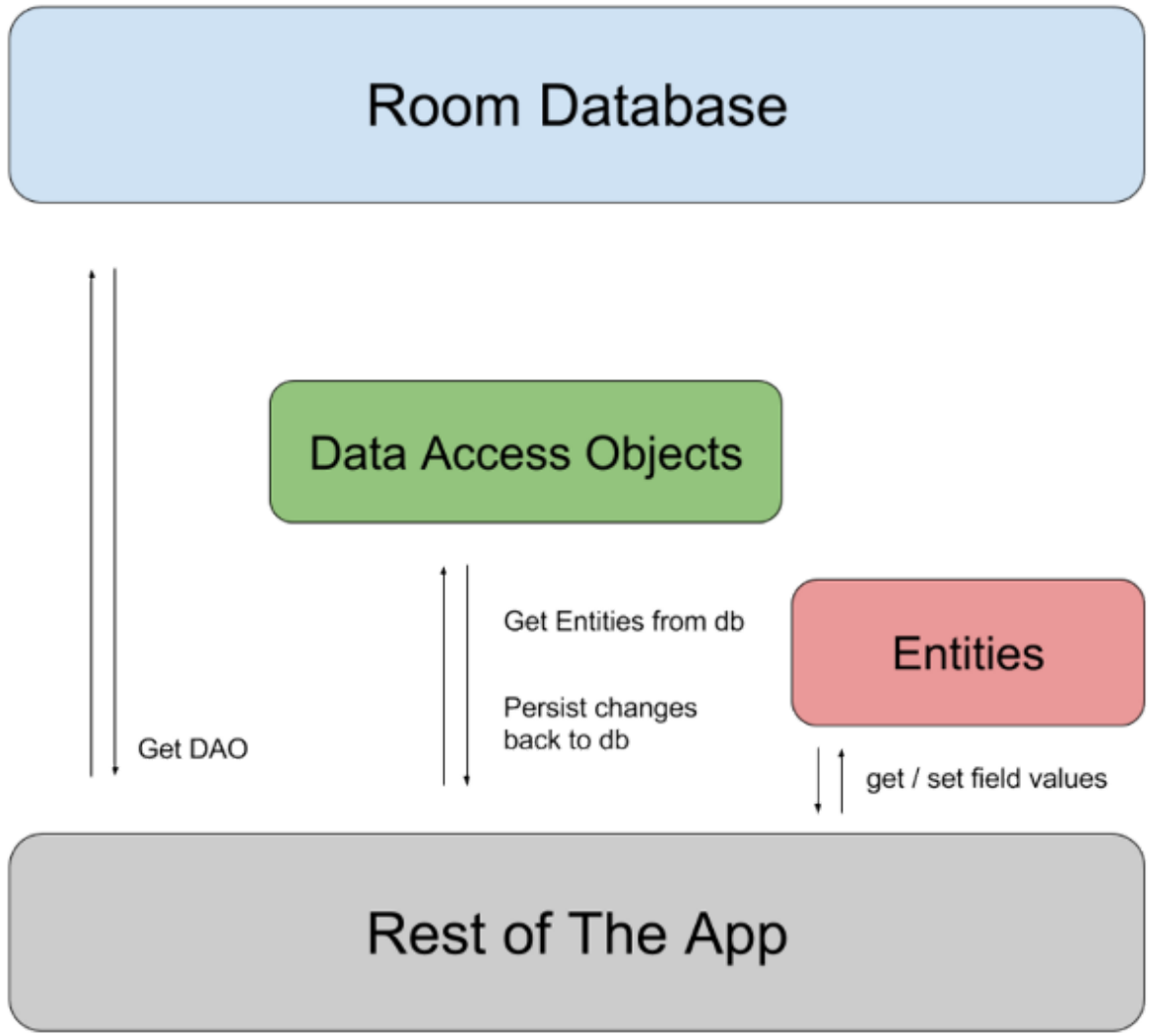
作用于Dao的注解用来作为Room的

Room做为jetpack的核心组件之一，其目的是通过对sqlLite进行封装，目的在于更为便捷的操作数据库。

注意`Room`是一个关系映射(`ORM`)库

Room对SQLLite进行抽象使用，在充分使用SQLLite的同时，并保持流畅的速度。



Room的配置与使用

1. 首先通过以下配置语句远程依赖使用Room

```
implementation 'androidx.room:room-runtime:2.2.4'
kapt 'androidx.room:room-compiler:2.2.5'
```

