**События на базе LiveData Android**

LiveData – это отличный инструмент для связывания состояния ваших данных и объектов с жизненным циклом (LifecycleOwner, обычно это Fragment или Activity).

Обычно LiveData помещаются во ViewModel и используются для обновления состояния вашего UI. Часто ViewModel может пережить LifecycleOwner и сохранить состояние LiveData. Такой механизм подходит, когда вам нужно сохранить данные и восстановить их через некоторое время, например, после смены конфигурации.

Но что, если мы хотим использовать механизм событий, а не состояний? Причем обязательно в контексте жизненного цикла обозревателя (LifecycleOwner). Например, нам нужно вывести сообщение после асинхронной операции при условии, что LifecycleOwner еще жив, имеет активных обозревателей и готов обновить свой UI. Если мы будем использовать LiveData, то мы будем получать одно и то же сообщение после каждой смены конфигурации, или при каждом новом подписчике. Одно из решений, которое напрашивается, это после обработки данных в некотором обозревателе обнулить эти данные в LiveData.

Например, такой код:

Observer {

handle(it)

yourViewModel.liveData.value = null

}

Но такой подход имеет ряд недостатков и не отвечает всем необходимым требованиям. Мне бы хотелось иметь механизм событий, который:

1. оповещает только активных подписчиков,
2. в момент подписки не оповещает о предыдущих данных,
3. имеет возможность выставить флаг handled в true, чтобы прервать дальнейшую обработку события.

Я реализовал класс MutableLiveEvent, который обладает всеми вышеперечисленными свойствами и который может работать, как обычный LiveData.

Как использовать:

//Создайте свой экземпляр EventArgs для передачи своих типов данных в событии

class MyIntEventArgs(data: Int) : EventArgs<Int>(data)

//Создайте обычную viewModel

class MainViewModel : ViewModel() {

private val myEventMutable = MutableLiveEvent<MyIntEventArgs>()

val myEvent = myEventMutable as LiveData<MyIntEventArgs>

fun sendEvent(data: Int) {

myEventMutable.value = MyIntEventArgs(data)

}

}

val vm = ViewModelProviders.of(this).get(MainViewModel::class.java)

vm.myEvent.observe(this, Observer {

//Обработка события

/\*

\* Если событие обработано, и вы не хотите, чтобы оно дошло до других обозревателей, то укажите handled = true

\*/

it.handled = true

})

Весь код доступен на GitHub, а ниже я немного расскажу о реализации.

**class** MutableLiveEvent<T : EventArgs<Any>> : MutableLiveData<T>() {  
  
 **internal val observers** = ArraySet<PendingObserver<**in** T>>()  
  
 @MainThread  
 **override fun** observe(owner: LifecycleOwner, observer: Observer<**in** T>) {  
 **val** wrapper = PendingObserver(observer)  
 **observers**.add(wrapper)  
  
 **super**.observe(owner, wrapper)  
 }  
  
 **override fun** observeForever(observer: Observer<**in** T>) {  
 **val** wrapper = PendingObserver(observer)  
 **observers**.add(wrapper)  
  
 **super**.observeForever(observer)  
 }  
  
 @MainThread  
 **override fun** removeObserver(observer: Observer<**in** T>) {  
  
 **when** (observer) {  
 **is** PendingObserver -> {  
 **observers**.remove(observer)  
 **super**.removeObserver(observer)  
 }  
 **else** -> {  
 **val** pendingObserver = **observers**.*firstOrNull* **{ it**.**wrappedObserver** == observer **}  
 if** (pendingObserver != **null**) {  
 **observers**.remove(pendingObserver)  
 **super**.removeObserver(pendingObserver)  
 }  
 }  
 }  
 }  
  
 @MainThread  
 **override fun** setValue(event: T?) {  
 **observers**.*forEach* **{ it**.awaitValue() **}  
 super**.setValue(event)  
 }  
}

Идея заключается в том, чтобы внутри класса MutableLiveEvent, в методах observe и observeForever, оборачивать обозреватели в специальный внутренний класс PendingObserver, который вызывает реальный обозреватель только один раз и только если выставлен флаг pending в true, и событие еще не обработано.

**internal class** PendingObserver<T : EventArgs<Any>>(**val wrappedObserver**: Observer<**in** T>) : Observer<T> {  
  
 **private var pending** = **false  
  
 override fun** onChanged(event: T?) {  
 **if** (**pending** && event?.**handled** != **true**) {  
 **pending** = **false  
 wrappedObserver**.onChanged(event)  
 }  
 }  
  
 **fun** awaitValue() {  
 **pending** = **true** }  
}

В PendingObserver флаг pending выставлен в false по умолчанию. Это решает п.2 (не оповещать о старых данных) из нашего списка.

А код в MutableLiveEvent

**override fun** setValue(event: T?) {  
 **observers**.*forEach* **{ it**.awaitValue() **}  
 super**.setValue(event)  
 }

Сначала выставляет pending в true и только потом обновляет данные внутри себя. Это обеспечивает выполнение п.1. (оповещение только активных подписчиков).

Последний момент, о котором я еще не рассказал, — это EventArgs. Это класс - обобщение, в котором есть флаг handled для прерывания дальнейшей обработки события (п.3.).

**open class** EventArgs<**out** T>(**private val content**: T?) {  
  
 **var handled**: Boolean = **false  
  
 val data**: T?  
 **get**() {  
 **return if** (**handled**) {  
 **null** } **else** {  
 **content** }  
 }  
}

На этом все, спасибо за внимание!