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

之三：ViewModel

上面两篇文章我们介绍了JetPack之LifeCycle组件与LiveData，以及两者连用，这边文章下面主要讲解如何编写防止页面重新创建的LiveData数据

问题：

ViewModel是什么？

下面是官网介绍截图：



个人理解的意思如下：

该ViewModel类旨在存储和管理 用户界面Activity里面相关LiveData数据 。保存Activity中的数据避免重新创建，比如屏幕旋转配置更改导致Activity重新创建的异常情况

简单来讲，就是方式Activity因为异常情况被回收/重新创建导致需要保存的数据临时丢失问题

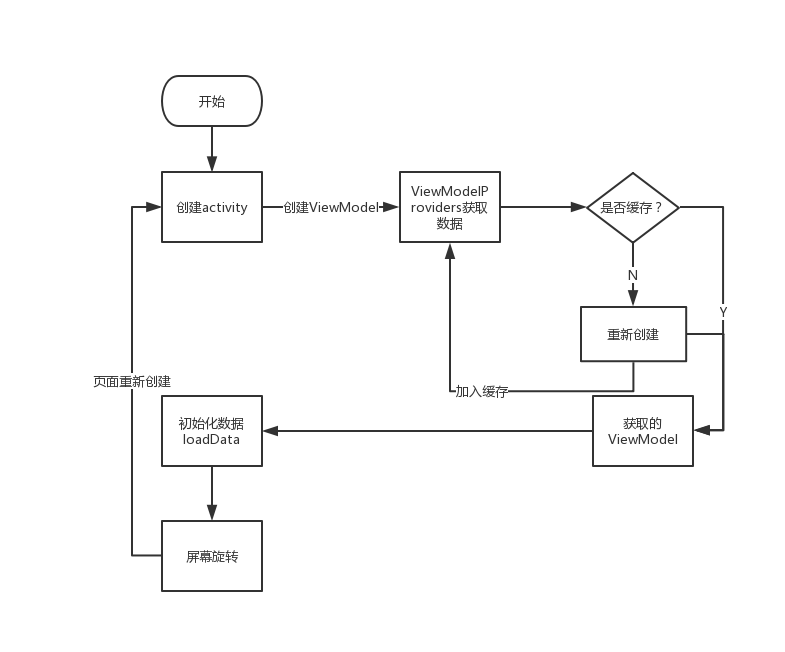
常见的处理方式：

作为开发者，我们经常需要考虑页面变化需要保存临时数据，你可能在Activity# onSaveInstanceState方法里面写入少量的序列化数据并且重新创建的时候来重新初始化，但是这个方法是存在弊端的

第一点：数据量少，不能写入过多数据

第二点：数据必须是可序列化的，如果没有序列化，那么将无法保存

ViewModel就是为了在解决此类问题，让开发者不再为了重新创建而烦恼数据的丢失，其实现方式就是采取的内存缓存，下面我们会编写并模拟实现方式



上图很好的诠释了如何获取ViewModel以及Activity重新创建如何获取之前的ViewModel

1.Activity执行onCreate，

2.从ViewModelProviders获取ViewModel数据，如果不存在，那么会创建ViewModel数据，同时加入内存缓存（其实就是一个集合里）并返回，如果存在，返回该数据

3.Activity加载返回的数据，用户执行操作

4.页面意外重新创建，重新走1流程，此时2流程里面已有数据，将直接返回

使用方法：

1.定义我们的ViewMoel数据：

public class KeepData extends ViewModel {  
 private MutableLiveData<String> liveData = new MutableLiveData<>();  
  
 public LiveData<String> getLiveData() {  
 return liveData;  
 }  
  
 public void setLiveData(String args) {  
 liveData.setValue(args);  
 }  
}

一个简单的ViewModel类就生成了，里面有一个LiveData类，用于观察Activity生命周期，与LifeCycle联用

2.Activity#onCreate创建ViewModel类：

KeepData keepData = ViewModelProviders.*of*(this).get(KeepData.class);  
keepData.getLiveData().observe(this, new Observer<String>() {  
 @Override  
 public void onChanged(@Nullable String s) {  
   
