Night Piece

Android构架系列之二--MVP&&Clean理解与实践之MVP

2016-05-02 | 技术

一万个人心中有一万个MVP，对MVP的理解千差万别，似乎也没有一个最权威的Demo来说明什么是MVP，求其是Android平台上（如果有，请告知）。至今最权威的资料可以说是这些：

- MVP的原始资料：微软在.NET中使用的技术，偏向于WEB技术，不一定完全适用于移动开发。
- android mvp ： github上star很多的关于Android中的MVP例子，入门教程。
- 一个mvp框架的实现：一个被引用很多的mvp示例，讲解了MVP的好处，强调了数据恢复的问题。
- Google的MVP的demo

关于MVP的学习路径，我推荐按照上面的顺序。微软的MVP在.Net中的应用虽然不是Java语言讲解的，但是对MVP设计思想讲解十分清晰，为什么这样设计，有哪些设计的变形，设计带来的好处与劣势都写的十分清晰。后面的几篇文章更倾向于MVP思想在Android平台上的具体应用。另外一篇国内文章Android MVP 详解（上）总结很全，可以参考。

首先，MVP不是构架模式

MVP不是整个软件的构架模式（Architectural Pattern），而仅仅是表现层（Presentation）的构架模式。叫他设计模式也不是很恰当。

- MVP is a **user interface** architectural pattern. – Wiki
- First thing to clarify is that MVP is not an architectural pattern, it’s only responsible for the presentation layer . – mvp-android
- MVP的命名中P即Presenter，往往翻译『主持者』，我觉得翻译成**『展示者』或者『表现者』**更恰当。

什么是MVP

这个问题建议通读微软关于MVP讲解的文章。学习前可以先参考阮一峰先生的这篇文章。
之前我的理解：

- M层，即 `model` ，这里的Model理解成业务层，不仅仅负责数据部分，如网络请求、数据库，获取数据，还有**业务逻辑**。由于轻客户端的设计趋势，Model应当很薄。
- V层，即 `view` ，负责数据展示，Android的View或者Fragemnt
- P层，即 `presenter` ,**是一切的主宰，是Master角色，它控制Model获取数据处理业务，再控制View的变化。**即它本身不含业务逻辑，会去**调用**业务逻辑(Model)。
- **核心：M与V不直接交互数据，必须通过P**
- 具体而言：V实现某个UIController接口，暴露自己实现的UI功能，如显示Progress等，P持有V的接口。V同时也会持有P，他们相互依赖（最好V/P都是通过接口依赖?）。

现在理解，核心角度：**哪里需要重用，哪里需要测试！**。在MVP中，业务逻辑M，展示逻辑P需要测试，而View自身往往不是测试的主要目标，View更多的目标是可以重用。既开发中很多界面具有相似性。

> 重用：一个View往往用在多个不同的模块中，比如ListView，自定义控件等。而Model与Presenter就不太会重用，开发新的模块往往是新写业务逻辑和展示逻辑，或者直接修改原来的业务逻辑达到目标。因而可测试对他们尤其重要！（使用Clean构架，业务逻辑也可以尽量重用）

为什么这样设计：

- 分离业务逻辑，代码清晰，分离的目的也是方便测试。
- **测试需要**，这是强制的，MVP的核心思想，分离出业务逻辑方便测试，却是我一直忽略的，接口的设计往往是为测试服务的：
  - 测试的目标是：业务逻辑，既Presenter与Model，由于Model很薄，所以Presenter是测试的重点。
  - 所以**View层必须是接口设计**，目的是在测试P时，可以用mockView代替真实的View，实现测试与View无关。
  - Model中推荐使用Repository模式设计Model中的**数据获取部分**，可以使用Mock数据来测试，在Clean构架尤其明显。
- **重用/复用需要**，这不是强制的，如果有需要，可以开发可复用的View，那么要求与View绑定的数据也是抽象的，如果View与Presenter绑定，Presenter也必须是抽象的。如果Presenter与Model绑定（可能是ViewModel，或者是在MVP的变种中）那么Model最好也是接口设计。可以参考此文。

  > 一个常见的需求：相同的界面，加载不同的数据

一些缺陷

在微软的文档中也提到了MVP的一下缺点，我们可以参考：

1. 最重要的一个问题，MVP是面向前端Web开发的，他们具有Model很薄的特点，即业务逻辑应当尽量都放到Server中，主要的场景都是request and response，且请求是没有状态的（即下一个请求如果需求之前的数据，需要重新请求这些数据）。所以，**在MVP的设计中通知机制被忽略了**，即model无法主动修改Presenter，或者说一个Presenter修改了Model，无法通知另外一个Presenter知晓！解决办法：使用Events机制实现通知，微软推荐的是Observer设计模式

