“终于懂了“系列：Jetpack AAC完整解析（二）LiveData 完全掌握！

胡飞洋 2020-12-06 22:01 10486

关注

> 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。

> 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！

上一篇介绍了Jetpack AAC 的基础组件 Lifecycle，它是用于管理Activity/Fragment的生命周期。这篇来介绍基于Lifecycle的用于处理数据的组件——LiveData。

一、LiveData介绍

1.1 作用

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

LiveData，原意是 活着的数据。 数据还能有生命？ 先来看下官方的定义：

> LiveData 是一种可观察的数据存储器类。与常规的可观察类不同，LiveData 具有生命周期感知能力，意指它遵循其他应用组件（如 Activity/Fragment）的生命周期。这种感知能力可确保 LiveData 仅更新处于活跃生命周期状态的应用组件观察者。

拆解开来：

1. LiveData是一个数据持有者，给源数据包装一层。
2. 源数据使用LiveData包装后，可以被observer观察，数据有更新时observer可感知。
3. 但 observer的感知，只发生在（Activity/Fragment）活跃生命周期状态（STARTED、RESUMED）。

也就是说，**LiveData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LifecycleOwner的活跃生命周期状态**。

1.2 特点

使用 LiveData 具有以下优势：

- **确保界面符合数据状态**，当生命周期状态变化时，LiveData通知Observer，可以在observer中更新界面。观察者可以在生命周期状态更改时刷新界面，而不是在每次数据变化时刷新界面。
- **不会发生内存泄漏**，observer会在LifecycleOwner状态变为DESTROYED后自动remove。
- **不会因 Activity 停止而导致崩溃**，如果LifecycleOwner生命周期处于非活跃状态，则它不会接收任何 LiveData事件。
- **不需要手动解除观察**，开发者不需要在onPause或onDestroy方法中解除对LiveData的观察，因为LiveData能感知生命周期状态变化，所以会自动管理所有这些操作。
- **数据始终保持最新状态**，数据更新时 若LifecycleOwner为非活跃状态，那么会在**变为活跃时接收最新数据**。例如，曾经在后台的 Activity 会在返回前台后，observer立即接收最新的数据。

二、LiveData的使用

下面介绍LiveData的使用，掌握使用方法也可以更好理解上面的内容。

2.1基本使用

gradle依赖在上一篇中已经介绍了。下面来看基本用法：

1. 创建LiveData实例，指定源数据类型
2. 创建Observer实例，实现onChanged()方法，用于接收源数据变化并刷新UI

3. LiveData实例使用observe()方法添加观察者，并传入LifecycleOwner
4. LiveData实例使用setValue()/postValue()更新源数据 （子线程要postValue()）

举个例子：

java 复制代码

```java
public class LiveDataTestActivity extends AppCompatActivity{

    private MutableLiveData<String> mLiveData;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lifecycle_test);

        //liveData基本使用
        mLiveData = new MutableLiveData<>();
        mLiveData.observe(this, new Observer<String>() {
            @Override
            public void onChanged(String s) {
                Log.i(TAG, "onChanged: "+s);
            }
        });
        Log.i(TAG, "onCreate: ");
        mLiveData.setValue("onCreate");//activity是非活跃状态，不会回调onChanged。变为活跃时，v
    }
    @Override
    protected void onStart() {
        super.onStart();
        Log.i(TAG, "onStart: ");
        mLiveData.setValue("onStart");//活跃状态，会回调onChanged。并且value会覆盖onCreate、on
    }
    @Override
    protected void onResume() {
        super.onResume();
        Log.i(TAG, "onResume: ");
        mLiveData.setValue("onResume");//活跃状态，回调onChanged
    }
    @Override
    protected void onPause() {
        super.onPause();
        Log.i(TAG, "onPause: ");
        mLiveData.setValue("onPause");//活跃状态，回调onChanged
    }
    @Override
    protected void onStop() {
        super.onStop();
```

```
        Log.i(TAG, "onStop: ");
        mLiveData.setValue("onStop");//非活跃状态，不会回调onChanged。后面变为活跃时，value被onS
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.i(TAG, "onDestroy: ");
        mLiveData.setValue("onDestroy");//非活跃状态，且此时Observer已被移除，不会回调onChanged
    }
}
```

