高级程序开发组件->Jetpack<架构组件>

- **Jetpack诞生的背景**

为了追求更高的代码质量，可读性，易维护性，出现了各种各样的架构来规范代码。例如**MVP,MVVM** 架构。在2018年，Google旨在帮助开发者编写出更加符合高质量代码规范，更具有架构设计的应用程序，推出了**Jetpack--开发组件工具集** 。

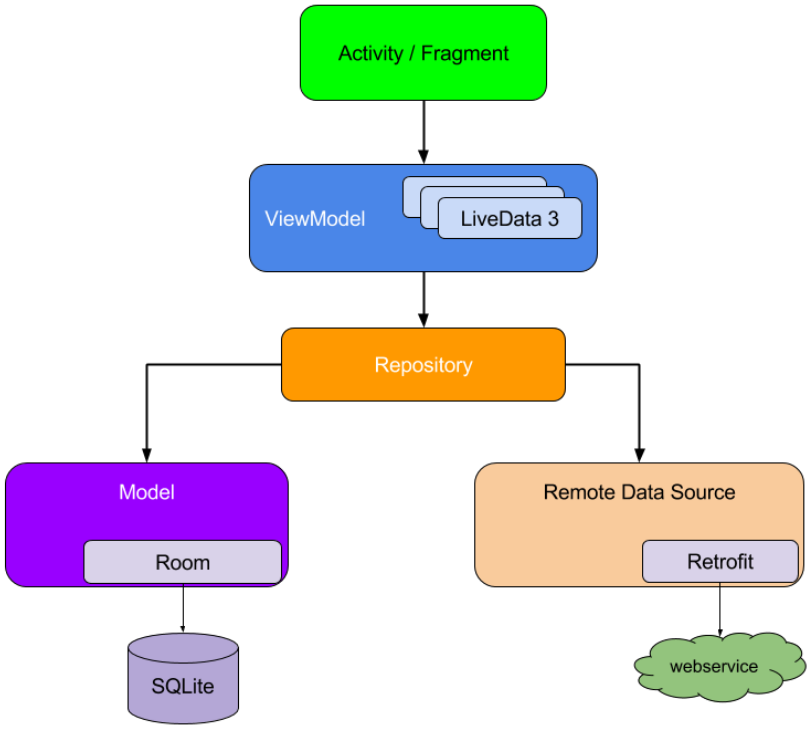
什么是Jetpack

> Android Jetpack 是一套组件、工具和指导，可以帮助您快速构建出色的 Android 应用，让我们的代码更加简洁，并且简化开发过程。

Jetpack全家福

Jetpack的架构组件是目前我们用的比较多的一个部分，其中的LiveData和ViewModel常和Rtrofit一起构建MVVM架构。

简单介绍一下MVVM架构与Jetpack的联系



整个架构解析如下：

1. `View` 层调用 `ViewModel` 获取数据
2. `ViewModel` 调用 `Repository` 获取数据
3. `Repository` 是数据仓库（里面一般封装了Rtrofit的一些网络请求或者访问数据库的方法），根据实际业务，再通过 `Dao` 访问本地数据库或者 `Retrofit` 访问服务器（通过网络请求拿到数据）。
4. `ViewModel` 中的 `LiveData` 类型数据得到更新
5. `View` 层的观察者 `Observer(LiveData是个可观察的数据类型)` 的回调方法 `onChanged()` 中收到新的数据，更新 `UI`。

如何使用Jetpack的架构组件

Databinding

DataBinding是谷歌官方推出的一个库，DataBinding库来写声明的layouts文件，可以用最少的代码来绑定你的app逻辑和layouts文件。（我个人理解就是不用通过回调去更新UI，而Ui直接和数据绑定在一起，直接通过改变数据的值，ui也就自动更新了。）

Databinding基本用法：

在app.gradle下，需要添加Data Binding到gradle构建文件里如下：

```
android {
    ....
    dataBinding {
        enabled = true
    }
}
```

在布局中就可以这样写：

```
<?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"
    xmlns:tools="http://schemas.android.com/tools"
    >

    <data>
        <variable name="user" type="demo.com.databindingdemo.User"/>
    </data>

    <android.support.constraint.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">


        <TextView
            android:id="@+id/user_name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.mUserName}"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/user_age"
            android:text='@{user.mUserage+""}'
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toBottomOf="@id/user_name" />
    </android.support.constraint.ConstraintLayout>
</layout>
```

