“终于懂了“系列：Jetpack AAC完整解析（三）ViewModel 完全掌握！

胡飞洋 2021-01-07 21:40 15675

关注

> Jetpack AAC 系列文章:
>
> “终于懂了“系列：Jetpack AAC完整解析（一）Lifecycle 完全掌握！
>
> “终于懂了“系列：Jetpack AAC完整解析（二）LiveData 完全掌握！
>
> “终于懂了“系列：Jetpack AAC完整解析（三）ViewModel 完全掌握！
>
> “终于懂了“系列：Jetpack AAC完整解析（四）MVVM - Android架构探索！
>
> “终于懂了“系列：Jetpack AAC完整解析（五）DataBinding 重新认知！

> 欢迎关注我的 **公 众 号**，微信搜索 **胡飞洋** ， 文章更新可第一时间收到。

上一篇介绍了Jetpack AAC 的数据处理组件 LiveData，它是使得 数据的更新 能以观察者模式被observer感知，且此感知只发生在活跃生命周期状态。 这篇来介绍与LiveData搭配使用的视图模型组件——ViewModel。

> 注意，如果你对MVVM架构中的VM和本篇的ViewModel都没有一定认识的话，那么就不要将两者进行联想了。目前，你就理解为没有任何关系。后面会有专门篇幅介绍MVVM。

一、ViewModel介绍

ViewModel是Jetpack AAC的重要组件，同时也有一个同名抽象类。

ViewModel，意为 视图模型，即 **为界面准备数据的模型**。简单理解就是，ViewModel为UI层提供数据。 官方文档定义如下：

> ViewModel 以注重生命周期的方式存储和管理界面相关的数据。(作用)

> ViewModel 类让数据可在发生屏幕旋转等配置更改后继续留存。(特点)

到这里，你可能还是不清楚ViewModel到底是干啥的，别急，往下看。

1.1 出场背景

在详细介绍ViewModel前，先来看下背景和问题点。

1. Activity可能会在某些场景（例如屏幕旋转）销毁和重新创建界面，那么存储在其中的界面相关数据都会丢失。例如，界面含用户信息列表，因配置更改而重新创建 Activity 后，新 Activity 必须重新请求用户列表，这会造成**资源的浪费**。能否直接恢复之前的数据呢？对于简单的数据，Activity 可以使用 **onSaveInstanceState()** 方法保存 然后从 onCreate() 中的Bundle**恢复数据**，但此方法仅适合可以序列化再反序列化的少量数据（IPC对Bundle有1M的限制），而**不适合数量可能较大的数据**，如用户信息列表或位图。 那么如何做到 因配置更改而新建Activity后的数据恢复呢?

2. UI层（如 Activity 和 Fragment）经常需要通过逻辑层（如MVP中的Presenter）进行异步请求，可能需要一些时间才能返回结果，如果逻辑层持有UI层应用（如context），那么UI层需要管理这些请求，确保界面销毁后清理这些调用以**避免潜在的内存泄露**，但此项管理**需要大量的维护工作**。 那么如何更好的避免因异步请求带来的内存泄漏呢?