注意到 LiveData实例mLiveData的创建是使用MutableLiveData，它是LiveData的实现类，且指定了源数据的类型为String。然后创建了接口Observer的实例，实现其onChanged()方法，用于接收源数据的变化。observer和Activity一起作为参数调用mLiveData的observe()方法，表示observer开始观察mLiveData。然后Activity的所有生命周期方法中都调用了mLiveData的setValue()方法。 结果日志打印如下：

java 复制代码

```
//打开页面，
2020-11-22 20:23:29.865 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onCreate:
2020-11-22 20:23:29.867 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onStart:
2020-11-22 20:23:29.868 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onChanged:
2020-11-22 20:23:29.869 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onResume:
2020-11-22 20:23:29.869 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onChanged:
//按Home键
2020-11-22 20:23:34.349 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onPause:
2020-11-22 20:23:34.349 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onChanged:
2020-11-22 20:23:34.368 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onStop:
//再点开
2020-11-22 20:23:39.145 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onStart:
2020-11-22 20:23:39.146 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onChanged:
2020-11-22 20:23:39.147 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onResume:
2020-11-22 20:23:39.147 13360-13360/com.hfy.androidlearning I/Lifecycle_Test: onChanged:
//返回键退出
2020-11-22 20:23:56.753 14432-14432/com.hfy.androidlearning I/Lifecycle_Test: onPause:
2020-11-22 21:23:56.753 14432-14432/com.hfy.androidlearning I/Lifecycle_Test: onChanged:
2020-11-22 20:23:58.320 14432-14432/com.hfy.androidlearning I/Lifecycle_Test: onStop:
2020-11-22 20:23:58.322 14432-14432/com.hfy.androidlearning I/Lifecycle_Test: onDestroy:
```

- 首先打开页面，onCreate()中setValue，由于activity是非活跃状态，不会立即回调onChanged。当走到onStart()变为活跃时，onChanged被调用，但value被onStart()中setValue的value覆盖，所以打印的是onChanged: onStart。（为啥不是连续打印两次呢？，是因为ON_START事件是在onStart() return之后，即onStart()走完之后才变为活跃<

详见上一篇>，此时observer接收最新的数据。） 接着走到onResume()，也setValue了，同样是活跃状态，所以立刻回调onChanged，打印onChanged: onResume

- 按Home键时，onPause()中setValue，活跃状态，立刻回调onChanged方法。onStop()执行时已经变为非活跃状态，此时setValue不会立即回调onChanged方法。

- 再点开时，走到onStart()变为活跃时，onChanged被调用，但value被onStart()中setValue的value覆盖，所以打印的是onChanged: onStart。接着走到onResume()，也setValue了，同样是活跃状态，所以立刻回调onChanged。

- 返回键退出时，onPause()/onStop()的效果和按Home键一样。onDestroy()中setValue，此时非活跃状态，且此时observer已被移除，不会回调onChanged。

另外，除了使用observe()方法添加观察者，也可以使用**observeForever**(Observer) 方法来注册未关联 LifecycleOwner的观察者。在这种情况下，观察者会被视为始终处于活跃状态。

2.2 扩展使用

扩展包括两点：

1. 自定义LiveData，本身回调方法的覆写：onActive()、onInactive()。
2. 实现LiveData为**单例**模式，便于在多个Activity、Fragment之间共享数据。

官方的例子如下：

java 复制代码

```java
public class StockLiveData extends LiveData<BigDecimal> {
        private static StockLiveData sInstance; //单实例
        private StockManager stockManager;

        private SimplePriceListener listener = new SimplePriceListener() {
            @Override
            public void onPriceChanged(BigDecimal price) {
                setValue(price);//监听到股价变化 使用setValue(price) 告知所有活跃观察者
            }
        };

    //获取单例
        @MainThread
        public static StockLiveData get(String symbol) {
            if (sInstance == null) {
```

```
16                 sInstance = new StockLiveData(symbol);
17             }
18             return sInstance;
19         }
20 
21         private StockLiveData(String symbol) {
22             stockManager = new StockManager(symbol);
23         }
24 
25            //活跃的观察者（LifecycleOwner）数量从 0 变为 1 时调用
26         @Override
27         protected void onActive() {
28             stockManager.requestPriceUpdates(listener);//开始观察股价更新
29         }
30 
31            //活跃的观察者（LifecycleOwner）数量从 1 变为 0 时调用。这不代表没有观察者了，可能是全都
32         @Override
33         protected void onInactive() {
34             stockManager.removeUpdates(listener);//移除股价更新的观察
35         }
36     }
37 
```

