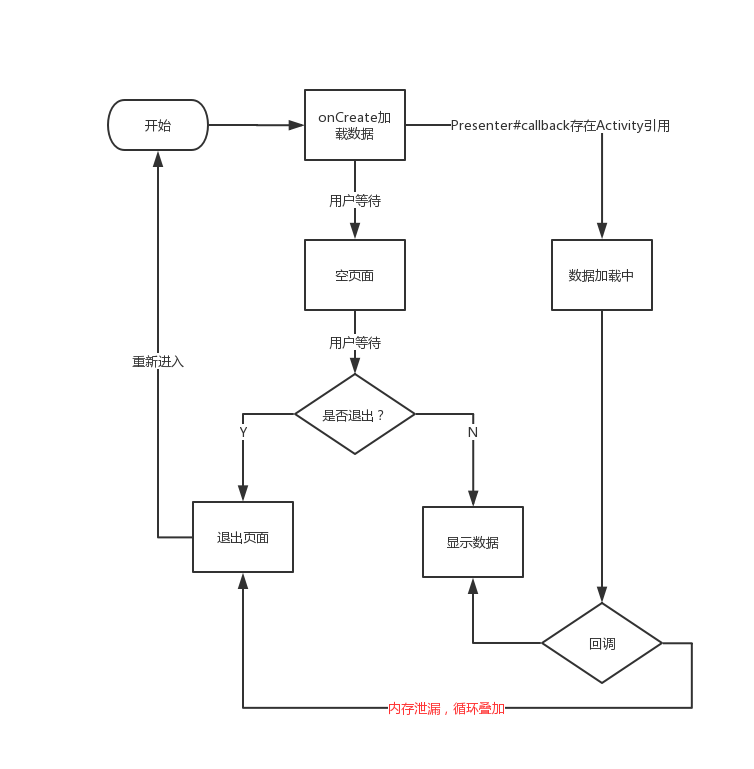
Android JetPack架构之LifeCycles+LiveData+ViewModel 使用与源码解析

之一：LifeCycles

问题场景：

平时我们在写页面代码的时候，常常会使用MVP的设计模式（针对这个模式我想初学者可以去自行了解，这里默认读者都会了-w-），针对这个模式，有一个设计弊端，就是P层执行加载数据的方法的时候，常常遇到网络不好或者加载时间过长的问题，导致数据回调时间过长，这个时候一般用户可能就不会感兴趣，直接点击返回上级页面了，这个时候如果我们作为开发者如果不取消回调那么就会常常出现内存泄漏的问题，内存泄漏一直都是开发者需要时刻关注的问题，因为不适当的使用会使手机内存消耗过大导致卡顿

