Android Room 使用详解

小村医 关注
1 2018.12.18 18:32:27 字数 1,835 阅读 17,062

一、概述

Room提供了一个访问SQLite的抽象层，以便在利用SQLite的全部功能的同时进行流畅的数据库访问。

> 要在项目中使用Room需要在程序的build.gradle文件中添加以下依赖

```
dependencies {
    def room_version = "2.1.0-alpha03"

    implementation "androidx.room:room-runtime:$room_version"
    annotationProcessor "androidx.room:room-compiler:$room_version" // use kapt for Ko

    // optional - RxJava support for Room
    implementation "androidx.room:room-rxjava2:$room_version"

    // optional - Guava support for Room, including Optional and ListenableFuture
    implementation "androidx.room:room-guava:$room_version"

    // optional - Coroutines support for Room
    implementation "androidx.room:room-coroutines:$room_version"

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

Room主要有以下三个部分组成：

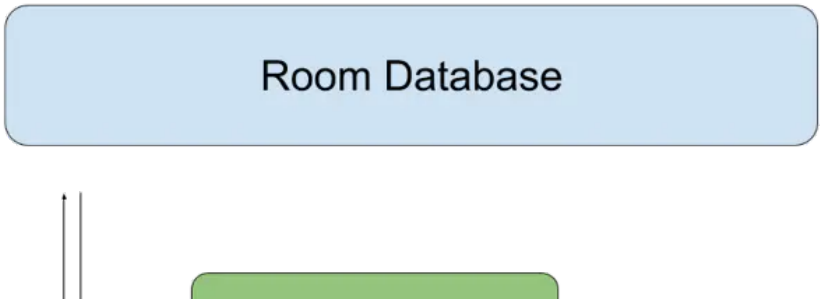
1. Database：

   标有 `@Database` 注解的类需要具备以下特征：

   - 继承 `RoomDatabase` 的抽象类
   - 在注释中包括与数据库关联的实体列表（@Database(entities ={ })）
   - 包含一个无参的抽象方法并返回一个使用 `@Dao` 注解的类

2. Entity：对应数据库中的表
3. DAO：包含访问数据库的方法

以上各部分的依赖关系如下图所示：



room_architecture.png

下面使用一个简单的例子介绍各部分如何使用：

User

```
@Entity
public class User {
    @PrimaryKey
    public int uid;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

UserDao

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);

    @Insert
    void insertAll(User... users);

    @Delete
    void delete(User user);
}
```

AppDatabase

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

在创建完上面的文件之后，可以使用以下代码获得数据库的实例：

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database-name").build();
```

> 在实例化**AppDatabase**对象时，应遵循单例设计模式，因为每个 `RoomDatabase` 实例都相当消耗性能，并且您很少需要访问多个实例。

二、 Entity定义数据

默认情况下，Room为实体中定义的每个字段创建一个列。如果实体有不想持久的字段，则可以使用 `@Ignore` 来注解它们。必须通过Database类中的 `entities` 数组引用实体类。
下面的代码片段显示了如何定义实体：

```
@Entity
public class User {
    @PrimaryKey
    public int id;

    public String firstName;
    public String lastName;
}
```

使用主键

每个实体必须定义至少1个字段作为主键。即使只有1个字段，仍然需要用 `@PrimaryKey` 注解字段。此外，如果您想Room自动分配IDs给实体，则可以设置 `@PrimaryKey` 的 `autoGenerate` 属性。
如果实体具有复合主键，则可以使用 `@Entity` 注解的 `primaryKeys` 属性，如下面的代码片段所示：

```
@Entity(primaryKeys = {"firstName", "lastName"})
public class User {
    public String firstName;
    public String lastName;
}
```

默认情况下，Room使用类名作为数据库表名。如果希望表具有不同的名称，请设置@Entity注解的 `tableName` 属性
SQLite中的表名不区分大小写。
与tableName属性类似，Room使用字段名称作为数据库中的列名。如果希望列具有不同的名称，请将 `@ColumnInfo` 注解添加到字段中，如下面的代码片段所示：

```
@Entity(tableName = "users")
public class User {
    @PrimaryKey
    public int id;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

索引

需要索引数据库中的某些列以加快查询速度。若要向实体添加索引，在 `@Entity` 注释中添加 `indices` 属性 ， 下面的代码片段演示了这个注解过程：

```
@Entity(indices = {@Index("name"),
        @Index(value = {"last_name", "address"})})
public class User {
    @PrimaryKey
    public int id;

    public String firstName;
    public String address;

    @ColumnInfo(name = "last_name")
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

有时，数据库中的某些字段或字段组必须是唯一的。可以通过将 `@Index` 注解的 `unique` 属性设置为true来强制执行此唯一性属性。下面的代码示例防止表中包含两行，它们包含firstName和lastName列的相同值集：

```
@Entity(indices = {@Index(value = {"first_name", "last_name"},
        unique = true)})
public class User {
    @PrimaryKey
    public int id;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

三、Dao访问数据

DAO既可以是接口，也可以是抽象类。如果是抽象类，它可以有一个构造函数，它把 `RoomDatabase` 作为唯一的参数。Room在编译时创建每个DAO实现。

Insert

当您创建一个DAO方法并用 `@Insert` 注解时，Room生成一个实现，在一个事务中将所有参数插入到数据库中

```
@Dao
public interface MyDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    public void insertUsers(User... users);

    @Insert
    public void insertBothUsers(User user1, User user2);