为了观察股票价格变动，继承LiveData自定义了StockLiveData，且为单例模式，只能通过get(String symbol)方法获取实例。 并且重写了onActive()、onInactive()，并加入了 开始观察股价更新、移除股价更新观察 的逻辑。

- onActive()调用时机为：活跃的观察者（LifecycleOwner）数量从 0 变为 1 时。
- onInactive()调用时机为：活跃的观察者（LifecycleOwner）数量从 1 变为 0 时。

也就是说，只有当 存在活跃的观察者（LifecycleOwner）时 才会连接到 股价更新服务 监听股价变化。使用如下：

java 复制代码

```
1      public class MyFragment extends Fragment {
2          @Override
3          public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceStat
4              super.onViewCreated(view, savedInstanceState);
5              //获取StockLiveData单实例，添加观察者，更新UI
6              StockLiveData.get(symbol).observe(getViewLifecycleOwner(), price -> {
7                  // Update the UI.
8              });
9          }
10     }
```

由于StockLiveData是单实例模式，那么多个LifycycleOwner（Activity、Fragment）间就可以共享数据了。

2.3 高级用法

如果希望在将 LiveData 对象分派给观察者之前对存储在其中的值进行更改，或者需要根据另一个实例的值返回不同的 LiveData 实例，可以使用LiveData中提供的Transformations类。

2.3.1 数据修改 - Transformations.map

java 复制代码

```java
//Integer类型的liveData1
MutableLiveData<Integer> liveData1 = new MutableLiveData<>();
//转换成String类型的liveDataMap
LiveData<String> liveDataMap = Transformations.map(liveData1, new Function<Integ
    @Override
    public String apply(Integer input) {
        String s = input + " + Transformations.map";
        Log.i(TAG, "apply: " + s);
        return s;
    }
});
liveDataMap.observe(this, new Observer<String>() {
    @Override
    public void onChanged(String s) {
        Log.i(TAG, "onChanged1: "+s);
    }
});

liveData1.setValue(100);
```

使用很简单：原本的liveData1 没有添加观察者，而是使用Transformations.map()方法 对liveData1的数据进行的修改 生成了新的liveDataMap，liveDataMap添加观察者，最后liveData1设置数据 。

此例子把 Integer类型的liveData1 修改为String类型的liveDataMap。结果如下：

java 复制代码

```java
2020-12-06 17:01:56.095 21998-21998/com.hfy.androidlearning I/Lifecycle_Test: apply: 100
2020-12-06 17:01:56.095 21998-21998/com.hfy.androidlearning I/Lifecycle_Test: onChanged1
```

2.3.2 数据切换 - Transformations.switchMap

如果想要根据某个值 切换观察不同LiveData数据，则可以使用Transformations.switchMap()方法。

java 复制代码

```java
//两个liveData，由liveDataSwitch决定 返回哪个livaData数据
      MutableLiveData<String> liveData3 = new MutableLiveData<>();
      MutableLiveData<String> liveData4 = new MutableLiveData<>();

//切换条件LiveData，liveDataSwitch的value 是切换条件
      MutableLiveData<Boolean> liveDataSwitch = new MutableLiveData<>();

//liveDataSwitchMap由switchMap()方法生成，用于添加观察者
      LiveData<String> liveDataSwitchMap = Transformations.switchMap(liveDataSwitch, n
          @Override
          public LiveData<String> apply(Boolean input) {
          //这里是具体切换逻辑：根据liveDataSwitch的value返回哪个liveData
              if (input) {
                  return liveData3;
              }
              return liveData4;
          }
      });

      liveDataSwitchMap.observe(this, new Observer<String>() {
          @Override
          public void onChanged(String s) {
              Log.i(TAG, "onChanged2: " + s);
          }
      });

      boolean switchValue = true;
      liveDataSwitch.setValue(switchValue);//设置切换条件值

      liveData3.setValue("liveData3");
      liveData4.setValue("liveData4");
```