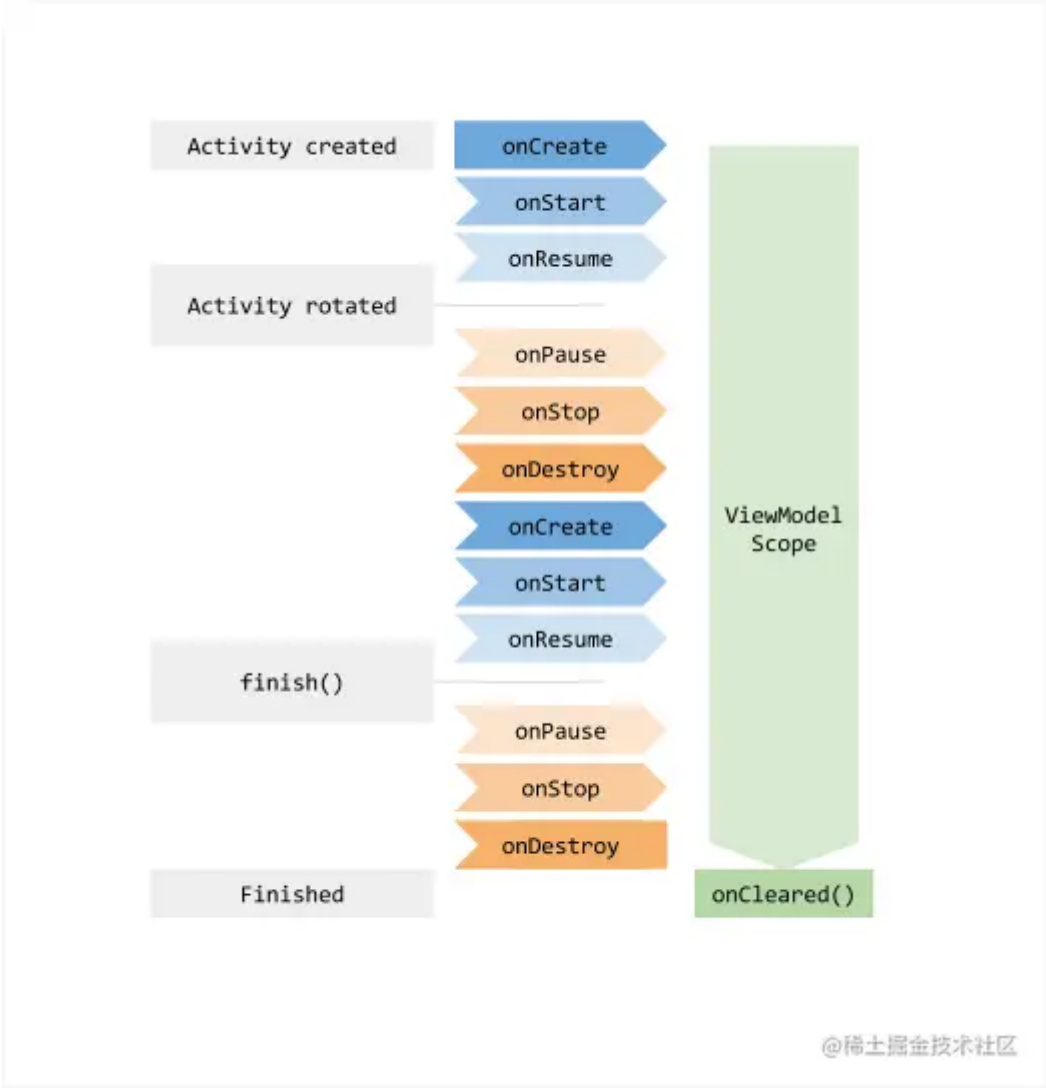
这时候ViewModel就闪亮出场了——**ViewModel用于代替MVP中的Presenter**，为UI层准备数据，用于解决上面两个问题。

1.2 特点

具体地，相比于Presenter，ViewModel有以下特点：

1.2.1 生命周期长于Activity

ViewModel最重要的特点是 **生命周期长于Activity**。来看下官网的一张图：



看到**在因屏幕旋转而重新创建Activity后，ViewModel对象依然会保留**。 只有Activity真正Finish的时ViewModel才会被清除。

也就是说，**因系统配置变更Activity销毁重建，ViewModel对象会保留并关联到新的Activity**。而Activity的正常销毁（系统不会重建Activity）时，ViewModel对象是会清除的。

那么很自然的，因系统配置变更Activity销毁重建，ViewModel内部存储的数据 就可供重新创建的Activity实例使用了。这就解决了第一个问题。

1.2.2 不持有UI层引用

我们知道，在MVP的Presenter中需要持有IView接口来回调结果给界面。

而ViewModel是不需要持有UI层引用的，那结果怎么给到UI层呢？答案就是使用上一篇中介绍的基于观察者模式的LiveData。 并且，ViewModel也不能持有UI层引用，因为ViewModel的生命周期更长。

