5.8. Архитектурные компоненты. Библиотека Room. Часть 2

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 5.8. Архитектурные компоненты. Библиотека Room. Часть 2

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 18:01

Оглавление

1. Room - несколько сущностей

До этого момента мы рассматривали базу данных с одной сущностью, у нас была всего одна таблица. На этом занятии мы сделаем несколько сущностей и посмотрим как в библиотеку Room их можно связать между собой.

Схема данных

Давайте разработаем схему данных для описания учебного процесса. В нашей базе будет четыре сущности, то есть четыре таблицы:

- Школа (сущность School)
- Директор (сущность Director)
- Учащийся (сущность Student)
- Предмет (сущность Subject)

Каждая таблица должна иметь первичный ключ, это особенно важно для организации объединения таблиц, выборке по нескольким таблицам сразу. Ключом может быть любое уникальное поле, обычно для этого в каждой сущности-таблице добавляют специальное дополнительное целочисленное поле id. Но давайте для простоты, все-таки у нас модельный, учебный проект, сделаем у каждой сущности ее имя ключевым полем. Конечно, в реальной базе это должно привести к коллизиям, наверняка в большой базе будет, например, несколько студентов с одинаковыми именами. Но на этом занятии мы сделаем это сознательно, чтобы не путаться в многочисленных id. И поля будем называть с префиксами, чтобы не путаться, например, в классе-таблице School будет поле schoolNameб а не просто name.

Отношения между сущностями

Посмотрим, как связаны наши сущности между собой. Чаще всего директор управляет одной школой и школа управляется одним директором. Значит отношение Director - School один к одному.

В одной школе учится много учеников. Но ученик посещает только одну школу (в этой модели мы рассматриваем базовое образование). Отношение School - Student один ко многим

Наконец, каждый ученик изучает несколько предметов и один предмет изучает несколько учеников. В этом случае отношение Student - Subject много ко многим

На этом занятии мы как раз и рассмотрим как при помощи библиотеки Room установить эти отношения в базе данных приложения. Опишем принципы подхода, который будем применять.

Проще всего реализовать связь один к одному. Для этого достаточно добавить в таблицу Director, название школы, которой он руководит. Или наоборот, в таблицу School добавить имя директора. Тогда мы сможем легко отвечать на вопросы типа "где живет директор Иванов Иван Иванович?". Для этого нужно просто объединить две таблицы School и Director, а база данных это умеет.

Как соединить школу и ученика? Тоже просто. У каждого ученика одна единственная школа. Достаточно ее имя добавить в сущность ученика.

А что делать с отношением ученик-предмет? Тут уже придется создавать новую таблицу. В которой и связывать ученика и предмет. У нас получится много записей ученик-предмет. В ней имена школ и имена учеников могут повторяться. Назовем новую таблицу-сущность StudentSubjectCrossRef.

2. Отношение "Один к одному"

Реализуем сущности

Самая простая сущность - School, она содержит лишь один первичный ключ - свое название.

```
@Entity
data class School(
    @PrimaryKey(autoGenerate = false)
    val schoolName: String
)
```

Класс Director содержит еще и имя школы

```
@Entity
data class Director(
    @PrimaryKey(autoGenerate = false)
    val directorName : String
    val schoolName: String
)
```

Реализуем связь

Одинаковые названия полей в двух таблицах не обеспечивают их связь. Нужно прописать ее в явном виде, при помощи отдельного data-класса.

```
data class SchoolAndDirector(
    @Embedded val school: School,
    @Relation(
        parentColumn = "schoolName",
        entityColumn = "schoolName"
  )
    val director: Director
)
```

Аннотация @Entity отсутствует, потому что это не отдельная сущность, не третья таблица, а только связь одной таблицы с другой.

Аннотация @Embedded говорит Room, что надо просто взять поля из School и считать их полями класса SchoolAndDirector.

Аннотация @Relation описывает собственно связь. Таблицы связываются между собой: поле schoolName из таблицы School будет связано с одноименным полем таблицы Director.

Создаем интерфейс DAO

Этот класс будет очень похож на аналогичный предыдущего занятия

```
@Dao
interface SchoolDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertSchool(school: School)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertDirector(director: Director)

    @Transaction
    @Query("SELECT * FROM school WHERE schoolName = :schoolName")
    fun getSchoolAndDirectorWithSchoolName(schoolName: String): List<SchoolAndDirector>
}
```

Единственное, на что в этом коде стоит обратить внимание, это параметр аннотации @Insert, мы не использовали этот параметр в проекте предыдущего занятия. Параметр onConflict задает поведение базы, когда мы попытаемся вставить в нее значение, которое уже имеется в базе. Значение OnConflictStrategy.REPLACE говорит, что запись нужно обновить. Этим мы получаем возможность неоднократно вставлять одинаковые данные.

И еще обратим внимание на аннотацию @Transaction. Дело в том, что данный запрос по сути вызывает несколько команд к базе данных, поэтому важно, чтобы все они выполнились как единое целое. Проблема может возникнуть при обращении к базе из разных потоков одновременно. Аннотация @Transaction гарантирует правильное поведение базы данных в многопоточной среде.

