**一、ViewModel介绍**

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)

## 1.1 出场背景

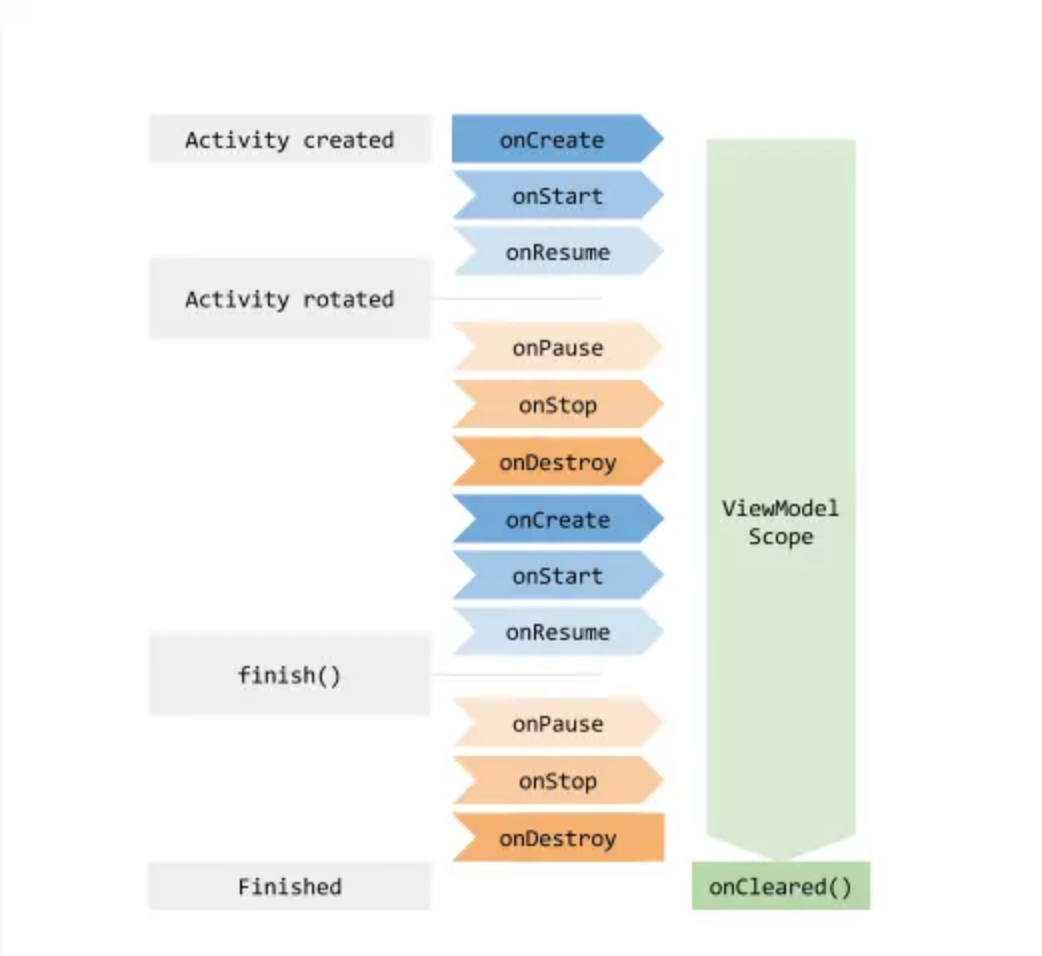
1. Activity可能会在某些场景（例如屏幕旋转）销毁和重新创建界面，那么存储在其中的界面相关数据都会丢失。例如，界面含用户信息列表，因配置更改而重新创建 Activity 后，新 Activity 必须重新请求用户列表，这会造成**资源的浪费**。能否直接恢复之前的数据呢？对于简单的数据，Activity 可以使用 **onSaveInstanceState()** 方法保存 然后从 onCreate() 中的Bundle**恢复数据**，但此方法仅适合可以序列化再反序列化的少量数据（IPC对Bundle有1M的限制），而**不适合数量可能较大的数据**，如用户信息列表或位图。 那么如何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

### 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、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。

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。

### **ViewModelProvider的构造方法**

public ViewModelProvider(@NonNull ViewModelStoreOwner owner) {

this(owner.getViewModelStore(), owner instanceof HasDefaultViewModelProviderFactory