liveData3、liveData4是两个数据源，有一个判断条件来决定 取哪一个数据 ，这个条件就是liveDataSwitch，如果值为true则取liveData3，false则取liveData4。Transformations.switchMap()就用于实现这一逻辑，返回值liveDataSwitchMap添加观察者就可以了。 结果如下：

java 复制代码

```
020-12-06 17:33:53.844 27347-27347/com.hfy.androidlearning I/Lifecycle_Test: switchValue
020-12-06 17:33:53.847 27347-27347/com.hfy.androidlearning I/Lifecycle_Test: onChanged2:

020-12-06 17:34:37.600 27628-27628/com.hfy.androidlearning I/Lifecycle_Test: switchValue
020-12-06 17:34:37.602 27628-27628/com.hfy.androidlearning I/Lifecycle_Test: onChanged2:
```

（Transformations对LivaData这两个用法和Rxjava简直一毛一样）

2.3.3 观察多个数据 - MediatorLiveData

MediatorLiveData 是 LiveData 的子类，允许合并多个 LiveData 源。只要任何原始的 LiveData 源对象发生更改，就会触发 MediatorLiveData 对象的观察者。

java 复制代码

```java
        MediatorLiveData<String> mediatorLiveData = new MediatorLiveData<>();

        MutableLiveData<String> liveData5 = new MutableLiveData<>();
        MutableLiveData<String> liveData6 = new MutableLiveData<>();

    //添加 源 LiveData
        mediatorLiveData.addSource(liveData5, new Observer<String>() {
            @Override
            public void onChanged(String s) {
                Log.i(TAG, "onChanged3: " + s);
                mediatorLiveData.setValue(s);
            }
        });
    //添加 源 LiveData
        mediatorLiveData.addSource(liveData6, new Observer<String>() {
            @Override
            public void onChanged(String s) {
                Log.i(TAG, "onChanged4: " + s);
                mediatorLiveData.setValue(s);
            }
        });

    //添加观察
        mediatorLiveData.observe(this, new Observer<String>() {
            @Override
            public void onChanged(String s) {
                Log.i(TAG, "onChanged5: "+s);
                //无论liveData5、liveData6更新，都可以接收到
            }
        });
```

```
31
32          liveData5.setValue("liveData5");
33          //liveData6.setValue("liveData6");
```

例如，如果界面中有可以从本地数据库或网络更新的 LiveData 对象，则可以向 MediatorLiveData 对象添加以下源：

- 与存储在本地数据库中的数据关联的 liveData5
- 与从网络访问的数据关联的 liveData6 **Activity 只需观察 MediatorLiveData 对象即可从这两个源接收更新。** 结果如下：

java 复制代码

```
1  2020-12-06 17:56:17.870 29226-29226/com.hfy.androidlearning I/Lifecycle_Test: onChanged3
2  2020-12-06 17:56:17.870 29226-29226/com.hfy.androidlearning I/Lifecycle_Test: onChanged5
```

（Transformations也是对MediatorLiveData的使用。）

LiveData的使用就讲完了，下面开始源码分析。

三、源码分析

前面提到 LiveData几个特点，能感知生命周期状态变化、不用手动解除观察等等，这些是如何做到的呢?

