singwhatiwanna
2018年06月26日 阅读 4159

Android Architecture Components 只看这一篇就够了

> 本文由 玉刚说写作平台 提供写作赞助
>
> 原作者： Boy·哈利波特
>
>

一、前言

1.1、Android Architecture Components 介绍

Android Architecture Components 是谷歌在Google I/O 2017发布一套帮助开发者解决Android 架构设计的方案。里面包含了两大块内容：

- 生命周期相关的 Lifecycle-aware Components
- 数据库解决方案 Room

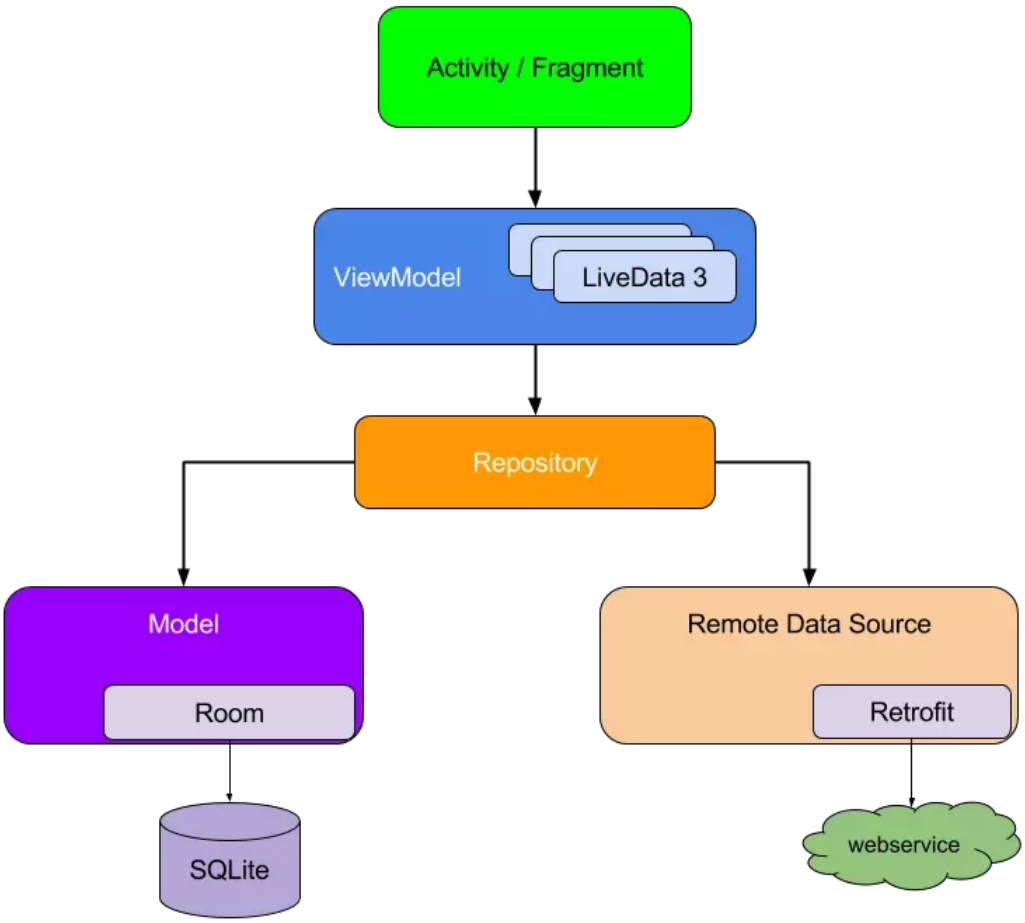
1.2、组件功能

官方给予 Google 组件的功能：A collection of libraries that help you design robust, testable, and maintainable apps. Start with classes for managing your UI component lifecycle and handling data persistence。

使用Google 提供的处理数据持久化和管理组件生命周期的类，有助于应用开发者们构建更加鲁棒性，可测的，稳定可靠的应用。

提供主要的组件有：

1.3、主要架构



1.4、使用组件

在项目根目录 build.gradle 文件添加仓库依赖：

```
allprojects {
    repositories {
        jcenter()
        google()
    }
}
```

如果遇到如下因 gradle 版本导致的编译失败问题：

```
Error:(6, 1) A problem occurred evaluating root project 'TestArc'.>
Could not find method google() for arguments [] on repository container;
```

```
maven {
   url 'https://maven.google.com'
}
```

然后在主 module 的 build.gradle 文件添加需要依赖的组件：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:26.+'

    def lifecycle_version = "1.1.1"