? ((HasDefaultViewModelProviderFactory) owner).getDefaultViewModelProviderFactory()

: NewInstanceFactory.getInstance());

}

ViewModelProvider构造方法接受一个ViewModelStoreOwner类型的参数。ViewModelStoreOwner是一个接口，内部只有一个getViewModelStore方法。接下来在构造方法中又调用了重载的构造方法，并且第一个调用了ViewModelStoreOwner的getViewModelStore方法来获取ViewModelStore，第二个参数通过ower判断类型来实例化ViewModelProviderFacory

在中ComponentActivity实现ViewModelStoreOwner接口和HasDefaultViewModelProviderFactory，ComponentActivity重写了getViewModelStore和getDefaultViewModelProviderFactory方法。

public class ComponentActivity extends androidx.core.app.ComponentActivity implements

ViewModelStoreOwner,

HasDefaultViewModelProviderFactory{

@NonNull

@Override

public ViewModelStore getViewModelStore() {

if (getApplication() == null) {

throw new IllegalStateException("Your activity is not yet attached to the "

+ "Application instance. You can't request ViewModel before onCreate call.");

}

// 确保ViewModelStore不为空

ensureViewModelStore();

return mViewModelStore;

}

@NonNull

@Override

// 创建了一个默认的ViewModelFactory

public ViewModelProvider.Factory getDefaultViewModelProviderFactory() {

if (getApplication() == null) {

throw new IllegalStateException("Your activity is not yet attached to the "

+ "Application instance. You can't request ViewModel before onCreate call.");

}

if (mDefaultFactory == null) {

mDefaultFactory = new SavedStateViewModelFactory(

getApplication(),

this,

getIntent() != null ? getIntent().getExtras() : null);

}

return mDefaultFactory;

}

}

ensureViewModelStore方法中如果ViewModelStore为null，则会尝试恢复或者实例化ViewModelStore，代码如下：

void ensureViewModelStore() {

if (mViewModelStore == null) {

// 获取上次配置数据实例

NonConfigurationInstances nc =

(NonConfigurationInstances) getLastNonConfigurationInstance();

if (nc != null) {

// 从NonConfigurationInstances中恢复ViewModelStore

mViewModelStore = nc.viewModelStore;

}

// NonConfigurationInstances中没有保存那么就实例化ViewModelStore

if (mViewModelStore == null) {

mViewModelStore = new ViewModelStore();

}

}

}

ViewModelProvider的构造方法最终调用了下边的这个重载的构造方法：

public ViewModelProvider(@NonNull ViewModelStore store, @NonNull Factory factory) {

mFactory = factory;

mViewModelStore = store;

}

在创建ViewModel的时候，Activity中会首先尝试恢复或者创建一个ViewModelStore。并将其保存到ViewModelProvider中待用。

ViewModelProvider的get方法

实例化了ViewModelProvider后会调用ViewModelProvider的get方法。代码如下：

public <T extends ViewModel> T get(@NonNull Class<T> modelClass) {

String canonicalName = modelClass.getCanonicalName();

// ... 省略校验

return get(DEFAULT\_KEY + ":" + canonicalName, modelClass);

}

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

ViewModel viewModel = mViewModelStore.get(key);

if (modelClass.isInstance(viewModel)) {

if (mFactory instanceof OnRequeryFactory) {

((OnRequeryFactory) mFactory).onRequery(viewModel);

}

return (T) viewModel;

} else {

//noinspection StatementWithEmptyBody

if (viewModel != null) {

// TODO: log a warning.

}

}

if (mFactory instanceof KeyedFactory) {

viewModel = ((KeyedFactory) mFactory).create(key, modelClass);

} else {

viewModel = mFactory.create(modelClass);

}

mViewModelStore.put(key, viewModel);

return (T) viewModel;

}

get方法中首先调用了mViewModelStore的get方法，并通过key来获取ViewModel。可见ViewModel本身应该是一个K-V的集合类。暂且不管ViewModelStore。接下来，如果modelClass已经实例化了，就直接返回ViewModel，否则就调用Factory来创建ViewModel，并将ViewModel存入ViewModelStore中。很显然，ViewModel的实例化时在Factory中进行的。ViewModelProvider的构造方法是允许我们自己传入一个工厂的，如果不传也会创建默认的工厂。本章的第一小节已经看到会默认实例化一个SavedStateViewModelFactory，这个工厂继承了KeyedFactory，我们来看下它的create是如何实现的：

