LiveData 源码分析，再来梳理一下吧~

345丶 郭霖 2022-09-07 08:00 发表于江苏

/ 今日科技快讯 /

近日，爱尔兰数据隐私监管机构证实，在对Meta Platform旗下社交网络Instagram处理儿童数据的方式进行调查后，该机构已决定对其处以4.05亿欧元(约合4.03亿美元)的罚款。

/ 作者简介 /

本篇文章来自345丶的投稿，文章主要对LiveData的源码进行了分析，相信会对大家有所帮助！同时也感谢作者贡献的精彩文章。

345丶的博客地址：

> https://juejin.cn/user/3597257779197021

/ 前言 /

LiveData 是一种持有可被观察的数据存储类,和其他可被观察的类不同的是，LiveData 是就要生命周期感知能力的，这意味着他可以在 Activity ，fragment 或者 service 生命周期活跃状态时 更新这些组件。

在日常开发过程中，LiveData 已经是必不可少的一环了，例如 MVVM 以及 MVI 开发模式中，都用到了 LiveData。

/ 了解 LiveData /

如果观察者(Activity/Fragment) 的生命周期处于 STARTED 或者 RESUMED 状态，LiveData 就会认为是活跃状态。LiveData 只会将数据更新给活跃的观察者。

在添加观察者的时候，可以传入 LifecycleOwner 。有了这关系，当 Lifecycle 对象状态为 DESTORYED 时，便可以移除这个观察者。

使用 LIveData 具有以下优势：

- 确保界面符合数据状态：数据发生变化时，就会通知观察者。我们可以再观察者回调中更新界面，这样就无需在数据改变后手动更新界面了。
- 没有内存泄漏，因为关联了生命周期，页面销毁后会进行自我清理。
- 不会因为Activity 停止而导致崩溃，页面处于非活跃状态时，他不会接收到任何 LiveData 事件。
- 数据始终保持最新状态，页面如果变为活跃状态，它会在变为活跃状态时接收最新数据。
- 配置更改后也会接收到最新的可用数据。
- 共享资源，可以使用单例模式扩展 LiveData 对象，以便在应用中共享他们。

/ LiveData 的使用 /

LiveData 是一种可用于任何数据的封装容器，通常 LiveData 存储在 ViewModel 对象中。

```
class JokesDetailViewModel : ViewModel() {
   //创建 LiveData
   private val _state by lazy { MutableLiveData<JokesUIState>() }
   val state : LiveData<JokesUIState> = _state

   private fun loadChildComment(page: Int, commentId: Int, parentPos: Int, curPos: Int) {
      viewModelScope.launch {
         launchHttp {
            jokesApi.jokesCommentListItem(commentId, page)//请求数据
         }.toData {
            //通知观察者
            _state.value = JokesUIState.LoadMoreChildComment(it.data, parentPos, curPos)
         }
      }
```

```
    }
}


//观察 LiveData
viewModel.state.observe(this, Observer {
    //更新 UI
})
```

/ LiveData 实现原理分析 /

LiveData 源码中主要用到的类：

- Observer：观察者接口
- LiveData：发送已经添加观察的逻辑都在其中
- ObserverWrapper：抽象的观察者包装类，提供了mLastVersion 和判断以及更新观察者是否活跃的方法
- LifecycleBoundleObserver：继承 ObserverWrapper，可以感知生命周期，会在页面活跃的时候更新观察者
- AlwaysActiveObserver：继承 ObserverWrapper ，无法感知生命周期，可以在任意时刻接收到通知。

添加观察者：observer

调用 observe，传入 LifecycleOwner 和 observer 。

通过 LifecycleOwner 可以对页面的生命周期进行观察，只需要获取 lifecycle添加对应的观察者即可。

```
    @MainThread
    public void observe(@NonNull LifecycleOwner owner, @NonNull Observer<? super T> observer
        //如果已经销毁，直接退出
        if (owner.getLifecycle().getCurrentState() == DESTROYED) {
            return;
        }
        //对 observer 进行包装
        LifecycleBoundObserver wrapper = new LifecycleBoundObserver(owner, observer);
        //进行保存，如果已经添加过，则会返回之前添加时 key 的 value，没添加过就进行添加，返回 null
```

```
    ObserverWrapper existing = mObservers.putIfAbsent(observer, wrapper);
    if (existing != null && !existing.isAttachedTo(owner)) {
      //....
    }
    //之前添加过
    if (existing != null) {
       return;
    }
    //注册 lifecycle，生命周期改变时会回调
    owner.getLifecycle().addObserver(wrapper);
  }
```

