Jetpack-ViewModel 笔记

> 源码版本 **androidx.lifecycle:lifecycle-viewmodel:2.6.1**

👍 ViewModel 思想：沉思篇-剖析Jetpack的ViewModel
https://juejin.cn/post/7306723174464061478

ViewModel 的基本功能就是管理 UI 数据。从职责来看，这算是对 Activity 和 Fragment 的一次功能拆分，以前的 UI 数据通常存储在 Activity 和 Fragment 内部，需要自己处理创建、存储、更新、恢复。另外，Activity 和 Fragment 作为 UI 的承载者，很多情况下需要共享数据、复用数据。基于以上痛点，ViewModel 被设计出来了，同时还将保存 UI 数据的能力强化了，支持设备配置变更后的数据保存与恢复。

ViewModel 从两层粒度划分来完成数据管理功能。第一层是对 ViewModel 自身数据的管理，目标是完成 **ViewModel 的创建**，对应的实体为 ViewModelProvider.Factory。第二层是对已存在的 **ViewModel 组的管理**，目标是保证意外情况下 ViewModel 的有效性，对应的实体为 ViewModelStore。同时，两层之间需要由粘合剂连接起来，对应的实体为 ViewModelProvider。

1. ViewModel 的创建

1.1 Factory

ViewModel 通过工厂类 ViewModelProvider.Factory 来完成创建，具体流程是：根据一个 ViewModel 子类的类信息创建对应的对象，如果创建过程需要参数，则从 CreationExtras 获取。

```
1 public interface Factory {
2     public fun <T : ViewModel> create(modelClass: Class<T>): T
3
4     public fun <T : ViewModel> create(modelClass: Class<T>, extras:
  CreationExtras): T
5 }
```

1.2 CreationExtras

CreationExtras 在 Lifecycle 2.5.0 版本引入，主要目的在于使得 ViewModelProvider.Factory 无状态，推荐阅读CreationExtras 来了，创建 ViewModel 的新方式 - 掘金。若自定义 ViewModel 的构造函数存在参数，先看下以前的写法：

```
class MainViewModel(val uid: String, val did: String) : ViewModel() {

    ...

    class Factory(
        private val uid: String,
        private val did: String
    ) : ViewModelProvider.Factory {
        override fun <T : ViewModel> create(modelClass: Class<T>): T {
            return MainViewModel(uid, did) as T
        }
    }
}
```

😅 这种写法的痛点：

- Factory 的参数可能来自 Activity 的 Intent、Fragment 的 arguments...，而且在真实项目中准备 VM 参数的代码要复杂得多
- 持有状态的 Factory 不利于复用，往往需要为每个 VM 配备专属的 Factory

Lifecycle 2.5.0 之后创建 VM 时可以通过 CreationExtras 获取初始化所需的参数，同时一个无状态的 Factory 可以更好地复用，例如可以在一个 Factory 中使用 when 处理所有类型的 VM 创建：

```
class ViewModelFactory : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>, extras:
CreationExtras): T {
        return when (modelClass) {
            MainViewModel::class.java -> {
                val uid = extras[MainViewModel.UID_KEY] ?:""
                MainViewModel(uid) as T
            }

            LoginViewModel::class.java -> {
                val did = extras[LoginViewModel.DID_KEY] ?:""
                LoginViewModel(did) as T
            }

            else -> {
```

```
            throw IllegalArgumentException("Unknown class $modelClass")
        }
    }
}
```

1.2.1 Key

```
public abstract class CreationExtras internal constructor() {
    internal val map: MutableMap<Key<*>, Any?> = mutableMapOf()

    // CreationExtras 元素的 Key, T 是具有该 Key 的元素的类型
    public interface Key<T>

    public abstract operator fun <T> get(key: Key<T>): T?
}
```

👍 Key 的泛型 T 代表对应 Value 的类型。相对于 Map<K，V>，这种定义方式可以更加类型安全地获取多种类型的键值对，CoroutineContext 也是采用这种设计

系统内置的 CreationExtras.Key:

CreationExtras.Key	Descriptions
ViewModelProvider.NewInstanceFactory.VIEW_MODEL_KEY	ViewModelProvider 可以基于 key 区分多个 VM 实例，VIEW_MODEL_KEY 用来提供当前 VM 的这个 key
ViewModelProvider.AndroidViewModelFactory.APPLICATION_KEY	提供当前 Application Context
SavedStateHandleSupport.SAVED_STATE_REGISTRY_OWNER_KEY	
SavedStateHandleSupport.VIEW_MODEL_STORE_OWNER_KEY	
SavedStateHandleSupport.DEFAULT_ARGS_KEY	

