Практическая работа. DiffUtil

Приложение должно быть настолько плавным и быстрым, насколько это возможно. Но если действовать неосторожно, большие и сложные наборы данных могут привести к сбоям в вашем представлении с переработкой. В этой работе мы представим `DiffUtil`: вспомогательный класс, который расширяет возможности представлений с переработкой. Вы научитесь использовать его для эффективного обновления представлений с переработкой. Вы узнаете, как объекты `ListAdapter` упрощают работу с `DiffUtil`. А заодно будет показано, как полностью избавиться от `findViewById()` реализацией связывания данных в коде представления с переработкой.

Представление с переработкой правильно отображает данные задач

В предыдущей работе мы добавили в приложение `Tasks` представление с переработкой, которое отображает свои данные точно так, как нам требуется. Каждая задача отображается на отдельной карточке, а на каждой карточке выводится имя задачи и признак ее завершения. Карточки выстраиваются в сетку из двух столбцов:

но при обновлении данных в представлении с переработкой происходит переход

Каждый раз, когда мы добавляем новую задачу, представление с переработкой перерисовывается, чтобы оно включало новую запись и оставалось актуальным. Однако при этом в представлении с переработкой происходит резкий переход. Каждый раз, когда возникает необходимость в обновлении представления с переработкой, перерисовывается весь список. Нет плавных переходов, которые бы указывали на изменения, и если список окажется очень длинным, пользователь перестает понимать, в какой позиции он сейчас находится. Кроме того, такой подход неэффективен для больших наборов данных и может привести к проблемам с быстродействием. Прежде чем браться за решение этих проблем, вспомним структуру приложения `Tasks`.

Снова о приложении Tasks

Как говорилось ранее, приложение `Tasks` предоставляет пользователю возможность вводить записи, которые сохраняются в базе данных `Room`. Приложение включает представление с переработкой, в котором отображаются все введенные записи. Главный экран приложения определяется фрагментом `TasksFragment`, который использует модель представления с именем `TasksViewModel`. Его макет — `fragment_tasks.xml` — включает представление с переработкой, отображающее сетку задач. Представление с переработкой использует адаптер с именем `TaskItemAdapter`, а размещение его элементов определяется файлом макета `task_item.xml`. Эти части приложения взаимодействуют по следующей схеме:

Как говорилось ранее, приложение `Tasks` предоставляет пользователю возможность вводить записи, которые сохраняются в базе данных Room. Приложение включает представление с переработкой, в котором отображаются все введенные записи. Главный экран приложения определяется фрагментом `TasksFragment`, который использует модель представления с именем `TasksViewModel`. Его макет — `fragment_tasks.xml` — включает представление с переработкой, отображающее сетку задач. Представление с переработкой использует адаптер с именем `TaskItemAdapter`, а размещение его элементов определяется файлом макета `task_item.xml`. Эти части приложения взаимодействуют по следующей схеме:

Как представление с переработкой получает свои данные

Когда возникает необходимость в обновлении данных представления с переработкой, происходит следующее:

1. `TasksFragment` оповещается о добавлении записи в базу данных. Это происходит из-за того, что он наблюдает за свойством tasks объекта `TasksViewModel`: списком `LiveData<List<Task>>`, который получает свои данные из базы данных.
2. `TasksFragment` задает свойство data объекта `TaskItemAdapter`, который содержит данные представления с переработкой. Ему присваивается новый список `List<Task>` (который он получает из свойства `tasks`), который включает последние изменения в записях.
3. `TaskItemAdapter` оповещает представление с переработкой о том, что данные изменились. Представление с переработкой реагирует на оповещение перерисовкой и повторным связыванием каждого элемента в списке.

