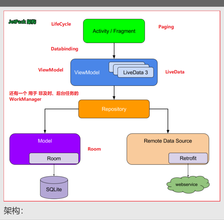
JetPack

Jetpack的优势



架构:

我理，官网这个 guide文档

在 dao（网络）和 repository 之间
又封装了一层 NetworkBoundResource --->

用以 实现 即在 显示缓存数据，后台拉取新数据 的同时，
有一个 刷新状态 通知。。。

androidx 一系列库

目标:
健壮、可测试、可扩展、高性能、低消耗

1、最好 一个 activity，加上 一堆 轻量级的 fragment

2、且 fragment 也尽量的 轻量，职责单一：
只负责 处理 用户的交互 和 系统响应。

其他的业务 都 剥离开， 交给 viewModel 或者 更下层 去处理

LifeCycle

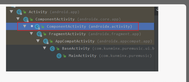
1、 轻 任务对象 提供 在 activity 或者 fragment 之外 来 简单感应 这些 周期 典型安卓的生命周期的接口 LifecycleOwner （比如activity） 其生命周期的 方法

getLifecycle().addObserver(observer)

2、 这样， 就可以 更方便的 解决 内存泄漏问题 了
--> LifecycleObserver 这个顶级接口，
没有处理 自动移除 自身监听的逻辑，所以，最好 在 自身监听到 onDestroy()，移除 自身，

3、 可以使用 DefaultLifecycleObserver 这个子接口，选择性的实现 需要的生命周期 回调，

原始接口，还要 加 @OnLifecycleEvent 注解，比较麻烦



1、我们在activity内部，通过 getLifecycle().addObserver(我们自定义的 生命周期监听类)，注入到 您的自定义的监听类；
在自定义类内部的方法，通过添加 OnLifecycleEvent 注解，声明具体的生命周期，即可在对应生命周期 被回调

2、原因：
主要是我们间的包下的：
androidx.activity.ComponentActivity 实现了 LifecycleOwner 接口，
在和onCreate() 时，注入了一个 不可见的 fragment 来感应 activity的生命周期；
ReportFragment.injectIfNeededIn(this);

这个fragment 在对应生命周期被 被回调时，会寻找 宿主activity的 lifeCycle 注册到 LifecycleRegistry ，
其实就是一个 集中管理 所有 宿主生命周期的观察者的 集合：
handleLifecycleEvent(event)

xmind

LifecycleOwner 生命周期 拥有者（宿主）的 实现类:

即 要拥有 自己 独特的 生命周期 才行，

2、还要实现 接口 Lifecycle getLifecycle()，让 其他人 能 注册、感知
其实 底层的实现 都是 LifecycleRegistry，由他 来分发 各类 生命周期事件

实现类：9 个

1、 供应级的 ComponentActivity

2、 Fragment
其实你对象的生命周期，由 FragmentManager 管理！
--> 通过 fragment 实例 的 getLifecycle() 获取

3、 FragmentViewLifecycleOwner
和 Fragment 的 mView 的 生命周期 紧密相关
没有view 就不会初始化，
注意 不是 Fragment的，是 其 view 的！
--> 通过 fragment 实例 的 getViewLifecycleOwner() 获取

4、 FragmentActivity的一个callback实现
忽略

5、 LifecycleRegistryOwner 这时的 子接口，
忽略

6、 LifecycleService 定制的 service子类，在其几个 生命周期内，也会 进行分发， 通过 getLifecycle() 也可以获取，注册

7、 NavBackStackEntry 导航相关。。。 没仔细研究

8、 ProcessLifecycleOwner
在 ProcessLifecycleOwnerInitializer 这个provider子类
初始create 时 来 操作 初始化，
--> 看起来 是想 通过 这个provider 注册时，
来 监控 整个app的 所有activity的生命周期，
对 activity生命的 创建、销毁 分别 进行计数。。。