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

// 判断是不是AndroidViewModel类型

boolean isAndroidViewModel = AndroidViewModel.class.isAssignableFrom(modelClass);

Constructor<T> constructor;

// 获取ViewModel的构造方法

if (isAndroidViewModel && mApplication != null) {

// 查找ANDROID\_VIEWMODEL\_SIGNATURE对应的构造方法

constructor = findMatchingConstructor(modelClass, ANDROID\_VIEWMODEL\_SIGNATURE);

} else {

// 查找VIEWMODEL\_SIGNATURE的构造方法

constructor = findMatchingConstructor(modelClass, VIEWMODEL\_SIGNATURE);

}

// doesn't need SavedStateHandle

if (constructor == null) {

// 通过Factory去创建ViewModel

return mFactory.create(modelClass);

}

SavedStateHandleController controller = SavedStateHandleController.create(

mSavedStateRegistry, mLifecycle, key, mDefaultArgs);

try {

T viewmodel;

// 通过构造方法反射实例化ViewModel

if (isAndroidViewModel && mApplication != null) {

viewmodel = constructor.newInstance(mApplication, controller.getHandle());

} else {

viewmodel = constructor.newInstance(controller.getHandle());

}

viewmodel.setTagIfAbsent(TAG\_SAVED\_STATE\_HANDLE\_CONTROLLER, controller);

return viewmodel;

} catch (IllegalAccessException e) {

throw new RuntimeException("Failed to access " + modelClass, e);

} catch (InstantiationException e) {

throw new RuntimeException("A " + modelClass + " cannot be instantiated.", e);

} catch (InvocationTargetException e) {

throw new RuntimeException("An exception happened in constructor of "

+ modelClass, e.getCause());

}

}

这里的mFactory是在SavedStateViewModelFactory实例化时创建的，可能是AndroidViewModelFactory类型或者NewInstanceFactory类型。

public SavedStateViewModelFactory(@Nullable Application application,

@NonNull SavedStateRegistryOwner owner,

@Nullable Bundle defaultArgs) {

// ...

mFactory = application != null

? ViewModelProvider.AndroidViewModelFactory.getInstance(application)

: ViewModelProvider.NewInstanceFactory.getInstance();

}

以AndroidViewModelFactory为例来看create

public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {

if (AndroidViewModel.class.isAssignableFrom(modelClass)) {

//noinspection TryWithIdenticalCatches

try {

return modelClass.getConstructor(Application.class).newInstance(mApplication);

} catch (NoSuchMethodException e) {

throw new RuntimeException("Cannot create an instance of " + modelClass, e);

} catch (IllegalAccessException e) {

throw new RuntimeException("Cannot create an instance of " + modelClass, e);

} catch (InstantiationException e) {

throw new RuntimeException("Cannot create an instance of " + modelClass, e);

} catch (InvocationTargetException e) {

throw new RuntimeException("Cannot create an instance of " + modelClass, e);

}

}

return super.create(modelClass);

}

其实就是通过构造方法反射创建了ViewModel而已。

总体来看ViewModelProvider的任务就是创建ViewModel并将ViewModel存储到ViewModelStore中。而ViewModel的创建都是通过Model.class的构造方法反射创建的。

上一节猜测ViewModelStore是一个存储K-V结构的数据类型，那么来看下它的源码。

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());

}

/\*\*

\* Clears internal storage and notifies ViewModels that they are no longer used.

\*/

public final void clear() {

for (ViewModel vm : mMap.values()) {

vm.clear();

}

mMap.clear();

}

}

原来ViewModelStore内部就是封装了一下HashMap，实例化的ViewModel都被存储到了HashMap中。另外注意到ViewModelStore中还有一个clear方法，用来清理ViewMode和Map集合。在哪里对ViewModelStore进行清楚呢？这个还需要到Activity中找答案了，注意到在ComponentActivity的构造方法中有如下代码：

public ComponentActivity() {

Lifecycle lifecycle = getLifecycle();

if (Build.VERSION.SDK\_INT >= 19) {

getLifecycle().addObserver(new LifecycleEventObserver() {

@Override

public void onStateChanged(@NonNull LifecycleOwner source,

@NonNull Lifecycle.Event event) {

if (event == Lifecycle.Event.ON\_STOP) {

Window window = getWindow();

final View decor = window != null ? window.peekDecorView() : null;

if (decor != null) {

decor.cancelPendingInputEvents();