Метод записи свойства data вызывает notifyDataSetChanged()

Представление с переработкой перерисовывает и заново связывает весь список из-за метода записи, который был добавлен к свойству data объекта `TaskItemAdapter`. Напомним, как выглядит код:

```
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>()
{
 var data = listOf<Task>()
 set(value) {
 field = value
 notifyDataSetChanged()
 }
 ...
}
```

Метод записи вызывается каждый раз, когда потребуется обновить свойство `data`. Как видно из листинга, он присваивает новое значение свойству `data`, а затем вызывает `notifyDataSetChanged()`. Этот метод сообщает всем наблюдателям, включая представление с переработкой, о том, что набор данных изменился, поэтому представление с переработкой перерисовывается с учетом последних изменений.

`notifyDataSetChanged()` перерисовывает весь список

Тем не менее использование `notifyDataSetChanged()` создает проблемы. При каждом вызове метод сообщает, что свойство data устарело в какомто отношении, но не указывает, в каком именно. Так как представление с переработкой не знает, что изменилось, оно реагирует на оповещение повторным связыванием и перерисовкой каждого элемента в списке. Когда все представление с переработкой заново связывает и перерисовывает элементы подобным образом, оно теряет текущую позицию пользователя в списке. Если список содержит хотя бы десяток записей, это может привести к переходу и смене отображаемой части списка. Кроме того, такой подход неэффективен для больших наборов

данных. Если представление с переработкой содержит много элементов, повторное связывание и перерисовка создают большой объем лишней работы и могут вызывать проблемы с быстродействием.

Передача представлению с переработкой информации о том,что должно измениться

В другом, более эффективном варианте метод `notifyDataSetChanged()` вызывается для оповещения представления с переработкой о том, какие элементы в списке изменились, чтобы представление обновило только эти элементы. Например, если в базу данных добавляется новая запись, то для представления с переработкой будет более эффективно просто добавить новый элемент, чем заново связывать и перерисовывать весь список. Проверять все изменения вручную было бы слишком хлопотно и потребовало бы слишком большого объема кода. К счастью, библиотека представлений с переработкой включает вспомогательный класс с именем `DiffUtil`, который выполняет всю черную работу за вас.

DiffUtil определяет отличия между списками

Класс `DiffUtil` специализируется на определении отличий между двумя списками, чтобы вам не приходилось заниматься этой работой. Каждый раз, когда адаптеру передается новая версия списка, используемого его представлением с переработкой, `DiffUtil` сравнивает ее со старой версией. Класс определяет, какие элементы были добавлены, удалены или обновлены, и сообщает представлению с переработкой, что следует изменить, наиболее эффективным способом из всех возможных:

Класс `DiffUtil` специализируется на определении отличий между двумя списками, чтобы вам не приходилось заниматься этой работой. Каждый раз, когда адаптеру передается новая версия списка, используемого его представлением с переработкой, `DiffUtil` сравнивает ее со старой версией. Класс определяет, какие элементы были добавлены, удалены или обновлены, и сообщает представлению с переработкой, что следует изменить, наиболее эффективным способом из всех возможных:

Что мы собираемся сделать

В этой работе мы внесем изменение в представление с переработкой приложения `Tasks`, чтобы в нем использовался класс `DiffUtil`, а представления заполнялись посредством связывания данных. Эти изменения повышают эффективность представления с переработкой и улучшают впечатления пользователя от работы с ним. Для этого необходимо:

1. Использовать `DiffUtil` в представлении с прокруткой Мы создадим новый класс с именем `TaskDiffItemCallback`, который использует `DiffUtil` для сравнения элементов в списке. Затем мы обновим код `TaskItemAdapter`, чтобы в нем использовался этот новый класс. Эти изменения повышают эффективность представления с переработкой, а работа пользователя с представлением становится более плавной и приятной.

2. Реализовать связывание данных в макете представления с переработкой. Мы удалим вызовы `findViewById()` в коде `TaskItemAdapter` и заполним представления каждого элемента с использованием связывания данных.

Для начала заставим представление с переработкой использовать `DiffUtil`.