    // ViewModel and LiveData
    compile "android.arch.lifecycle:extensions:$lifecycle_version"
    // alternatively - just ViewModel
    compile "android.arch.lifecycle:viewmodel:$lifecycle_version" // use -ktx for Kotlin
    // alternatively - just LiveData
    compile "android.arch.lifecycle:livedata:$lifecycle_version"
    // alternatively - Lifecycles only (no ViewModel or LiveData).
    // Support library depends on this lightweight import
    compile "android.arch.lifecycle:runtime:$lifecycle_version"

    annotationProcessor "android.arch.lifecycle:compiler:$lifecycle_version"
    // alternately - if using Java8, use the following instead of compiler
    compile "android.arch.lifecycle:common-java8:$lifecycle_version"

    // optional - ReactiveStreams support for LiveData
    compile "android.arch.lifecycle:reactivestreams:$lifecycle_version"

    // optional - Test helpers for LiveData
    // compile "android.arch.core:core-testing:$lifecycle_version"
    compile 'com.squareup.retrofit2:retrofit:2.1.0'

    compile 'com.squareup.retrofit2:converter-gson:2.1.0'
    compile 'com.facebook.stetho:stetho:1.3.1'

    //  room
    compile 'android.arch.persistence.room:runtime:1.1.0'
    annotationProcessor 'android.arch.persistence.room:compiler:1.1.0'
    compile "android.arch.persistence.room:rxjava2:1.1.0"
}
```

二、Lifecycle 管理生命周期

2.1、Lifecycle 介绍

Lifecycle 组件指的是 android.arch.lifecycle 包下提供的各种类与接口，可以让开发者构建能感知其他组件（主要指Activity 、Fragment）生命周期（lifecycle-aware）的类。

2.2、常规 MVP Presenter 使用

比如我们需要监听某个 Activity 生命周期的变化，在生命周期改变的时候打印日志，一般做法构造回调的方式，先定义基础 BaseActivityPresenter 接口：

```
public interface BaseActivityPresenter extends BasePresenter{

    void onCreate();

    void onStart();

    void onResume();

    void onPause();

    void onStop();

    void onDestroy();

}
```

在实现类中增加自定义操作（打印日志）：

```
public class ActivityPresenter implements BaseActivityPresenter {

    private static String TAG = ActivityPresenter.class.getSimpleName();

    @Override
    public void onCreate() {
        LogUtil.i(TAG, "onCreate()");
    }
```

```
            LogUtil.i(TAG, "onStart()");
        }

        @Override
        public void onResume() {
            LogUtil.i(TAG, "onResume()");
        }

        @Override
        public void onPause() {
            LogUtil.i(TAG, "onPause()");
        }

        @Override
        public void onStop() {
            LogUtil.i(TAG, "onStop()");
        }

        @Override
        public void onDestroy() {
            LogUtil.i(TAG, "onDestroy()");
        }

    }
```

然后在需要监听的 Activity 中依次回调方法：

```
    @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            mBasePresenter = new ActivityPresenter();
        }

        @Override
        protected void onStart() {
            super.onStart();
            mBasePresenter.onStart();
        }

        @Override
        protected void onResume() {
            super.onResume();
            mBasePresenter.onResume();
        }
```

```
        super.onPause();
        mBasePresenter.onPause();
    }

    @Override
    protected void onStop() {
        super.onStop();
        mBasePresenter.onStop();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mBasePresenter.onDestroy();
    }
```

在 Activity 的 onCreate() 方法中创建 BasePresenter，监听 Activity 的生命周期方法。

2.3、使用 Lifecycle

上述写可以实现基础的功能，但是不够灵活，假如除了 ActivityPresenter 类，还有别的类要监听 Activity 生命周期变化，那也需要添加许多生命周期的回调方法，比较繁琐。那我们是否可以当 Activity 生命周期发生变化的时候主动通知需求方呢？答案就是使用 Lifecycle 提供的 LifecycleObserver：

```
public class ActivityLifeObserver implements BaseActivityPresenter,
 LifecycleObserver {

    private String TAG = ActivityLifeObserver.class.getSimpleName();

    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
    @Override
    public void onCreate() {
        LogUtil.i(TAG, "onCreate()");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    @Override
    public void onStart() {
        LogUtil.i(TAG, "onStart()");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
```

```
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    @Override
    public void onPause() {
        LogUtil.i(TAG, "onPause()");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    @Override
    public void onStop() {
        LogUtil.i(TAG, "onStop()");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
    @Override
    public void onDestroy() {
        LogUtil.i(TAG, "onDestroy()");
    }