   > **在移动开发中并不是这样的！**虽然使用场景不多但是还是需要的这种机制的。可以参考文章：界面之下：还原真实的MV*模式。参考MVC中的通知方法实现是在M与V之间使用Observer机制，但是MVP，M与V必须解耦（V不能引用M），可以在M与P之间实现Observer，或者M用Event来通知P。

2. Presenter过于繁重，它需要操控很多元素。解决方案：通过设计可以避免，记住，Presenter只操控Model获取结果数据，Clean的设计体现了这一点。

3. 在哪里创建组合Presenter与View呢？Google的Demo中推荐Activity，而Fragment作为View。其他方案中多在View中构造Presenter。

我个人觉得的一些问题：

- 待补充
- 虽然理论上应当是薄Model的，但是实际中Model还是有相当部分的代码，P与M过于耦合，Model层如何测试，应当通过接口分离出Model，类似Clean

MVP的两种变体

了解这两种遍体更能帮助我们清晰MVP中各个部分的职责与关系。**十分有利于实践中我们代码的设计！**本质上也是编程实现层面的问题。

Supervising Presenter

一些逻辑简单的UI操作操作可能不需要使用Presenter来完成，比如更新一个name，同时修改model中一个变量，之间没有复杂的展示逻辑（注意Presenter只管展示逻辑）。我们不需要Presenter参与。同时项目中也有一些的UI逻辑的操作，如显示name，同时刷新并展示用户相关信息，这就需要Presenter参与。
因此，**为了避免在某些情况下Presenter的"多余"，该模式中添加了Model与View的直接通路**，类似于DataBinding。但是，注意对应的使用情景，见下图。

Passive View

其实在MVP中View就是被动的了，Presenter是Master，这里的被动View是什么呢？因为，**MVP中的View还是被动的不够完全**。一个简单的程序逻辑如下：

> View.init()–>Presenter.loadData–>操作View的showData(参数是List)

但是在Passive View中逻辑如下：

> View的构造函数中构造Presenter并把自己传入其中–>Presenter的构造函数中调用view.init()–>Presenter.loadData–>在presenter中循环调用view.addItem(参数是一个子view)添加数据。

可以看到这里的View完全被动了，更宽泛的说，在普通的MVP中View自己还可以调用一些展示逻辑，如在OnClick中show个Toast，但是在Passive View模式中，View只要吧onclick的event告知Presenter，然后Presenter来根据事件弹出Toast。
参考文章是这篇。

MVP与MVC

这里的MVP与MVC都是指的是**客户端**的构架！

> MVP/MVC都是对于有GUI界面而言的。web服务端的GUI界面在Browser中，对服务端而言只是一个接口/模板的存在，**他们所说的MVC**基本上就是客户端的MVP（更确切的说不应当叫MVC，应当叫**Model2构架**，参考此文），因为它不符合MVC中关于**View必须直接订阅Model**的关系，而MVP/Model2中数据必须经过X中转。

一言以蔽之，MVP与MVC都是分层的方法，其中最大的不同就是数据流向。

- MVP:V -> P -> M -> P -> V
- MVC:1.V向M注册自己(注意：V不能直接修改M） 2.V -> C -> M 3.M通知V

那MVVM呢

MVVP中是讲MVP中的P换成了VM即ViewModel，特点是ViewModel与View会双向绑定，当View变换时，ViewModel会收到Notification，当ViewModel中值被修改时，View会自动显示这个变换（即ViewModel中的内容被映射到View中）

ViewModel即视图Model，职责是负责保存View的属性和状态和更新Model。

MVVM可以看做MVP的一个实现？吧V与P部分的逻辑写入了框架中，减少了程序员的工作。

Android中的MVP

为什么要用？

- 可行性，Android本质上是薄Model的，View与Model直接可以是类似于Request/Reponse之间的关系，保持无状态。因此可以使用MVP。
- 业务逻辑解耦，代码可读性需要，将业务逻辑放入Model，将视图逻辑放入P，是代码可读性的需要。
- 测试需要，十分重要的一点，通过MVP各层的接口设计，可以实现测试Mock的使用，方便视图逻辑与业务逻辑的测试。

额外的一个好处 – 后台线程问题解决

这一点值得单独拿出来说，在Android中有这么一类问题：在配置改变/Activity重启时Activity会重新构建（各种情况参考一个mvp框架的实现），如果后台线程持有Activity的引用会导致Activity泄露，最常见的代码是， `new Thread(runnable)` 中Runnable是非静态内部类，默认会隐式持有外部的Activity的引用（更不要说我们有时会显示的持有了），不知不觉中内存泄漏了。。