Реализация DiffUtil.ItemCallback

Чтобы использовать `DiffUtil` для представления с переработкой приложения `Tasks`, необходимо создать новый класс (назовем его `TaskDiffItemCallback`), реализующий абстрактный класс `DiffUtil.ItemCallback`. Этот класс используется для вычисления разности между двумя элементами списка, отличными от `null`, и помогает значительно повысить эффективность представлений с переработкой. Реализуя `DiffUtil.ItemCallback`, сначала необходимо задать тип объектов, с которыми он работает. Для этого будет использоваться обобщение:

```
class TaskDiffItemCallback : DiffUtil.ItemCallback<Task>()
```

Также необходимо переопределить два метода: areItemsTheSame() и areContentsTheSame(). Метод areItemsTheSame() используется для проверки того, ссылаются ли два переданных ему объекта на один элемент. Для его реализации будет использоваться следующий код:

```
override fun areItemsTheSame(oldItem: Task, newItem: Task)
 = (oldItem.taskId == newItem.taskId)
```

Если оба объекта имеют одинаковые значения `taskId`, это означает, что они ссылаются на один элемент, и метод возвращает `true`. Метод `areContentsTheSame()` используется для проверки совпадения содержимого двух объектов и вызывается только в том случае, если `areItemsTheSame()` возвращает `true`. Так как `Task` является классом данных, этот метод может быть реализован следующим кодом:

```
override fun areContentsTheSame(oldItem: Task, newItem: Task) = (oldItem ==
newItem)
```

Создание TaskDiffItemCallback.kt

Чтобы создать новый класс, выделите пакет `com.hfad.tasks` в папке *app/src/main/java*, затем выберите команду `File→New→Kotlin Class/File`. Введите имя файла «TaskDiffItemCallback» и выберите вариант `Class`. После того как файл будет создан, обновите его код:

```
package com.hfad.tasks
import androidx.recyclerview.widget.DiffUtil
class TaskDiffItemCallback : DiffUtil.ItemCallback<Task>() {
 override fun areItemsTheSame(oldItem: Task, newItem: Task)
 = (oldItem.taskId == newItem.taskId)
 override fun areContentsTheSame(oldItem: Task, newItem: Task) = (oldItem ==
newItem)
}
```

ListAdapter получает аргумент DiffUtil.ItemCallback

После определения `TaskDiffItemCallback` необходимо использовать его в коде адаптера. Для этого мы обновим класс `TaskItemAdapter`, чтобы он расширял класс `ListAdapter` вместо `RecyclerView.Adapter.ListAdapter` — разновидность `RecyclerView.Adapter`, спроектированная для работы со списками. Он предоставляет собственный резервный список, чтобы вам не приходилось определять его самостоятельно, и получает `DiffUtil.ItemCallback` в конструкторе. Мы укажем, что `TaskItemAdapter` является разновидностью `ListAdapter` с собственным списком `List<Task>`, и передадим ему экземпляр `TaskDiffItemCallback`. Для этого используется следующий код:

```
import androidx.recyclerview.widget.ListAdapter
class TaskItemAdapter
 : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {
 ...
}
```

Остальной код TaskItemAdapter можно упростить

После того как класс `TaskItemAdapter` был изменен для расширения `ListAdapter`, можно удалить его свойство `data` типа `List<Task>` вместе с его методом записи. Это свойство стало ненужным, потому что `ListAdapter` содержит собственный резервный список, так что вам не придется определять его самостоятельно. Также можно удалить метод `getItemCount()` объекта `TaskItemAdapter`. Он был необходим, когда адаптер расширял `RecyclerView.Adapter`, но `ListAdapter` предоставляет собственную реализацию, поэтому этот метод стал лишним. Наконец, необходимо обновить метод `onBindViewHolder()` адаптера, чтобы вместо

```
val item = data[position]
```

для получения элемента в определенной позиции свойства `data` использовался код

```
val item = getItem(position)
```

Этот код получает элемент в заданной позиции резервного списка адаптера. Весь этот код будет приведен на следующей странице.