LiveData 地址 同上

类似于 事件总线 EventBus：

通过它 包装数据，
1、 往 这个数据 bean 添加 观察者，
2、 也用它 来 set/post 数据，发送数据变更

这个对象 可以直接new，也可以 通过 Room数据库 查询的时候，直接返回。。

...
优势：
能感知 生命周期：
（1）只有 宿主 处于 active 状态，才会 回调 观察者的 onChange() 回调
（2）在 宿主 销毁时，自动 清空 观察者，防止 泄露！

通过 livedata 传入的 observer 会被封装进 LifecycleBoundObserver ，

1、 先加入了 livedata 自身的 观察者集合

ObserverWrapper existing = mObservers.putIfAbsent(observer, wrapper);

2、 再加入 activity 等 LifecycleOwner 的 mObserverMap ，
来感知 其生命周期回调。
当 生命周期变更时，
会回调 LifecycleBoundObserver.onStateChanged() 时
（1）它 做了个判断，
if (mOwner.getLifecycle().getCurrentState() == DESTROYED) {
removeObserver(mObserver);
return;
}

如果 宿主 销毁了， 调用 removeObserver(mObserver);
会将这个observer 被引用的地方 都清掉（
livedata 的 map 会删除它，
LifecycleRegistry 会删除它
）

所以 livedata 的 observer 会很安全，不会 内存泄露！ ！

（2）否则，才会进行下一步判断

a、 可视状态 是否改变，没有的话，直接 return!

b、 当前宿主 至少是 androidx.lifecycle.Lifecycle.State#STARTED ，
才能 刷新数据
dispatchingValue(this);
----> 为了 避免 不可见时 还刷新UI的 功耗问题。。
因为 下次可见时，还会回调 来刷新的 ，不给 数据不准确！

而 刷新的逻辑，
就是比较 你这个观察者的 数据版本 和 我 livedata 最新的版本 大小

if (observer.mLastVersion >= mVersion) {
return;
}

observer.mLastVersion = mVersion;
observer.mObserver.onChanged((T) mData);

只有 observer的版本 较小，表示持有的是 旧数据，才会把最新的数据 发给他，
调用其 onChange() 回传 新数据！

LiveData#dispatchingValue()
即 刷新 观察者 数据的流程：

1、 lifecycle 的 状态 由 不可见 --> 可见

2、 显式的调用了
livedata 的 setValue() 或者 postValue() 方法

ViewModel

1、view 和 model（ui 相关数据） 之间的 桥梁，
是 最稳定于 view 层 的，
还需要 Databinding 帮它绑定：
---> databinding 绑定 layout，然后 设置 vm 进去，

并且 屏幕旋转 也不会被销毁！

通常是在 view 所在的 xml 直接 声明 vm变量 并使用，

（1）首先 要将 这xml 改写成 绑定型布局 layout
（2）在<data> 节点下，声明<variable>节点，
即 声明一个变量 供当前的vm使用，
name 是 变量名；
type 即 类名
（3）通常 vm 就是在这里使用的，
指定 xmlFlr 某个view的属性 或者 action
直接关联 对应 variable的某个属性 或 方法

2、 通常 其内部 会包装一堆的 livedata 对象，
或者 observable对象

vm 实例的获取方式：

1、 不要 直接 new，这样 不好管理，会导致 内存泄漏

2、 要 存放到 一个 ViewModelStore 里面 去管理，
通过 ViewModelProvider工具类 来回去
（1）一个 storeOwner 对应一个 store，通过 new 一个provider 去获取 vm实例，
（2）provider 需要 一个 store ，一个能创建vm的factory
不要用 旧的方式 ViewModelProviders.of() 来获取 provider；
用 直接 new，并传入 factory的方式；
（3）之后，传入 vm的class字节码对象，就可以获取 一个 vm实例了

这里，provider，代理了 store 罢了，保证 一个scope 只有一个 vm实例对象，不会重复创建！

ViewModelStoreOwner 接口，
即 ViewModelStore 这个 vm 容器的管理类
或者说 是 一个 scope，
可以是 一个activity，一个 fragment 等等