在data内描述了一个名为user的变量属性，使其可以在这个layout中使用

```
<variable name="user" type="demo.com.databindingdemo.User"/>
```

在layout的属性表达式写作@{}，下面是一个TextView的text设置为user的mUserName属性：

```
android:text="@{user.mUserName}"
```

2)Data对象

```
public class User {

    public final String mUserName;
    public final int mUserage;

    public User(String userName, int userAge) {
        this.mUserName = userName;
        mUserage = userAge;
    }


}
```

3）Binding数据

默认情况下，一个Binding类会基于layout文件的名称而产生,上述的layout文件是activity_main.xml，因此生成的类名是ActivityMainBinding

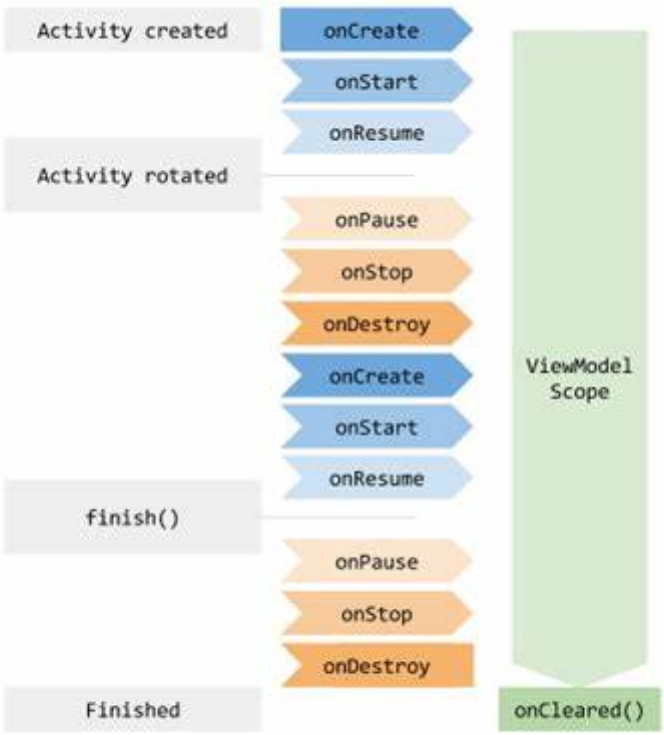
```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityMainBinding binding =
 DataBindingUtil.setContentView(this, R.layout.activity_main);
        User user = new User("sam",11);
        binding.setUser(user);
    }
}
```

这就是Databinding的最基本的用法，剩下的一些深入的用法例如：

1.import 2.(alias)当类名有冲突时，其中一个类名可以重命名 3.导入的类型还可以在表达式中使用static属性和方法（一般可以用于给button设置监听事件啥的）.下面我可以通过一个例子来梳理一遍Databinding的流程。

ViewModel

ViewModel的重要作用就是可以帮助Activity分担一部分工作,用于调用Repository（仓库层）获取数据，用来保存View中的数据。当手机发生横竖屏翻转的时候，Activity会被重建，同时存放在Activity的数据就会丢失。而ViewModel的生命周期不同，它可以保证在翻转的时候不会被重建，只要咱们的数据交给ViewModel控制的话，我们就不会存在翻转数据丢失的问题了。



如果我们想使用ViewModel组件的话，需要添加如下的依赖：

```
dependencies {
        implementation "androidx.lifecycle:lifecycle-extensions:2.1.0"
}
```

ViewModel中设置一些变量或者封装一些Repository（网络请求拿数据）的方法来保存数据

在View中创建ViewMode的实例:

```
viewModel=ViewModelProviders.of(this<你的Activity或者Fragment实例
>).get(MainViewModel<你的ViewModel>::class.java)
```

不能通过直接去创建ViewModel的实例，因为ViewModel有自己独立的生命周期，并且长于Activity的周期，如果在onCreate中创建实例的话，每次Oncreate就会有个实例，当数据翻转的时候，就保存不了数据了。

```
MainViewModel::class.java
```

这个相当于Java中的MainViewModel.class这种写法。

- **如何向ViewModel传递参数**

由于ViewModel的实例都是通过上述的ViewModelProviders构建的，因此我们没有地方可以向ViewModel的构造函数传递参数，之前的功能可以在翻转的时候保存数据，那我们可不可以让程序重新启动的时候也可以保存当时退出的数据。

基本思路可以这样：退出的时候将counter保存（数据库）--->重新启动的时候读取之前的数据-->传递给ViewModel.好像就解决了。关键是如何传值呢?