1.2.2 创建

CreationExtras 是一个抽象类，无法直接实例化，需要使用其子类 MutableCreationExtras 来创建实例，这是一种读写分离的设计思想，保证了使用处的不变性：

```
public class MutableCreationExtras(initialExtras: CreationExtras = Empty) :
CreationExtras() {

    init {
        map.putAll(initialExtras.map)
    }

    public operator fun <T> set(key: Key<T>, t: T) {
        map[key] = t
    }

    public override fun <T> get(key: Key<T>): T? {
        @Suppress("UNCHECKED_CAST")
        return map[key] as T?
    }
}
```

ViewMdelProvider 的次构造函数中，通过 defaultCreationExtras(owner) 获取 CreationExtras:

```
internal fun defaultCreationExtras(owner: ViewModelStoreOwner): CreationExtras
{
    return if (owner is HasDefaultViewModelProviderFactory) {
        owner.defaultViewModelCreationExtras
    } else CreationExtras.Empty
}
```

看下 ComponentActivity 的默认实现：

```
// ComponentActvity.java

public CreationExtras getDefaultViewModelCreationExtras() {
    MutableCreationExtras extras = new MutableCreationExtras();
    if (getApplication() != null) {
        extras.set(ViewModelProvider.AndroidViewModelFactory.APPLICATION_KEY,
getApplication());
    }
    extras.set(SavedStateHandleSupport.SAVED_STATE_REGISTRY_OWNER_KEY, this);
    extras.set(SavedStateHandleSupport.VIEW_MODEL_STORE_OWNER_KEY, this);
    if (getIntent() != null && getIntent().getExtras() != null) {
```

```
        extras.set(SavedStateHandleSupport.DEFAULT_ARGS_KEY,
getIntent().getExtras());
    }
    return extras;
}
```

以 Activity 为例，我们可以重写 getDefaultViewModelCreationExtras() 来注入自定义的 CreationExtras:

```
override val defaultViewModelCreationExtras: CreationExtras
    get() = MutableCreationExtras(super.defaultViewModelCreationExtras).apply {
        set(MainViewModel.UID_KEY, "123")
    }
```

1.3 ViewModelProvider

在 Activity 中获取 ViewModel 最常见的写法:

```
class MainActivity : AppCompatActivity() {

    val viewModel = ViewModelProvider(this).get(MainViewModel::class.java)

}
```

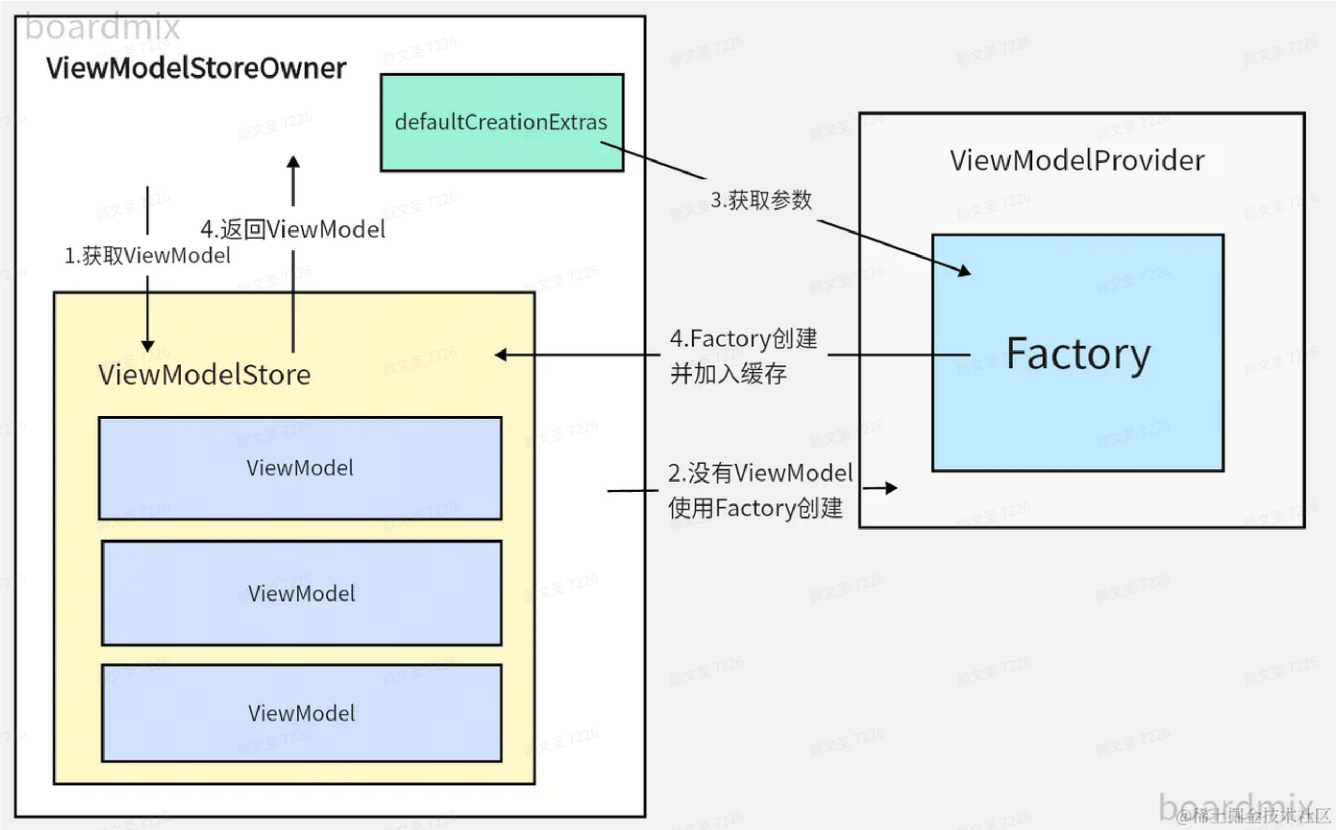
获取 ViewModel 的唯一方式是通过 ViewModelProvider，先看下 ViewModelProvider 有哪些构造函数:

```
public open class ViewModelProvider

@JvmOverloads
constructor(
    private val store: ViewModelStore,
    private val factory: Factory,
    private val defaultCreationExtras: CreationExtras = CreationExtras.Empty,
) {

    public constructor(
        owner: ViewModelStoreOwner
    ) : this(owner.viewModelStore, defaultFactory(owner),
defaultCreationExtras(owner))

