Night Piece

Android构架系列之二--MVP&&Clean理解与实践之Clean

2016-05-08 | 技术

一万个人心中有十万个Clean。对Clean的理解每个人都不相同，网上也有很多很多的实现，我试图从最原始的资料出发，结合自己的理解讲解一下Clean构架。
首先，要明确的一点是Clean是一种**分层架构**，是一种软件系统架构模式/思想。

> 其他架构有：事件驱动架构、微内核架构、微服务架构、基于空间的架构。—《软件架构模式》

这与MVP不同，MVP关注UI层（对应Clean中Presentation）的设计，更偏向与代码的设计，而Clean可以理解成一种自上而下的，全面的构架（会考虑性能等因素），更加抽象。

在开始之前，我想强调的一点是，**Clean构架不适用于所有移动应用，它适用的范围是那些Model非常复杂的应用，但是移动应用的设计趋势是薄的Model，在这种情况下Clean略显臃肿。单独使用MVP可能更适应于快速开发的要求。**可以说Clean的复杂性很大一部分是来自于把Model独立抽出的需求，在移动App上，这很可能没有必要。

原始资料如下：

- 原始文章：Bob大叔关于Clean的文章，适用于很多场景，不限于Android。比较抽象，难以理解，不建议首先学习！
- Clean构架详细的解析文章，原文，中文翻译：Android平台上Clean的一种实现，Github上Star很多的项目，可以直接学习这一篇文章，GitHub有详细的代码和讲解，注意作者迭代了两个版本,第一版是没有依赖注入，RxJava那些框架的，相对比较好理解。
- Clean构架的GitHub Issue：这个**重点推荐**，里面很多问题很有价值。

其他参考文章：

- 国人讲解的Clean框架：如果上面的有困难，建议优先学习这篇文章，构架在在一个工程Module中，代码干净，比较好理解，但是可能对Clean的理解并不透彻。
- Android MVP 详解（下）：也是Clean的总结，加入了自己的理解，可以参考。

> 架构是一种高层思想，我这里只是从Demo中学习，其实Clean有无数种实现方式，比如，依赖的解耦可以用接口，也可以用命令等等，这些其实是细节！

理解Clean

这张图可是说是理解Clean的核心，第一次学习Clean就看到了这张图，留下了满满的疑惑，甚至被误解了。相关误解看这个Issue

> The one you are citing shows how data flow through the layers, not dependencies or relation of the layers between themselves

最重要的： **洋葱图是依赖图，右下角的图片是控制流图**，不要混淆。

依赖原则与控制流

依赖原则和控制流是Clean构架的核心，只有理解了依赖原则和控制流，才能理解Clean为什么这么设计。

- 向内依赖：洋葱图中，内环里的所有项不能了解外环所发生的东西(内环使用接口操作外环，外环可以直接调用内环）。注意：最内层的Entity是Domain层的业务实体。

  > - source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in the an inner circle. That includes, functions, classes. variables, or any other named software entity.
  > - The outer circles are mechanisms. The inner circles are policies.**内环是策略，外环是机制。**

- 数据结构也要遵循依赖原则，因此要经常转变数据结构。

- 控制流图中，注意 `User Case Interactor` 的箭头！它就是最内层的，但是它是控制流的Master。可以这样理解，**Controller（如用户操作的UI事件）启动了Interator之后，Interactor主管所有事务，包括调用Data层，Interactor内部处理数据，最后塞给Presentation层。**

> 存疑：
>
> - 区分Enterprise Business Rules 与 Application Bussiness Rules
> - 这里的控制流，是单一的吗？即一个Interactor执行一个动作就一个回调。还是Interactor可以不断给Presentation塞数据？比如有一个需求，5S刷新一次界面。

依赖原则与接口设计

为了满足依赖原则，内核不知道外环的具体实现，因此，必须使用接口设计（未必，可以用其他方式！）：

- 内环Domain是Master的角色，它操控其他层的接口（Data层数据输入的接口和Presentation层数据输出的接口），来实现业务。
- 为了满足内环的独立性，**接口要设计在Domain那一层。**这种独立性可能是分module的要求，下面会单独讨论是否需要分module设计。

Clean与性能

Clean构架中把Model完全分离到了Domain与Data层中，这有一个巨大的好处，可以极大可能地避免应用的性能问题。
我们知道，Android开发中的一大问题是ANR，在主线程进行太多的业务操作，导致应用卡顿。使用Clean时，在设计框架的过程中，**我们在框架中可以强制Domain层在异步线程中运行。**这个设计思想可以**与 `AsyncTask` 类比**，如同 `doInBackground()` 方法放入了Domain与Data中， `pre` 和 `post` 则做了Presentation的工作。
由此，各层设计如下：

layer	thread	interface

layer	thread	interface
Presentation	Main Thread	
Domain	Work Thread	**异步回调**
Data	与Domain同线程	**同步调用**

可以发现Data层使用同步接口，这样的优势是Domain使用Data获取数据的逻辑是否简单，减少了线程同步的操作。

Clean与MVP

Clean是一种**分层构架**，软件架构模式/思想！MVX更像设计模式。构架是一种静态的形式，Clean构架告诉你需要把业务逻辑从M中单独拿出来，分成BLL和DAL。所以Clean是更高的抽象，MVX更详细的设计，还会告诉你数据如何流动。这样的表述更合理：**Clean构架是在MVP上使用了三层构架！**