}

}

}

});

}

getLifecycle().addObserver(new LifecycleEventObserver() {

@Override

public void onStateChanged(@NonNull LifecycleOwner source,

@NonNull Lifecycle.Event event) {

// Activity的销毁事件

if (event == Lifecycle.Event.ON\_DESTROY) {

// Clear out the available context

mContextAwareHelper.clearAvailableContext();

// And clear the ViewModelStore

if (!isChangingConfigurations()) {

getViewModelStore().clear();

}

}

}

});

getLifecycle().addObserver(new LifecycleEventObserver() {

@Override

public void onStateChanged(@NonNull LifecycleOwner source,

@NonNull Lifecycle.Event event) {

ensureViewModelStore();

getLifecycle().removeObserver(this);

}

});

}

全局搜素发现只有在ComponentActivity构造方法中有一处调用了ViewModelStore的clear方法，很显然，这里是通过LifeCycle监听Activity的销毁事件，然后才对ViewModel进行了清理。

4.ViewModelStore的状态保存

我们知道，屏幕旋转时候Activity会经历销毁和重建的过程，那么ViewModelStore是什么时候被保存的呢？其实是在ComponentActivity的onRetainNonConfigurationInstance方法中，代码如下：

// ComponentActivity

public final Object onRetainNonConfigurationInstance() {

// Maintain backward compatibility.

Object custom = onRetainCustomNonConfigurationInstance();

ViewModelStore viewModelStore = mViewModelStore;

if (viewModelStore == null) {

// No one called getViewModelStore(), so see if there was an existing

// ViewModelStore from our last NonConfigurationInstance

NonConfigurationInstances nc =

(NonConfigurationInstances) getLastNonConfigurationInstance();

if (nc != null) {

viewModelStore = nc.viewModelStore;

}

}

if (viewModelStore == null && custom == null) {

return null;

}

NonConfigurationInstances nci = new NonConfigurationInstances();

nci.custom = custom;

nci.viewModelStore = viewModelStore;

return nci;

}

ViewModelStore的恢复则是在getLastNonConfigurationInstance中，代码如下：

// Activity

public Object getLastNonConfigurationInstance() {

return mLastNonConfigurationInstances != null

? mLastNonConfigurationInstances.activity : null;

}

getLastNonConfigurationInstance的调用其实前边代码已经有提到，就是在ensureViewModelStore中。ensureViewModelStore除了在getViewModelProvider中调用外，还会在生命周期的回调中被调用，代码在上边也有贴出。

当屏幕旋转时，会调用Activity的onRetainNonConfigurationInstance方法。ViewModel组件正是通过该方法将ViewModel保存起来，给重建的Activity使用。

//androidx.activity:activity:1.2.2@aar

//ComponentActivity.java

public final Object onRetainNonConfigurationInstance() {

// Maintain backward compatibility.

Object custom = onRetainCustomNonConfigurationInstance();

ViewModelStore viewModelStore = mViewModelStore;

if (viewModelStore == null) {

// No one called getViewModelStore(), so see if there was an existing

// ViewModelStore from our last NonConfigurationInstance

NonConfigurationInstances nc =

(NonConfigurationInstances) getLastNonConfigurationInstance();

if (nc != null) {

viewModelStore = nc.viewModelStore;

}

}

if (viewModelStore == null && custom == null) {

return null;

}

NonConfigurationInstances nci = new NonConfigurationInstances();

nci.custom = custom;

nci.viewModelStore = viewModelStore;

return nci;

}

Activity还有个类似的方法onSaveInstanceState() ，onSaveInstanceState() 和onRetainNonConfigurationInstance() 的区别是：

onSaveInstanceState() 调用的场景是：activity1启动activity2。生命周期调用顺序如下：

activity1.onPause()->activity2.onCreate()->activity2.onStart()->activity2.onResume()->activity1.onStop()->activity1.onSaveInstanceState()，

