DiffUtil RecyclerView

Упрощаем обновление списка

В предыдущей статье мы породили баг : если список пустой, то мы при добавлении шутки в список сразу же видели изменения. А при удалении из списка не отображалась текстовка ошибки с фиолетовым фоном. Почему? Потому что исключение о пустом списке в кеше у нас в дата слое и получается при вызове метода получения списка. Так же нам требуется позиция элемента чтобы передать ее в адаптер ресайкла чтобы тот обновился. Но кто-то скажет : а нет более простого решения? Мы должны добавить или удалить элемент и после вызывать разные методы и находить позицию. Было бы классно чтобы кто-то сам занимался этим вопросом и облегчил нам жизнь. И да, такое решение есть. Называется DiffUtilCallback.

Давайте перепишем код таким образом чтобы это было возможно.

```
class CommonDiffUtilCallback : DiffUtil.Callback() {
    override fun getOldListSize(): Int {
        TODO( reason: "Not yet implemented")
    }

    override fun getNewListSize(): Int {
        TODO( reason: "Not yet implemented")
    }

    override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {
        TODO( reason: "Not yet implemented")
    }

    override fun areContentsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {
        TODO( reason: "Not yet implemented")
    }
}
```

Во-первых нам нужно создать класс, который наследуется от DiffUtil.Callback. И как видите нам нужно определить 4 метода. 2 из них простые : размер старого списка и размер нового списка. Мы просто передадим в конструктор старый список и новый списки и у них вызовем методы получения размера. Нас сейчас больше интересуют методы areItemsTheSame и areContentsTheSame.

Давайте подробнее остановимся на каждом из них чтобы вы понимали разницу.

Просто откроем исходный код и почитаем джавадок. Для этого просто перейдем по стрелочке вверх слева. Итак, вызывается чтобы понять что 2 элемента одинаковы, т.е. одно и то же. Зачем это нужно? Сейчас если вы напишите код который просто удалит весь список и поставит новый то будет некрасиво : без анимации замена всего списка. Мы используем дифутилс для того, чтобы это было красиво при изменении элементов. И потому нам нужно понимание что элементы одинаковые, т.е. элемент тот же что и был. Если мы просто напишем false для этого метода то при изменении вы увидете мерцание : перерисовку элементов. А если мы напишем правильный код, то там где в новом списке те же элементы

что и были в старом списке они не будут меняться. Теперь как видите сама документация говорит о том, что нужно сравнивать айди как уникальные поля элементов.

```
/**
 * Called by the DiffUtil to decide whether two object represent the same Item.
 * <p>
 * For example, if your items have unique ids, this method should check their id equality.
 *
 * @param oldItemPosition The position of the item in the old list
 * @param newItemPosition The position of the item in the new list
 * @return True if the two items represent the same object or false if they are different.
 */
public abstract boolean areItemsTheSame(int oldItemPosition, int newItemPosition);
```

Давайте напишем код сейчас для этого метода.

```
class CommonDiffUtilCallback<E>(
    private val oldList:List<CommonUiModel<E>>,
    private val newList:List<CommonUiModel<E>>
) : DiffUtil.Callback() {
    override fun getOldListSize() = oldList.size

    override fun getNewListSize() = newList.size

    override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {
        return oldList[oldItemPosition].same(newList[newItemPosition])
    }
```

Как видите я просто передал списки в конструктор и у нас 2 разных элемента с разными типами айди у поля то и у нас будет дженерик класс. И самое интересное, что айди у классов приватные, значит нам нужен метод для сравнения. Давайте напишем метод у обьекта CommonUiModel. Так как у нас 3 наследника у абстрактного класса то напишем опен метод с дефолтным фолс. Ведь айди есть только у избранного элемента

```
abstract class CommonUiModel<T>(private val first: String, private val second: String) {
    open fun same(model: CommonUiModel<T>): Boolean = false
```

И напишем для начала реализацию для избранного

```
class FavoriteCommonUiModel<E>(private val id: E, text: String, punchline: String) :
    CommonUiModel<E>(text, punchline) {
    override fun same(model: CommonUiModel<E>): Boolean {
        return model is FavoriteCommonUiModel<E> && model.id == id
    }
```

Поле айди только у избранного и потому нам нужно проверить что имеем дело с этим же наследником. Да, у нас проверка через is и это не самое красивое решение. Но давайте пока оставим как есть.