3. Отношение "Один ко многим"

Добавим класс-сущность Student

```
@Entity
data class Student(
    @PrimaryKey(autoGenerate = false)
    val studentName: String,
    val schoolName: String
)
```

и реализуем связь учеников со школой. Так чтобы получать список всех учеников, которые учатся в школе с данным названием.

```
data class SchoolWithStudents(
    @Embedded val school: School,
    @Relation(
        parentColumn = "schoolName",
        entityColumn = "schoolName"
  )
    val students: List<Student>
)
```

Этот класс очень похож на класс DirectorAndSchool6 с той разницей что в классе не один студент, а целый список студентов.

Еще нужно добавить прототип транзакции в DAO-интерфейс:

```
...
@Transaction
@Query("SELECT * FROM school WHERE schoolName = :schoolName")
fun getSchoolWithStudents(schoolName: String): List<SchoolWithStudents>
...
```

4. Отношение "Многие ко многим"

Остается самое интересное. Сам класс Subject очень прост.

```
@Entity
data class Subject(
    @PrimaryKey(autoGenerate = false)
    val subjectName: String
)
```

Здесь нет ничего нового. Но вот создание связи делается особым образом.

Мы уже обсудили, что для отношения такого типа создается третья таблица, а значит новая сущность-Entity

```
@Entity(primaryKeys = ["studentName", "subjectName"])
data class StudentSubjectCrossRef(
    val studentName: String,
    val subjectName: String
)
```

Здесь могут повторяться и ученики и предметы, но вот *пара из них является уникальной*. Это так называемый "составной первичный ключ".

Конечно, создание такой таблицы не связывает таблицу Students с таблицей Subjects. Больше того, мы еще нигде не указали эти имена. Связь нужно оформить отдельным классом, как мы делали раньше с классами SchoolAndDirector и SchoolWithStudents.

```
data class StudentWithSubjects(
    @Embedded val student: Student,
    @Relation(
        parentColumn = "studentName",
        entityColumn = "subjectName",
        associateBy = Junction(StudentSubjectCrossRef::class)
    )
    val subjects: List<Subject>
)
```

Все аналогично, но мы должны добавить связь: сущности Student и Subject связаны через третью сущность. Это мы и указываем в параметре accociateBy.

Расширяем DAO

```
@Transaction
@Query("SELECT * FROM subject WHERE subjectName = :subjectName")
fun getStudentsOfSubject(subjectName: String): List<SubjectWithStudents>
```

Аналогично можно (и нужно) создать класс для получения списка учеников, которые изучают данный предмет.

5. Класс базы данных и заполнение данными

Осталось создать базу данных и наполнить ее данными

Код собственно базы данных практически ничем не отличается от аналогичного кода предыдущего занятия

```
@Database(
    entities = [
        School::class,
        Student::class,
        Director::class,
        Subject::class,
        StudentSubjectCrossRef::class
  ],
    version = 1
)
abstract class SchoolDatabase : RoomDatabase() {

    abstract val schoolDao: SchoolDao
  abstract class SchoolDataBase: RoomDatabase() {
        abstract fun resultsDao(): SchoolDao
  }
}
```

Единственное, на что обратим внимание, это количество сущностей. Реально мы будем работать с четырьмя, но для обеспечения работы БД у нас их пять.

Заполняем базу значениями

Положим в базу несколько значений. Для этого можно использовать файл DataExample.kt с примером значений из репозитория.

Приведем фрагменты этого файла

```
class DataExample {
    companion object {
        val directors = listOf(
            Director("Mike Litoris", "Jake Wharton School"),
            ...
        )
        val schools = listOf(
            School("Jake Wharton School"),
            ...
        )
        val subjects = listOf(
            Subject("Dating for programmers"),
            Subject("Avoiding depression"),
            ...
        )
        val students = listOf(
            Student("Beff Jezos", "Kotlin School"),
            Student("Mark Suckerberg", "Jake Wharton School"),
            ...
        )
        val studentSubjectRelations = listOf(
            StudentSubjectCrossRef("Beff Jezos", "Dating for programmers"),
            StudentSubjectCrossRef("Beff Jezos", "Avoiding depression"),
            ...
        )
    }
}
```

В активности написать

```
val dao = db.schoolDao;
lifecycleScope.launch {
    DataExample.directors.forEach { dao.insertDirector(it) }
    DataExample.schools.forEach { dao.insertSchool(it) }
    DataExample.subjects.forEach { dao.insertSubject(it) }
    DataExample.students.forEach { dao.insertStudent(it) }
    DataExample.studentSubjectRelations.forEach { dao.insertStudentSubjectCrossRef(it) }
}
```

И в инспекторе баз данных Android Studio увидеть результат.

Полностью проект можно увидеть в репозитории Интерфейс в этом проекте полностью отсутствует. Он реализуется точно так же, как в предыдущем занятии.

Начать тур для пользователя на этой странице