}
```

让我们的业务类实现 ActivityLifeObserver 接口，同时在每一个方法实现上增加 @OnLifecycleEvent(Lifecycle.Event.XXXX) 注解，OnLifecycleEvent 对应了 Activity 的生命周期方法。被监听的 Actiivty 实现 LifecycleOwner 接口，然后在需要监听的 Activity 中注册：

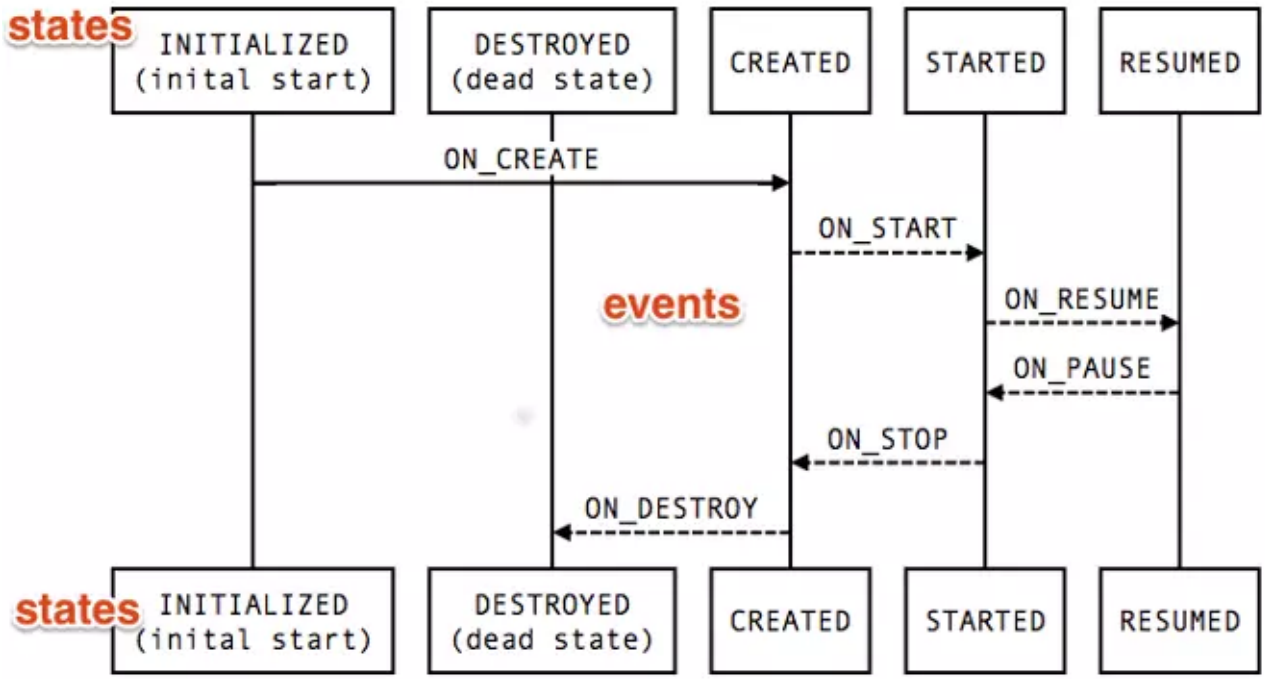
```
public class DetailActivity extends AppCompatActivity implements LifecycleOwner{

    private static String TAG = DetailActivity.class.getSimpleName();
    private LifecycleRegistry mLifecycleRegistry;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_detail);
        mLifecycleRegistry = new LifecycleRegistry(this);
        // 注册需要监听的 Observer
        mLifecycleRegistry.addObserver(new ActivityLifeObserver());
        mLifecycleRegistry.addObserver(new LocationLifeObserver());
    }

    @Override
    public Lifecycle getLifecycle() {
        return mLifecycleRegistry;
    }
```

运行如下：

```
com.troy.androidrc I/ActivityLifeObserver: onCreate()
com.troy.androidrc I/ActivityLifeObserver: onStart()
com.troy.androidrc I/ActivityLifeObserver: onResume()
com.troy.androidrc I/ActivityLifeObserver: onPause()
com.troy.androidrc I/ActivityLifeObserver: onStop()
com.troy.androidrc I/ActivityLifeObserver: onDestroy()
```

其中 Lifecycle 使用两个主要的枚举类来表示其所关联组件的生命周期：

- Event 事件 从组件或者Lifecycle类分发出来的生命周期，它们和Activity / Fragment生命周期的事件一一对应。(ON_CREATE, ON_START, ON_RESUME, ON_PAUSE, ON_STOP, ON_DESTROY)；
- State 状态 当前组件的生命周期状态(INITIALIZED, DESTROYED, CREATED, STARTED, RESUMED)。



LifecycleRegistry 类用于注册和反注册需要观察当前组件生命周期的 Observer，用法如下：