当使用MVP模式开发时，线程操作应当在P中（或者P使用的M中），只要我们**在Activity的 `onDestroy()` 中，清空Presenter持有的View的引用即可。**或者，**Presenter干脆持有的是View的弱引用**，可以杜绝此类问题。

要处理的一个问题 – Presenter的保持

同样，Activty的销毁与恢复，引来了另外一个问题，我有时需要：

- 配置改变时，保持Presenter并重新与View绑定。
- Activity销毁重建（如打开不保留回收）时，保持一些值到Bundle

> 这不是必须的，在大多数情况下，在Activity重建时重新构造一个新的Presenter并重新绑定View，这并没有什么问题。

我们不希望重新构建Presenter，而继续使用之前的，这样的好处是不需要重新启动线程，重头开始一些操作。一些思路如下：

- 最简单：使用xxActivity的静态变量保持Presenter，Activity重建时判空，如果Presenter非空，重新绑定Activity即可。问题是：
  - 什么时候清空Presenter？不能在onDestroy中清空，容易泄露Presenter，正确方式参考nucleus框架。
  - 一个Activity类只能有一个Presenter（如果你要启动多个相同的Activity只能共享Presenter）
- nucleus框架：本质上和第一个相同，但是解决了上面的问题，他会在一个全局的静态池中保留所有的Presenter，配置改变时，Activity会维护自己持有的Presenter的ID（通过onSaveInstance，解决了上面第二个问题）但是不清空池子中的Presenter。而Activity销毁时会清空池中Presenter，重建时会使用保存的Bundle构建新的Presenter（通过onSaveInstance，Presenter也会有自己的Bundle）。

> 注意:在改框架中是通过 `Activity##isChangingConfigurations()` 来判断是否是配置改变的情况。

- 使用Activity的Fragment setRetainInstance(true)来在**配置改变时**保持Presenter，问题：必须是Fragment的成员变量，**在Activity重建时无法保持Presenter**，需要自己实现类似bundle的功能恢复Presneter。

- 使用Android Loader机制：本质上setRetainInstance内部就是使用的此方法，使用自定义Loader保持Presenter可以应用在Activity，View任何场合。问题：与上面的类似，在Activity重建时无法保持Presenter。

综上，大部分情况下，可能Presenter并不需要保持，我们无需使用任何手段，只需确保Presenter的逻辑正确。对于有需要的场合（如含有后台线程，频繁转动屏幕引起问题），**个人建议使用Fragment保持机制最为简单可行**（这也是Google Demo的作法）。

争论：谁是P

由于Android中Activity等控件特殊性（如具有生命周期），有很多的MVP实现使用Activity、Fragment、Adapter作为Presenter。Android中其他的View（ViewGroup）作为V，这样的好处是Presenter自带生命周期，使用是否方便。
但是，个人认为这个方法有可借鉴之处，但是并不好理解，没有单独一个Presenter类更加直观，并不适合实际使用。同样我们参考Google demo中的实现，使用了另外一种方案：**建议使用单独的Presenter类，Fragment作为View，而Activity是构建和结合他们的地方**。（强调必须使用Fragment，即使只有一个页面）。这种方式是可选的，职责清晰，虽然所有的Activity必须有Fragment麻烦了一点，这个代价还是值得的。

纯MVP实践–参考Google代码

并不想贴自己的代码，毕竟没有Google的优雅，尤其是在细节地方，建议大家都参考google的官方demo的MVP分支。总结一下：

- View – Fragment、View 与 xxxContract.View接口
- Presenter – xxxPresenter与xxxContract.Presenter接口
- Model – xxxRepository，Model的内部实现以后再讲，也是接口分离各个数据来源。
- Activity – **组合器**,the overall controller that creates and connects views and presenters.
- xxxContract – **契约类**，每个业务都有，里面有View和Presenter一对接口，这种设计逻辑十分清晰，可以借鉴。

遗留问题

listview Adapter的MVP设计 – 在疑问解答中说明
Model的内部实现与缓存设计 – 单独文章

补充

- 界面之下：还原真实的MV*模式：我见过的最清晰的描述MVX文章

> - MVC中保持了依赖同一块Model的不同View显示数据的实时性和准确性
> - 总结了各个模式的优缺点

总结

至此，MVP的问题基本说明白了，参考了原始资料，加入自己的理解。但是，还是建议以原始资料学习最佳，一万个人心中有一万个MVP，每个人的理解可能均不相同。

Android # 主框架

Android构架系列之二--MVP&&Clean理解与实践之疑问 Android构架系列之二--MVP&&Clean理解与实践之Clean

Like

Issue Page

Error: Comments Not Initialized

Write Preview Login with GitHub

Leave a comment

Styling with Markdown is supported

Comment

Powered by Gitment