3.1 添加观察者

LiveData原理是观察者模式，下面就先从LiveData.observe()方法看起：

java 复制代码

```
1      /**
2       * 添加观察者. 事件在主线程分发. 如果LiveData已经有数据，将直接分发给observer。
3       * 观察者只在LifecycleOwner活跃时接受事件，如果变为DESTROYED状态，observer自动移除。
4       * 当数据在非活跃时更新，observer不会接收到。变为活跃时 将自动接收前面最新的数据。
5       * LifecycleOwner非DESTROYED状态时，LiveData持有observer和 owner的强引用，DESTROYED状态时
6       * @param owner    控制observer的LifecycleOwner
7       * @param observer 接收事件的observer
8       */
9      @MainThread
10     public void observe(@NonNull LifecycleOwner owner, @NonNull Observer<? super T> obse
```

```
        assertMainThread("observe");
        if (owner.getLifecycle().getCurrentState() == DESTROYED) {
            // LifecycleOwner是DESTROYED状态，直接忽略
            return;
        }
        //使用LifecycleOwner、observer 组装成LifecycleBoundObserver，添加到mObservers中
        LifecycleBoundObserver wrapper = new LifecycleBoundObserver(owner, observer);
        ObserverWrapper existing = mObservers中.putIfAbsent(observer, wrapper);
        if (existing != null && !existing.isAttachedTo(owner)) {
        //!existing.isAttachedTo(owner)说明已经添加到mObservers中的observer指定的owner不是传进
            throw new IllegalArgumentException("Cannot add the same observer"
                    + " with different lifecycles");
        }
        if (existing != null) {
            return;//这里说明已经添加到mObservers中,且owner就是传进来的owner
        }
        owner.getLifecycle().addObserver(wrapper);
    }
```

首先是判断LifecycleOwner是DESTROYED状态，就直接忽略，不能添加。接着使用LifecycleOwner、observer 组装成LifecycleBoundObserver包装实例wrapper，使用putIfAbsent方法observer-wrapper作为key-value添加到观察者列表mObservers中。（putIfAbsent意思是只有列表中没有这个observer时才会添加。）

然后对添加的结果进行判断，如果mObservers中已经存在此observer key，但value中的owner不是传进来的owner，就会报错“不能添加同一个observer却是不同LifecycleOwner”。如果是相同的owner，就直接returne。

最后用LifecycleOwner的Lifecycle添加observer的封装wrapper。

另外，再看observeForever方法：

java 复制代码

```
    @MainThread
    public void observeForever(@NonNull Observer<? super T> observer) {
        assertMainThread("observeForever");
        AlwaysActiveObserver wrapper = new AlwaysActiveObserver(observer);
        ObserverWrapper existing = mObservers.putIfAbsent(observer, wrapper);
        if (existing instanceof LiveData.LifecycleBoundObserver) {
            throw new IllegalArgumentException("Cannot add the same observer"
                    + " with different lifecycles");
        }
        if (existing != null) {
            return;
```

```
        }
        wrapper.activeStateChanged(true);
    }
```

和observe()类似，只不过 会认为观察者一直是活跃状态，且不会自动移除观察者。

3.2 事件回调

LiveData添加了观察者LifecycleBoundObserver，接着看如何进行回调的：

java 复制代码

```java
    class LifecycleBoundObserver extends ObserverWrapper implements LifecycleEventObserv
        @NonNull
        final LifecycleOwner mOwner;

        LifecycleBoundObserver(@NonNull LifecycleOwner owner, Observer<? super T> observ
            super(observer);
            mOwner = owner;
        }

        @Override
        boolean shouldBeActive() { //至少是STARTED状态
            return mOwner.getLifecycle().getCurrentState().isAtLeast(STARTED);
        }

        @Override
        public void onStateChanged(@NonNull LifecycleOwner source,
                @NonNull Lifecycle.Event event) {
            if (mOwner.getLifecycle().getCurrentState() == DESTROYED) {
                removeObserver(mObserver);//LifecycleOwner变成DESTROYED状态，则移除观察者
                return;
            }
            activeStateChanged(shouldBeActive());
        }

        @Override
        boolean isAttachedTo(LifecycleOwner owner) {
            return mOwner == owner;
        }

        @Override
        void detachObserver() {
            mOwner.getLifecycle().removeObserver(this);
```

```
33          }
34      }
```

LifecycleBoundObserver是LiveData的内部类，是对原始Observer的包装，把LifecycleOwner和Observer绑定在一起。当LifecycleOwner处于活跃状态，就称 LifecycleBoundObserver是活跃的观察者。

它实现自接口LifecycleEventObserver，实现了onStateChanged方法。上一篇Lifecycle中提到onStateChanged是生命周期状态变化的回调。