---> 注意哦！

我们可以让 自定义的 application 实现 这个接口，就可以 让app来存放一个 跟随app 全生命周期的 状态共享 vm了；
在跨activity，fragment的跳转 获取 vm，就 可以直所这些共享状态，得到 变更通知！ ！

它的职责是：

1、在 因 配置改变时 scope需要 销毁 重建时，
要 保留住 ViewModelStore

2、 而在 自身 scope（比如 activity）真的销毁时，要 调用 ViewModelStore.clear() 清理 vm，
防止 内存泄漏！

public final void clear() {
for (ViewModel vm : mMap.values()) {
vm.clear();--> 让 vm 自身 也做 清理！
}
mMap.clear();
}

有4个 实现类：
1、activity 的 极简单，在 配置改变时，会自己缓存到 NonConfigurationInstances

2、fragment
当时是从 Fragment 内部的 FragmentManagerViewModel 那里 去拿 自己对应的

3、还是用 activity 的

4、用自己的 NavControllerViewModel --> 说明完。。

所以 vm 其 生命周期 比 相应的 view 要长。

所以，vm 不要 直接引用 对应的 view对象！

1、activity 在不是 因为配置改变 而导致的 销毁，重建时，身上 附带的 viewmodel 会被清掉， 保证 内存不会泄露！

2、操作是，通过监听 自己的生命周期
if (event == Lifecycle.Event.ON_DESTROY) {
if (!isChangingConfigurations()) {
getViewModelStore().clear();
}
}
}
3、其实 跟踪 也没关系，因为 在 onRetainNonConfigurationInstance 时（需要重建 而保留数据），
已经 保存到 NonConfigurationInstances.viewModelStore 中 去了。

xmind

DataBinding 地址 同上

Databinding 早期 相关情况:

mvvm 架构的由来

1、 抽象类，帮助 我们 绑定 layout 和 vm
layout 中，就可以直接 使用 vm 对象的属性了！

2、 一旦我们在 某个 xml 布局中，顶级标签是 <layout>
即 这个view 启用 Databinding 了！

会由 APT 帮 我们生成 这个layout 真正的 ViewDataBinding 类 和 其 Impl 类 ，
（1）分别为： layoutB类 + Binding后缀，和 layoutB类 + BindingImpl 后缀
（2）如果有两、一堆一堆，
那么 后一个就是 layoutB类 + BindingLandImpl 后缀，如右图
（3）layout文件中 带有 vm 信息的 livedata、observableBoolean 等，
会被 localFieldCount 统计，之后在 生成的 ViewDatabindingImpl类的 executeBindings()
监听中，被包括成 WeakListener（弱引用 持有 binding对象，防止泄露）， 放到
mLocalFieldObservers 数组中。
这样的话，这些 对象，就可以 即时 做主的 生命周期了 --> 不会内存泄漏

3、 vm 和 layout 的 真正绑定 是由都使用的 DataBindingUtil

--> 调用其 DataBindingUtil.bind() 方法 在 ViewDataBinding 实例对象，
其实 就是 解析出 rootView，当参数 构造 对应的 databindingB对象

binding 对象的 获取方式，都是静态方法，推荐 DataBindingUtil 的 1 和 3

（1）activity的 binding：
DataBindingUtil.setContentView(this, R.layout.activity_main);

（2）fragment的 binding：
FragmentLXXXBinding.bind(之前 inflate 出来的 view)

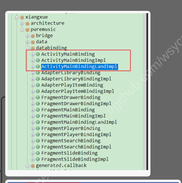
（3）其他任意 layout 的：
DataBindingUtil.inflate(inflater layoutId , parent, attachToParent);

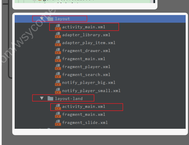
4、通过 bindingB对象 的 getRoot() 方法 --> 可以拿到 根 view！

