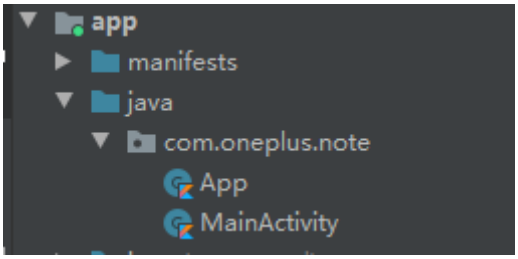
# Application Architecture

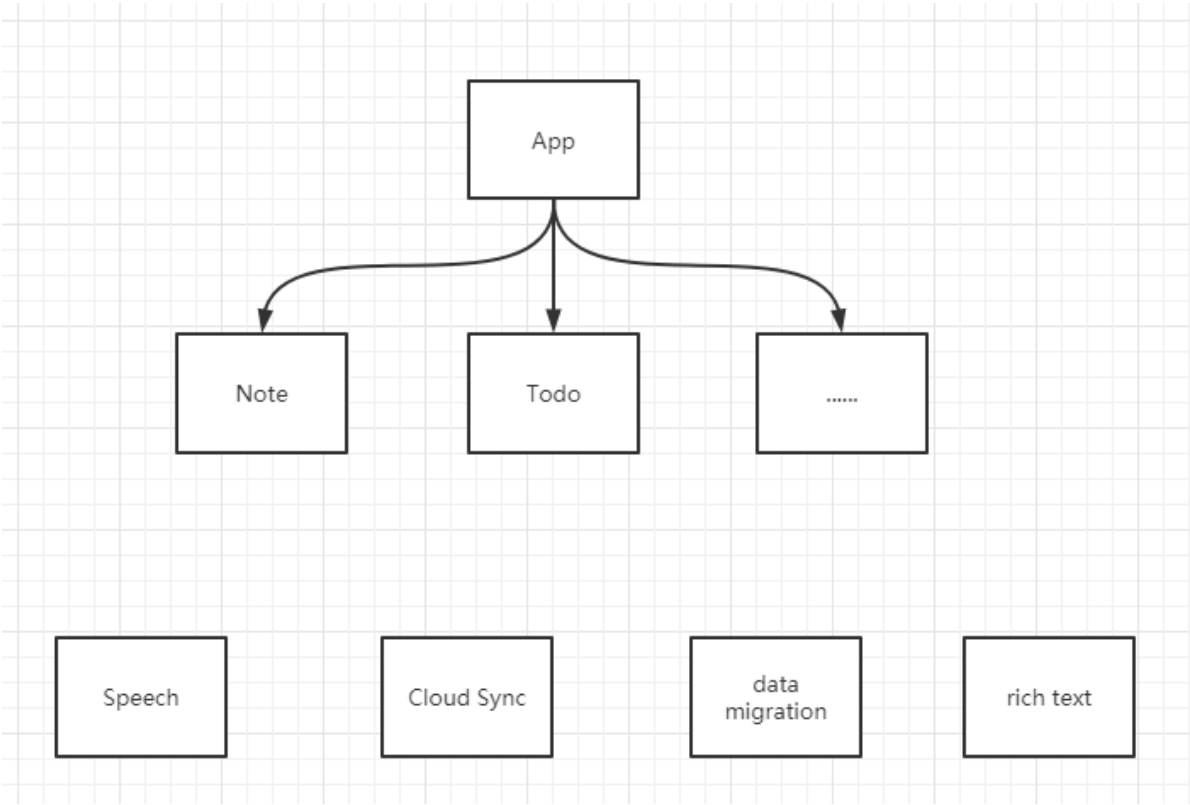
便签将Note以及Todo功能拆分开，feature module之间相互隔离

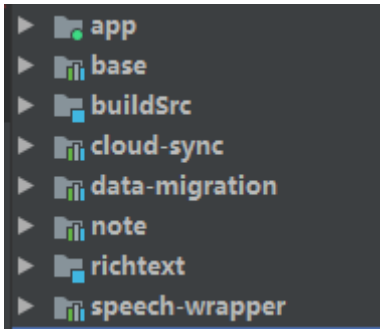



app module作为一个壳，根据Gradle配置，去集成不同的feature module

会存在几个问题

1. module间通信问题
2. note、todo动态配置问题
3. ……






Module间通信问题，基于LiveData的数据通信总线。

在Android开发中，正常且推荐的 App 页面架构应该为包含多个 Activity，这样做的好处在于系统内存告急时可以回收处于后台的 Activity，保证更多的资源给前台的页面和任务。但是我们知道，启动新 Activity 是在 ActivityManagerService 中运行，它需要执行一系列耗时的操作，我们能明显地意识到页面切换这个动作。