在使用讲解之前，先简单的介绍以下room的一些注解

注解	含义
Entity	数据表名
ColumnInfo	作用于Entity，对某些字段进行命名,如不使用：则Entity对象中的成员变量则会以其命名变作为数据表表中的字段名
PrimaryKey	主键，设置autoGenerate是否为自增
Dao	数据库操作接口
Query	作用于Dao内可用于查询，也可用户条件删除
Insert	作用于Dao内用于插入数据
OnConflictStrategy	Insert的配置选项
Delete	作用于Dao内用于删除数据
Database	数据库的注解，用以表明数据库，并且返回业务所需的Dao对象
entities	作用于Database注解，表示业务所需多少张数据表。

2.构建Entity 、 Dao 、DataBase

1. entity的构建

   首先利用Entity注解表明示例数据表表名为：kille_tom_user;
   其次利用@PrimaryKey(autoGenerate = false) 表明用户id为主键并且不可递增;

   其次利用ColumnInfo 对realName以及nickName分别进行数据字段命名。

```
@Entity(tableName = "kille_tom_user")
class UserEntity(

    @PrimaryKey(autoGenerate = false)
    val id: String,
    @ColumnInfo(name = "real_name")
    val realName: String?,
    @ColumnInfo(name = "nick_name")
    val nickName: String,
    val sex: String
)
```

2. dao的构建

   Dao 的使用是声明好需要注解的接口然后利用相关的注解将接口内的方法进行注解，然后room会根据其内部的规则将相应的操做转化为实际的sql操作。

```
@Dao
interface UserDao {

    @Query("select * from kille_tom_user where real_name is not null")
    fun getRealUsers():List<UserEntity>
```

```
    @Query("select max(age) from kille_tom_user")
    fun getMaxAge():Int?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUpdate(userEntities:List<UserEntity>)


}
```

3. dataBase的构建

   dataBase则需要声明一个抽象的类并且继承RoomDataBase并且利用相关的注解将实际需要的数据表与它关联起来，并且通过使用的抽象方法需要返回操作的DAO对象，其余的实际操作则会由Room在其内部实现好实际需要返回的数据库对象让开发者操作。

```
@Database(
    entities = [
        (UserEntity::class)], version = 1, exportSchema = true
)
abstract class AppDB : RoomDatabase(){

    abstract fun getUserDao(): UserDao
}
```

4. 数据库的初始化以及全局单列

   在前面的代码示例演示了如何声明数据库的关键部件，接下来利用`Room.databaseBuilde`将声明好的数据库对象进行构建；

   注意为了避免内存泄漏最好使用Application中的context进行初始化，并且使用单列的方式去创建数据库，用以避免不必要的初始化。

   如下可采用类似的方式去构建初始化数据库。

```
object AppDBManager {

    private var db: AppDB? = null

    fun initDB(context: Application, uuid: String) {

        if (db == null){
            synchronized(AppDB::class.java){
                if (db == null){

                    db = Room.databaseBuilder(context.applicationContext,
AppDB::class.java, "${uuid}_${context::class.java.simpleName}_app_db")
                        .allowMainThreadQueries()
                        .build()
                }
            }
        }


    }
```

```
    @Throws
    fun getDB(): AppDB {
        return db ?: throw RuntimeException("LandRomDB Null")
    }

    fun clear() {
        db = null
    }
}
```

空

常用的SQL操作

在前一节中我们学习了如何构建并且初始化Room，在这一节中，我们会学习到常用的SQL操作，例如简单的增删改查、模糊搜索、min()、max()等操作。

查询操作

在sql中我们常常需要做一些查询操作，例如查询某一列、查求指定的最小值、最大值，或者模糊搜寻匹配等。

如下代码示例针对上一节中的UserEnitity进行演示讲解：

1. 指定条件的查询
   在UserEntity中，当realName不为空时则代表用户已经实名了，那么如何从user表中检索出已经实名的用户呢可以通过以下方式去实现：

```
@Query("select * from kille_tom_user where real_name is not null")
fun getRealUsers():List<UserEntity>
```