LiveData.LifecycleBoundObserver

继承 ObserverWrapper 的包装类，实现了LifecycleEventObserver 接口，也就是可以接收到页面生命周期的通知。

```
class LifecycleBoundObserver extends ObserverWrapper implements LifecycleEventObserver {
  @NonNull
  final LifecycleOwner mOwner;

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

  //当前状态大于或者等于 STARTED 返回 true
  @Override
  boolean shouldBeActive() {
    return mOwner.getLifecycle().getCurrentState().isAtLeast(STARTED);
  }

  //生命周期相关回调
  @Override
  public void onStateChanged(@NonNull LifecycleOwner source,
      @NonNull Lifecycle.Event event) {
    Lifecycle.State currentState = mOwner.getLifecycle().getCurrentState();
    //如果已经销毁，移除观察者
    if (currentState == DESTROYED) {
       removeObserver(mObserver);
       return;
    }
    Lifecycle.State prevState = null;
   //循环更新状态
    while (prevState != currentState) {
       prevState = currentState;
```

```
            //修改活跃状态
            activeStateChanged(shouldBeActive());
            currentState = mOwner.getLifecycle().getCurrentState();
        }
    }

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

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

ObserverWrapper

```
void activeStateChanged(boolean newActive) {
  //如果等于之前状态
  if (newActive == mActive) {
    return;
  }
  //设置活跃状态
  mActive = newActive;
  //修改活跃数量，活跃 +1，不活跃 -1
  changeActiveCounter(mActive ? 1 : -1);
  //如果状态变成了活跃状态，直接调用 dispatchingValue，传入当前的观察者
  if (mActive) {
    dispatchingValue(this);
  }
}
```

LiveData.changeActiveCounter

```
void changeActiveCounter(int change) {
    int previousActiveCount = mActiveCount;//之前活跃的数量
    mActiveCount += change;//总活跃数量
    if (mChangingActiveState) {//如果正在更改，退出
        return;
    }
    mChangingActiveState = true;//更改中
    try {
        while (previousActiveCount != mActiveCount) {
            boolean needToCallActive = previousActiveCount == 0 && mActiveCount > 0;
```

```
        boolean needToCallInactive = previousActiveCount > 0 && mActiveCount == 0;
        previousActiveCount = mActiveCount;
        //如果当前是第一个激活的，调用 onActive
        if (needToCallActive) {
           onActive();//当活动的观察者从0 变成 1的时候调用
        //如果没有激活的为 0 ，调用 onInactive
        } else if (needToCallInactive) {
           onInactive();//懂活跃的观察者变成 0 时调用
        }
      }
    } finally {
      mChangingActiveState = false;
    }
}
```

LiveData.dispatchingValue 分发数据

```
void dispatchingValue(@Nullable ObserverWrapper initiator) {
 //如果正在分发，退出
 if (mDispatchingValue) {
     mDispatchInvalidated = true;
     return;
  }
  mDispatchingValue = true;
  do {
     mDispatchInvalidated = false;
     //如果观察者不为空
     if (initiator != null) {
        considerNotify(initiator);
        initiator = null;
     } else {
      //如果为空，遍历所有的观察者，将数据发送给所有观察者
        for (Iterator<Map.Entry<Observer<? super T>, ObserverWrapper>> iterator =
             mObservers.iteratorWithAdditions(); iterator.hasNext(); ) {
           considerNotify(iterator.next().getValue());
           if (mDispatchInvalidated) {
              break;
           }
        }
     }
  } while (mDispatchInvalidated);
  mDispatchingValue = false;
}
```

LiveData.considerNotify 对数据进行派发，通知观察者

```
private void considerNotify(ObserverWrapper observer) {
    //如果是不活跃状态，退出
    if (!observer.mActive) {
        return;
    }
    //再次进行判断，如果是不活跃的，则会更新状态，然后退出
    if (!observer.shouldBeActive()) {
        observer.activeStateChanged(false);
        return;
    }
    //观察者版本是否低于当前的版本
    //初始值 mLastVersion 为 -1，mVersion 为 0
    //小于等于表示没有需要更新的数据
    if (observer.mLastVersion >= mVersion) {
        return;
    }
    //更新版本
    observer.mLastVersion = mVersion;
    //通知观察者
    observer.mObserver.onChanged((T) mData);
}
```

我们来梳理一下上面的流程：

我们通过 observe 添加一个观察者，这个观察者会被 LifecycleBoundObserver 进行一个封装，LifecycleBoundObserver继承 ObserverWrapper，并且实现了 LifecycleEventObserver。之后就会将观察添加到 Observers 中，最后注册页面生命周期的 observer。

当生命周期发生变化后，就会回调到 LifecycleBoundObserve 中的 onStateChanged 方法中，如果是销毁了，就会移除观察者，如果不是就会循环更新当前状态。

在更新状态的时候就会判断是否为活跃状态，如果是活跃状态就会进行分发，分发的时候如果观察者为 null ，就会遍历所有的观察者进行分发，否则就分发传入的观察者。

最后会再次判断活跃状态，已经判断观察者版本是否低于当前版本，如果都满足，就会更新观察者。

其中：

- onActive 方法会在活动的观察者从 0 变成 1 的时候调用
- onInactive 方法会在活动的观察者从 1 变成 0 的时候调用

添加观察者：observeForever

另外，除了 observe 观察以外，还有 observeForever ，这种方式不会进行感知生命周期，具体的流程如下：