所以，ViewModel不需要也不能 持有UI层引用，那么就避免了可能的内存泄漏，同时实现了解耦。这就解决了第二个问题。

二、ViewModel使用

2.1 基本使用

了解了ViewModel作用解特点，下面来看看如何结合LivaData使用的。（gradle依赖在第一篇中已经介绍过了。）

步骤：

1. 继承ViewModel自定义MyViewModel
2. 在MyViewModel中编写获取UI数据的逻辑
3. 使用LiveData将获取到的UI数据抛出
4. 在Activity/Fragment中使用ViewModelProvider获取MyViewModel实例
5. 观察MyViewModel中的LiveData数据，进行对应的UI更新。

举个例子，如果您需要在Activity中显示用户信息，那么需要将获取用户信息的操作分放到ViewModel中，代码如下：

java 复制代码

```java
public class UserViewModel extends ViewModel {

    private MutableLiveData<String> userLiveData ;
    private MutableLiveData<Boolean> loadingLiveData;

    public UserViewModel() {
        userLiveData = new MutableLiveData<>();
        loadingLiveData = new MutableLiveData<>();
    }

    //获取用户信息,假装网络请求 2s后 返回用户信息
    public void getUserInfo() {

        loadingLiveData.setValue(true);

        new AsyncTask<Void, Void, String>() {
            @Override
            protected void onPostExecute(String s) {
```

```
                loadingLiveData.setValue(false);
                userLiveData.setValue(s);//抛出用户信息
            }
            @Override
            protected String doInBackground(Void... voids) {
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                String userName = "我是胡飞洋，公众号名字也是胡飞洋，欢迎关注~";
                return userName;
            }
        }.execute();
    }

    public LiveData<String> getUserLiveData() {
        return userLiveData;
    }
    public LiveData<Boolean> getLoadingLiveData() {
        return loadingLiveData;
    }
}
```

UserViewModel继承ViewModel，然后逻辑很简单：假装网络请求 2s后 返回用户信息，其中userLiveData用于抛出用户信息，loadingLiveData用于控制进度条显示。

再看UI层：

java 复制代码

```
public class UserActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        Log.i(TAG, "onCreate: ");

        TextView tvUserName = findViewById(R.id.textView);
        ProgressBar pbLoading = findViewById(R.id.pb_loading);
   //获取ViewModel实例
        ViewModelProvider viewModelProvider = new ViewModelProvider(this);
        UserViewModel userViewModel = viewModelProvider.get(UserViewModel.class);
        //观察 用户信息
        userViewModel.getUserLiveData().observe(this, new Observer<String>() {
            @Override
            public void onChanged(String s) {
```

```
                // update ui.
                tvUserName.setText(s);
            }
        });

        userViewModel.getLoadingLiveData().observe(this, new Observer<Boolean>() {
            @Override
            public void onChanged(Boolean aBoolean) {
                pbLoading.setVisibility(aBoolean?View.VISIBLE:View.GONE);
            }
        });
        //点击按钮获取用户信息
        findViewById(R.id.button).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                userViewModel.getUserInfo();
            }
        });
    }

    @Override
    protected void onStop() {
        super.onStop();
        Log.i(TAG, "onStop: ");
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.i(TAG, "onDestroy: ");
    }
}
```

页面有个按钮用于点击获取用户信息，有个TextView展示用户信息。 在onCreate()中先 **创建ViewModelProvider实例，传入的参数是ViewModelStoreOwner，Activity和Fragment都是其实现。然后通过ViewModelProvider的get方法 获取ViewModel实例**，然后就是 **观察ViewModel中的LiveData**。

运行后，点击按钮 会弹出进度条，2s后展示用户信息。接着旋转手机，我们发现用户信息依然存在。来看下效果：

旋转手机后确实是重建了Activity的，日志打印如下：

```java
1  2021-01-06 20:35:44.984 28269-28269/com.hfy.androidlearning I/UserActivity: onStop:
2  2021-01-06 20:35:44.986 28269-28269/com.hfy.androidlearning I/UserActivity: onDestroy:
3  2021-01-06 20:35:45.025 28269-28269/com.hfy.androidlearning I/UserActivity: onCreate:
```

