5.7. Архитектурные компоненты. Библиотека Room. Часть 1

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 5.7. Архитектурные компоненты. Библиотека Room. Часть 1

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 18:54

Оглавление

1. Введение

При изучении предыдущего раздела вас наверняка посещала мысль, что в коде слишком много повторов. Данные, которые мы хранили в базе данных практически никак не связаны с классом, который представляет эти данные.

Каждый раз при обращении к базе данных мы были вынуждены вспоминать о том, в какой таблицы хранятся данные и в каких полях. А нельзя ли сгеренеровать код, который свяжет класс, который представляет данные, с самой базой данных? Можно, именно это и выполняет библиотека Room.

Библиотека Room обеспечивает уровень абстракции над SQLite, чтобы обеспечить более надежный доступ к базе данных, используя всю мощь SQLite.

Room является частью компонентов архитектуры Android, представленных в Google I/O 2016. Это не просто ORM, это целая библиотека, которая позволяет нам создавать базы данных SQLite и управлять ими проще, с помощью аннотаций мы можем определять наши базы данных, их таблицы и операции; Room автоматически переведет эти аннотации в инструкции/запросы SQLite для выполнения соответствующих операций в движке БД.

Таким образом, Room — это высокоуровневый интерфейс для низкоуровневых привязок SQLite, встроенных в Android Он выполняет большую часть своей работы во время компиляции, создавая API-интерфейс поверх встроенного SQLite API, поэтому вам не нужно работать с Cursor-ом.

2. Обзор архитектуры Room

Библиотека Room состоит из трех основных компонентов

Entity (сущность)

Entity представляет собой таблицу в базе данных и должна быть аннотирована с помощью *@Entity.*

Каждая сущность состоит как минимум из одного поля, в котором должен быть определен первичный ключ.

DAO (объект доступа к базе данных)

В Room мы используем объекты доступа к данным для доступа и управления вашими данными. DAO является основным компонентом библиотеки Room и включает методы, которые предлагают доступ к базе данных ваших приложений, он должен быть аннотирован с помощью *@Dao* .

Объект DAO используется вместо построителей запросов и позволяет вам разделять различные компоненты базы данных, например текущие данные и статистику.

База данных

Служит держателем базы данных и основной точкой доступа к реляционным данным. Класс должен быть аннотирован с помощью *@Database* и *расширяет RoomDatabase*.

Он содержит и возвращает DAO.

В этой теме мы переделаем приложение, созданное в предыдущей теме и в процессе реализуем каждую из компонент.

3. Реализация на простом примере

В этом разделе мы проведем рефакторинг приложения, разработанного в предыдущей теме. Это простейший менеджер заметок с одним строковым и одним числовым полем.

Подключение зависимостей

Для работы с Room нужно отредактировать grade-файл уровня модуля, добавить в него зависимости

```
dependencies {
  def room_version = "2.2.6"

  implementation "androidx.room:room-runtime:$room_version"
  kapt "androidx.room:room-compiler:$room_version"

  // optional - Kotlin Extensions and Coroutines support for Room
  implementation "androidx.room:room-ktx:$room_version"

  // optional - Test helpers
  testImplementation "androidx.room:room-testing:$room_version"
}
```

Entity

После импорта зависимостей мы можем определить Entity, представляющее таблицу в базе данных. В нашем примере у нас есть простой элемент со строкой и числом.

Библиотека Room основана на аннотациях. Нам достаточно аннотировать элементы Entity и библиотека сгенерирует необходимое.

```
@Entity(tableName = "results")
data class ResultEntity(
    @PrimaryKey(autoGenerate = true)
    var id: Int,
    @ColumnInfo(name = "name") var name: String,
    @ColumnInfo(name = "result") var result: Int
)
```

Мы получаем сразу и data-класс и представление данных в базе данных!

DAO

Теперь мы можем создать DAO, который будет содержать все наши запросы

```
@Dao
interface ResultsDao {
    @Query("SELECT * FROM results ORDER BY :order")
    fun getAll(order:String): List<ResultEntity>
    @Insert
    fun insert(vararg result: ResultEntity)
    @Delete
    fun delete(result: ResultEntity)
    @Update
    fun update(vararg result: ResultEntity)
}
```

Здесь мы просто определяем базовые функции базы данных SQL, такие как вставка и удаление записей. При этом детализации требует только запрос на поиск, все остальные действия стандартны.