onRetainNonConfigurationInstance() 调用场景是当configuration发生改变时，例如：旋转屏幕。

那么问题来了，一共有三个

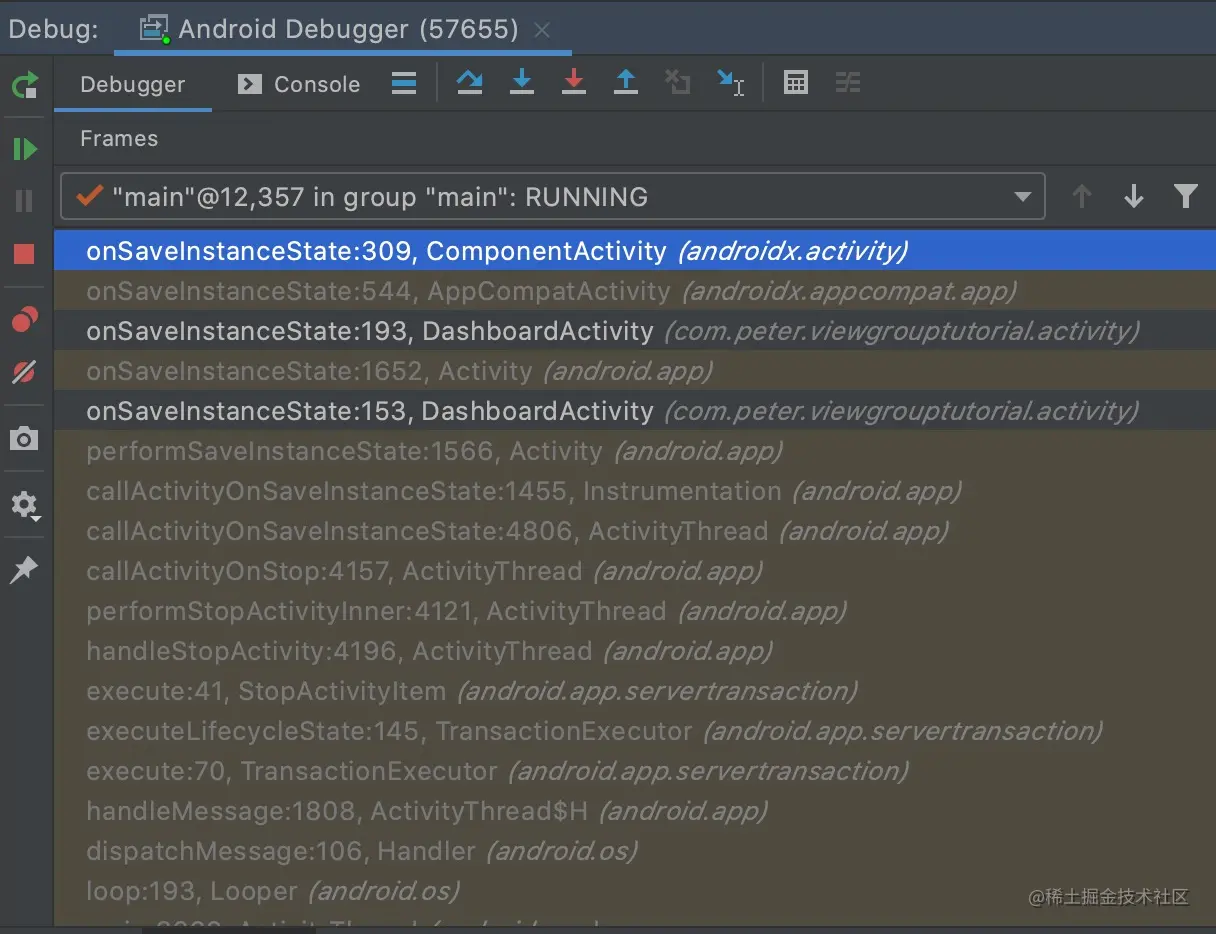
它们存储的状态数据颗粒度一样吗？

它们把状态数据存储到哪里去了？

如果系统后台将Activity杀掉后，它们都能把状态恢复回来吗？

比如：ActivityManagerService的activityStopped方法的远程代理调用找不到，在群友们的帮助下，最终顺利找到，交流的过程中还是有不少收获。

首先在ComponentActivity的onSaveInstanceState(Bundle outState) 方法中加个断点。调用栈如下：



重点关注ActivityThread.callActivityOnSaveInstanceState(Bundle outState)

//ActivityThread.java

private void callActivityOnSaveInstanceState(ActivityClientRecord r) {

r.state = new Bundle();

r.state.setAllowFds(false);

if (r.isPersistable()) {

r.persistentState = new PersistableBundle();

mInstrumentation.callActivityOnSaveInstanceState(r.activity, r.state,

r.persistentState);