Обновленный код TaskItemAdapter.kt

Ниже приведен обновленный код `TaskItemAdapter`; обновите файл `TaskItemAdapter.kt`:

```
package com.hfad.tasks
import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.CheckBox
import androidx.recyclerview.widget.ListAdapter
```

```
import android.widget.TextView
import androidx.cardview.widget.CardView
import androidx.recyclerview.widget.RecyclerView
class TaskItemAdapter : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>
(TaskDiffItemCallback()) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
 : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)
 override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
    val item = getItem(position)
    holder.bind(item)

 class TaskItemViewHolder(val rootView: CardView) :
RecyclerView.ViewHolder(rootView) {
 ...
 }
}
```

Заполнение списка ListAdapter

Последнее, что осталось сделать, — передать список записей `Task` резервному списку `TaskItemAdapter`. Ранее для этого мы отдавали команду `TasksFragment` наблюдать за свойством `tasks` объекта `TasksViewModel`. Каждый раз, когда свойство изменялось, фрагмент обновлял свойство `data` объекта `TaskItemAdapter` новым значением свойства `tasks`. Напомним, как выглядел код, который для этого использовался:

```
viewModel.tasks.observe(viewLifecycleOwner, Observer {
 it?.let {
 adapter.data = it
 }
})
```

Итак, теперь адаптер использует резервный список вместо свойства `data`, и решение необходимо слегка изменить.

Для передачи списка задач резервному списку `TaskItemAdapter` будет использоваться метод `submitList()`. Этот метод обновляет резервный список `ListAdapter` новым объектом `List`, поэтому он идеально подходит для этой ситуации. Ниже приведен новый код, который необходимо добавить в `TasksFragment`; мы добавим его на следующей странице:

```
viewModel.tasks.observe(viewLifecycleOwner, Observer {
 it?.let {
 adapter.data = it

 }
})
```

Когда адаптер получает новый список, он использует класс `TaskDiffItemCallback` для того, чтобы сравнить его со старой версией. Затем представление с переработкой обновляется найденными различиями — вместо замены всего списка. Такой подход более эффективен, а представление работает более плавно. Давайте посмотрим, как выглядит обновленный код `TasksFragment`.

Обновленный код TasksFragment.kt

Ниже приведен обновленный код TasksFragment; обновите файл `TasksFragment.kt`:

```
class TasksFragment : Fragment() {
 private var _binding: FragmentTasksBinding? = null
 private val binding get() = _binding!!
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 _binding = FragmentTasksBinding.inflate(inflater, container, false)
 val view = binding.root
 val application = requireNotNull(this.activity).application
 val dao = TaskDatabase.getInstance(application).taskDao
 val viewModelFactory = TasksViewModelFactory(dao)
 val viewModel = ViewModelProvider(
 this, viewModelFactory).get(TasksViewModel::class.java)
 binding.viewModel = viewModel
 binding.lifecycleOwner = viewLifecycleOwner
 val adapter = TaskItemAdapter()
 binding.tasksList.adapter = adapter
 viewModel.tasks.observe(viewLifecycleOwner, Observer {
 it?.let {
 adapter.submitList(it)
 }
 })
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
 }
}
```

Посмотрим, что происходит при выполнении приложения.

Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

1. При запуске приложения `MainActivity` отображает `TasksFragment`. `TasksFragment` использует `TasksViewModel` в качестве модели представления.

2. `TasksFragment` создает объект `TaskItemAdapter` и назначает его адаптером для представления с переработкой.

3. `TasksFragment` наблюдает за свойством `tasks` объекта `TasksViewModel`. Это свойство содержит тип `LiveData<List<Task>>`, в котором хранится актуальный список записей из базы данных.

4. Каждый раз, когда свойство tasks получает новое значение, `TasksFragment` передает свой список `List<Task>` адаптеру `TaskItemAdapter`.