5、有了 rootView，也可以 获取 binding 对象，然后 设置 layout 所需的 vm对象

--> 通过 DataBindingUtil.getBinding (rootView) 方法，
其实 就是调用了 ViewDataBinding.getBinding(view); 方法，
--> 原理是： view.getTag(R.id.dataBinding);

因为，databindingB对象 在 定义时
将自身以 R.id.dataBinding 为 tag键，设置给了 rootView





ViewBinding 接口的 抽象子类:

所以，可以用它来 直接引用 layout 中 指定 id 的那些 子view，进行 view的 属性设置，修改

layoutB中 给 view设置的如下，没有前缀的操作：

allowDrawerOpen="@{vm.allowDrawerOpen}"
isOpenDrawer="@{vm.openDrawer}"

实际上 就是 @BindingAdapter
即 = 号前的 是一个value，= 号后的是 传入的值

需要在 一个 XXXBindingAdapter的类中
写对应的处理函数，处理这个 设置 行为；

函数名任意（最好相关），
然后 注解的 value 就是 = 号前的value，
函数参数，依次是：view对象，= 后的参数（可以多个参数一起写，参数类型 就是 对应类型了）

这些 函数，会被 databindingImpl 生成类的 executeBindings()，使用！
--> 即 每次执行绑定时 回调（绑定到window，刷新新一帧，显式 重绑定等）---> 设置给 各 view，响应 设置

因为 livedata 感知 BaseObservableField 这些 vm 内部的字段，
被修改过的情况，会修改 flag，然后 调用 bindingImpl 的
notifyPropertyChange()，进行上述 重绑定

Room

1、官方 研发的，和 sqlite 数据库 打交道的 orm 框架；

2、优势：
（1）dao层 和 database层 实体类
完全由 APT 生成，全部使用 数据库事务，保证 数据安全；
（2）可以直接返回 各种数据类型：
原始bean、rxjava、flowable、livedata

3、为 livedata ，可以感应 数据库变化，
同步更新 view --->

即使有 网络请求新数据 也要 先更新到 数据库，
在变动 view ，保证 数据来源唯一，数据 一致性！

xmind

Navigation

路由框架：

1、可以做 activity，主要是 fragment 的导航

2、主要是 通过 有向无环图 来做，其实就是
在res/navigation文件夹下 创建 xml 文件

比如：nav_main.xml
（1）对应 一个 NavHostFragment 对象的 navGraph 属性
（专门用来做导航用的 fragment 容器）

（2）这个xml 声明 一个导航关系图：

navigation 是 根节点：

里面有 多个 fragment 节点，以及 startDestination 表示 起始 fragment；

fragment 节点：
声明 其 布局内的 id，name 类型，名称 table，
layout 布局（这就是 普通的 layout 布局文件了）

内部的 action 节点：表示一个 跳转操作
id 用来 代码识别，destination 指定 跳转目标，还有 其他各种动画配置

3、 路由xml的使用：
在 真正的layout.xml 文件中，通过 指定 一个 fragment节点，
（1）指定其类型为：
android:name="androidx.navigation.fragment.NavHostFragment"
（2）指定 路由xml
app:navGraph="@navigation/nav_graph_main"
（3）打开 路由键，让 navHost 代理 操作 即可
app:defaultNavHost="true"

4、action 使用：
Navigation.findNavController(view).navigate(R.id.action_page2)
最后的参数 id 就是 路由action id

xmind

Paging

分页 加载数据，按需加载。
1、目前 主要是 适配了 RecyclerView

2、

md

WorkManager

1、对于 非及时性的后台任务，都可以使用

官方保证 省电、操作 完成

2、内部实现方式：
（1）数据库 记录操作，所以可以 断点 继续
（2）6.0之前使用 广播+闹钟， 之后 使用 jobScheduler
（3）可以对 这些任务 加 限制条件（必须联网，不能低电量，
网络怎样才可以，磁盘空间限制，设备是否空闲 等）

3、4.0 之后 可以使用

xmind

XMind