} else {

mInstrumentation.callActivityOnSaveInstanceState(r.activity, r.state);

}

}

我们注意到r.state = new Bundle()， 原来outState参数是在这里创建的。Bundle可以用来组件间传递数据，也可以用来进程间传递数据。

重点关注ActivityThread.handleStopActivity()

public void handleStopActivity(IBinder token, boolean show, int configChanges,

PendingTransactionActions pendingActions, boolean finalStateRequest, String reason) {

final ActivityClientRecord r = mActivities.get(token);

r.activity.mConfigChangeFlags |= configChanges;

final StopInfo stopInfo = new StopInfo();

performStopActivityInner(r, stopInfo, show, true /\* saveState \*/, finalStateRequest, reason);

if (localLOGV) Slog.v(

TAG, "Finishing stop of " + r + ": show=" + show

+ " win=" + r.window);

updateVisibility(r, show);

// Make sure any pending writes are now committed.

if (!r.isPreHoneycomb()) {

QueuedWork.waitToFinish();

}

stopInfo.setActivity(r);

stopInfo.setState(r.state);

stopInfo.setPersistentState(r.persistentState);

pendingActions.setStopInfo(stopInfo);

mSomeActivitiesChanged = true;

}

注意到pendingActions.setStopInfo(stopInfo)

@Override

public void run() {

// Tell activity manager we have been stopped.

try {

if (DEBUG\_MEMORY\_TRIM) Slog.v(TAG, "Reporting activity stopped: " + mActivity);

// TODO(lifecycler): Use interface callback instead of AMS.

ActivityManager.getService().activityStopped(

mActivity.token, mState, mPersistentState, mDescription);

} catch (RemoteException ex) {

// Dump statistics about bundle to help developers debug

final LogWriter writer = new LogWriter(Log.WARN, TAG);

final IndentingPrintWriter pw = new IndentingPrintWriter(writer, " ");

pw.println("Bundle stats:");

Bundle.dumpStats(pw, mState);

pw.println("PersistableBundle stats:");

Bundle.dumpStats(pw, mPersistentState);

if (ex instanceof TransactionTooLargeException

&& mActivity.packageInfo.getTargetSdkVersion() < Build.VERSION\_CODES.N) {

Log.e(TAG, "App sent too much data in instance state, so it was ignored", ex);

return;

}

throw ex.rethrowFromSystemServer();

}

}

该方法调用了ActivityManager.getService().activityStopped(mActivity.token, mState, mPersistentState, mDescription)方法。还将2.1中创建的mState当参数传进来了。

//ActivityManagerService.java

@Override

public final void activityStopped(IBinder token, Bundle icicle,

PersistableBundle persistentState, CharSequence description) {

if (DEBUG\_ALL) Slog.v(TAG, "Activity stopped: token=" + token);

// Refuse possible leaked file descriptors

if (icicle != null && icicle.hasFileDescriptors()) {

throw new IllegalArgumentException("File descriptors passed in Bundle");

}

final long origId = Binder.clearCallingIdentity();

synchronized (this) {

final ActivityRecord r = ActivityRecord.isInStackLocked(token);

if (r != null) {

r.activityStoppedLocked(icicle, persistentState, description);

}

}

trimApplications();

Binder.restoreCallingIdentity(origId);

}

重点关注r.activityStoppedLocked(icicle, persistentState, description)

final void activityStoppedLocked(Bundle newIcicle, PersistableBundle newPersistentState, CharSequence description) {

final ActivityStack stack = getStack();

if (mState != STOPPING) {

Slog.i(TAG, "Activity reported stop, but no longer stopping: " + this);

stack.mHandler.removeMessages(STOP\_TIMEOUT\_MSG, this);

return;

}

if (newPersistentState != null) {

persistentState = newPersistentState;

service.notifyTaskPersisterLocked(task, false);

}

if (DEBUG\_SAVED\_STATE) Slog.i(TAG\_SAVED\_STATE, "Saving icicle of " + this + ": " + icicle);

if (newIcicle != null) {

icicle = newIcicle;

haveState = true;

launchCount = 0;

updateTaskDescription(description);