代码设计

Presentation属于视图层，一般使用MVP，Domain和Data属于Model层，在移动App中应当很薄（很多Demo就是透传了数据。。）这里粗浅的说一下，以后有专门的文章讲Model层设计。
具体代码看下一篇– Android构架系列之二–MVP&&Clean理解与实践之实例分析

Presentation层 – MVP

由Clean与MVP的关系，我们可以很容易地设计Presentation层，MVP的设计参考前一篇文章。
注意Presenter中的**视图逻辑**。

> 疑问：
>
> - 分页是哪里的逻辑

Domain层 – 命令模式

Domain层属于Model的BLL，业务逻辑

> 疑问：
>
> - 粒度问题
> - Domain数据是否需要缓存？
> - one Call one Callback? one Call many Callback–>notify?

Data层 – 数据仓库模式(Repository)

Data层属于Model的DAL
一个误区：这里的Data是业务的Data，不是我们理解的文件，图片的缓存这些东西。如果他们有业务含义则可以放到Data层中，否则只能算

工具类（如Glide）

> 疑问：
>
> - Model的原始数据是否需要缓存？

单独讲讲各层之间的接口

各层之间的接口都是成对出现的，分为调用接口和回调接口

其他问题讨论

是否需要分Module？

这个问题在Issues上也有讨论，first layer,then feature?分析了各种优缺点，可以参考。原来的工程是分Module这样设计的，作者的理由是便于测试和强制的分层，但是随着Android测试工具的发展，这种理由已经变弱，因此作者也开发了相应的合并分支。
一种折中的方案是：还是分Module，但是Module内部使用功能再分包。

数据通知的问题

由于MVP构架的天然缺陷。数据通知问题需要特殊处理。
参考这个Issues,总结如下

使用EventBus

1. 什么时候可以加入EventBus机制

> So Event busses can be useful in the case of “Data layer lets the Presentation layer know a User was updated in a background service to we can show a toast”. But they are not to be used in case of Commands, eg: “User clicks on a button to update a user”. As said, events indicate that something has happened, not that something should happen.

2. 哪里发布消息，哪里接收消息

> From the architectural standpoint presented in this repository, an event bus certainly can be implemented, and most probably in the Data Layer. You could have some kind of background service that would poll your Api from time to time. When an update happens, that background service would have a references to the Event Bus (or Aggregator) and post a UserUpdatedEvent to it. Any subscribers on the Event Bus would then be notified. Another discussion is where these subscribers should exist, but that depends on the scope of your application. Usually this will be in the Presentation Layer.（理解：事件的结果需要展示的化就是Presentation层接收–绝大部分情况，不需要展示则Domain层接收）
> 另一个人say:
> In general, i can say, that i use event bus only in order to pass events from the data layer to the presentation layer. Events, but not the data

3. 发送的Event数据是什么

> Something of note is that Events indicate that something has happen, but be careful with what data you pass with the Event. Ideally the Event holds minimal immutable data to inform subscribers. In the case of a UserUpdatedEvent, the Event holds the ID of the user that was updated, but not the User Object itself. If you would do this, the subscriber would consume the Event and use that data (eg, show the new User's name in a Toast) but might skip over a lot of Data or Business logic. The subscriber should use the ID passed by the event to get the new User through a Use Case. This way, you pass by your business rules and any Data implementation that might exist. For example, a business rule states that a user with a long name should verify if this name is correct. But if the subscriber in the Presenter consumes an entire User object and just shows the Toast, the business rules that are applied when getting a User through a Use Case are not enforced. (keep in mind, i'm just making up silly rules here)

4. 区分Action与Event(do not confuse Events with Actions.)

> - events：被动接收：Background service -> cloud service -> callback -> push event -> listeners are notified。实践上，events表示已经发生的事情，比如数据已经更新了。
> - Actions:主动发出到回调：GUI -> domain -> data -> cloud service -> callback -> data -> domain -> GUI。实践上，actions表示要/正在发生的事情。用户点击按钮，触发了行为。

5. Eventbus虽然完全解耦，但是容易失控！

> they work the same way as GOTO instruction ！就像很多语言中的goto语句。导致最后难以Debug。event spaghetti

使用RxJava

待补充

总结

本文详细论述了什么是Clean构架，以及讲解了Clean构架的难点，如同MVP的理解，Clean也有各种各样的分析，我试图从一些原始资料出发分析什么是Clean，然而，建议学习时还是看原始资料能更好的理解Clean。最后我想强调的是Clean不是万能的构架，有自身的问题，这需要我们根据业务需求具体问题具体分析。
下一篇文章讲解具体代码，以Google构架Demo的clean分支来讲解，由于Clean包含了MVP部分，所以MVP的部分一并说明。

Android # 主框架

‹ Android构架系列之二--MVP&&Clean理解与实践之MVP

Android构架系列之二--MVP&&Clean理解与实践之实例分析 ›