使用单activity，多fragment的方案消耗更少的资源，能更快地响应页面间切换和交互。但是它也有些短处，在层次深的页面进行现场保存和还原会消耗更多的资源和时间，所以适合在页面层级结构不深的应用或场合中应用。

Activity更倾向于一个整体模块容器，而Fragment是其中的子模块，Activity的存在可以对应用更好的结构化和模块化的划分，让应用有更健壮和清晰的层次，而Fragment可以让将应用的功能细化和具象化。

便签作为一个功能单一的应用，页面层级结构不深，所以是比较适合使用单activity，多fragment方案的。

但是要使用Fragment 来取代 Activity 的职能还需要做一系列的工作，我们需要把一些 Activity 中常用的功能搬运到 Fragment 中，并且我们还需要另外维护一份 Fragment 页面的栈队。进行Fragment之间切换、回退等操作。在使用不当的情况下可能会出现framgnet重叠问题，由于activity已经销毁导致使用Context crash等问题。

而Jetpack Navigation组件就可以解决这些问题。

**Navigation**

在Android中，activity 和 fragment是主要的视图控制器，界面间的跳转也是围绕着 activity / fragment 进行的。

在没有Navigation时，

```
// 跳转 activity
val intent = Intent(this, SecondActivity::class.java)
intent.putExtra("key", "value")
startActivity(intent)

// 跳转 fragment
supportFragmentManager.commit {
    addToBackStack(null)
    val args = Bundle().apply { putString("key", "value") }
    replace<HomeFragment>(R.id.container, args = args)
}
```

可能还会将KEY抽取为变量，并且在activity 和 fragment 中通过静态方法告诉调用者该界面需要什么参数

```
// NoteListActivity
companion object {
    @JvmStatic
    fun startSearchActivity(context: Context, param: String) {
        val intent = Intent(context, SearchActivity::class.java)
        intent.putExtra(Constant.DISPLAY_STYLE_KEY, param)
        context.startActivity(intent)
    }
}

// SearchFragment
companion object {
    fun newInstance(param: String) = SearchFragment().apply {
        arguments = Bundle().also {
            it.putString(Constant.DISPLAY_STYLE_KEY, param)
        }
    }
}
```

得益于 kotlin 的扩展函数，界面间跳转的代码已足够简洁。

但是

- 如果在每个界面加上跳转动画呢?
- 当项目比较复杂时，如何能快速理清界面间的跳转关系?
- 在单 activity 的项目中，如何控制几个相关的 fragment 有着相同的 ViewModel 作用域，而不是整个 activity 共享?
- 组件间的界面跳转?

Jetpack Navigation组件为应用内导航提供了强大的导航框架，它封装着应用内导航的API

- 支持fragment, activity, 或者是自定义的 destination 间的跳转
- Navigation UI 库 支持 Drawer，Toolbar 等 UI 组件
- 在Android Studio中提供可视化管理的工具

现在我们对 Navigation 有一个初步的认识，接下来我们看看 Navigation 的职能边界

**Navigation能做什么?**

- 简化界面跳转的配置

  在Navigation Graph中可视化配置，可以在resource下创建，也可以通过代码动态创建
- 管理返回栈
- 自动化 fragment transaction
- 类型安全地传递参数