}

if (!stopped) {

if (DEBUG\_STATES) Slog.v(TAG\_STATES, "Moving to STOPPED: " + this + " (stop complete)");

stack.mHandler.removeMessages(STOP\_TIMEOUT\_MSG, this);

stopped = true;

setState(STOPPED, "activityStoppedLocked");

mWindowContainerController.notifyAppStopped();

if (finishing) {

clearOptionsLocked();

} else {

if (deferRelaunchUntilPaused) {

stack.destroyActivityLocked(this, true /\* removeFromApp \*/, "stop-config");

mStackSupervisor.resumeFocusedStackTopActivityLocked();

} else {

mStackSupervisor.updatePreviousProcessLocked(this);

}

}

}

}

我们注意到最终bundle数据会保存在ActivityRecord的icicle对象中。

总结：onSaveInstanceState方法是当Activity调用了onStop后，会调用到ActivityThread的callActivityOnSaveInstanceState()方法，把Activity需要保存的数据放入Bundle对象中，并且随后通过IPC进程间通信机制，调用ActivityManagerService的activityStopped方法，将Bundle对象保存到AMS端的ActivityRecord中。

被杀端后恢复数据过程

//ActivityStackSupervisor.java

final boolean realStartActivityLocked(ActivityRecord r, ProcessRecord app,

boolean andResume, boolean checkConfig) throws RemoteException {

// 忽略其它代码

// Create activity launch transaction.

final ClientTransaction clientTransaction = ClientTransaction.obtain(app.thread, r.appToken);

clientTransaction.addCallback(LaunchActivityItem.obtain(new Intent(r.intent),

System.identityHashCode(r), r.info,

// TODO: Have this take the merged configuration instead of separate global

// and override configs.

mergedConfiguration.getGlobalConfiguration(),

mergedConfiguration.getOverrideConfiguration(), r.compat,

r.launchedFromPackage, task.voiceInteractor, app.repProcState, r.icicle,

r.persistentState, results, newIntents, mService.isNextTransitionForward(), profilerInfo));

// 忽略其它代码

}

我们看到最终是通过ActivityRecord.icicle恢复数据。

//LaunchActivityItem.java

@Override

public void execute(ClientTransactionHandler client, IBinder token,

PendingTransactionActions pendingActions) {

Trace.traceBegin(TRACE\_TAG\_ACTIVITY\_MANAGER, "activityStart");

ActivityClientRecord r = new ActivityClientRecord(token, mIntent, mIdent, mInfo,

mOverrideConfig, mCompatInfo, mReferrer, mVoiceInteractor, mState, mPersistentState,

mPendingResults, mPendingNewIntents, mIsForward,

mProfilerInfo, client);

client.handleLaunchActivity(r, pendingActions, null /\* customIntent \*/);

Trace.traceEnd(TRACE\_TAG\_ACTIVITY\_MANAGER);

}

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent){

activity.mCalled = false;

if (r.isPersistable()) {

mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);