Обратите внимание на передачу параметра в функцию `getAll()`.

Database

Остается объявить собственно базу данных.

```
@Database(entities = arrayOf(ResultEntity::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun resultsDao(): ResultsDao
}
```

Волшебные три строки сгенерируют все. Мы конкретизируем только класс объектов, которые база будет содержать.

Пользуемся

Для использования базы данных теперь достаточно создать базу

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "results.db"
).build()
```

Получить из нее объект DAO и использовать его для запросов к базе. Единственное, нужно проводить действия с базой данной в отдельном потоке. Для Room это не рекомендация, а жесткое правило, иначе выкидывается ошибка

```
 java.lang.IllegalStateException: Cannot access database on the main thread since it may potentially lock the UI for a long
period of time.
```

Удобно использовать для этого корутины.
Исправим класс активности:

```
class MainActivity : AppCompatActivity() {
    val db by lazy {
        Room.databaseBuilder(
            this,
            AppDatabase::class.java, "results.db"
        ).build()
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val context = this
        val dbHelper = SimpleDBHelper(context)
        btnInsert.setOnClickListener {
            if (editTextName.text.toString().isNotEmpty() &&
                editTextResult.text.toString().isNotEmpty()
            ) {
                //val result = Result(editTextName.text.toString(), editTextResult.text.toString().toInt())
                //dbHelper.insert(result)
                val result = ResultEntity(
                    0,
                    editTextName.text.toString(),
                    editTextResult.text.toString().toInt()
                )
                GlobalScope.launch {
                    db.resultsDao().insert(result)
                }
                clearFields()
            } else {
                Toast.makeText(context, "Please Fill All Data's", Toast.LENGTH_SHORT).show()
            }
        }
        btnRead.setOnClickListener {
//        val allRecords = mutableListOf<Result>()
//        val cursor = readableDatabase.query(
//            DBContract.Entry.TABLE_NAME, null, null,
//            null, null, null, order
//        )
//        cursor.moveToFirst()
//        while (cursor.moveToNext()) {
//            allRecords.add(
//                Result(
//                    cursor.getString(cursor.getColumnIndex(DBContract.Entry.COLUMN_NAME_NAME)),
//                    cursor.getInt(cursor.getColumnIndex(DBContract.Entry.COLUMN_NAME_RESULT))
//                )
//            )
//        }
            GlobalScope.launch {
                val data = db.resultsDao().getAll("RESULT DESC")
                withContext(Main) {
                    tvResult.text = ""
                    for (d in data) {
                        tvResult.append("${d.name} ${d.result}\n")
                    }
                }
            }
        }
    }


    private fun clearFields() {
        editTextName.text.clear()
        editTextResult.text.clear()
    }
}
```

В комментариях мы оставили строки предыдущей версии.

LiveData - вишенка на Room Database

Сейчас в проекте мы должны обновлять состояние экрана кнопкой Read. Понятно, что в нашей программе можно вынести получение списка в отдельную функцию и вызывать ее не только по кнопке, но и сразу после добавления записи. Но что делать, если данные будут поставляться в программу и из других источников, например, из сети?

Можно изменить DAO-класс таким образом, чтобы возвращалась LiveData со списком, тогда интерфейс будет получать оповещение о том, что база изменилась (неважно почему!) и сможет обновляться сразу после изменения.