  Safe Args
- 管理转场动画

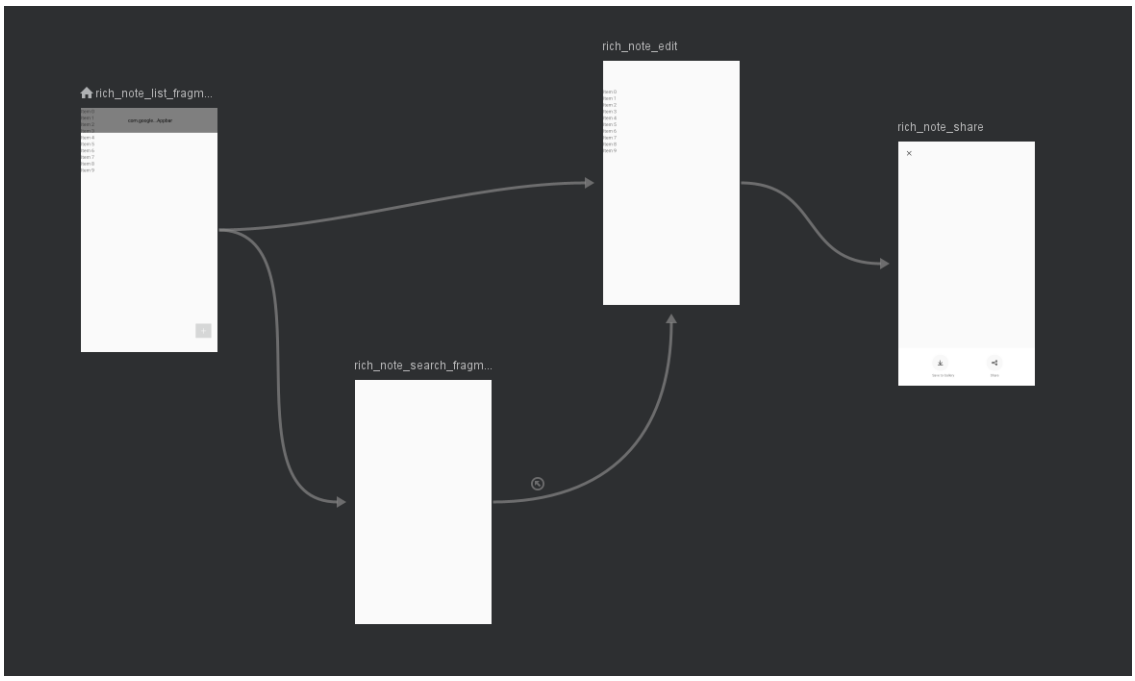
  enterAnim, exitAnim, popEnterAnim, popExitAnim
- 简化 deep link