总结下：

1. ViewModel的使用很简单，作用和原来的Presenter一致。只是要结合LiveData，UI层观察即可。
2. ViewModel的创建必须通过ViewModelProvider。
3. 注意到ViewModel中没有持有任何UI相关的引用。
4. 旋转手机重建Activity后，数据确实恢复了。

2.2 Fragment间数据共享

Activity 中的多个Fragment需要相互通信是一种很常见的情况。假设有一个ListFragment，用户从列表中选择一项，会有另一个DetailFragment显示选定项的详情内容。在之前 你可能会定义接口或者使用EventBus来实现数据的传递共享。

现在就可以使用 ViewModel 来实现。这**两个 Fragment 可以使用其 Activity 范围共享 ViewModel 来处理此类通信**，如以下示例代码所示：

java 复制代码

```java
//ViewModel
public class SharedViewModel extends ViewModel {
//被选中的Item
    private final MutableLiveData<UserContent.UserItem> selected = new MutableLiveData<U

    public void select(UserContent.UserItem user) {
        selected.setValue(user);
    }
    public LiveData<UserContent.UserItem> getSelected() {
        return selected;
    }
}

//ListFragment
public class MyListFragment extends Fragment {
   ...
    private SharedViewModel model;
   ...
    public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        //获取ViewModel，注意ViewModelProvider实例传入的是宿主Activity
        model = new ViewModelProvider(requireActivity()).get(SharedViewModel.class);
        adapter.setListner(new MyItemRecyclerViewAdapter.ItemCLickListner(){
            @Override
            public void onClickItem(UserContent.UserItem userItem) {
                model.select(userItem);
            }
```

```
        });
    }
}

//DetailFragment
public class DetailFragment extends Fragment {

    public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        TextView detail = view.findViewById(R.id.tv_detail);
        //获取ViewModel,观察被选中的Item
        SharedViewModel model = new ViewModelProvider(requireActivity()).get(SharedViewM
        model.getSelected().observe(getViewLifecycleOwner(), new Observer<UserContent.Us
            @Override
            public void onChanged(UserContent.UserItem userItem) {
                //展示详情
                detail.setText(userItem.toString());
            }
        });
    }
}
```

