彼此不了解  
3 Facade还充当中介，处理表示层(用户交互)和应用程序核心之间的通信。

4 状态管理使用效果来处理异步操作  
5 异步服务可以使用多种协议进行通信，并且应该公开非常简单的API，这样它们就可以更容易地在沙箱中使用  
6 配置文件用于为不同类型的环境存储常量  
7 国际化对于支持多种语言和将标签名称保存在一个本地位置非常重要

8 实用程序提供了一个可以在多个项目中共享的库  
9 服务工作人员和延迟加载是优化应用程序性能的基本特性

们希望您能够很好地理解如何从零开始创建可伸缩的应用程序体系结构。我们已经为添加新特性打下了良好的基础，如果我们的应用程序开始成长，我们也不应该感到害怕。现在，当我们完成了基本的开发之后，就可以开始额外的开发了。