5. `TaskItemAdapter` использует `TaskDiffItemCallback` для сравнения старых данных с новыми. Для определения того, что изменилось в данных, используются методы `areItemsTheSame()` и `areContentsTheSame()` объекта `TaskDiffItemCallback`.

6. `TaskItemAdapter` передает представлению с переработкой информацию об изменениях. Представление с переработкой заново связывает и перерисовывает необходимые элементы.

Когда вы запускаете приложение, `TasksFragment`, как и прежде, отображает сетку карточек в представлении с переработкой. Если ввести имя новой задачи и щелкнуть на кнопке, в представление с переработкой добавляется новая карточка задачи, а существующие карточки смещаются, чтобы освободить для нее место.

Представление с переработкой ведет себя подобным образом, потому что вместо замены всего списка для передачи изменений используется класс `DiffUtil`.

Представления с переработкой могут использовать связывание данных

Другое возможное усовершенствование представления с переработкой приложения `Tasks` — переход на связывание данных. Как вы помните, класс `TaskItemViewHolder`, являющийся внутренним классом `TaskItemAdapter`, использует метод `findViewById()` для получения ссылок на представления всех элементов в представлении с переработкой. Затем метод `bind()` держателя представления использует эти ссылки для добавления данных в каждое представление. Напомним, как выглядит код:

```
class TaskItemViewHolder(val rootView: CardView)
 : RecyclerView.ViewHolder(rootView) {
 val taskName = rootView.findViewById<TextView>(R.id.task_name)
 val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)
 ...
 fun bind(item: Task) {
 taskName.text = item.taskName
 taskDone.isChecked = item.taskDone
 }
}
```

Если перевести представление с переработкой на связывание данных, можно удалить вызовы `findViewById()` и поручить каждому представлению загрузку его собственных данных.

Как реализовать связывание данных

Представление с переработкой переводится на связывание данных примерно по той же схеме, как и для фрагментов. Необходимо сделать следующее:

1. Добавить переменную связывания данных в `task_item.xml`. Мы сделаем `<layout>` корневым элементом макета и создадим переменную связывания данных с именем task и типом `Task`. При этом будет сгенерирован класс связывания с именем `TaskItemBinding`.
2. Присвоить значение переменной связывания данных в `TaskItemAdapter`. Объект `TaskItemBinding` используется для заполнения макета каждого элемента, а его переменной связывания данных присваивается объект `Task` для этого элемента.
3. Использовать переменную связывания данных для заполнения представлений данными. Наконец, мы обновим файл `task_item.xml`, чтобы каждое представление загружало свои данные из объекта `Task` макета.

Начнем с определения переменной связывания данных.

Включение переменной связывания данных в task_item.xml

Начнем с добавления элемента `<layout>` в корневой элемент `task_item.xml` и определения переменной связывания данных. Но вместо того чтобы использовать его для привязки представлений к модели представления, мы укажем, что оно имеет тип `Task`. Ниже приведен код для решения этой задачи; обновите файл `task_item.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">
 <data>
 <variable
 name="task"
 type="com.hfad.tasks.Task" />
 </data>
 <androidx.cardview.widget.CardView
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_margin="8dp"
 app:cardElevation="4dp"
 app:cardCornerRadius="4dp" >
 ...
 </androidx.cardview.widget.CardView>
</layout>
```

Назначение элемента `<layout>` корневым элементом файла `task_item.xml` сообщает Android, что макет должен использоваться со связыванием данных, поэтому он генерирует новый класс связывания с именем `TaskItemBinding`. Мы воспользуемся этим классом для заполнения приведенного выше макета и присвоим его переменной связывания данных объект `Task`.

Макет заполняется в коде держателя представления адаптера

При создании представления с переработкой мы заполнили файл макета `task_item.xml` в `TaskItemViewHolder` — внутреннем классе `TaskItemAdapter`. Теперь необходимо изменить код, чтобы он работал с классом связывания `TaskItemBinding`. Но прежде чем браться за дело, напомним текущую версию кода.