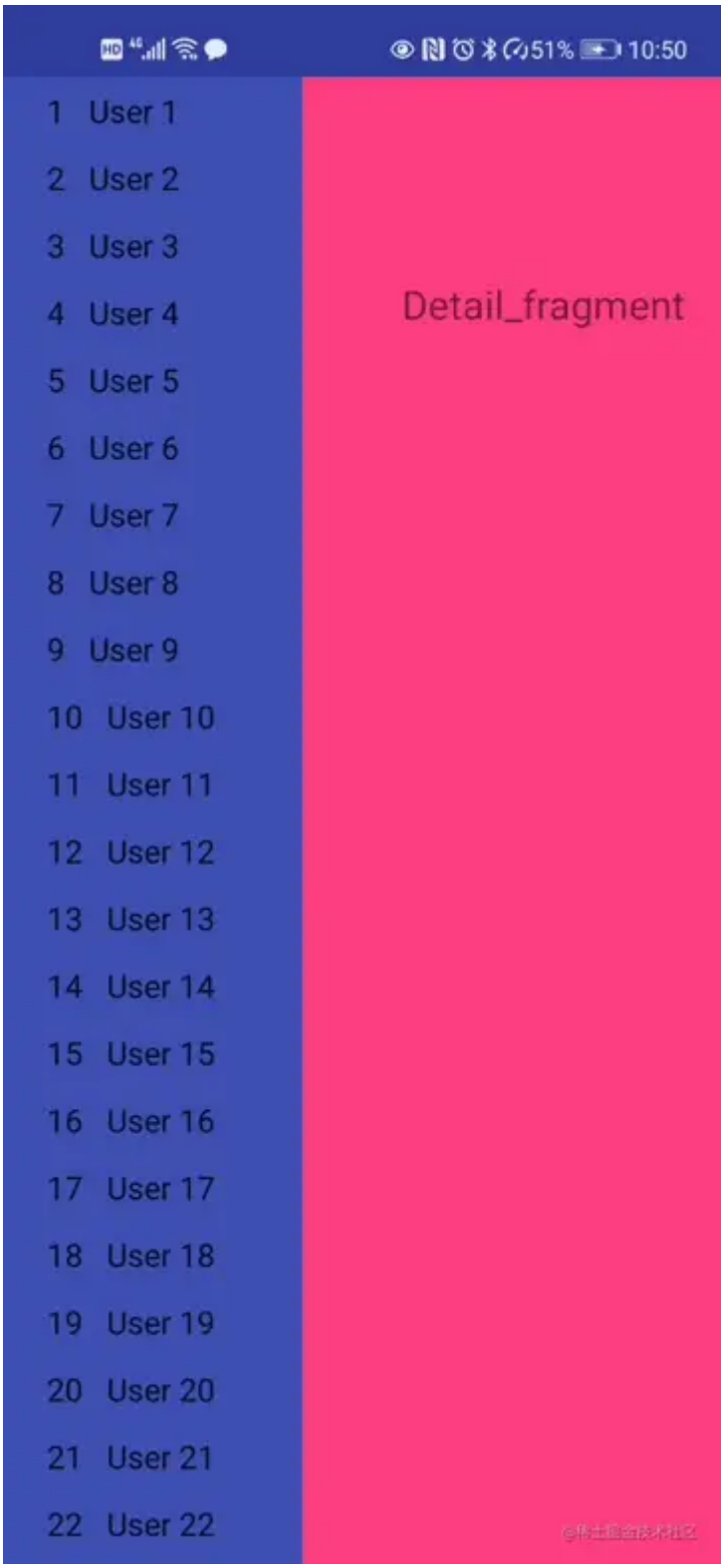
代码很简单，ListFragment中在点击Item时更新ViewModel的LiveData数据，然后DetailFragment监听这个LiveData数据即可。

要注意的是，这两个 Fragment 通过ViewModelProvider获取ViewModel时 传入的都是它们宿主 Activity。这样，当这两个 Fragment 各自获取 ViewModelProvider 时，它们会收到相同的 SharedViewModel 实例（其范围限定为该 Activity）。

此方法具有以下 **优势**：

1. Activity 不需要执行任何操作，也不需要对此通信有任何了解。
2. 除了 SharedViewModel 约定之外，Fragment 不需要相互了解。如果其中一个 Fragment 消失，另一个 Fragment 将继续照常工作。
3. 每个 Fragment 都有自己的生命周期，而不受另一个 Fragment 的生命周期的影响。如果一个 Fragment 替换另一个 Fragment，界面将继续工作而没有任何问题。

最后来看下效果：



三、源码分析

经过前面的介绍，我们知道ViewModel的核心点 就是 因配置更新而界面（Activity/Fragment）重建后，ViewModel实例依然存在，这个如何实现的呢？ 这就是我们源码分析的重点了。

在获取ViewModel实例时，我们并不是直接new的，而是使用ViewModelProvider来获取，猜测关键点应该就在这里了。

3.1 ViewModel的存储和获取

先来看下ViewModel类：

java 复制代码

```java
public abstract class ViewModel {
    ...
    private volatile boolean mCleared = false;
    //在ViewModel将被清除时调用
    //当ViewModel观察了一些数据，可以在这里做解注册 防止内存泄漏
    @SuppressWarnings("WeakerAccess")
    protected void onCleared() {
    }
    @MainThread
    final void clear() {
        mCleared = true;
        ...
        onCleared();
    }
...
}
```