И теперь можем вернуться в дифутилс и разобраться со вторым методом

```
/**
 * Called by the DiffUtil when it wants to check whether two items have the same data.
 * DiffUtil uses this information to detect if the contents of an item has changed.
 * <p>
 * DiffUtil uses this method to check equality instead of {@link Object#equals(Object)}
 * so that you can change its behavior depending on your UI.
 * For example, if you are using DiffUtil with a
 * {@link RecyclerView.Adapter RecyclerView.Adapter}, you should
 * return whether the items' visual representations are the same.
 * <p>
 * This method is called only if {@link #areItemsTheSame(int, int)} returns
 * {@code true} for these items.
 *
 * @param oldItemPosition The position of the item in the old list
 * @param newItemPosition The position of the item in the new list which replaces the
 *                        oldItem
 * @return True if the contents of the items are the same or false if they are different.
 */
public abstract boolean areContentsTheSame(int oldItemPosition, int newItemPosition);
```

И здесь мы понимаем что в нашем конкретном случае этот метод не нужен. Мы лишь добавляем полностью новый элемент в список и удаляем его. Но метод сравнения контента нам бы понадобился если бы мы имели состояния к примеру. Предположим я бы написал изменения в юай для элемента и в зависимости от значения полей бы менял что-то. Например если вы смотрите мои стримы на ютуб, то там сейчас в списке есть схлопывающиеся и расхлопывающиеся заголовки. И там у нас будет метод сравнения контента: схлопнут ли заголовок или нет. Чтобы его не перерисовывать полностью и не мерцать мы заюзаем этот метод. А в текущей ситуации давайте просто там напишем return false. Но если хотите напишите методы которые сравнивают поля класса кроме айди.

```
class CommonDiffUtilCallback<E>(
    private val oldList: List<CommonUiModel<E>>,
    private val newList: List<CommonUiModel<E>>
) : DiffUtil.Callback() {
    override fun getOldListSize() = oldList.size
    override fun getNewListSize() = newList.size
    override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int) =
        oldList[oldItemPosition].same(newList[newItemPosition])
    override fun areContentsTheSame(oldItemPosition: Int, newItemPosition: Int) = false
}
```

Итого для нашей ситауции дифутилколбек будет выглядеть таким образом. Что же дальше? Нам нужно использовать его в адаптере. Но как?

```
fun update() {
    val result = communication.getDiffResult()
    result.dispatchUpdatesTo( adapter: this)
}
```

Для начала я переписал метод обновления таким образом, что мы получаем результат изменения и обновляем адаптер. Теперь никаких notifyItemInserted, notifyItemRemoved, notifyDataSetChanged и там еще несколько методов еще для случаев если не один элемент изменился а несколько сразу. Т.е. я удалил второй метод update с аргументом из адаптера. Ок, давайте тогда посмотрим как должен выглядеть метод в комуникации

```
private lateinit var diffResult: DiffUtil.DiffResult
override fun getDiffResult() = diffResult
```

У меня просто переменная в которую сохраним разницу и вернем в адаптер. Мы бы могли написать метод типа dispatchResult(adapter:RecyclerView.Adapter) и в адаптере передать себя комуникациям. Сделайте так если хотите. Особо разницы нет конечно, кроме той, что у нас метод называется гет и мне оно не нравится. Ну да ладно. Теперь нам нужно переписать метод обновления списка в комуникациях.

```
override fun showDataList(list: List<CommonUiModel<T>>) {
    val callback = CommonDiffUtilCallback( oldList: listLiveData.value ?: emptyList(), list)
    diffResult = DiffUtil.calculateDiff(callback)
    listLiveData.value = ArrayList(list)
}
```

Там где я постю новый список просто создам разницу из старого списка (текущее значение или пустой список) и нового списка. Там где у меня обсервится ливдата будет вызываться метод апдейт у адаптера и вызовется метод получения дифрезалт. Единственное нам нужно теперь чтобы методы добавления и удаления элементов возвращали список избранных и мы их постили в коммуникацию.

fun removeItem(id: T) : Int удалим этот метод и реализацию из коммуникации

```
override fun changeItemStatus(id: T) {
    viewModelScope.launch(dispatcher) { this: CoroutineScope
        interactor.removeItem(id)
        communication.showDataList(interactor.getItemList().toUiList())
    }
}
```

В принципе можно вот так: удалили элемент и вызвали метод получения списка за ним. Не обязательно переписывать все методы у интерактора репозитория и т.д.
И в принципе это все что мы должны были сделать. Запустим проект и посмотрим.
И теперь как видите у нас плавные изменения в списке и когда не остается элементов в избранном то возвращается полноценная ошибка на фиолетовом фоне.

Пока что это все что вам достаточно знать о дифутилс. Если в вашем ресайклере есть добавление и/или удаление элементов и вы не хотите сами находить позиции и удалять их и добавлять, то просто используйте дифутилс.