```

```
    public constructor(owner: ViewModelStoreOwner, factory: Factory) : this(
        owner.viewModelStore,
        factory,
        defaultCreationExtras(owner)
    )
}
```

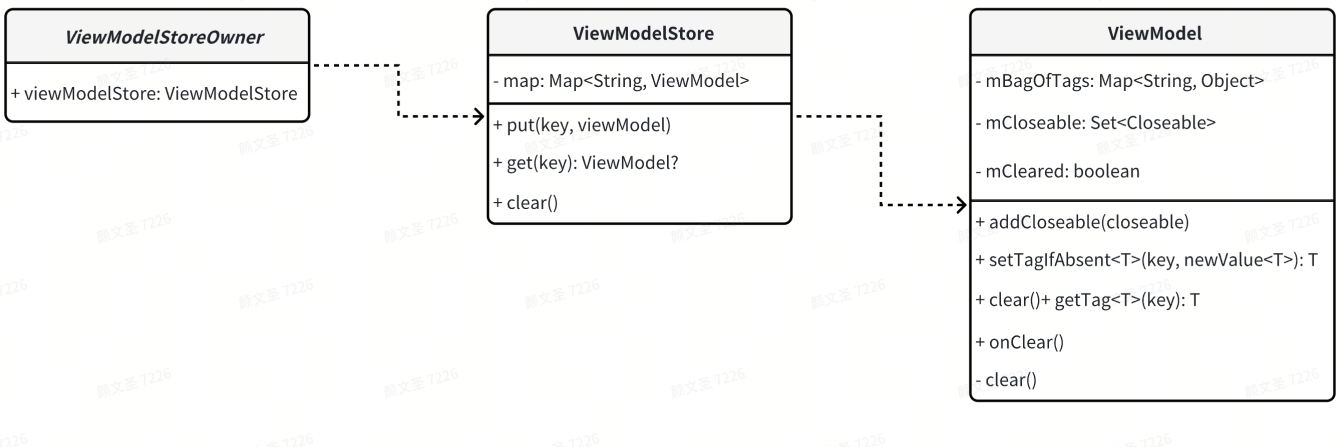
实例化 ViewModelProvider 后，调用 get() 获取 ViewModel:

```
public open operator fun <T : ViewModel> get(modelClass: Class<T>): T {
    val canonicalName = modelClass.canonicalName?: throw
IllegalArgumentException("Local and anonymous classes can not be ViewModels")
    return get("$DEFAULT_KEY:$canonicalName", modelClass) // 1. 根据类名创建默认
key
}

@MainThread
public open operator fun <T : ViewModel> get(key: String, modelClass:
Class<T>): T {
    val viewModel = store[key] // 2. 从 ViewModelStore 通过 key 获取 ViewModel
    if (modelClass.isInstance(viewModel)) {
        return viewModel as T // 3. 返回已有的 ViewModel
    } else {
        if (viewModel != null) {
            // TODO: log a warning.
        }
    }
    val extras = MutableCreationExtras(defaultCreationExtras)
    extras[VIEW_MODEL_KEY] = key
    return try {
        factory.create(modelClass, extras) // 4. 创建 ViewModel
    } catch (e: AbstractMethodError) {
        factory.create(modelClass)
    }.also { store.put(key, it) } // 5. 将新创建的 ViewModel 存入
ViewModelStoreOwner
}
```