在LifecycleOwner生命周期状态变化时 判断如果是DESTROYED状态，则移除观察者。LiveData自动移除观察者特点就来源于此。 如果不是DESTROYED状态，将调用父类ObserverWrapper的activeStateChanged()方法处理 这个生命周期状态变化，shouldBeActive()的值作为参数，至少是STARTED状态为true，即活跃状态为true。

java 复制代码

```java
1       private abstract class ObserverWrapper {
2           ...
3           void activeStateChanged(boolean newActive) {
4               if (newActive == mActive) {
5                   return;//活跃状态 未发生变化时，不会处理。
6               }
7               mActive = newActive;
8               boolean wasInactive = LiveData.this.mActiveCount == 0;//没有活跃的观察者
9               LiveData.this.mActiveCount += mActive ? 1 : -1;//mActive为true表示变为活跃
10              if (wasInactive && mActive) {
11                  onActive();//活跃的观察者数量 由0变为1
12              }
13              if (LiveData.this.mActiveCount == 0 && !mActive) {
14                  onInactive(); //活跃的观察者数量 由1变为0
15              }
16              if (mActive) {
17                  dispatchingValue(this);//观察者变为活跃，就进行数据分发
18              }
19          }
20      }
```

ObserverWrapper也是LiveData的内部类。mActive是ObserverWrapper的属性，表示此观察者是否活跃。如果活跃状态 未发生变化时，不会处理。

LiveData.this.mActiveCount == 0 是指 LiveData 的活跃观察者数量。活跃的观察者数量 由0变为1、由1变为0 会分别调用LiveData的 onActive()、onInactive()方法。这就是前面提到的 扩展使

用 的回调方法。

最后观察者变为活跃，就使用LiveData的dispatchingValue(observerWrapper)进行数据分发:

java 复制代码

```java
void dispatchingValue(@Nullable ObserverWrapper initiator) {
    if (mDispatchingValue) {
        mDispatchInvalidated = true;//如果当前正在分发，则分发无效，return
        return;
    }
    mDispatchingValue = true; //标记正在分发
    do {
        mDispatchInvalidated = false;
        if (initiator != null) {
            considerNotify(initiator); //observerWrapper不为空，使用considerNotify()通
            initiator = null;
        } else { //observerWrapper为空，遍历通知所有的观察者
            for (Iterator<Map.Entry<Observer<? super T>, ObserverWrapper>> iterator
                    mObservers.iteratorWithAdditions(); iterator.hasNext(); ) {
                considerNotify(iterator.next().getValue());
                if (mDispatchInvalidated) {
                    break;
                }
            }
        }
    } while (mDispatchInvalidated);
    mDispatchingValue = false;
}
```

如果当前正在分发，则分发无效；observerWrapper不为空，就使用considerNotify()通知真正的观察者，observerWrapper为空 则遍历通知所有的观察者。 observerWrapper啥时候为空呢？这里先留个疑问。 继续看considerNotify()方法：

java 复制代码

```java
private void considerNotify(ObserverWrapper observer) {
    if (!observer.mActive) {
        return; //观察者非活跃 return
    }
    //若当前observer对应owner非活跃，就会再调用activeStateChanged方法，并传入false，其内部会
    if (!observer.shouldBeActive()) {
        observer.activeStateChanged(false);
        return;
    }
    if (observer.mLastVersion >= mVersion) {
        return;
```

```
        }
        observer.mLastVersion = mVersion;
        observer.mObserver.onChanged((T) mData);//回调真正的mObserver的onChanged方法
    }
```

先进行状态检查：观察者是非活跃就return；若当前observer对应的owner非活跃，就会再调用activeStateChanged方法，并传入false，其内部会再次判断。最后回调真正的mObserver的onChanged方法，值是LivaData的变量mData。

到这里回调逻辑也通了。

3.3 数据更新

LivaData数据更新可以使用setValue(value)、postValue(value)，区别在于postValue(value)用于 子线程:

java 复制代码

```java
//LivaData.java
    private final Runnable mPostValueRunnable = new Runnable() {
        @SuppressWarnings("unchecked")
        @Override
        public void run() {
            Object newValue;
            synchronized (mDataLock) {
                newValue = mPendingData;
                mPendingData = NOT_SET;
            }
            setValue((T) newValue); //也是走到setValue方法
        }
    };

    protected void postValue(T value) {
        boolean postTask;
        synchronized (mDataLock) {
            postTask = mPendingData == NOT_SET;
            mPendingData = value;
        }
        if (!postTask) {
            return;
        }
        ArchTaskExecutor.getInstance().postToMainThread(mPostValueRunnable);//抛到主线程
    }
```