  Deep Links配置
- 集中并且可视化管理导航

Navigation Graph

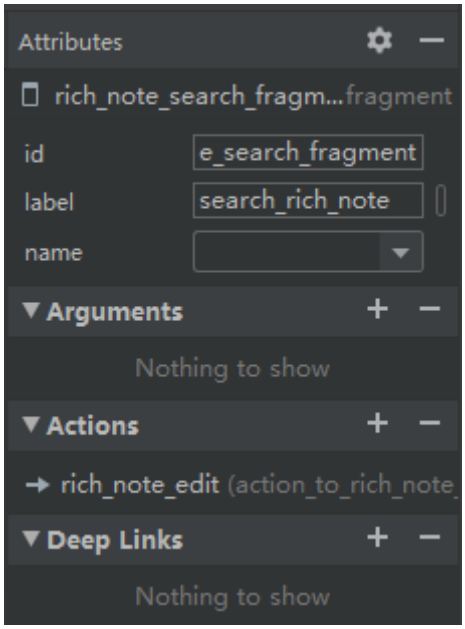
Navigation主要分三个部分

- Navigation Graph

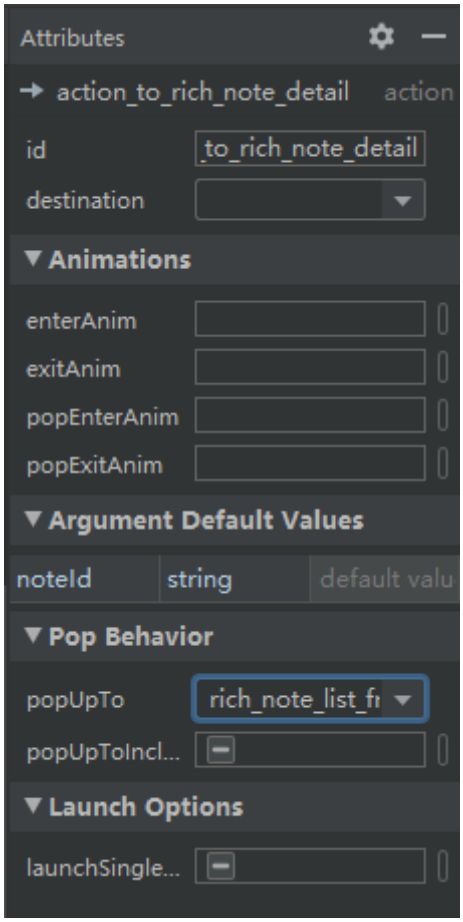



Navigation Graph中每一个界面叫 Destination，可以是fragment ， activity，或者自定义的Destination。Navigation管理的就是Destination间的跳转。

点击 Destination，可以在屏幕右侧看见 deep link 等信息的配置




Navigation Graph 中的带箭头的线叫：Action，它代表着 Destination 间不同的路径，它是从一个界面跳转另个一个界面的抽象。可以看到Action 的详细配置，可以配置跳转动画，Destination 间跳转传递的参数，操作返回栈，以及Launch Options




个人感觉Navigation Graph有点像有向图，其中的Destination 和 Action 对应的是点和边。

- NavHostFragment

  用于显示 `navigation graph` 中的 destination。Navigation 提供一个默认的 `NavHost` 实现 `NavHostFragment`，它显示 fragment destination。

  可以这样使用

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/nav_main">
</androidx.fragment.app.FragmentContainerView>
```

  所有的 fragment Destination都是通过 NavHostFragment管理，相当于一个容器。每个 NavHostFragment 都有一个NavController，用于定义 navigation host 中的导航。它还包括 navigation graph以及 navigation 状态（例如当前位置和返回栈），它们将与 NavHostFragment 本身一起保存和恢复

- NavController

  是Navigation的大管家，帮助 `NavHost` 管理导航，其内部持有 `NavGraph` 和 `Navigator`（专为NavGraph元素构建），NavGraph决定了界面间的跳转逻辑，Navigator定义了在应用内导航的机制。`Navigator` 是对 `Destination` 之间跳转的封装。每种 `Navigator` 都有自己的导航策略，例如 `ActivityNavigator` 使用 `starActivity` 来进行导航。

**同一个graph中共享ViewModel**

我们都知道 fragment 可以使用 activity 级别共享 ViewModel，但是对于单 activity 项目，这就意味着所有的 fragment 都能拿到这个共享的 ViewModel。本着最少知道原则，这不是一个好的设计。

Navigation引入了navigation graph 内共享的 ViewModel，使得 ViewModel 的作用域得到了细化，业务之间可以很好地被隔离。

```
private val richNoteViewModel: RichNoteListViewModel by
navGraphViewModels(R.id.nav_note)
```

### Safe Args

根据 navigation 文件生成代码，安全地在 `Destination` 之间传递数据

为什么要设计这样一个插件呢?

我们知道使用 bundle 或者 intent 传递数据时，如果出现类型不匹配或者其他异常，是在 Runtime 中才能发现的，而 `Safe Args` 把校验转移到了编译期。

支持的参数类型：

Int, Float, Long, Boolean, String, Resource Reference, Custom Parcelable, Custom Serializable, Custom Enum

启用 Safe Args 后，生成的代码将为每个操作以及每个发送和接收 destination 创建以下类型安全的类和方法。

- 为拥有 action 的发送 destination 的创建一个类，该类的类名为 destination 名 + Directions
- 为每个传递参数的 action 创建一个内部类，如果action 叫 action_to_rich_note_edit_fragment，则会创建 ActionToRichNoteEditFragment类。
- 为接收 destination 创建一个类，该类的类名为 destination 名 + Args

如何在发送 destination 传递参数

```
private fun navigateToRichNote(
    note: RichNoteListItem,
    view: View
) {
    val directions =
RichNoteListFragmentDirections.actionToRichNoteEditFragment(note.localId)
    view.findNavController().navigate(directions)
}
```

如何在接收 destination 中取出传来的参数，kotlin 可以使用 `by navArgs()` 获取参数