利用ViewModelProvider.Factory接口就可以实现了，重写里面的Create方法就可以传递参数了。

```
class ViewModelFactory(private val countReserved:Int):ViewModelProvider.Factory
{
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return MainViewModel(/*countReserved*/)as T
    }
}
```

Lifecycles

用于在一个非Activity类中去时刻感知Activity的生命周期，一般的解决方法就是利用观察者模式，在Activity中重写相应的生命周期的方法，然后再通知MyObserver去执行一些操作。例如：

```
class MyObserver {
    fun activityStart(){
    }
    fun activityStop(){
    }
}
```

```
class MainActivity : AppCompatActivity() {
    lateinit var observer: MyObserver
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        observer= MyObserver()
    }

    override fun onStart() {
        super.onStart()
        observer.activityStart()
    }

    override fun onStop() {
        super.onStop()
        observer.activityStop()
    }
}
```

然而我们利用Lifecycle的话就不需要在Activity中写这么多逻辑处理。

```
class MyObserver :LifecycleObserver {
   @OnLifecycleEvent(Lifecycle.Event.ON_START)
   fun activityStart(){
       Log.d("MyObserver","activityStart")
   }
    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    fun activityStop(){
        Log.d("MyObserver","activityStop")
    }
}
```

主要是使用了@OnLifecycleEvent（）这个注解，当Activity中出现对应生命周期的时候，去通知这个MyObserver，就回调到对应生命周期的方法了。现在Activity是如何去通知呢?

```
lifecycleOwner.lifecycle.addObserver(MyObserver())
```

首先调用lifecycleOwner的getlifecycle方法获得一个Lifecycle实例，再给它添加一个观察者就好了，LifecycleOwner该如何获取呢？在Activity中是继承自AppCompatActivity的,或者Fragment继承AndroidX.fragment.app.Frament

他们本身自己就是一个LifecycleOwner。所以上述代码可以简化成.

```
lifecycle.addObserver(MyObserver())
```

这样MyObserver就可以感知到Activity周期的变化了，严格上说是被动的去感知。要想主动感知

```
class MyObserver(val lifecycle: Lifecycle) :LifecycleObserver
```

只需要传一个lifecycle进去就可以了，然后调用lifecycle的currentState方法来主动感知当前的生命周期。

lifecycle和前文提到的viewModel没有多大直接的关系。而接下来的LiveData可谓是Jetpack中一个非常非常重要的组件了

LiveData

前面我们一直都是在Activity中手动获取ViewModel中的数据这种交互方式来更新ui。但是如果ViewModel中有一些数据是需要耗时操作才能获得的，那么还按照刚才那种操作的话，就会出现，ViewModel还在获取数据，而Activity就已经在获取新的数据，这样肯定还是以前的数据。

为了解决上述问题，我们希望在ViewModel中的数据获取之后去主动通知Activity更新ui.这样逻辑就很清楚了。但是切忌不能将Activity的实例传给ViewModel,然后去通知Activity更新ui。因为ViewModel的生命周期是长于Activity的，就很可能因为Activity无法释放而造成内存泄漏。

吗

所以就引出了LiveData,一种活着的数据。可以在数据发生变化的时候通知观察者，也就是说数据可以用LiveData包装一下，然后再Activity中观察它，这样就可以主动将数据变化通知给Activity了。

具体实现：

```
class MainViewModel(countReserved:Int):ViewModel() {
    val counter=MutableLiveData<Int>()
    init {
        counter.value=countReserved
    }
    fun plusOne(){
        val count=counter.value?:0
        counter.value=count+1
    }
    fun clear(){
        counter.value=0
    }
}
```

这里，我们将counter变量变成了一个MutableLiveData的对象，并制定了泛型Int.MutableLiveData是一种可变的LiveData.主要有三种读写数据的方法，getvalue(),setvalue(),postvalue()，getvalue（）就是获取数据。set和Post都是设置数据，只是set只能在主线程中调用，而post用于非主线程调用。不过不用去记，因为kotlin已经帮我们自动识别了，get,set有对应的语法糖。