一个简单的流程图



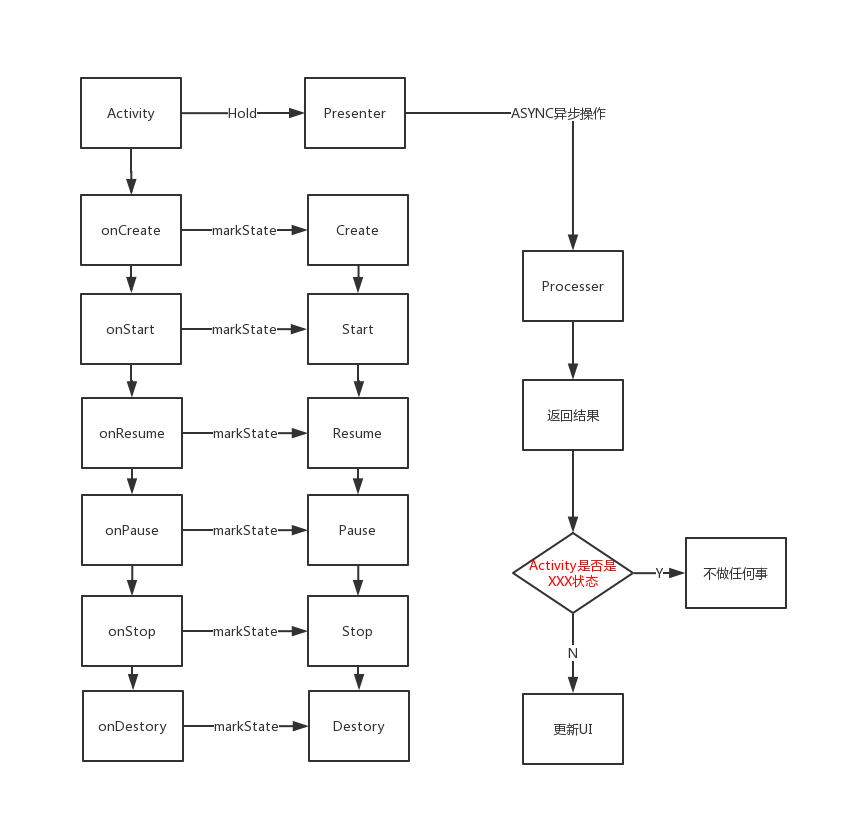
红色字体是产生内存泄漏的地方，如果反复进入退出，容易导致泄漏

问题：

针对这些对Activity生命周期比较敏感的数据，我们是如何处理的呢？业界常规做法都是Activity进入啥状态了，通知Presenter进入啥状态，Presenter中写入标记位，并且在P中回调的时候查看该标记位是否处于destory，如果页面关闭了，那么这些数据将不需要回调了，那么就不会存在activity引用了，这样内存就不会泄漏了，当然了，Presenter也可以实时查询Activity状态，相当于状态同步了

还一种比较好的做法就是Presenter弱引用Activity的Callback，这一点可以解决内存泄漏，但是针对查询activity状态还需要实现自定义接口方法，因此不如上面一种同步好

第一种流程图如下：



问题：

这样看上去确实可以解决这个问题，可以针对每个activity与每个presenter都要这样写通知与state这样的结构，我们下面需要将这些整合到我们自己写的基类BaseActivity里面

仔细分析一下这个结果，activity进入某个状态，通知P层进入某个状态，这个看上去是不是有点像观察者模式呢？

嗯，我们可以在activity中持有P层的观察者，activity作为被观察者可以这样哟~

下面开始我们的代码吧，第一步，activity作为被观察者，需要存储观察者的集合：

写一个LifecycleRegistry类：

public class LifecycleRegistry implements LifecycleOwner {  
 */\*\*  
 \* 定义activity状态值，ON\_CREATE,...ON\_DESTROY  
 \* 参考{****@link*** *android.arch.lifecycle.Lifecycle}  
 \*/* private int statue;  
  
 */\*\*  
 \* 定义一个观察者集合  
 \*/* private Set<LifecycleObserver> holdData = new HashSet<>();  
  
 @Override  
 public int getCurrentState() {  
 return statue;  
 }  
  
 */\*\*  
 \* 添加观察者  
 \** ***@param*** *observer observer  
 \*/* public void addObserver(LifecycleObserver observer){  
 holdData.add(observer);  
 }  
  
 */\*\*  
 \* 同流程图，activity生命周期调用  
 \** ***@param*** *activityState {****@link*** *LifecycleRegistry#statue}  
 \*/* public void markStatue(int activityState) {  
 for (LifecycleObserver observer : holdData) {  
 observer.onStatueChanged(statue, activityState);  
 }  
 // 改变为最新状态  
 this.statue = activityState;  
 }  
}  
}

//被观察者用于回调查询当前状态接口定义

public interface LifecycleOwner {  
 public int getCurrentState();  
}

//观察者实现接口定义

public interface LifecycleObserver {  
 void onStatueChanged(int oldStatue,int newStatue);  
}

这里为什么需要定义2个接口呢？

其实P层持有activity内部被观察就可以通过其实现的接口LifecycleOwner来主动查询其状态，这里就不需要观察者模式了，但是存在P对LifecycleOwner的引用

第二种则是典型的观察者模式，需要P层实现该接口，并且通过时候P实现的回调接口来通知P状态改变，这里P对LifecycleOwner不存在引用

这里我们把2种方式都加入，后面我们也可以对第二种方式进行改造

下面开始我们的BaseActivity编码：

public class SupportActivity extends AppCompatActivity {  
  
 */\*\*  
 \* 观察者注册类  
 \*/* private LifecycleRegistry lifecycleRegistry = new LifecycleRegistry();  
  