```
public void observeForever(@NonNull Observer<? super T> observer) {
   assertMainThread("observeForever");
   //和上面不同的是，这里用的是 AlwaysActiveObserver 进行包装
   AlwaysActiveObserver wrapper = new AlwaysActiveObserver(observer);
   ObserverWrapper existing = mObservers.putIfAbsent(observer, wrapper);
   //...
   // 设置为活跃状态
   wrapper.activeStateChanged(true);
}
```

LiveData.AlwaysActiveObserver

```
private class AlwaysActiveObserver extends ObserverWrapper {

   AlwaysActiveObserver(Observer<? super T> observer) {
      super(observer);
   }

  //判断活跃状态的时候直接为 true
   @Override
   boolean shouldBeActive() {
      return true;
   }
}
```

ObserverWrapper

```
//observeForever添加的观察者，全部都是活跃的
void activeStateChanged(boolean newActive) {
   if (newActive == mActive) {
      return;
   }
```

```
    mActive = newActive;
    changeActiveCounter(mActive ? 1 : -1);
    if (mActive) {
        //分发
        dispatchingValue(this);
    }
}
```

其他的都一样，没啥可看的了。

发送数据 setValue

```
protected void setValue(T value) {
    assertMainThread("setValue");
    mVersion++; //livedata 的版本 +1
    mData = value; // 保存数据
    dispatchingValue(null); //分发数据
}


void dispatchingValue(@Nullable ObserverWrapper initiator) {
    //.......
    do {
        mDispatchInvalidated = false;
        if (initiator != null) {
            considerNotify(initiator);
            initiator = null;
        } else {
            //分发给所有观察者
            for (Iterator<Map.Entry<Observer<? super T>, ObserverWrapper>> iterator =
                    mObservers.iteratorWithAdditions(); iterator.hasNext(); ) {
                considerNotify(iterator.next().getValue());
                if (mDispatchInvalidated) {
                    break;
                }
            }
        }
    } while (mDispatchInvalidated);
    mDispatchingValue = false;
}


private void considerNotify(ObserverWrapper observer) {
    if (!observer.mActive) {
        return;
    }
    //是否活跃状态
    if (!observer.shouldBeActive()) {
```

```
        observer.activeStateChanged(false);
        return;
    }
    //判断版本，如果发过了直接退出
    if (observer.mLastVersion >= mVersion) {
        return;
    }
    observer.mLastVersion = mVersion;
    observer.mObserver.onChanged((T) mData);
}
```

整个过程还是非常简单的，如果是非活跃状态，就不会发送数据，当生命周期变为活跃状态时，就会自行分发数据。

发送数据 postValue

```
protected void postValue(T value) {
    boolean postTask;
    //同步待发送的数据
    synchronized (mDataLock) {
        postTask = mPendingData == NOT_SET; //没有待办 true
        mPendingData = value;
    }
    if (!postTask) {//有待办直接 return
        return;
    }
    //将runnable 扔到主线程执行
    ArchTaskExecutor.getInstance().postToMainThread(mPostValueRunnable);
}


private final Runnable mPostValueRunnable = new Runnable() {
    @SuppressWarnings("unchecked")
    @Override
    public void run() {
        Object newValue;
        //同步数据
        synchronized (mDataLock) {
            newValue = mPendingData;
            mPendingData = NOT_SET; //设置为没有待办
        }
        //发送数据
        setValue((T) newValue);
    }
};
```

/ LiveData 相关操作 /

MutableLiveData

由于 LiveData 的发送数据方法是 protected 修饰的，所以不能直接调用。就有了 MutableLiveData 继承 LiveData 将发送数据的方法改为了 public