图片来自: https://juejin.cn/post/7306723174464061478#heading-17

2. ViewModel 组的管理



ComponentActivity 和 Fragment 实现了 ViewModelStoreOwner，作为 ViewModelStore 的管理者

2.1 ComponentActivity

```
1 // ComponentActivity.java
2
```

```java
private ViewModelStore mViewModelStore;

@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.");
    }
    ensureViewModelStore();
    return mViewModelStore;
}

/**
 * 初始化 ViewModelStore，如果 Activity 由于配置更改而重建，
 * 那么新 Activity 实例将会复用上次的 ViewModelStore
 */
void ensureViewModelStore() {
    if (mViewModelStore == null) {
        // NonConfigurationInstances 与 Activity 因配置改变而重建有关
        NonConfigurationInstances nc = (NonConfigurationInstances) getLastNonConfigurationInstance();
        if (nc != null) {
            // Restore the ViewModelStore from NonConfigurationInstances
            mViewModelStore = nc.viewModelStore;
        }
        if (mViewModelStore == null) {
            mViewModelStore = new ViewModelStore();
        }
    }
}
```

🤔 为什么 ViewModel 的数据在 Activity 重建后还能存在?

这里仅作简单解释：当 Activity 因配置更改而销毁时，会保存 ViewModelStore 实例至 NonConfigurationInstances 中，在之后新 Activity onCreate 时，会读取 NonConfigurationInstances 来复用旧 Activity 的 ViewModelStore，所以重建后的 ViewModelStore 是同一个实例。具体源码分析可参考：屏幕旋转导致Activity销毁重建，ViewModel是如何恢复数据的

当 Activity 正常销毁时，ViewModelStore 中的 ViewModel 也会被清理：

```java
// ComponentActivity.java
public ComponentActivity() {
```

```
    getLifecycle().addObserver(new LifecycleEventObserver() {
        @Override
        public void onStateChanged(@NonNull LifecycleOwner source, @NonNull
Lifecycle.Event event) {
            if (event == Lifecycle.Event.ON_DESTROY) {
                if (!isChangingConfigurations()) { // Activity 正常销毁
                    getViewModelStore().clear();
                }
            }
        }
    });
}

// ViewModelStore.kt
open class ViewModelStore {
    private val map = mutableMapOf<String, ViewModel>()

    fun clear() {
        for (vm in map.values) {
            vm.clear()
        }
        map.clear()
    }
}

// ViewModel.java
public abstract class ViewModel {
    @Nullable
    private final Map<String, Object> mBagOfTags = new HashMap<>();
    @Nullable
    private final Set<Closeable> mCloseables = new LinkedHashSet<>();
    private volatile boolean mCleared = false;

    final void clear() {
        mCleared = true;
        if (mBagOfTags != null) {
            synchronized (mBagOfTags) {
                for (Object value : mBagOfTags.values()) {
                    closeWithRuntimeException(value);
                }
            }
        }
        if (mCloseables != null) {
            synchronized (mCloseables) {
                for (Closeable closeable : mCloseables) {
                    closeWithRuntimeException(closeable);
                }
```

```
            }
        }
        onCleared();
    }

    protected void onCleared() {
    }
}
```

2.2 Fragment

Fragment 也实现了 ViewModelStoreOwner 接口

```
// Fragment.java

FragmentManager mFragmentManager;

@NonNull
@Override
public ViewModelStore getViewModelStore() {
    if (mFragmentManager == null) {
        throw new IllegalStateException("Can't access ViewModels from detached fragment");
    }
    if (getMinimumMaxLifecycleState() == Lifecycle.State.INITIALIZED.ordinal()) {
        throw new IllegalStateException("Calling getViewModelStore() before a Fragment "
                + "reaches onCreate() when using setMaxLifecycle(INITIALIZED) is not "
                + "supported");
    }
    return mFragmentManager.getViewModelStore(this);
}
```

3. SavedStateHandle

我们都知道，Activity 因配置更改而导致销毁时，ViewModel 是不会被销毁的，在之后 Activity 重新创建时，会复用同一个 ViewModel 实例。除了因配置更改而导致 Activity 重建，还有其他导致销毁重建的因素，例如系统内存不足时需要回收部分后台应用进程，这种场景下就需要 ViewModel 配合

SavedStateHandle 来进行保存和恢复数据了。推荐阅读ViewModel的局限，销毁重建的方案 SavedStateHandle - 掘金

4. ViewModel 最佳实践