ViewModel类 是抽象类，内部没有啥逻辑，有个clear()方法会在ViewModel将被清除时调用。

然后ViewModel实例的获取是通过ViewModelProvider类，见名知意，即ViewModel提供者，来看下它的构造方法：

java 复制代码

```java
    public ViewModelProvider(@NonNull ViewModelStoreOwner owner) {
        this(owner.getViewModelStore(), owner instanceof HasDefaultViewModelProviderFact
                ? ((HasDefaultViewModelProviderFactory) owner).getDefaultViewModelProvid
                : NewInstanceFactory.getInstance());
    }

    public ViewModelProvider(@NonNull ViewModelStoreOwner owner, @NonNull Factory factor
        this(owner.getViewModelStore(), factory);
    }

    public ViewModelProvider(@NonNull ViewModelStore store, @NonNull Factory factory) {
        mFactory = factory;
        mViewModelStore = store;
    }
```

例子中我们使用的是只需传ViewModelStoreOwner的构造方法，最后走到两个参数ViewModelStore、factory的构造方法。继续见名知意：ViewModelStoreOwner——ViewModel存储器拥有者；ViewModelStore——ViewModel存储器，用来存ViewModel的地方；Factory——创建ViewModel实例的工厂。

ViewModelStoreOwner是个接口：

java 复制代码

```java
public interface ViewModelStoreOwner {
   //获取ViewModelStore，即获取ViewModel存储器
    ViewModelStore getViewModelStore();
}
```

实现类有Activity/Fragment，也就是说 Activity/Fragment 都是 ViewModel存储器的拥有者，具体是怎样实现 获取ViewModelStore的呢?

先不急，我们先看 ViewModelStore 如何存储ViewModel、以及ViewModel实例如何获取的。

java 复制代码

```java
/**
 * 用于存储ViewModels.
 * ViewModelStore实例 必须要能 在系统配置改变后 依然存在。
 */
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);
    }
    Set<String> keys() {
        return new HashSet<>(mMap.keySet());
    }
    /**
     * 调用ViewModel的clear()方法，然后清除ViewModel
     * 如果ViewModelStore的拥有者（Activity/Fragment）销毁后不会重建，那么就需要调用此方法
     */
```

```
    public final void clear() {
        for (ViewModel vm : mMap.values()) {
            vm.clear();
        }
        mMap.clear();
    }
}
```

ViewModelStore代码很简单，viewModel作为Value存储在HashMap中。

再来看下创建ViewModel实例的工厂Factory，也就是NewInstanceFactory：

java 复制代码

```
    public static class NewInstanceFactory implements Factory {
   ...
        @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,
            } catch (IllegalAccessException e) {
                throw new RuntimeException("Cannot create an instance of " + modelClass,
            }
        }
    }
```

很简单，就是通过传入的class 反射获取ViewModel实例。

回到例子中，我们使用 `viewModelProvider.get(UserViewModel.class)` 来获取UserViewModel实例，那么来看下get()方法：

java 复制代码

```
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 V

   //拿到Key，也即是ViewModelStore中的Map的用于存 ViewModel的 Key
        return get(DEFAULT_KEY + ":" + canonicalName, modelClass);
    }

public <T extends ViewModel> T get(@NonNull String key, @NonNull Class<T> modelClass) {
```

```
    //从ViewModelStore获取ViewModel实例
        ViewModel viewModel = mViewModelStore.get(key);

        if (modelClass.isInstance(viewModel)) {
            if (mFactory instanceof OnRequeryFactory) {
                ((OnRequeryFactory) mFactory).onRequery(viewModel);
            }
            //如果从ViewModelStore获取到，直接返回
            return (T) viewModel;
        }

        if (mFactory instanceof KeyedFactory) {
            viewModel = ((KeyedFactory) (mFactory)).create(key, modelClass);
        } else {
        //没有获取到，就使用Factory创建
            viewModel = (mFactory).create(modelClass);
        }
        //存入ViewModelStore 然后返回
        mViewModelStore.put(key, viewModel);
        return (T) viewModel;
    }
```

逻辑很清晰，先尝试从ViewModelStore获取ViewModel实例，key是"androidx.lifecycle.ViewModelProvider.DefaultKey:xxx.SharedViewModel"，如果没有获取到，就使用Factory创建，然后存入ViewModelStore。