```
public class MutableLiveData<T> extends LiveData<T> {
    /**
     * Creates a MutableLiveData initialized with the given {@code value}.
     *
     * @param value initial value
     */
    public MutableLiveData(T value) {
        super(value);
    }
    /**
     * Creates a MutableLiveData with no value assigned to it.
     */
    public MutableLiveData() {
        super();
    }

    @Override
    public void postValue(T value) {
        super.postValue(value);
    }

    @Override
    public void setValue(T value) {
        super.setValue(value);
    }
}
```

扩展 LiveData

如果观察者的生命周期处于 STARTED 或者 RESUMED 状态，LiveData 就会认为观察者处于活跃状态，我们可以通过重写 LiveData 来判断当前是否为活跃状态。

例如页面上的一些操作需要在活跃状态下进行，如下载数据，或者需要实时的监听后台数据，就可以重新下面方法，并完成对应逻辑。

```
class StockLiveData(url: String) : LiveData<String>() {
   private val downloadManager = DownloadManager(url)

   override fun onActive() {
    if(!downloadManager.isStart()){
       downloadManager.start()
    }
   }

   override fun onInactive() {
      downloadManager.stop()
   }
}
```

当具有活跃的观察者时，就会调用 onActive 方法。

当没有任何活跃的观察者时，就会调用 onInactive 方法。

当然这只是我想到的场景，开发中可以根据不同的业务场景做出不同的判断。

转换 LiveData

Transformations.map()

在数据分发给观察者之前对其中存储的值进行更改，返回一个新的 LiveData，可以使用此方法。

```
 val strLiveData = MutableLiveData<String>()
 val strLengthLiveData = Transformations.map(strLiveData) {
   it.length
 }
 strLiveData.observe(this) {
  Log.e("---345---> str: ", "$it");
 }
 strLengthLiveData.observe(this) {
  Log.e("---345---> strLength: ", "$it");
 }
strLiveData.value = "hello word"
```

```
E/---345---> str: : hello word
E/---345---> strLength: : 10
```

Transformations.switchMap()

相当于对上面的做了个判断，根据不同的需求返回不同的 LiveData。

```
val idLiveData = MutableLiveData<Int>()

val userLiveData = Transformations.switchMap(idLiveData) { id->
    getUser(id)
}
```

也可以通过 id 去判断，返回对应的 livedata 即可。

合并多个 LiveData

```
val live1 = MutableLiveData<String>()
val live2 = MutableLiveData<String>()

val mediator = MediatorLiveData<String>()
mediator.addSource(live1) {
 mediator.value  = it
 Log.e("---345---> live1", "$it");
}
mediator.addSource(live2){
 mediator.value  = it
 Log.e("---345---> live2", "$it");
}
mediator.observe(this, Observer {
 Log.e("---345---> mediator", "$it");
})
live1.value = "hello"


E/---345---> mediator: hello
E/---345---> live1: hello
```

通过 MediatorLiveData 将两个 MutableLiveData 合并到一起，这样当任何一个发生变化，MediatorLiveData 都可以感知到。

/ 相关问题 /

LiveData 发送的数据是粘性的

例如再没有观察者的时候发送数据，此时 mVersion +1，等到真正添加了观察者后，生命周期也是活跃的，那么就会将这个数据重新分发到观察者。所以说发送数据这个操作数粘性的。

如果需要去除粘性事件，可以再添加完 observe 后去通过反射修改 mVersion 和 观察者包装类中的 mLastVersion 的值，将 mVersion 赋值给 mLastVersion 即可去掉粘性事件。

数据倒灌现象

一般情况下，LiveData 都是存放在 ViewModel 中的，当Activity重建的时候，观察者会被 remove 掉，重建后会添加一个新的观察者，添加后新的观察者版本号就是 -1，所以就会出现数据再次被接收到。

这种解决方式和上面一样，反射修改版本号就可以解决。

非活跃状态的观察者转为活跃状态后，只能接收到最后一次发送的数据。

一般情况下我们都需要的是最新数据，如果非要所有数据，只能重写 LiveData 了。

推荐阅读：

我的新书，《第一行代码 第3版》已出版！

Android 13 Developer Preview一览

一个解决滑动冲突的新思路，无缝嵌套滑动

▼

长按上图，识别图中二维码即可关注

阅读原文

喜欢此内容的人还喜欢

零日漏洞：暴利且危险

数世咨询

76 张图，剖析 Spring AOP 源码，小白居然也能看懂，大神，请收下我的膝盖！

楼仔

uber-go/dig 源码阅读

golang算法架构leetcode技术php