```
class TaskItemAdapter...{
 ...
 class TaskItemViewHolder(val rootView: CardView)
 : RecyclerView.ViewHolder(rootView) {
 val taskName = rootView.findViewById<TextView>(R.id.task_name)
 val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)
 companion object {
 fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
 val layoutInflater = LayoutInflater.from(parent.context)
 val view = layoutInflater
 .inflate(R.layout.task_item, parent, false) as CardView
 return TaskItemViewHolder(view)
 }
 }
 fun bind(item: Task) {
 taskName.text = item.taskName
 taskDone.isChecked = item.taskDone
 }
 }
}
```

Использование класса связывания для заполнения макета

Первое изменение, которое мы внесем в `TaskItemViewHolder`, — заполнение `task_item.xml` с использованием класса `TaskItemBinding`. Для этого мы воспользуемся методом `inflateFrom()` держателя представления:

```
fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
 val layoutInflater = LayoutInflater.from(parent.context)
 val view = layoutInflater.inflate(R.layout.task_item, parent, false) as CardView
 val binding = TaskItemBinding.inflate(layoutInflater, parent, false)
 return TaskItemViewHolder(view binding)
}
```

Обратите внимание на передачу переменной связывания — объекта `TaskItemBinding` — конструктору `TaskItemViewHolder`. А значит, также нужно обновить определение класса `TaskItemViewHolder` и привести его к следующему виду:

```
class TaskItemViewHolder(val binding: TaskItemBinding)
 : RecyclerView.ViewHolder(binding.root) {
 ...
}
```

Присваивание объекта Task переменной связывания данных макета

Теперь в приложении класс `TaskItemBinding` используется для заполнения макета `task_item.xml`, и мы можем воспользоваться им для присваивания переменной связывания данных `task`. Для этого мы изменим метод `bind()` объекта `TaskItemViewHolder`, чтобы он присваивал `task` текущему элементу `Task` представления с переработкой:

```
fun bind(item: Task) {
 binding.task = item
}
```

Строки, в которых заполняются представления `task_name` и `task_done` макета, стали лишними. Это означает, что мы также можем удалить свойства `taskName` и `taskDone` из держателя представления. Полный код `TaskItemAdapter` (включая внутренний класс `TaskItemViewHolder`) приведен на следующей странице.

Полный код TaskItemAdapter.kt

Ниже приведен обновленный код `TaskItemAdapter`; обновите файл `TaskItemAdapter.kt`:

```
package com.hfad.tasks
import android.view.LayoutInflater
import android.view.ViewGroup

import androidx.recyclerview.widget.ListAdapter

import androidx.recyclerview.widget.RecyclerView
import com.hfad.tasks.databinding.TaskItemBinding

class TaskItemAdapter
 : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {
 override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
 : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)
 override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
 val item = getItem(position)
 holder.bind(item)
 }

class TaskItemViewHolder(val binding: TaskItemBinding) :
RecyclerView.ViewHolder(binding.root)){
```

```
        companion object {
    fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
    val layoutInflater = LayoutInflater.from(parent.context)
    val binding = TaskItemBinding.inflate(layoutInflater, parent, false)
    return TaskItemViewHolder()
    }
    }

    fun bind(item: Task) {
        binding.task = item
    }
    }
}
```

Использование связывания данных для заполнения представлений макета

Теперь, когда мы присвоили переменной связывания данных `task` макета `task_item.xml` элемент `Task` держателя представления, мы можем воспользоваться связыванием данных для заполнения представлений макета. Код решения этой задачи вам уже знаком. Например, для заполнения текста представления `task_name` именем задачи можно использовать следующий код:

```
<TextView
 android:id="@+id/task_name"
 ...
 android:text="@{task.taskName}" />
```

а состояние флажка `task_done` может задаваться следующим кодом:

