[Урок 3 LiveData Дополнительные возможности](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/526-urok-3-livedata.html)

В прошлом уроке мы разобрались, как отправлять и получать данные в LiveData. В этом уроке рассмотрим несколько дополнительных возможностей. Как преобразовать тип данных с помощью map. Как создать свой LiveData. Как объединить несколько LiveData в один с помощью MediatorLiveData.

Transformations

Вы можете поменять типа данных в LiveData с помощью [Transformations.map](https://developer.android.com/reference/android/arch/lifecycle/Transformations.html" \l "map(android.arch.lifecycle.LiveData%3CX%3E,%20android.arch.core.util.Function%3CX,%20Y%3E)" \t "_blank).

Рассмотрим пример, в котором LiveData<String> будем превращать в LiveData<Integer>:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | LiveData<String> liveData = ...;    LiveData<Integer> liveDataInt = Transformations.map(liveData, new Function<String, Integer>() {     @Override     public Integer apply(String input) {         return Integer.parseInt(input);     }  }); |

В метод map передаем имеющийся LiveData<String> и функцию преобразования. В этой функции мы будем получать String данные из LiveData<String>, и от нас требуется преобразовать их в Integer. В данном случае просто парсим строку в число.

На выходе метода map получим LiveData<Integer>. Можно сказать, что он подписан на LiveData<String> и все получаемые String значения будет конвертировать в Integer и рассылать уже своим подписчикам.

Рассмотрим более сложный случай. У нас есть LiveData<Long>, нам необходимо из него получить LiveData<User>. Конвертация id в User выглядит так:

|  |  |
| --- | --- |
| 1  2  3 | private LiveData<User> getUser(long id) {     // ...  } |

По id мы получаем LiveData<User> и на него надо будет подписываться, чтобы получить объект User.

В этом случае мы не можем использовать метод map, т.к. мы получим примерно такой результат:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | LiveData<Long> liveDataId = ...;    LiveData<LiveData<User>> liveDataUser = Transformations.map(liveDataId, new Function<Long, LiveData<User>>() {     @Override     public LiveData<User> apply(Long id) {         return getUser(id);     }  }); |

На выходе будет объект LiveData<LiveData<User>>. Чтобы избежать этого, используем [switchMap](https://developer.android.com/reference/android/arch/lifecycle/Transformations.html" \l "switchMap(android.arch.lifecycle.LiveData%3CX%3E,%20android.arch.core.util.Function%3CX,%20android.arch.lifecycle.LiveData%3CY%3E%3E)" \t "_blank) вместо map.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | LiveData<Long> liveDataId = ...;    LiveData<User> liveDataUser = Transformations.switchMap(liveDataId, new Function<Long, LiveData<User>>() {     @Override     public LiveData<User> apply(Long id) {         return getUser(id);     }  }); |

switchMap уберет вложенность LiveData и мы получим LiveData<User>.

Свой LiveData

В некоторых ситуациях удобно создать свою обертку LiveData.

Рассмотрим пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class LocationLiveData extends LiveData<Location> {       LocationService.LocationListener locationListener = new LocationService.LocationListener() {         @Override         public void onLocationChanged(Location location) {             setValue(location);         }     };       @Override     protected void onActive() {         LocationService.addListener(locationListener);     }       @Override     protected void onInactive() {         LocationService.removeListener(locationListener);     }    } |

Класс LocationLiveData расширяет LiveData<Location>.

Внутри него есть некий locationListener - слушатель, который можно передать в LocationService и получать обновления текущего местоположения. При получении нового Location от LocationService, locationListener будет вызывать метод setValue и тем самым обновлять данные этого LiveData.

LocationService - это просто какой-то сервис, который предоставляет нам текущую локацию. Его реализация в данном примере не важна. Главное - это то, что мы подписываемся (addListener) на сервис, когда нам нужны данные, и отписываемся (removeListener), когда данные больше не нужны.