    @Insert
    public void insertUsersAndFriends(User user, List<User> friends);
}
```

如果@Insert方法只接收1个参数，则可以返回一个Long型的值，这是插入项的新rowId。如果参数是数组或集合，则应该返回long[] 或者 List类型的值。
有时插入数据和更新数据会产生冲突,所以就有了冲突之后要怎么解决,SQLite对于事务冲突定义了5个方案
OnConflictStrategy

- REPLACE:见名知意,替换,违反的记录被删除，以新记录代替之
- ignore: 违反的记录保持原貌，其它记录继续执行
- fail: 终止命令，违反之前执行的操作得到保存
- abort 终止命令，恢复违反之前执行的修改
- rollback 终止命令和事务，回滚整个事务

Update

`@Update` 注解在数据库中用于修改一组实体的字段。它使用每个实体的主键来匹配查询。

```
@Dao
public interface MyDao {
    @Update

```

```
    public void updateUsers(User... users);
}
```

Delete

用于从数据库中删除给定参数的一系列实体，它使用主键匹配数据库中相应的行

```
@Dao
public interface MyDao {
    @Delete
    public void deleteUsers(User... users);
}
```

Query

`@Query` 是DAO类中使用的主要注解。它允许您在数据库上执行读/写操作。每个@Query方法在编译时被验证，因此，如果存在查询问题，则会发生编译错误而不是运行时故障。

Room还验证查询的返回值，这样如果返回对象中字段的名称与查询响应中的相应列名不匹配，则Room将以以下两种方式之一提醒您：

- 如果只有一些字段名匹配，则发出警告。
- 如果没有字段名匹配，则会出错。

1、简单查询

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM user")
    public User[] loadAllUsers();
}
```

2、将参数传递到查询中

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM user WHERE age > :minAge")
    public User[] loadAllUsersOlderThan(int minAge);
}
```

3、传递参数集合

有些查询可能要求您传递一个可变数量的参数，其中参数的确切数目直到运行时才知道。例如，您可能希望从区域的子集检索有关所有用户的信息

```
@Dao
public interface MyDao {
    @Query("SELECT first_name, last_name FROM user WHERE region IN (:regions)")
    public LiveData<List<User>> loadUsersFromRegionsSync(List<String> regions);
}
```

4、Observable查询

当执行查询时，您经常希望应用程序的UI在数据更改时自动更新。要实现这一点，请在查询方法描述中使用类型 `LiveData` 的返回值。当数据库被更新时，Room生成所有必要的代码来更新LiveData。

```
@Dao
public interface MyDao {
    @Query("SELECT first_name, last_name FROM user WHERE region IN (:regions)")
    public LiveData<List<User>> loadUsersFromRegionsSync(List<String> regions);
}
```

5、RXJava的响应式查询

Room还可以从您定义的查询中返回RXJava2 `Publisher` 和 `Flowable` 对象。若要使用此功能，请将androidx.room:room-rxjava2库添加到gradle的依赖关系中。

```
@Dao
public interface MyDao {
    @Query("SELECT * from user where id = :id LIMIT 1")
    public Flowable<User> loadUserById(int id);

    // Emits the number of users added to the database.
    @Insert
    public Maybe<Integer> insertLargeNumberOfUsers(List<User> users);

    // Makes sure that the operation finishes successfully.
    @Insert
    public Completable insertLargeNumberOfUsers(User... users);

    /* Emits the number of users removed from the database. Always emits at
       least one user. */
    @Delete
    public Single<Integer> deleteUsers(List<User> users);
}
```

6、直接Cursor访问

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM user WHERE age > :minAge LIMIT 5")
    public Cursor loadRawUsersOlderThan(int minAge);
}
```

四、 数据库升级

在应用程序中添加和更改特性时，你需要修改实体类以反映这些更改。当用户更新应用程序到最新版本时，不希望它们丢失所有现有数据，尤其是如果无法从远程服务器恢复数据。

如果您不提供必要的迁移，则Room会重新构建数据库，这意味着您将丢失数据库中的所有数据

```
Room.databaseBuilder(getApplicationContext(), MyDb.class, "database-name")
        .addMigrations(MIGRATION_1_2, MIGRATION_2_3).build();

static final Migration MIGRATION_1_2 = new Migration(1, 2) {
    @Override
    public void migrate(SupportSQLiteDatabase database) {
        database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, "
                + "`name` TEXT, PRIMARY KEY(`id`))");
    }
};

static final Migration MIGRATION_2_3 = new Migration(2, 3) {
    @Override
    public void migrate(SupportSQLiteDatabase database) {
        database.execSQL("ALTER TABLE Book "
                + " ADD COLUMN pub_year INTEGER");
    }
};
```

五、类型转换器

TypeConverter，它将自定义类转换为Room可以保留的已知类型。例如，如果想要持久化实例Date，可以编写以下内容 TypeConverter 来在数据库中存储等效的Unix时间戳

```
public class Converters {

    @TypeConverter
    public static Date fromTimestamp(Long value) {
        return value == null ? null : new Date(value);
    }

```

```
    @TypeConverter
    public static Long dateToTimestamp(Date date) {
        return date == null ? null : date.getTime();
    }

}
```

上面的例子中定义了两个函数，一个转换Date对象到Long对象，另一个执行逆变换，从Long到Date。由于Room是知道如何持久化Long对象的，因此它可以使用此转换器来持久保存Date类型的值。接下来，添加 @TypeConverters 注释到AppDatabase类，这样Room就可以在AppDatabase中的实体和Dao上使用上面的定义的类型转换器。还可以限制 @TypeConverters 到不同的范围，包括单个实体，DAO和DAO方法。