2. 模糊搜索查询

   在UserEntity中，birthday代表着用户设置的生日，那么我们如何实现查询指定年份出生的用户呢，例如1993年出生的用户,可以使用`like`左匹配去建立查询

   `注意在Room中使用like进行匹配搜索 % 符号是不能写在sql语句中，要通过这样的方式去实现`

   `注意在Room中传递的参数必须用作sql语句中的调用否则编译时期就会报错，如何引用通过利用 : 符号与参数进行调用例如如下示例`

```
   @Query("select * from kille_tom_user where birthday like :year")
   fun getBirthYearUser(year:String):List<UserEntity>

   AppDBManager.getDB().getUserDao().getBirthYearUser("1993%")
```

3. 组合条件查询

   针对1、2示例我们可以实现类似的一个需求，查询已经实名的用户并且是指定年份出生的用户，例如1993年出生的用户并且已实名。

```
@Query("select * from kille_tom_user where real_name is not null and
birthday like :year")
fun getRealInBirthYearUsers(year:String):List<UserEntity>
```

4. 求某列的最大值以及最少值

   在在UserEntity中，age代表着用户设置的年龄，那么我们要求出年龄的最大值或者最小值则需要使用 max 以及 min 函数去求。例如这样

```
    @Query("select max(age) from kille_tom_user")
    fun getMaxAge():Int?

    @Query("select min(age) from kille_tom_user")
    fun getMinAge():Int?
```

插入操作

在插入数据时，往往会存在一些已有的数据，当需要覆盖插入的时候则需要利用`OnConflictStrategy.REPLACE`这一配置，默认配置为`OnConflictStrategy.ABORT`

一般在客户端中对数据进行缓存保存时，往往服务器的数据都是唯一最新的时候一般都会这样使用，例如统一更新用户信息并且替换已有用户信息来保证数据的唯一性例如

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
fun insertUpdate(userEntities:List<UserEntity>)
```

Room的兼容与升级

在Room中，数据库也是存在版本号的，目的在于区分版本数据，并且将低版本的转向为高版本，但是需要开发者去配置数据库进行兼容升级。

还记得我们的`AppDB`吗，不记得没关系，我们再次温故一下：

```
@Database(
    entities = [
        (UserEntity::class)], version = 1, exportSchema = true
)
abstract class AppDB : RoomDatabase(){

    abstract fun getUserDao(): UserDao
}
```

在上述代码中，entities代表这这个数据库中有多少张数据表，version顾名思义就是数据库的版本号。

还记得我们得UserEntity吗？在UserEntity中我们有realName代表者用户是否实名了，那么相应的用户实名信息，我们应该如何存储？有两种方式：

1. 新增一张实名数据表，通过操作数据表对用户执行相应的操作。
2. 直接在用户数据表中扩展数据字段存储响应数据。

针对以上两种方式我们可以分别探讨:

在探讨前先抽离出我们所需要通用的数据；例如：用户所在的国家、详细地址、用户真实姓名、以及用户惟一证件id，例如我们的身份证号码。

那么我们现在开始通过这两种方式示例代码进行探讨Room的兼容与升级。

新增表方式的实现

基于前面提出的三大基本数据：国家、地址、证件id；我们可以设计一张这样数据表

```
@Entity(tableName = "real_name_infor")
class RealNameInforEntity(

    @PrimaryKey
    var id: String = "",

    val userId: String,

    val country: String,

    val address: String,

    val cardID: String,

    val userRealName:String
)
```

对应Dao可以这样去声明

```
@Dao
interface RealNameInforDao {

    @Query("select * from real_name_infor ")
    fun getAllRealData():List<RealNameInforEntity>

    @Query("select * from real_name_infor where userId=:id")
    fun getRealDataForUserId(id:String):RealNameInforEntity?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUpdate(entities:List<RealNameInforEntity>)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUpdate(entity: RealNameInforEntity)

}
```

对应的Dao以及Entity有了，那么我们的DB版本也必须进行修改，避免程序升级安装后，可能因为数据没做兼容出现的一系列问题。

首先进行这样一个简单的配置