```
private val args: RichNoteEditFragmentArgs by navArgs()

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    val noteId = if (TextUtils.isEmpty(args.noteId)) {
        mViewModel.mRichData?.metadata?.localId
    } else {
        args.noteId
    }
}
```

**如果想要引用其他 module 中的 graph，可以使用 `include` 标签**

```
<!-- (root) nav_main.xml -->
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_main"
    app:startDestination="@id/nav_note">

    <include app:graph="@navigation/nav_note"/>

</navigation>
```

```
<!-- (root) nav_note.xml -->
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_note"
    app:startDestination="@id/rich_note_list_fragment">

    <fragment
        android:id="@+id/rich_note_list_fragment"

 android:name="com.oneplus.note.richnote.ui.homepage.RichNoteListFragment"
        android:label="rich_note_list"
        tools:layout="@layout/fragment_rich_note_list">

        ...


    </fragment>

    ......

</navigation>
```

### LiveData

LiveData符合标准的观察者模式，具备扩展性强、耦合性低的特性，同时它还是一个存储数据的容器，当容器数据改变时会触发观察者，即**数据驱动**

为什么要改变数据？无非是想让数据使用者感知到而已，而LiveData可以优雅的实现这一流程，将 改变、通知 两步操作合并为一步，即省事也提高了安全性。

LiveData的特性决定了它非常适合去做数据驱动UI，引入liveData后改变数据会自动触发UI渲染，将两步操作合并为一步，大大降低出错的概率。

什么是数据驱动UI，通俗一点说就是当数据改变时对应的UI也要跟着变，反过来说当需要改变UI只需要改变对应的数据即可。为了解决数据，UI一致性问题。

`LiveData` 在 `Lifecycle` 的加持下可以实现只在可见状态接收通知，说的通俗一点 `Activity` 执行了 `onStop()` 后内部的 `LiveData` 就无法收到通知。举个例子，ActivityA `和` ActivityB `共享同一个` LiveData，伪代码如下