到这里，我们知道了 ViewModel如何存储、实例如何获取的，但开头说的分析重点：“因配置更新而界面重建后，ViewModel实例依然存在”，这个还没分析到。

3.2 ViewModelStore的存储和获取

回到上面的疑问，看看 Activity/Fragment 是怎样实现 获取ViewModelStore的，先来看ComponentActivity中对ViewModelStoreOwner的实现：

java 复制代码

```
//ComponentActivity.java
    public ViewModelStore getViewModelStore() {
        if (getApplication() == null) {
        //activity还没关联Application，即不能在onCreate之前去获取viewModel
            throw new IllegalStateException("Your activity is not yet attached to the "
                    + "Application instance. You can't request ViewModel before onCreate
        }
        if (mViewModelStore == null) {
```

```
        //如果存储器是空，就先尝试 从lastNonConfigurationInstance从获取
            NonConfigurationInstances nc =
                    (NonConfigurationInstances) getLastNonConfigurationInstance();
            if (nc != null) {
                mViewModelStore = nc.viewModelStore;
            }
            if (mViewModelStore == null) {
            //如果lastNonConfigurationInstance不存在，就new一个
                mViewModelStore = new ViewModelStore();
            }
        }
        return mViewModelStore;
    }
```

这里就是重点了。先尝试 从NonConfigurationInstance从获取 ViewModelStore实例，如果NonConfigurationInstance不存在，就new一个mViewModelStore。 并且还注意到，在onRetainNonConfigurationInstance()方法中 会把mViewModelStore赋值给NonConfigurationInstances：

java 复制代码

```
    //在Activity因配置改变 而正要销毁时，且新Activity会立即创建，那么系统就会调用此方法
    public final Object onRetainNonConfigurationInstance() {
        Object custom = onRetainCustomNonConfigurationInstance();

        ViewModelStore viewModelStore = mViewModelStore;
        ...
        if (viewModelStore == null && custom == null) {
            return null;
        }

   //new了一个NonConfigurationInstances，mViewModelStore赋值过来
        NonConfigurationInstances nci = new NonConfigurationInstances();
        nci.custom = custom;
        nci.viewModelStore = viewModelStore;
        return nci;
    }
```

onRetainNonConfigurationInstance()方法很重要：在Activity因配置改变 而正要销毁时，且新Activity会立即创建，那么系统就会调用此方法。 也就说，配置改变时 系统把viewModelStore存在了NonConfigurationInstances中。

NonConfigurationInstances是个啥呢?

java 复制代码

```java
//ComponentActivity
    static final class NonConfigurationInstances {
        Object custom;
        ViewModelStore viewModelStore;
    }
```

ComponentActivity静态内部类，依然见名知意，**非配置实例**，即 与系统配置 无关的 实例。所以屏幕旋转等的配置改变 不会影响到这个实例？ 继续看这个猜想是否正确。

我们看下getLastNonConfigurationInstance()：

java 复制代码

```java
//Acticity.java

NonConfigurationInstances mLastNonConfigurationInstances;

//返回onRetainNonConfigurationInstance()返回的实例
public Object getLastNonConfigurationInstance() {
    return mLastNonConfigurationInstances != null ? mLastNonConfigurationInstances.activ
}

static final class NonConfigurationInstances {
        Object activity;
        HashMap<String, Object> children;
        FragmentManagerNonConfig fragments;
        ArrayMap<String, LoaderManager> loaders;
        VoiceInteractor voiceInteractor;
    }
```

方法是在Acticity.java中，它返回的是Acticity.java中的NonConfigurationInstances的属性activity，也就是onRetainNonConfigurationInstance()方法返回的实例。（注意上面那个是ComponentActivity中的NonConfigurationInstances，是两个类）

来继续看mLastNonConfigurationInstances是哪来的，通过寻找调用找到在attach()方法中：

java 复制代码

```java
final void attach(Context context, ActivityThread aThread, ...
            NonConfigurationInstances lastNonConfigurationInstances,... ) {
            ...
            mLastNonConfigurationInstances = lastNonConfigurationInstances;
            ...
        }
```

mLastNonConfigurationInstances是在Activity的attach方法中赋值。 在《Activity的启动过程详解》中我们分析过，attach方法是为Activity关联上下文环境，是在Activity 启动的核心流程——ActivityThread的performLaunchActivity方法中调用，这里的lastNonConfigurationInstances是存在 ActivityClientRecord中的一个组件信息。