而在Activiy中，我们应该去观察这个LiveData，这样LiveData就可以主动将数据变化通知给Activity了。

主要的代码如下：

```
viewModel.counter.observe(this, Observer {
    count->infoText.text=count.toString()
})
```

这里主要是调用了counter这个MutableLiveData对象中的observe()方法来观察数据的变化。observe（）方法接受两个参数，一个是LifecycleOwner对象，一个是Observer接口，当counter数据发生了变化的时候，就会回调到这里Observer里的OnChanged（）方法，更新界面。也就是说，上面我们提到的，在ViewModel中通过一些耗时操作，例如网络请求拿到数据后，才会回调到这个observer里，去更新ui。

但是，上述写法还是有点问题，就是ViewModel中的数据应该是封装的，也就是说不能通过外面去给counter设置数据，但是上面写法把这个可变的counter暴露了出去。所以，我们还是需要给这个counter封装一层，让它暴露出去的是不可变的，而它的实际是可变的却不暴露出去。怎么理解呢？就是说暴露给外面的counter是不可改变的LiveData,而在ViewModel中通过可以改变的_counter去改变这个值。例如：

```
class MainViewModel(countReserved:Int):ViewModel() {
    val counter:LiveData<Int>
    get() = _counter
    private val _counter=MutableLiveData<Int>()
    init {
        _counter.value=countReserved
    }
    fun plusOne(){
        val count=_counter.value?:0
        _counter.value=count+1
```

```
    }
    fun clear(){
        _counter.value=0
    }
}
```

我们把可变的_counter设置了private，达到了封装数据的效果，暴露出去的是个不可改变的LiveData，也就是个counter。这样当外部调用counter变量的时候，实际上获取的是__counter的实例。但是你无法给counter设置数据，因为counter是不可变的。从而保证了ViewModel的数据封装性。

---

map和switchMap

LiveData为了能够应对各种不同的需求场景，提供了两种转换方法：map()和swithMap()

假如有个用户信息的基类，User类

```
data class User(var  firstName:String,var  lastName:String,var age:Int)
```

可是，如果在Activity中明确只关注名字，不关心年龄，意思就是不观察年龄这个数据的变化，这个时候把整个user的LiveData数据暴露在外面就不太合适。所以就需要转换成只暴露名字的User的LiveData.

具体的写法如下：

```
private val userLiveData=MutableLiveData<User>()

val userName:LiveData<String>=Transformations.map(userLiveData){
    user->"${user.firstName} ${user.lastName}"
}
```

这里调用了Transformations的map()方法，来对LiveData进行转换，这里就将User对象的LiveData数据转换成了一个只包含用户姓名的字符串。每当userLiveData这个数据改变的时候，map()方法就会被调用，并执行一段逻辑，将转换后的数据通知给userName的观察者。

接下来我们介绍一下用得更多的switchMap()方法，我们之前说到的LiveData的对象都是在ViewModel中创建的，然而在一些项目中，很有可能ViewModel中的某个LiveData是通过调用Repository仓库层的网络请求方法获取的。

例如有一个Repository的单例类，代码如下：

```
object Repository {
    fun getUser(userId:String):LiveData<User>{
        val liveData=MutableLiveData<User>()
        liveData.value= User(userId,userId,0)
        return liveData
    }
}
```

这个单例类中的getUser（）方法，传入一个userId,返回一个包含User数据的LiveData对象。每次传

入一个UserId都会创建新的User对象

然后我们在ViewModel层中也封装一个getUser()的方法，用于获取LiveData对象，

```
fun getUser(userId: String):LiveData<User>{
    return Repository.getUser(userId)
}
```

那么Activity如何观察这个LiveData的数据变化呢。

可不可以这样写呢?

`viewModel.getUser(userId).observe(this,Observer{` user->....