```
@Dao
interface ResultsDao {

    @Query("SELECT * FROM results ORDER BY :order")
    fun getAll(order: String): LiveData<List<ResultEntity>>

    @Insert
  fun insert(vararg result: ResultEntity)
    @Delete
  fun delete(result: ResultEntity)
    @Update
  fun update(vararg result: ResultEntity)
}
```

Заменим в разметке TextView со скроллингом на RecycleView

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  tools:context=".MainActivity">

        <EditText
  android:id="@+id/editTextName"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="10dp"
  android:padding="8dp" />

        <EditText
  android:id="@+id/editTextResult"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="10dp"
  android:autofillHints="Age"
  android:inputType="number"
  android:padding="8dp"
  android:textColor="@android:color/background_dark" />

        <Button
  android:id="@+id/btnInsert"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="10dp"
  android:padding="8dp"
  android:text="Add data" />

    <androidx.recyclerview.widget.RecyclerView
  android:id="@+id/result_list"
  android:layout_width="match_parent"
  android:layout_height="match_parent" />

</LinearLayout>
```

И сделаем для него простейший адаптер (для просты не будем делать собственную разметку элементов списка, а возьмем стандартную с двумя TextView).

```
class ResultAdapter (private val results: List<ResultEntity>) :
    RecyclerView.Adapter<ResultAdapter.ResultViewHolder>() {

    override fun getItemCount() = results.size

  class ResultViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        var name: TextView? = null
 var result: TextView? = null

 init {
            name = itemView.findViewById(android.R.id.text1)
            result = itemView.findViewById(android.R.id.text2)
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ResultViewHolder {
        val itemView = LayoutInflater.from(parent.context).inflate(
            android.R.layout.simple_list_item_2, parent, false)
        return ResultViewHolder(itemView)
    }

    override fun onBindViewHolder(holder: ResultViewHolder, position: Int) {
        holder.name?.text = results[position].name
  holder.result?.text = results[position].result.toString()
    }
}
```

и в методе активности `onCreate()` сделаем наблюдателя

```
result_list.layoutManager = LinearLayoutManager(this)
db.resultsDao().getAll("RESULT DESC").observe(this,
    // при изменении базы данных мы просто пересоздаем адаптер с новым списком
    { results -> result_list.adapter = ResultAdapter(results)})
```

Полностью код активности может выглядеть так:

```
class MainActivity : AppCompatActivity() {
    val db by lazy {
  Room.databaseBuilder(
            this,
            AppDatabase::class.java, "results.db"
  ).build()
    }

  override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val context = this //      val dbHelper = SimpleDBHelper(context)
  btnInsert.setOnClickListener {
  if (editTextName.text.toString().isNotEmpty() &&
                editTextResult.text.toString().isNotEmpty()
            ) {
//val result =
//    Result(editTextName.text.toString(), editTextResult.text.toString().toInt()) //dbHelper.insert(result)
  val result = ResultEntity(
                    0,
                    editTextName.text.toString(),
                    editTextResult.text.toString().toInt()
                )
                GlobalScope.launch {
  db.resultsDao().insert(result)
                }
  clearFields()
            } else {
                Toast.makeText(context, "Please Fill All Data's", Toast.LENGTH_SHORT).show()
            }
        }
//        btnRead.setOnClickListener {
//            GlobalScope.launch {
//                val data = db.resultsDao().getAll("RESULT DESC")
//                withContext(Main) {
//                    tvResult.text = ""
//                    for (d in data) {
//                        tvResult.append("${d.name} ${d.result}\n")
//                    }
//                }
//            }
//        }
  result_list.layoutManager = LinearLayoutManager(this)
         db.resultsDao().getAll("RESULT DESC").observe(this,
             { results -> result_list.adapter = ResultAdapter(results)})

    }

    private fun clearFields() {
        editTextName.text.clear()
        editTextResult.text.clear()
    }
}
```

Итоги

Внешне приложение почти не изменилось, мы только убрали кнопку Read и расположили имя и результат на разных строках (потому что не делали своей разметки элементов, а воспользовались стандартной).

Но внутри приложение очень сильно преобразилось.

Теперь мы используем всего три класса (Сущности, Доступа к данным и, собственно, базы), при этом в них практически отсутствует копипаст. В классе DAO мы описали только нестандартный запрос SELECT, а функции удаления, добавления и изменения просто объявили: благодаря аннотациям стандартный код этих запросов генерируется автоматически.

И, наконец, мы использовали объекты LiveData для того, чтобы обновлять отображение базы сразу после ее изменения.

Полный код рефакторинга приложения можно посмотреть на гитхабе.

Начать тур для пользователя на этой странице