```
<CheckBox
 android:id="@+id/task_done"
 ...
 android:checked="@{task.taskDone}" />
```

Полный код приведен на следующей странице.

Полный код task_item.xml

Ниже приведен обновленный код `task_item.xml`; обновите этот файл:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">
```

```
 <data>
 <variable
 name="task"
 type="com.hfad.tasks.Task" />
 </data>
 <androidx.cardview.widget.CardView
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_margin="8dp"
 app:cardElevation="4dp"
 app:cardCornerRadius="4dp" >
 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:orientation="vertical" >
 <TextView
 android:id="@+id/task_name"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:padding="8dp"
 android:text="@{task.taskName}" />
<CheckBox
 android:id="@+id/task_done"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:padding="8dp"
 android:clickable="false"
 android:text="Done?"
 android:checked="@{task.taskDone}" />
 </LinearLayout>
 </androidx.cardview.widget.CardView>
</layout>
```

Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

1. `task_item.xml` определяет переменную связывания данных `Task` с именем `task`. Так как `task_item.xml` содержит корневой элемент `<layout>`, для этого макета генерируется класс связывания с именем `TaskItemBinding`.

2. `TasksFragment` создает объект `TaskItemAdapter` и назначает его адаптером представления с переработкой.

3. `TasksFragment` передает `List<Task>` объекту `TaskItemAdapter`. `List<Task>` содержит обновленный список записей из базы данных.

4. Метод `onCreateViewHolder()` объекта `TaskItemAdapter` вызывается для каждого элемента, который должен отображаться в представлении с переработкой. `onCreateViewHolder()`

вызывает метод `TaskItemViewHolder.inflateFrom()`, который создает объект `TaskItemBinding`. Он заполняет макет объекта и использует его для создания объекта `TaskItemViewHolder`.

5. Метод `onBindViewHolder()` объекта `TaskItemAdapter` вызывается для каждого объекта `TaskItemViewHolder`. При этом вызывается метод `bind()` объекта `TaskItemViewHolder`, который использует объект `TaskItemBinding`, чтобы присвоить переменной `task` объект `Task` элемента.

6. Код связывания данных в `task_item.xml` использует свойство `task` для назначения представлений для каждого элемента. Свойству `text` представления `task_name` присваивается `task.taskName`, а свойству checked представления `task_done` присваивается `task.taskDone`.

При запуске приложения `TasksFragment` отображает сетку карточек в представлении с переработкой. Представление ведет себя точно так же, как и в предыдущей версии, но на этот раз в его внутренней реализации используется связывание данных.

Поздравляем! Вы узнали, как реализовать связывание данных в представлении с переработкой, а также научились использовать `DiffUtil`.

Резюме

- Каждый раз, когда вызывается метод `notifyDataSetChanged()`, представление с переработкой выполняет повторное связывание и перерисовку каждого элемента.
- `DiffUtil` определяет, какие элементы изменились, и обновляет представление с переработкой наиболее эффективным образом.
- `ListAdapter` — разновидность `RecyclerView.Adapter`, работающая со списками. Он предоставляет собственный резервный список, а его конструктор получает `DiffUtil.ItemCallback`.
- `DiffUtil.ItemCallback` определяет различия между двумя элементами списка, отличными от `null`.
- Чтобы указать, ссылаются ли два объекта на один и тот же элемент, переопределите метод `areItemsTheSame()` объекта `DiffUtil.ItemCallback`.
- Чтобы указать, имеют ли два объекта одинаковое содержимое, переопределите метод `areContentsTheSame()` объекта `DiffUtil.ItemCallback`.
- Используйте метод `submitList()` для передачи новой версии списка `ListAdapter`.
- Представления с переработкой могут использовать связывание данных.
- Определите переменную связывания данных в макете, используемом для элементов представления с переработкой.
- Значение переменной связывания данных задается в коде адаптера представления с переработкой.