 }  
});

创建一个变量，同时使用我们第二章讲解的观察Activity生命周期变化

以上就是ViewModel的用法，看上去几行代码，就能解决数据丢失问题，下面我们将追踪源码分析其实现方式：

1. ViewModelProviders. of(Activity).get方法解析：

1.1 off方法是一个静态方法，其返回ViewModelProvider对象，此时只是返回ViewModelProvider对象，还没有生成我们的KeepData，

1.2 get方法解析：ViewModelProvider#122

ViewModel viewModel = mViewModelStore.get(key);

public <T extends ViewModel> T get(@NonNull String key, @NonNull Class<T> modelClass) {  
 ViewModel viewModel = mViewModelStore.get(key);  
  
 if (modelClass.isInstance(viewModel)) {  
 //noinspection unchecked  
 return (T) viewModel;  
 } else {  
 //noinspection StatementWithEmptyBody  
 if (viewModel != null) {  
 // TODO: log a warning.  
 }  
 }  
  
 viewModel = mFactory.create(modelClass);  
 mViewModelStore.put(key, viewModel);  
 //noinspection unchecked  
 return (T) viewModel;  
}

这里可以看出，这是ViewModelStore对象获取出来的，我们可以把它想象成一个Set集合，其核心实现就是ViewModelStore存取ViewModel，ViewModelStore定义如下：

public class ViewModelStore {  
  
 private final HashMap<String, ViewModel> mMap = new HashMap<>();  
  
 final void put(String key, ViewModel viewModel) {  
 ViewModel oldViewModel = mMap.put(key, viewModel);  
 if (oldViewModel != null) {  
 oldViewModel.onCleared();  
 }  
 }  
  
 final ViewModel get(String key) {  
 return mMap.get(key);  
 }  
  
 */\*\*  
 \* Clears internal storage and notifies ViewModels that they are no longer used.  
 \*/* public final void clear() {  
 for (ViewModel vm : mMap.values()) {  
 vm.onCleared();  
 }  
 mMap.clear();  
 }  
}

了了几行，可以看出，就是Map实现的存取，如果不存在，那么执行ViewModelProvider#134

viewModel = mFactory.create(modelClass);具体实现很简单，就是使用反射创建实例

public static class NewInstanceFactory implements Factory {  
  
 @SuppressWarnings("ClassNewInstance")  
 @NonNull  
 @Override  
 public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {  
 //noinspection TryWithIdenticalCatches  
 try {  
 return modelClass.newInstance();  
 } catch (InstantiationException e) {  
 throw new RuntimeException("Cannot create an instance of " + modelClass, e);  
 } catch (IllegalAccessException e) {  
 throw new RuntimeException("Cannot create an instance of " + modelClass, e);  
 }  
 }  
}

最后创建完实例，加入ViewModelStore里面并返回

以上就是源码的实现方式，简单来讲就是使用ViewModelStore来存取数据，不与Activity绑定，存在于内存中，我们可以使用静态来实现，而与Activity绑定的是ViewModel里面的LiveData数据，观察Activity变化，在Activity销毁的时候，我们第二章介绍过会移除LiveData与Activity的绑定关系，这样LiveData存在于静态内存ViewModelStore里面，同步不会引用Activity而导致内存泄漏

下面我们就来模拟源码实现

第一步：创建我们的ViewModelProvider，这是什么呢，就是一个Provider，创建我们的ViewModelStore，想要不在Activity重新创建，我们可以创建一个HolderFragment依附到activity上面，并且设置setRetainInstance(true)保证activity旋转不会导致HolderFragment销毁，同时HolderFragment提供返回ViewModelStore作为存储器

public class HolderFragment extends Fragment {  
 public static final String *HOLDER\_TAG* = "android.arch.lifecycle.state.StateProviderHolderFragment";  
  
 private final ViewModelStore mViewModelStore = new ViewModelStore();  
  