```
@Database(
    entities = [
        (UserEntity::class),
        (RealNameInforEntity::class)],
    version = 2, exportSchema = true
)
abstract class AppDB : RoomDatabase(){

    abstract fun getUserDao(): UserDao

    abstract fun getRealNameDao():RealNameInforDao
}
```

然后利用对`Migration`的实现去对room进行兼容升级

例如这样针对新增表的例子去实现

```
  private val DB_MARGIN_1_to_2 =
// 这里描述版本1到2的升级
object :Migration(1,2){
        override fun migrate(database: SupportSQLiteDatabase) {
            //利用database执行sql语句对数据库进行兼容升级
            database.execSQL("create table real_name_infor (id BLOB primary key
autoincrement not null,userId text not null,country text not null,address text
not null,cardID text not null,userRealName text not null)")
        }
    }
```

最后这样调用实现数据库兼容升级

```
object AppDBManager {

    private var db: AppDB? = null

    fun initDB(context: Application, uuid: String) {

        if (db == null){
            synchronized(AppDB::class.java){
                if (db == null){

                    db = Room.databaseBuilder(context.applicationContext,
AppDB::class.java, "${uuid}_${context::class.java.simpleName}_app_db")
                        .allowMainThreadQueries()
                        .addMigrations(DB_MARGIN_1_to_2)
                        .build()
                }
            }
        }


    }

    private val DB_MARGIN_1_to_2 = object :Migration(1,2){
        override fun migrate(database: SupportSQLiteDatabase) {
```

```
            database.execSQL("create table real_name_infor (id BLOB primary key
autoincrement not null,userId text not null,country text not null,address text
not null,cardID text not null,userRealName text not null)")
        }
    }
}
```

新增字段方式的实现

在之前userEntity中已经存在了realName那么我们这样对UserEntity进行拓展升级

```
@Entity(tableName = "kille_tom_user")
class UserEntity(
    @PrimaryKey(autoGenerate = false)
    val id: String,
    @ColumnInfo(name = "real_name")
    val realName: String?,
    @ColumnInfo(name = "nick_name")
    val nickName: String,
    val sex: String,
    var birthday:String?,
    var age:Int,
    @ColumnInfo(name = "real_country")
    val realCountry: String,
    @ColumnInfo(name = "real_address")
    val realAddress: String,
    @ColumnInfo(name = "card_id")
    val cardID: String
)
```

对于字段拓展我们需要使用到`alter`语句去对UserEntity进行拓展，还记得`Migration`吗?

对的只要是对数据进行兼容升级都需要创建对应版本的的`Migration`,有同学肯定问道，那怎么保证兼容的时序，在Room里面只要你设置好版本升级对应的方式，他就会自动执行了相应方式，所以我们不需要担心兼容升级不成功。

OK,我们回到正题

```
//先修改下数据库配置
@Database(
    entities = [
        (UserEntity::class),(RealNameInforEntity::class)], version = 3,
exportSchema = true
)
abstract class AppDB : RoomDatabase(){

    abstract fun getUserDao(): UserDao

    abstract fun getRealNameDao():RealNameInforDao
}
//其次实现兼容
object AppDBManager {

    private var db: AppDB? = null

    fun initDB(context: Application, uuid: String) {
```

```
        if (db == null){
            synchronized(AppDB::class.java){
                if (db == null){

                    db = Room.databaseBuilder(context.applicationContext,
AppDB::class.java, "${uuid}_${context::class.java.simpleName}_app_db")
                        .allowMainThreadQueries()
                        .addMigrations(DB_MARGIN_1_to_2,DB_MARGIN_2_to_3)
                        .build()
                }
            }
        }


    }

    private val DB_MARGIN_2_to_3 = object :Migration(2,3){
        override fun migrate(database: SupportSQLiteDatabase) {
            database.execSQL("alter table kille_tom_user add column real_country
text default null")
            database.execSQL("alter table kille_tom_user add column real_address
text default null")
            database.execSQL("alter table kille_tom_user add column card_id text
default null")
        }
    }
```

总结

使用Room切记基本注释的概念，以及针对模糊索引 `like` 需要在补上`%`的使用，以及针对数据库兼容升级时，需要对`Migration`的实现以及调用。

对于上诉针对Room所讲的`Entity`、`Dao`、`DataBase`、`Migration`可通过如下图片简单的掌握下它们间关系：