```
class ActivityA{
    liveData?.observe(this, Observer { value->
            textView.text = value
        })
}
class ActivityB{
    liveData?.observe(this, Observer { value->
            textView.text = value
        })
}
```

当`ActivityA`启动`ActivityB`后多次改变`liveData`值，等回到`ActivityA`时 你肯定不希望`Observer`收到多次通知而引发`textView`多次重绘。

引入`Lifecycle`后这个问题便可迎刃而解，`liveData`绑定`Lifecycle`后，当回到`ActivityA`时只会取`liveData`最新的值然后做通知，当不再需要引用时，会自动清理它们，从而避免多余的操作引发的性能问题。

Jetpack ViewModel用于做状态托管，有对应的作用域可跟随Activity/Fragment生命周期，这种特性恰好可以充当MVVM ViewModel的角色，分割数据层和视图层并做数据托管。

ViewModel要时刻保证最新状态分发到视图层，如果数据的承载以及分发交给LiveData，而ViewModel专注于托管LiveData保证不丢失，就可以更好的实现Fragment之间的通讯。

ViewModel内部的viewModelScope是一个协程的扩展函数，`viewModelScope`生命周期跟随`ViewModel`对应的`Lifecycle（Activity/Fragment）`，当页面销毁时会一并结束`viewModelScope`协程作用域，所以将耗时操作直接放在`viewModelScope`即可。

在界面销毁时会调用`ViewModel`的`onClear`方法，可以在该方法做一些释放资源的操作，进一步降低内存泄露的风险。

在当前的Android中可以使用DataBinding实现同样的效果，DataBinding最大的优点跟唯一的作用就是数据 UI双向绑定，UI和数据修改任何一方另外一方都会自动同步，这样的好处其实跟LiveData的类似。

ViewModel从Repository拿到数据暂存到ViewModel对应的ObservableFiled即可实现数据驱动UI。但这里有一个前提是，从Repository拿到的数据可以直接用，如果在Activity或者Adapter做数据二次处理再notify UI，这就违背了数据驱动UI的核心思想了。所以要实现数据驱动UI必须要有合理的分层（UI层拿到的数据无需处理，可以直接用），Data Mapper恰好解决了这一个问题，同时也可规避大量编写BindAdapter。

在一个界面足够复杂的时候，那对应的ViewModel代码可能会有成千上百行，很臃肿可读性也非常差。关于这点，可以单独去写一个use case处理，use case通常放在ViewModel与数据层之间，业务逻辑以及Data Mapper都应该放在use case中，每一个行为对应一个use case。这样就解决了ViewModel臃肿的问题，同时更方便编写测试用例。但流程很单一并且后期改动的可能也不太大的情况就没必要去写一个use case，DataMapper放在数据层即可。

使用 Room 将数据保存到本地数据库，Room是一个对象映射库，可利用最少的样板代码实现本地数据持久性。它还允许观察对数据库数据（包括集合和连接查询）的更改，并使用LiveData对象公开这类更改。

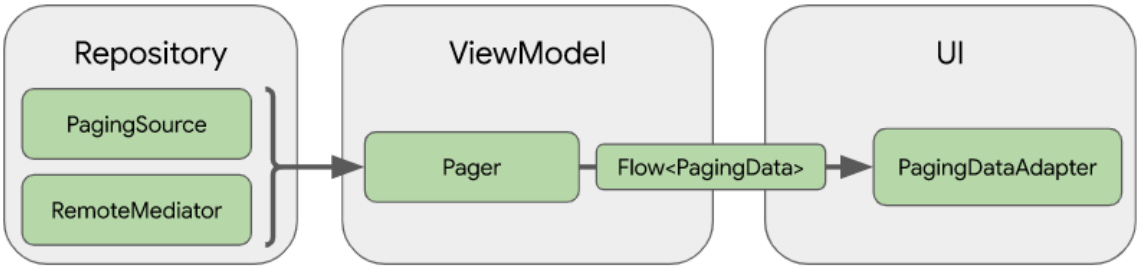