 */\*\*  
 \* 确保不会因为activity旋转重新创建  
 \*/* public HolderFragment() {  
 setReenterTransition(true);  
 }  
  
 public static HolderFragment holderFragmentFor(FragmentActivity activity) {  
 FragmentManager fm = activity.getSupportFragmentManager();  
 Fragment holder = fm.findFragmentByTag(*HOLDER\_TAG*);  
 if (holder instanceof HolderFragment) {  
 return (HolderFragment) holder;  
 } else {  
 HolderFragment holderFragment = new HolderFragment();  
 fm.beginTransaction().add(holder, *HOLDER\_TAG*).commitAllowingStateLoss();  
 return holderFragment;  
 }  
 }  
  
 @Override  
 public void onDestroy() {  
 mViewModelStore.clear();  
 super.onDestroy();  
 }  
}

holderFragmentFor 方法用于查找或者创建HolderFragment，成员ViewModelStore作为存储器，保存ViewModel数据，其中HolderFragment构造函数设置fragment持久化，并在fragment真正destory的时候关闭清除ViewModelStore存储器

第二部：提供get创建ViewModel方法：

public <T extends ViewModel> T get(@NonNull Class<T> modelClass) {  
 String canonicalName = modelClass.getCanonicalName();  
 if (canonicalName == null) {  
 throw new IllegalArgumentException("Local and anonymous classes can not be ViewModels");  
 }  
 ViewModel viewModel = mViewModelStore.get(canonicalName);  
  
 if (modelClass.isInstance(viewModel)) {  
 // noinspection unchecked  
 return (T) viewModel;  
 } else {  
 // noinspection StatementWithEmptyBody  
 if (viewModel != null) {  
 // *TODO: log a warning.* }  
 }  
 viewModel = NewInstanceFactory.*create*(modelClass);  
 mViewModelStore.put(canonicalName, viewModel);  
 // noinspection unchecked  
 return (T) viewModel;  
}  
  
public static class NewInstanceFactory {  
 @SuppressWarnings("ClassNewInstance")  
 @NonNull  
 public static <T extends ViewModel> T create(@NonNull Class<T> modelClass) {  
 // noinspection TryWithIdenticalCatches  
 try {  
 return modelClass.newInstance();  
 } catch (InstantiationException e) {  
 throw new RuntimeException("Cannot create an instance of " + modelClass, e);  
 } catch (IllegalAccessException e) {  
 throw new RuntimeException("Cannot create an instance of " + modelClass, e);  
 }  
 }  
}

简单的创建ViewModel并加入ViewModelStore以及返回创建的ViewModel

以上就是我们模拟源码android.arch.lifecycle.ViewModelProvider创建一个自己的ViewModelProvider，至于使用ViewModelProviders.of方法就是返回ViewModelProvider对象，源码也是使用new返回创建一个ViewModelProvider，并且创建holderFragment来生成一个ViewModelStore（通过holderFragmentFor(activity).getViewModelStore()方法，就是使用Fragment来进行数据存储，并且该Fragment是持久化的）

疑惑解答：

笔者在使用ViewModel的时候，我们可以看下ViewModelStore类的源码定义，其只提供了2个公开方法，一个是get方法，获取单个数据实体，第二个就是clearAll，区别发清除内存集合所有数据，那么如果想要删除集合中的单个key（根据className）保存的数据，确实没有办法，所以这里如果想要扩展这个ViewModelStore类，个人建议可以自己写这样一个结构

疑惑解答之二：

ViewModelStores#holderFragmentFor(activity)源码流程

1. 创建一个HolderFragment依附到activity上面，并且设置setRetainInstance(true)保证activity旋转不会导致HolderFragment销毁

2. HolderFragment在执行onDestory的时候清除ViewModelStore存储，做到activity真正退出才会清除ViewModel的数据

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

参考：

android.arch.lifecycle.ViewModel

android.arch.lifecycle.ViewModelProviders （implementation"android.arch.lifecycle:extensions:1.1.1"）