ActivityClientRecord是存在ActivityThread的mActivities中：

java 复制代码

```java
//ActivityThrtead.java
final ArrayMap<IBinder, ActivityClientRecord> mActivities = new ArrayMap<>();
```

那么，**ActivityThread 中的 ActivityClientRecord 是不受 activity 重建的影响，那么ActivityClientRecord中lastNonConfigurationInstances也不受影响，那么其中的Object activity也不受影响，那么ComponentActivity中的NonConfigurationInstances的viewModelStore不受影响，那么viewModel也就不受影响了。**

那么，到这里 核心问题 “配置更改重建后ViewModel依然存在” 的原理就分析完了。

四、对比onSaveInstanceState()

系统提供了onSaveInstanceState()用于让开发者保存一些数据，以方便界面销毁重建时恢复数据。那么和 使用ViewModel恢复数据 有哪些区别呢?

4.1 使用场景

在我很久之前一篇文章《Activity生命周期》中有提到：

> onSaveInstanceState调用时机：
>
> 当某个activity变得“容易”被系统销毁时，该activity的onSaveInstanceState就会被执行，除非该activity是被用户主动销毁的，例如当用户按BACK键的时候。 注意上面的双引号，何为“容易”？言下之意就是该activity还没有被销毁，而仅仅是一种可能性。
>
> 这种可能性有哪些？有这么几种情况：
>
> 1、当用户按下HOME键时。 这是显而易见的，系统不知道你按下HOME后要运行多少其他的程序，自然也不知道activity A是否会被销毁，故系统会调用onSaveInstanceState，让用户有机会保存某些非永久性的数据。以下几种情况的分析都遵循该原则 。

> 2、长按HOME键，选择运行其他的程序时。
>
> 3、按下电源按键（关闭屏幕显示）时。
>
> 4、从activity A中启动一个新的activity时。
>
> 5、**屏幕方向切换时**，例如从竖屏切换到横屏时。 在屏幕切换之前，系统会销毁activity A，在屏幕切换之后系统又会自动地创建activity A，所以onSaveInstanceState一定会被执行。

> 总而言之，onSaveInstanceState的调用遵循一个重要原则，即当系统“未经你许可”时销毁了你的activity，则onSaveInstanceState会被系统调用，这是系统的责任，因为它必须要提供一个机会让你保存你的数据（当然你不保存那就随便你了）。

而使用ViewModel恢复数据 则 只有在 因配置更改界面销毁重建 的情况。

4.2 存储方式

ViewModel是存在内存中，读写速度快，而通过onSaveInstanceState是在 序列化到磁盘中。

4.3 存储数据的限制

ViewModel，可以存复杂数据，大小限制就是App的可用内存。而 onSaveInstanceState只能存可序列化和反序列化的对象，且大小有限制（一般Bundle限制大小1M）。

五、总结

本文先介绍了ViewModel的概念——为界面准备数据的模型，然后它的特点：因配置更改界面销毁重建后依然存在、不持有UI应用；接着介绍了 使用方式、Fragment数据共享。最后详细分析了ViewModel源码及核心原理。

并且可以看到LiveData和ViewModel搭配使用，可以代替MVP中的Presenter解决很多问题。ViewModel是我们后续建立MVVM架构的重要组件。 这也是我们必须掌握和理解的部分。

下一篇将介绍基于LifeCycle、LiveData、ViewModel的MVVM架构，终于要到MVVM了，敬请关注。

今天就到这里啦~

.

感谢与参考：

ViewModel官方文档

.

> **你的 点赞、评论，是对我的巨大鼓励！**

> 欢迎关注我的 **公 众 号**，微信搜索 **胡飞洋** ， 文章更新可第一时间收到。

标签： Android Jetpack

文章被收录于专栏：

JetPack 架构组件系列

JetPack 架构组件 全面解析~

订阅专栏