Обратите внимание, что мы переопределили методы [onActive](https://developer.android.com/reference/android/arch/lifecycle/LiveData.html" \l "onActive()" \t "_blank) и [onInactive](https://developer.android.com/reference/android/arch/lifecycle/LiveData.html" \l "onInactive()" \t "_blank). onActive будет вызван, когда у LiveData появится хотя бы один подписчик. А onInactive - когда не останется ни одного подписчика. Соответственно эти методы удобно использовать для подключения/отключения нашего слушателя к LocationService.

Получилась удобная обертка, которая при появлении подписчиков сама будет подписываться к LocationService, получать Location и передавать его своим подписчикам. А когда подписчиков не останется, то LocationLiveData отпишется от LocationService.

Осталось сделать из LocationLiveData синглтон и можно использовать его в разных Activity и фрагментах.

MediatorLiveData

[MediatorLiveData](https://developer.android.com/reference/android/arch/lifecycle/MediatorLiveData.html) дает возможность собирать данные из нескольких LiveData в один. Это удобно если у вас есть несколько источников из которых вы хотите получать данные. Вы объединяете их в один и подписываетесь только на него.

Рассмотрим, как это делается, на простом примере.

|  |  |
| --- | --- |
| 1  2  3  4 | MutableLiveData<String> liveData1 = new MutableLiveData<>();  MutableLiveData<String> liveData2 = new MutableLiveData<>();    MediatorLiveData<String> mediatorLiveData = new MediatorLiveData<>(); |

У нас есть два LiveData<String>: liveData1 и liveData2. Мы хотим объединить их в один. Для этого нам понадобится MediatorLiveData.

Добавляем LiveData к MediatorLiveData

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | mediatorLiveData.addSource(liveData1, new Observer<String>() {     @Override     public void onChanged(@Nullable String s) {         mediatorLiveData.setValue(s);     }  });      mediatorLiveData.addSource(liveData2, new Observer<String>() {     @Override     public void onChanged(@Nullable String s) {         mediatorLiveData.setValue(s);       }  }); |

Метод [addSource](https://developer.android.com/reference/android/arch/lifecycle/MediatorLiveData.html" \l "addSource(android.arch.lifecycle.LiveData%3CS%3E,%20android.arch.lifecycle.Observer%3CS%3E)" \t "_blank) требует от нас два параметра.

Первый - это LiveData из которого MediatorLiveData собирается получать данные.

Второй параметр - это колбэк, который будет использован для подписки на LiveData из первого параметра. Обратите внимание, что в колбэке нам надо самим передавать в MediatorLiveData данные, получаемые из LiveData. Это делается методом setValue.

Таким образом mediatorLiveData будет получать данные из двух LiveData и постить их своим получателям.

Подпишемся на mediatorLiveData

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | mediatorLiveData.observe(this, new Observer<String>() {     @Override     public void onChanged(@Nullable String s) {         log("onChanged " + s);     }  }); |

Сюда теперь должны приходить данные из liveData1 и liveData2. Будем их просто логировать.

Отправим данные в liveData1 и liveData2:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | liveData1.setValue("1");  liveData2.setValue("a");  liveData1.setValue("2");  liveData2.setValue("b");  liveData1.setValue("3");  liveData2.setValue("c"); |

Смотрим лог:

*onChanged 1*  
*onChanged a*  
*onChanged 2*  
*onChanged b*  
*onChanged 3*  
*onChanged c*

Все данные, что мы передавали в liveData1 и liveData2 пришли в общий mediatorLiveData.

Немного усложним пример. Допустим, нам надо отписаться от liveData2, когда из него придет значение "finish".

Код подписки mediatorLiveData на liveData1 и liveData2 будет выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | mediatorLiveData.addSource(liveData1, new Observer<String>() {     @Override     public void onChanged(@Nullable String s) {         mediatorLiveData.setValue(s);     }  });      mediatorLiveData.addSource(liveData2, new Observer<String>() {     @Override     public void onChanged(@Nullable String s) {         if ("finish".equalsIgnoreCase(s)) {             mediatorLiveData.removeSource(liveData2);             return;         }         mediatorLiveData.setValue(s);       }  }); |

В случае с liveData1 ничего не меняется.