postValue方法把Runable对象mPostValueRunnable抛到主线程，其run方法中还是使用的setValue()，继续看：

java 复制代码

```
    @MainThread
    protected void setValue(T value) {
        assertMainThread("setValue");
        mVersion++;
        mData = value;
        dispatchingValue(null);
    }
```

setValue()把value赋值给mData，然后调用dispatchingValue(null)，参数是null，对应前面提到的observerWrapper为空的场景，即 遍历所有观察者 进行分发回调。

到这里观察者模式完整的实现逻辑就梳理清晰了：LivaData通过observe()添加 与LifecycleOwner绑定的观察者；观察者变为活跃时回调最新的数据；使用setValue()、postValue()更新数据时会通知回调所有的观察者。

3.4 Transformations原理

最后来看下Transformations的map原理，如何实现数据修改的。switchMap类似的。

java 复制代码

```
//Transformations.java
    public static <X, Y> LiveData<Y> map(@NonNull LiveData<X> source,@NonNull final Func
        final MediatorLiveData<Y> result = new MediatorLiveData<>();
        result.addSource(source, new Observer<X>() {
            @Override
            public void onChanged(@Nullable X x) {
                result.setValue(mapFunction.apply(x));
            }
        });
        return result;
    }
```

new了一个MediatorLiveData实例，然后将 传入的livaData、new的Observer实例作为参数 调用addSource方法：

java 复制代码

```java
//MediatorLiveData.java
    public <S> void addSource(@NonNull LiveData<S> source, @NonNull Observer<? super S>
        Source<S> e = new Source<>(source, onChanged);
        Source<?> existing = mSources.putIfAbsent(source, e);
        if (existing != null && existing.mObserver != onChanged) {
            throw new IllegalArgumentException(
                    "This source was already added with the different observer");
        }
        if (existing != null) {
            return;
        }
        if (hasActiveObservers()) {
        //MediatorLiveData有活跃观察者，就plug
            e.plug();
        }
    }
```

MediatorLiveData是LiveData的子类，用来观察其他的LiveData并在其OnChanged回调时 做出响应。传入的livaData、Observer 包装成Source实例，添加到列表mSources中。