}`

这样是不行的。因为每次调用一个getUser()方法的时候返回的都是一个新的LiveData实例，而上面的写法一直是观察一个老的LiveData实例，怎么说呢，就比如第一次传入的是“zhangsan”，得到的就是zhangsan这个用户对象，第二次调用getUser（）的时候传入的是“lisi”,得到的就是lisi这个对象，新的对象是产生了，但是新的观察对象没有产生。你还是在观察zhangsan就离谱了啊！这就好比你周围的女孩子不停变换，而你观察的始终是你的前任，这就不行呀！

为了解决上面的问题，随着LiveData实例的更新，也要紧跟着更换你的观察对象。也就是说这个逻辑：外面Activity每次调用getuser方法会产生新的对象，而新的对象产生必须对应新的观察对象的产生。这个就需要运用到switchMap()的方法，具体如下：

```
private  val userIdLiveData=MutableLiveData<String>()

val user:LiveData<User> = Transformations.switchMap(userIdLiveData){
    userId->Repository.getUser(userId)
}
 fun getUser(userId:String){
        userIdLiveData.value=userId
    }
```

这个先创建了一个userIdLiveData的对象用来观察userId的数据变化，然后，调用Transformations的swithchMap()方法把它转换成一个可以观察的LiveData对象，这个逻辑就是，每当userIdLiveData这个数据变化的时候，就调用Repository的getuser()方法产生一个新的对象，这样就产生一个新的LiveData就是个可观察对象usr。而我们的Activity在外面观察的就肯定是新产生的user这个LiveData对象就OK了。

```
 viewModel.user.observe(this, Observer {
    user ->infoText.text=user.firstName
})
```

整个流程梳理下就是：每当外面调用viewModel中的getUser（）方法的时候，产生一个新的对象，这就意味着ViewModel中的useridLiveData的数字改变了，这个时候观察userIdLiveData的

switchMap（）方法会调用，会执行转换函数中的Repository方法获取数据，等待获取数据完成后，这个时候就会产生一个新的可观察的LiveData对象，也就是外面Activity需要观察的user.这样就可以更新ui了。

总的流程就是Activity-->传入一个参数请求（可能是网络请求的一些参数）调用Repository层的方法获取新的对象（网络请求的结果），ViewModel等待网络请求完成后，产生一个新的可观察的LiveData对象（新的网络请求后的数据结果对象）---->Activity观察这个新的对象--->更新ui。

通过一个例子来说明：

一个基类：

```
data class Location(val lng:String,val lat:String)
```

一个观察lng，lat变换的LiveData的对象：

```
private val locationLiveData=MutableLiveData<Location>()
```

ViewModel中暴露在外界通过传入网络请求的参数，去改变locationLiveData的lng和lat

```
fun refreshWeather(lng:String,lat:String){
    locationLiveData.value=Location(lng,lat)
}
```

当lng,lat参数改变的时候，也就是locationLiveData改变后，调用一个Repository方法转换成一个新的可观察对象。

```
val weatherLiveData=Transformations.switchMap(locationLiveData){ location ->
    Repository.refreshWeather(location.lng,location.lat)
}
```

Repository中的调用网络请求返回新的LiveData数据的方法

```
fun refreshWeather(lng: String, lat: String) = fire(Dispatchers.IO) {
    coroutineScope {
        val deferredRealtime = async {
            SunnyWeatherNetwork.getRealtimeWeather(lng, lat)
        }
        val deferredDaily = async {
            SunnyWeatherNetwork.getDailyWeather(lng, lat)
        }
        val realtimeResponse = deferredRealtime.await()
        val dailyResponse = deferredDaily.await()
        if (realtimeResponse.status == "ok" && dailyResponse.status == "ok") {
            val weather = Weather(
                realtimeResponse.result.realtime,
                dailyResponse.result.daily
            )
            Result.success(weather)
        } else {
            Result.failure(
                RuntimeException(
```

```
                    "realtime response status is${realtimeResponse.status}" +
                            "daily response status is ${dailyResponse.status}"
                )
            )
        }
    }
}
```

这里主要关注的就是，传入新的lng和新的lat之后，利用协程中的asyn函数让他们并发的发起网络请求，然后再分别调用他们的await()方法，就可以保证在两次网络请求都成功响应后，拿到进一步的数据，最后再将两个请求数据封装成一个Weather对象，再用Result.success（）封装后emit发送出去。

而在外面Activity观察新产生的这个weatherLiveData。把result发送出去的封装数据拿出来，进行更新Ui.

```
viewModel.weatherLiveData.observe(this, Observer { result->
    val  weather=result.getOrNull()
    if (weather!=null){
        showWeatherInfo(weather)//更新ui的方法。
    }else{
        Toast.makeText(this,"无法成功获取天气信息",Toast.LENGTH_SHORT).show()
        result.exceptionOrNull()?.printStackTrace()
    }
    swipeRefresh.isRefreshing=false
})
```

总的思路就是这样.