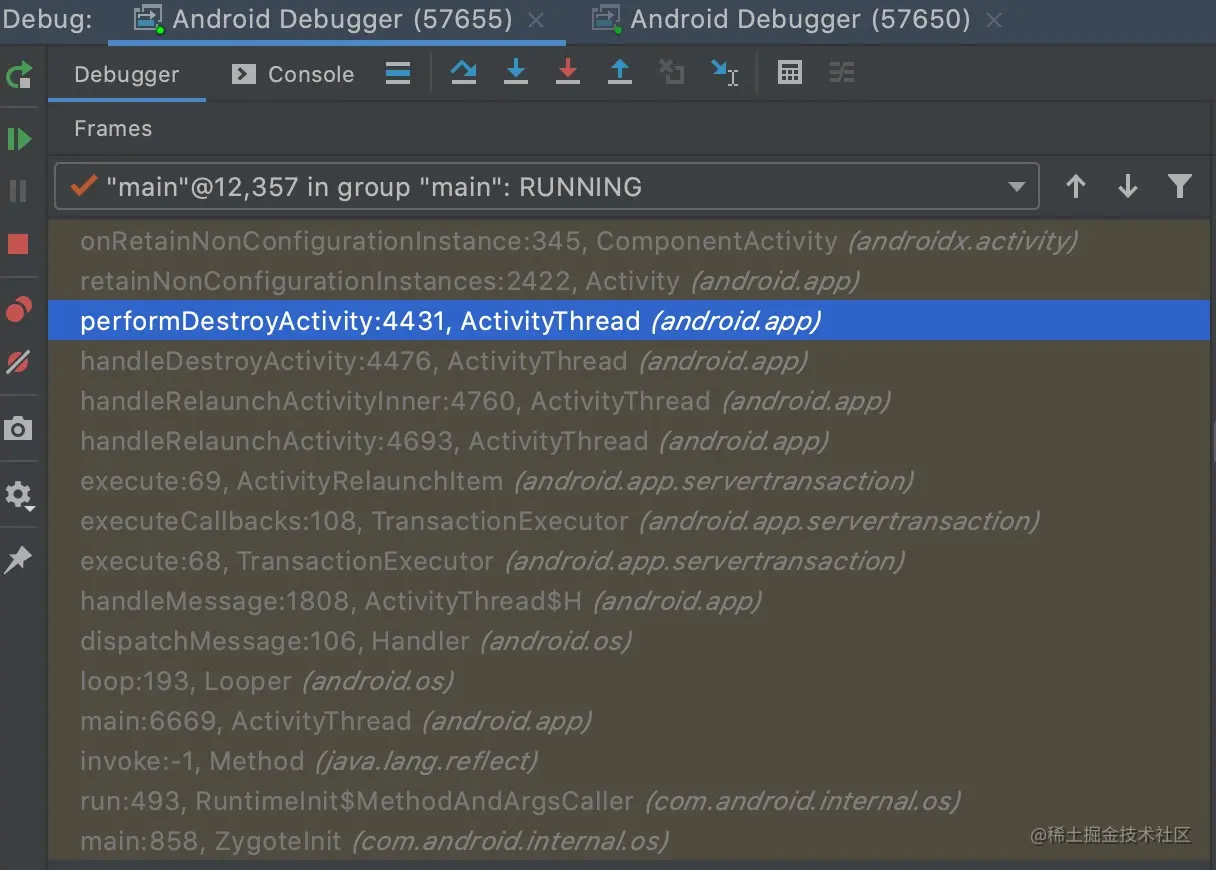
} else {

mInstrumentation.callActivityOnCreate(activity, r.state);

}

}

onRetainNonConfigurationInstance()



该方法是在重建Activity时调用performDestoryActivity时会保存数据。

ActivityClientRecord performDestroyActivity(IBinder token, boolean finishing,

int configChanges, boolean getNonConfigInstance, String reason) {

ActivityClientRecord r = mActivities.get(token);

//省略一些代码

if (getNonConfigInstance) {

try {

r.lastNonConfigurationInstances

= r.activity.retainNonConfigurationInstances();

} catch (Exception e) {

if (!mInstrumentation.onException(r.activity, e)) {

throw new RuntimeException(

"Unable to retain activity "

+ r.intent.getComponent().toShortString()

+ ": " + e.toString(), e);

}

}

}

我们可以看到onRetainNonConfigurationInstance方法返回的Object会赋值给ActivityClientRecord的lastNonConfigurationInstances。

3.1 ViewModel的存储和获取

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：

public ViewModelProvider(@NonNull ViewModelStoreOwner owner) {

this(owner.getViewModelStore(), owner instanceof HasDefaultViewModelProviderFactory

? ((HasDefaultViewModelProviderFactory) owner).getDefaultViewModelProviderFactory()

: NewInstanceFactory.getInstance());

}

public ViewModelProvider(@NonNull ViewModelStoreOwner owner, @NonNull Factory factory) {

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：

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。

/\*\*

\* 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：

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, e);

} catch (IllegalAccessException e) {

throw new RuntimeException("Cannot create an instance of " + modelClass, e);

}

}

}

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：

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");

//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。

## 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：

//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 call.");

}

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：

//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。

//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？

//Acticity.java

NonConfigurationInstances mLastNonConfigurationInstances;

//返回onRetainNonConfigurationInstance()返回的实例

public Object getLastNonConfigurationInstance() {

return mLastNonConfigurationInstances != null ? mLastNonConfigurationInstances.activity : null;

}

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：

final void attach(Context context, ActivityThread aThread, ...

NonConfigurationInstances lastNonConfigurationInstances,... ) {

...

mLastNonConfigurationInstances = lastNonConfigurationInstances;

...

}

mLastNonConfigurationInstances是在Activity的attach方法中赋值。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：

//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。

# 三、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

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）。