А вот при получении данных от liveData2 мы смотрим, что за значение пришло. Если это значение "finish", то методом [removeSource](https://developer.android.com/reference/android/arch/lifecycle/MediatorLiveData.html" \l "removeSource(android.arch.lifecycle.LiveData%3CS%3E)" \t "_blank) отписываем mediatorLiveData от liveData2 и не передаем это значение дальше.

Отправим несколько значений

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | liveData1.setValue("1");  liveData2.setValue("a");  liveData2.setValue("finish");  liveData1.setValue("2");  liveData2.setValue("b");  liveData1.setValue("3");  liveData2.setValue("c"); |

liveData2 отправляет здесь значения "a", "finish", "b" и "c". Через mediatorLiveData должно пройти только "a". А значение из liveData1 должны пройти все.

Запускаем, смотрим лог:

*onChanged 1*  
*onChanged a*  
*onChanged 2*  
*onChanged 3*

Все верно. При получении "finish" от liveData2, mediatorLiveData отписался от него и последующие его данные мы уже не получали.

RxJava

Мы можем конвертировать LiveData в Rx и наоборот. Для этого есть инструмент [LiveDataReactiveStreams](https://developer.android.com/reference/android/arch/lifecycle/LiveDataReactiveStreams.html" \t "_blank).

Чтобы его использовать добавьте в dependencies:

|  |  |
| --- | --- |
| 1 | implementation "android.arch.lifecycle:reactivestreams:1.0.0" |

Чтобы получить LiveData из Flowable или Observable, используем метод [fromPublisher](https://developer.android.com/reference/android/arch/lifecycle/LiveDataReactiveStreams.html" \l "fromPublisher(org.reactivestreams.Publisher%3CT%3E)" \t "_blank):

|  |  |
| --- | --- |
| 1  2 | Flowable<String> flowable = ... ;  LiveData<String> liveData = LiveDataReactiveStreams.fromPublisher(flowable); |

LiveData будет подписан на Flowable, пока у него (у LiveData) есть подписчики.

LiveData не сможет обработать или получить onError от Flowable. Если в Flowable возникнет ошибка, то будет крэш.

Неважно в каком потоке работает Flowable, результат в LiveData всегда придет в UI потоке.

Чтобы получить Flowable или Observable из LiveData нужно выполнить два преобразования. Сначала используем метод [toPublisher](https://developer.android.com/reference/android/arch/lifecycle/LiveDataReactiveStreams.html" \l "toPublisher(android.arch.lifecycle.LifecycleOwner,%20android.arch.lifecycle.LiveData%3CT%3E)" \t "_blank), чтобы получить Publisher. Затем полученный Publisher передаем в метод Flowable.fromPublisher:

|  |
| --- |
| LiveData<String> liveData = … ;  Flowable<String> flowable = Flowable.fromPublisher(          LiveDataReactiveStreams.toPublisher(this, liveData)); |

Прочие методы LiveData

[hasActiveObservers()](https://developer.android.com/reference/android/arch/lifecycle/LiveData.html#hasActiveObservers()) - проверка наличия активных подписчиков

[hasObservers()](https://developer.android.com/reference/android/arch/lifecycle/LiveData.html#hasObservers()) - проверка наличия любых подписчиков

[observeForever (Observer<T> observer)](https://developer.android.com/reference/android/arch/lifecycle/LiveData.html#observeForever(android.arch.lifecycle.Observer%3CT%3E)) - позволяет подписаться без учета Lifecycle. Т.е. этот подписчик будет всегда считаться активным.

[removeObserver (Observer<T> observer)](https://developer.android.com/reference/android/arch/lifecycle/LiveData.html#removeObserver(android.arch.lifecycle.Observer%3CT%3E)) - позволяет отписать подписчика

[removeObservers (LifecycleOwner owner)](https://developer.android.com/reference/android/arch/lifecycle/LiveData.html#removeObservers(android.arch.lifecycle.LifecycleOwner)) - позволяет отписать всех подписчиков, которые завязаны на Lifecycle от указанного LifecycleOwner.