外面Activity调用ViewModel中暴露在外面的refreshWeather()方法，改变lng,和Lat,从而转换函数启动，调用Repository中的网络请求函数，将新的数据通过Result发送出去，ViewModel从而产生一个新的可观察的LiveData对象，外界观察这个对象变换后，获取到Result发送出去的数据。进行ui更新。

假如，我们ViewModel中暴露在外面的方法是不需要传入参数的。这种情况下如何触发数据改变呢?

其实可以这样写:

```
fun refresh(){
refreshLiveData.value=refreshLiveData.value
}
```

只是把原来的数据取出来重新设置一遍，没错就是这种类似套娃的操作，只要调用了一次setValue()方法，就会触发数据变化事件，这个时候，就会回调switchMap（）的方法，执行Repository的方法，产生一个新的可观察的LiveData对象，进而外界观察其变换更新ui。

Room

之前学到相关SQLite数据库的知识，难免和SQL语句打交道。如果有一个封装好了ORM（对象关系映射,就是java是面对对象，而数据库是关系型的数据库，将面对对象的语言和面向关系的数据库之间建立的一种映射关系就是ORM)框架就好了.而Room就是Android官方推出的一个ORM框架。

ORM框架主要包括

- Entity：用于定义封装实际数据的实体类。
- Dao：在这里对数据库的各种操作进行封装，实际编程中只需要和Dao层打交道。也就是封装了SQL语句
- Database：用于定义数据库中的关键信息，例如数据库版本号，包含哪些实体类以及提供Dao层的访问实例。

  要使用Room就得添加如下依赖：

```
apply plugin: 'kotlin-kapt'
```

```
implementation "androidx.room:room-runtime:2.1.0"
kapt "androidx.room:room-compiler:2.1.0"
```

Entity:

```
@Entity
data class User(var  firstName:String,var  lastName:String,var age:Int){
    @PrimaryKey(autoGenerate = true)
    var id:Long = 0
}
```

其中用@Entity注解，将它声明成一个实体类，@PrimaryKey注解是将id字段设为了主键，再把autoGenerate 设置成true，就是使得主键的值是自动生成的。

Dao

Room中最关键的地方，封装了一系列的访问数据库的操作，Dao层是一个接口，类似于Rtrofit的API接口实现。

```
@Dao
interface UserDao {
    @Insert
    fun insertUser(user: User):Long

    @Update
    fun updateUser(newUser: User)

    @Query("select * from User")
    fun loadAllUsers():List<User>

    @Query("select * from User where age > :age")
    fun  loadUsersOlderThan(age:Int):List<User>

    @Delete
    fun deleteUser(user: User)

    @Query("delete from User where lastName = :lastName")
    fun deleteUserByLastName(lastName:String):Int
}
```

- @Insert注解，插入操作，传入一个user对象，并返回自动生成的主键Id
- @Update注解，表示将传入的user对象更新到数据库中
- @Query注解，前两个是查询操作，传入具体的SQL语句进行具体的查询操作。
- @Delete注解，删除操作，用于删除数据。
- 前面传入的都是一个实体来进行数据库的相关操作，如果传入的不是实体，例如最后一个，这个时候就必须用@Query注解加SQL语句进行插入，删除，查询操作。

Database

这部分只需要定义3个部分的内容，数据库的版本号，包括了哪些实体类，以及提供Dao层的访问实例。

```
@Database(version = 1,entities = [User::class])
abstract class AppDatabase :RoomDatabase(){
abstract fun userDao():UserDao

companion object{
private var instance:AppDatabase?=null
//instance来缓存AppDatabase的实例
@Synchronized
fun getDatabase(context: Context):AppDatabase{
    instance?.let {
        return it
    }//instance 不为空则就直接返回

    //instance 为空就调用databaseBuilderL来创建一个AppDatabase的实例
    return Room.databaseBuilder(context.applicationContext,
        AppDatabase::class.java,"app_database")
        .addMigrations(MIGRATION_1_2, MIGRATION_2_3)
        .build().apply {
            instance=this
            }
        }
    }
}
```

@Database注解表示是一个Database，在里面声明了版本号以及包含了哪些实体类。

然后用companion object单例模式，获取一个AppDatabase的实例。用instance来缓存，如果instance为空则调用

```
 Room.databaseBuilder(context.applicationContext,
        AppDatabase::class.java,"app_database")
```

来创建一个AppDatabase的实例。第三个参数是数据库名字。

Room创建完毕之后，我们改如何在Activity中测试呢?