我们不能把数据库DAO操作，直接交给ViewModel去操作，上面说过ViewModel应该只拿数据，而不关心数据是如何处理得到的。所以我们可以将ViewModel数据获取过程委托给一个新的模块，即Repository。

Repository模块会处理数据操作。可以将存储区视为不同数据源（持久性模型、网络服务和缓存）之间的媒介。这一部分的重要作用：从应用的其余部分中提取数据源。可采用依赖注入模式使类能够定义其依赖项而不构造它们，提供清晰的依赖项管理模式。

在便签中，首页进入到编辑页中，我们通过传递对应Item的noteId，在编辑页通过noteId，通过数据库去获取对应的Note数据，而不是将Item对应的Note数据直接传递，这样做的原因是，有可能存在传递过去的Note数据不是最新的数据了，这样会导致UI显示的数据并不是数据库中最新的数据，我们将数据库作为单一可信的数据来源，通过noteId去获取实时最新的Note数据。

便签首页以及搜索页面都是使用Paging3实现的列表分页。

Room从2.3.0开始支持返回值类型为PagingSource的@Query注释方法生成实现。




PagingSource 为单一的数据源

PagingData 为单词分页数据的容器

Pager 用来构建`Flow<PagingData>`的类， 实现数据加载完成的回调

PagingDataAdapter 分页加载数据的RecyclerView的适配器，其中的关键就是AsyncPagingDataDiffer

ViewModel层利用末端操作符来消费来自数据层的数据流。使用`Flow.asLiveData()`扩展函数将Flow转化为LiveData，共享Flow的底层订阅，同时根据观察者的生命周期管理订阅。此外，LiveData可以为后续添加的观察者提供最新的数据，其订阅在配置发生变更的时候依旧能够生效。

MVVM最核心点是通过ViewModel做数据驱动UI以及双向绑定的操作用来解决数据/UI的一致性问题。

- View 专门做视图渲染以及UI逻辑的处理
- Repository:代表远程仓库，从Repository取需要的数据
- ViewModel: Repository取出的数据需暂存到ViewModel，同时将数据映射到视图层

Jetpack是Android官方为确立标准化开发而提供的一套框架，`Lifecycle`可以让开发者不用过多考虑生命周期引发的一系列问题 ~ 有了DataBinding的支持让数据UI双向绑定成为了可能 ~ `LiveData`的存在解除`ViewModel`跟`Activity`双向依赖的问题。归根到底`Jetpack`就是一套开发框架，只是让MVVM更简单，更安全。

Google在应用架构指南中，提出两个常见的架构原则：

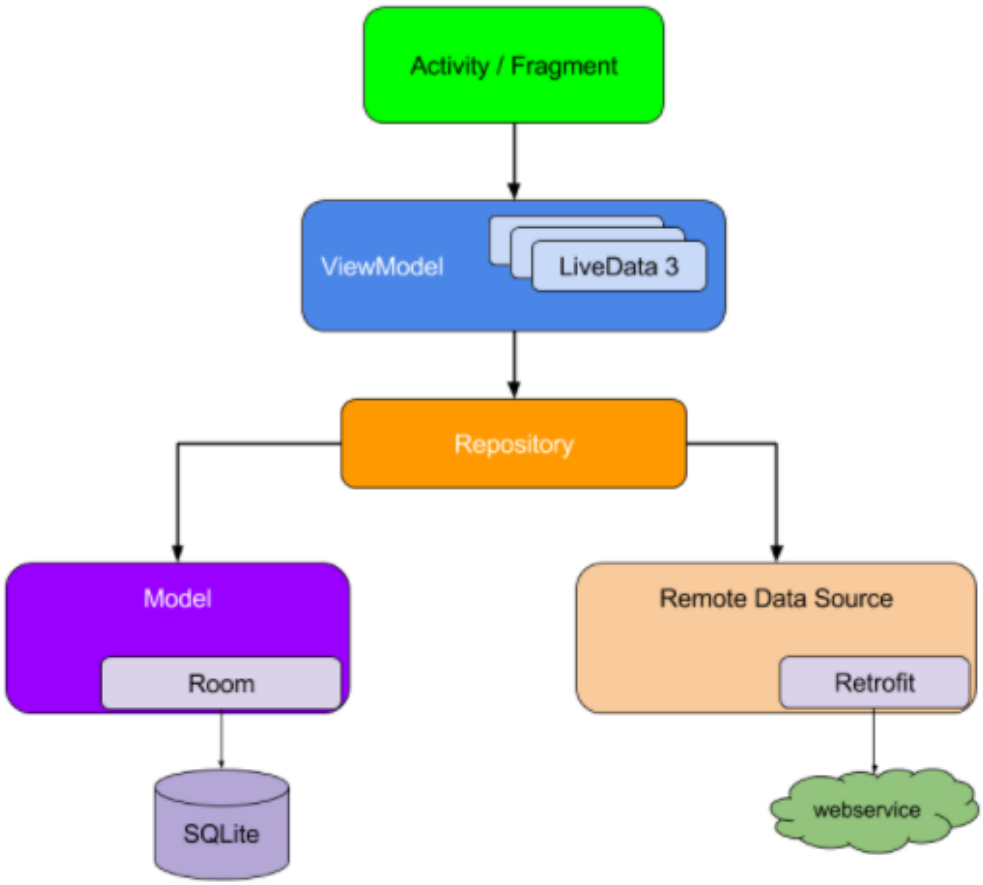
**分离关注点**：基于界面的类应仅包含处理界面和操作系统交互的逻辑。应使这些类尽可能保持精简，这样可以避免许多与生命周期相关的问题。

**通过模型驱动界面**：最好是持久性模型。模型是负责处理应用数据的组件。独立于应用中View对象和应用组件，因此不受应用的生命周期以及相关的关注点的影响。

为什么最好是持久性模型呢?

- 如果Android操作系统销毁应用以释放资源，用户不会丢失数据。
- 当网络连接不稳定或不可用时，应用会继续工作。

应用所基于的模型类应明确定义数据管理职责，这样将使应用更可测试且更一致。




**总结一下，引入Jetpack架构后**

- **Lifecycle 解决了生命周期 同步问题**
- **LiveData 实现了真正的状态驱动**
- **ViewModel 可以让 Fragment 通讯变得更优雅**
- **DataBinding 让双向绑定成为了可能**
- **Jetpack 只是让 MVVM 更简单、更安全**