如果MediatorLiveData有活跃观察者，就调用plug()：

java 复制代码

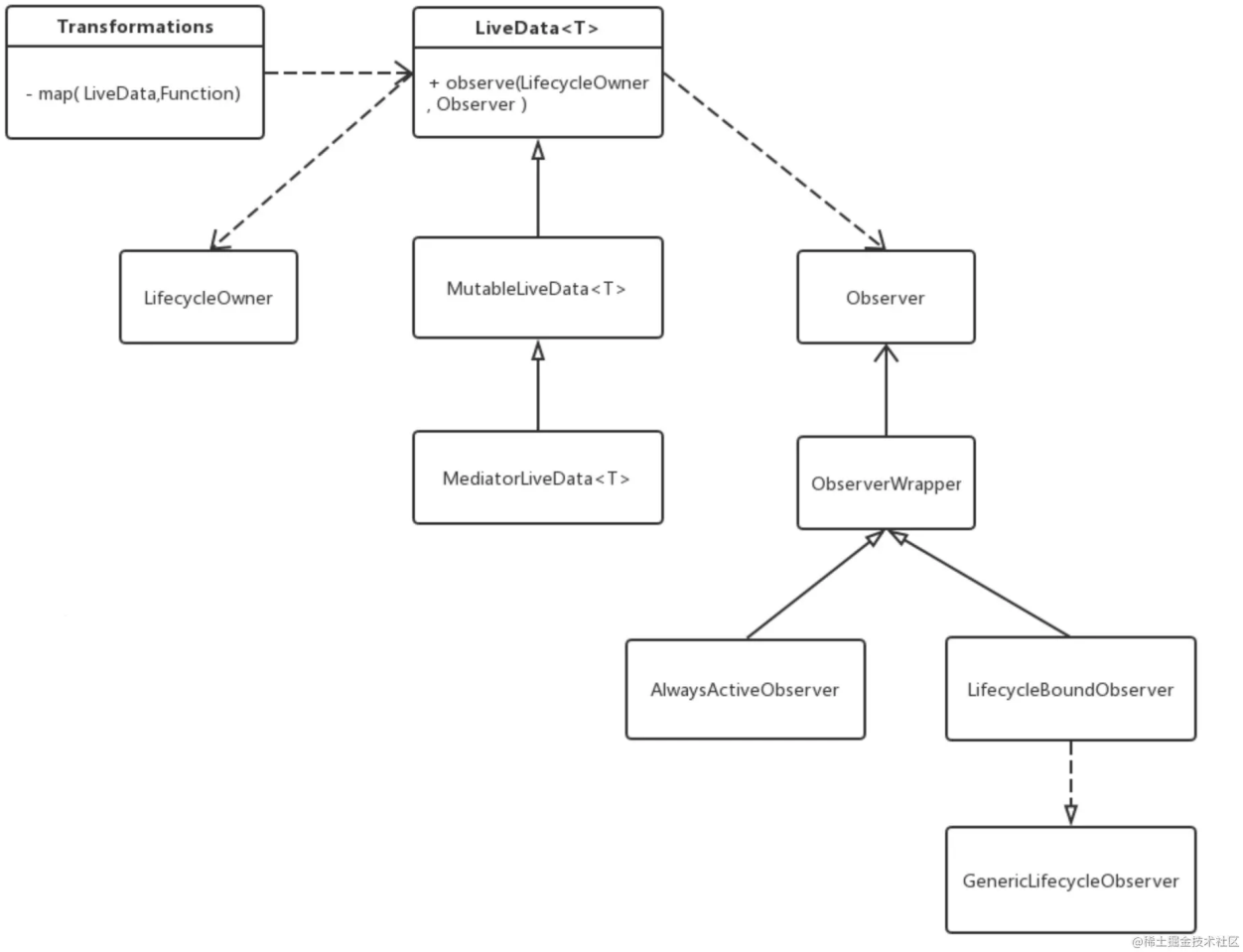
```java
//MediatorLiveData.java
    private static class Source<V> implements Observer<V> {
        final LiveData<V> mLiveData;
        final Observer<? super V> mObserver;
        int mVersion = START_VERSION;

        Source(LiveData<V> liveData, final Observer<? super V> observer) {
            mLiveData = liveData;
            mObserver = observer;
        }

        void plug() {
            mLiveData.observeForever(this);//observeForever
        }

        void unplug() {
            mLiveData.removeObserver(this);
        }

        @Override
        public void onChanged(@Nullable V v) {
            if (mVersion != mLiveData.getVersion()) {
                mVersion = mLiveData.getVersion();
```

```
                mObserver.onChanged(v);//源LiveData数据变化时及时回调到 传入的
            }
        }
    }
```

Source是MediatorLiveData的内部类，是对源LiveData的包装。plug()中让源LiveData调用observeForever方法添加永远观察者-自己。 这里为啥使用observeForever方法呢，这是因为源LiveData在外部使用时不会调用observer方法添加观察者，这里永远观察是为了在源LiveData数据变化时及时回调到 mObserver.onChanged(v)方法，也就是Transformations map方法中的nChanged方法。 而在e.plug()前是有判断 MediatorLiveData 确认有活跃观察者的。

最后map方法中的nChanged方法中有调用MediatorLiveData实例的setValue(mapFunction.apply(x)); 并返回实例。而mapFunction.apply()就是map方法传入的修改逻辑Function实例。

最后类关系图：



四、总结

本文先介绍了LiveData的概念——使用观察者并可以感知生命周期，然后是使用方式、自定义LivaData、高级用法Transformations。最后详细分析了LiveData源码及原理。

并且可以看到Lifecycle如何在LiveData中发挥作用，理解了观察者模式在其中的重要运用。LiveData是我们后续建立MVVM架构的核心。 LiveData同样是我们必须掌握和理解的部分。

下一篇将介绍ViewModel，同样是AAC中的核心内容。 今天就到这里啦~

.

感谢与参考：

Livedata官方文档

Android Jetpack架构组件（五）一文带你了解LiveData(原理篇)

.

> **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