通过AppDatabase获取到Dao层的实例用于后续进行数据库的操作

```
val userDao=AppDatabase.getDatabase(this).userDao()
```

```
val user1=User("Tom","Brady",40)
val user2=User("Tom","Hanks",63)
```

创建两个实体类，因为数据库的操作是耗时操作，不方便在主线程中执行，所以，需要开线程执行插入，删除，查询的操作。这个时候就不需要和SQL语句打交道了，直接调用userDao的封装好的方法，例如：

```
thread {
    user1.id=userDao.insertUser(user1)
    user2.id=userDao.insertUser(user2)
}
```

进行插入操作，返回一个主键id值，这个id值是更新，删除操作的基本。

```
thread{
    userDao.deleteUserByLastName("Hanks")
}
```

Room数据库的升级。

例如当我们想在数据库中添加一张Book表。这个时候就需要更新Room。同样的，首先要创建一个Book的实例类。

```
@Entity
data class Book (var name:String,var pages:Int,var author:String){
    @PrimaryKey(autoGenerate = true)
    var id :Long=0
}
```

然后就是Book的Dao层

```
@Dao
interface BookDao {
    @Insert
    fun insertBook(book:Book):Long

    @Query("select * from Book")
    fun loadAllBooks():List<Book>
}
```

接下来就是修改下AppDatabase的代码。

```
@Database(version = 2,entities = [User::class,Book::class])
```

```
val MIGRATION_1_2=object : Migration(1,2){
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("create table Book (id integer primary key
autoincrement not null," +
                "name text not null,pages integer not null)")

    }
}
```

增加更新的内容在那个单例类中。最后将这个更新操作添加在创建AppDatabase实例中就可以了。

execSQL中是一些建表的语句。

```
return Room.databaseBuilder(context.applicationContext,
    AppDatabase::class.java,"app_database")
    .addMigrations(MIGRATION_1_2)
    .build().apply {
        instance=this
    }
```

WorkManager

适合用于处理一些要求定时执行的任务，是一个处理定时任务的工具，可以保证即使在应用退出或者手机重启的情况下，之前注册的任务也会得到执行。

首先要用WorkManager需要导入依赖

```
implementation "androidx.work:work-runtime:2.2.0"
```

WorkManager的基本用法分为3部分：

1. 定义一个后台任务，并实现具体的逻辑
2. 配置后台任务的运行条件和约束信息，并构建后台任务请求
3. 将改后台任务请求传给WorkManager的enqueue（）方法，系统就会在合适的时间运行

定义一个后台任务

```
class SimpleWorker(context:
Context,params:WorkerParameters):Worker(context,params) {
    override fun doWork(): Result {
        Log.d("SimpleWorker","do work in SimpleWorker")
        return Result.success()
    }
}
```

后台任务继承Worker类，重写dowork()的方法,然后编写具体的后台任务，并成功就返回Result.success（），失败就返回Result.failure()

配置后台任务的运行条件和约束信息

```
val request=OneTimeWorkRequest.Builder(SimpleWorker::class.java).build()
```

将任务请求传给WorkManager的enqueue（）方法

```
WorkManager.getInstance(this).enqueue(request)
```

WorkManager的基本用法就介绍完了。

通过设置一些运行条件和约束信息可以进行一些复杂的操作

```
val request=OneTimeWorkRequest.Builder(SimpleWorker::class.java)
.setInitialDelay(5,TimeUnit.MINUTES)
.build()
```

进行延时后运行。

WorkManager可以进行链式任务，比如：实现先同步，再压缩，最后上传的功能，这个链式任务，只有在上一个任务成功运行之后才会执行下一个后台任务。

```
WorkManager.getInstance(this)
.beginwith(sync)
.then(compress)
.then(upload)
.enqueue(request)
```