 */\*\*  
 \* P层  
 \*/* private BasePresenter presenter;  
  
 @Override  
 protected void onCreate(@Nullable Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 lifecycleRegistry.markStatue(Lifecycle.State.*CREATED*);  
  
 lifecycleRegistry.addObserver(presenter);  
 }  
  
 @Override  
 protected void onStart() {  
 super.onStart();  
 lifecycleRegistry.markStatue(Lifecycle.State.*STARTED*);  
 }  
  
 @Override  
 protected void onResume() {  
 super.onResume();  
 lifecycleRegistry.markStatue(Lifecycle.State.*RESUMED*);  
 }  
  
 @Override  
 protected void onPause() {  
 //这里一般没有状态，表示可以更新，如果需要，可以定义常量通知加入  
 super.onPause();  
 }  
  
 @Override  
 protected void onStop() {  
 //这里一般没有状态，表示可以更新，如果需要，可以定义常量通知加入  
 super.onStop();  
 }  
  
 @Override  
 protected void onDestroy() {  
 lifecycleRegistry.markStatue(Lifecycle.State.*DESTROYED*);  
 super.onDestroy();  
 }  
}

好了，一个简单的生命周期管理类使用观察者模式，关联了P层在Activity状态关闭的时候通知P层状态了，P层自己实现观察者接口LifecycleObserver并自行记录状态即可，一个简单的Presenter如下：

public class BasePresenter implements LifecycleObserver {  
 */\*\*  
 \* {****@link*** *Lifecycle.State}  
 \*/* private int activityState = Lifecycle.State.*INITIALIZED*;  
   
 private UICallback uiCallback;  
  
 public BasePresenter(UICallback uiCallback) {  
 this.uiCallback = uiCallback;  
 }  
  
 @Override  
 public void onStatueChanged(int oldStatue, int newStatue) {  
 activityState = newStatue;  
 if(activityState == Lifecycle.State.*DESTROYED*) {  
 //如果activity进入了DESTROYED，那么需要移除对activity的回调引用  
 uiCallback = null;  
 }  
 }  
  
 */\*\*  
 \* 这是读取数据方法  
 \*/* public void loadPage(){  
 //  
 //...长时间读取数据  
 //  
  
 //判断状态是否需要回调数据，如果activity不是XXX状态那么更新UI，否则啥事不做  
 if(activityState > Lifecycle.State.XXX) {  
 if(uiCallback != null) {  
 uiCallback.onComplete(result);  
 }  
 }  
 }  
}

这就是一个简单的生命周期敏感类如何处理并查看activity状态，在合适的时候触发回调，页面关闭状态的移除回调，避免内存泄漏

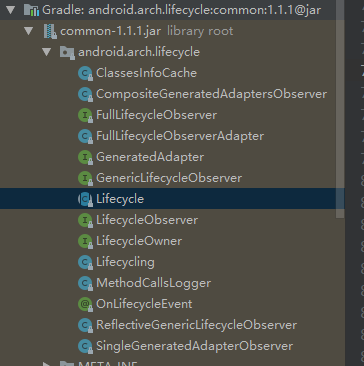
这就是一个简单的观察者模式，activity持有P层观察者接口通知P层状态变化，P层实现UI释放与状态保存，对Activity回调的时候判断状态作不同处理

对于JetPack组件架构LifeCycle 其实现原理就是上面例子

参考：

JetPack组件架构LifeCycle

android.arch.lifecycle.Lifecycle



疑惑解答，为什么组件架构LifecycleObserver是个空接口，那么P实现该接口，怎么通过注册注解的方式来实现回调生命周期方法？

解答：

这里我们可以模拟接口，使用反射调用就可以，修改通知方法，通过查询观察者的方法，只要实现对应注解然后反射调用就可以了，这里只要开发者对自定义注解稍微有点了解都可以明白，也能自己写出这样的接口与注解（默认读者会基本的java注解的自定义与反射调用咯~）

以上就是笔者对jetPack组件LifeCycle的理解与模拟实现，如有不当之处，欢迎读者回复指出，谢谢~