```
//  初始化
mLifecycleRegistry = new LifecycleRegistry(this);
mActivityLifeObserver = new ActivityLifeObserver();
//  注册观察者
mLifecycleRegistry.addObserver(mActivityLifeObserver);
```

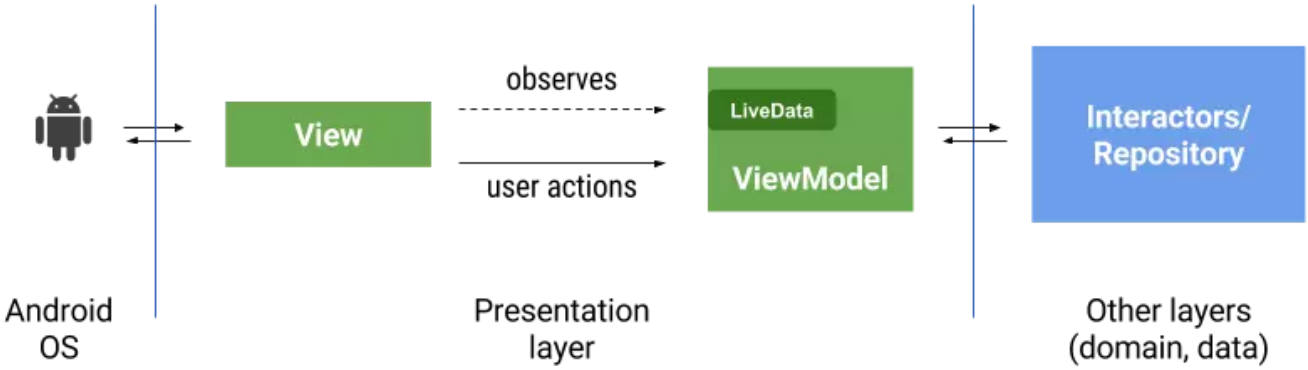
```
//  移除观察者
mLifecycleRegistry.removeObserver(mActivityLifeObserver);
```

三、LiveData && ViewModel

3.1、LiveData && ViewModel 介绍

LiveData 是一种持有可被观察数据的类（an observable data holder class）。和其他可被观察的类不同的是，LiveData是有生命周期感知能力的（lifecycle-aware,），这意味着它可以在 activities, fragments, 或者 services 生命周期是活跃状态时更新这些组件。

ViewModel 与 LiveData 之间的关系图如下：



3.2、LiveData && ViewModel 使用

在 Activity 页面有一 TextView，需要展示用户 User 的信息，User 类定义：

```
public class User {

    public String userId;

    public String name;

    public String phone;

    @Override
    public String toString() {
        return "User{" +
```

```
                '}';
        }
    }
```

常规的做法：

```
// 获取 User 的数据后
mTvUser.setText(user.toString());
```

这样做的一个问题，如果获取或者修改 User 的来源不止一处，那么需要在多个地方更新 TextView，并且如果在多处 UI 用到了 User，那么也需要在多处更新。

使用 LiveData 与 ViewModel 的组合，将LiveData 持有 User 实体，作为一个被观察者，当 User 改变时，所有使用 User 的地方自动 change。构建一个 UserViewModel 如下：

```
public class UserViewModel extends ViewModel
 implements BaseViewModel<User> {

    private String TAG = UserViewModel.class.getSimpleName();

    private MutableLiveData<User> liveUser;

    public MutableLiveData<User> getData(){
        if(liveUser == null){
            liveUser = new MutableLiveData<User>();
        }

        liveUser.setValue(loadData());
        return this.liveUser;
    }

    public void changeData(){
        if(liveUser != null){
            liveUser.setValue(loadData());
        }
    }

    @Override
    public User loadData() {
        User user = new User();
        user.userId = RandomUtil.getRandomNumber();
        user.name = RandomUtil.getChineseName();
```

```
            return user;
        }

        @Override
        public void clearData() {

        }
    }
```

自定义的UserViewModel 继承系统的 ViewModel，将 User 封装成 MutableLiveData：

```
if(liveUser == null){ liveUser = new MutableLiveData<User>(); }
```

在使用User 的地方增加观察：

```
//  view model.observe
mUserViewModel = ViewModelProviders.of(this).get(UserViewModel.class);
mUserViewModel.getData().observe(this, new Observer<User>() {
   @Override
   public void onChanged(@Nullable User user) {
       if(user != null){
           mTvUser.setText(user.toString());
       }
   }
});
```

数据源发送改变的时候：

```
//      改变 User 内容
mButtonUser.setOnClickListener(new View.OnClickListener() {
     @Override
     public void onClick(View v) {
         if(mUserViewModel != null && mUserViewModel.getData() != null){
             mUserViewModel.changeData();
         }
     }
});

//      setValue
public void changeData(){
   if(liveUser != null){
        liveUser.setValue(loadData());
    }
}
```

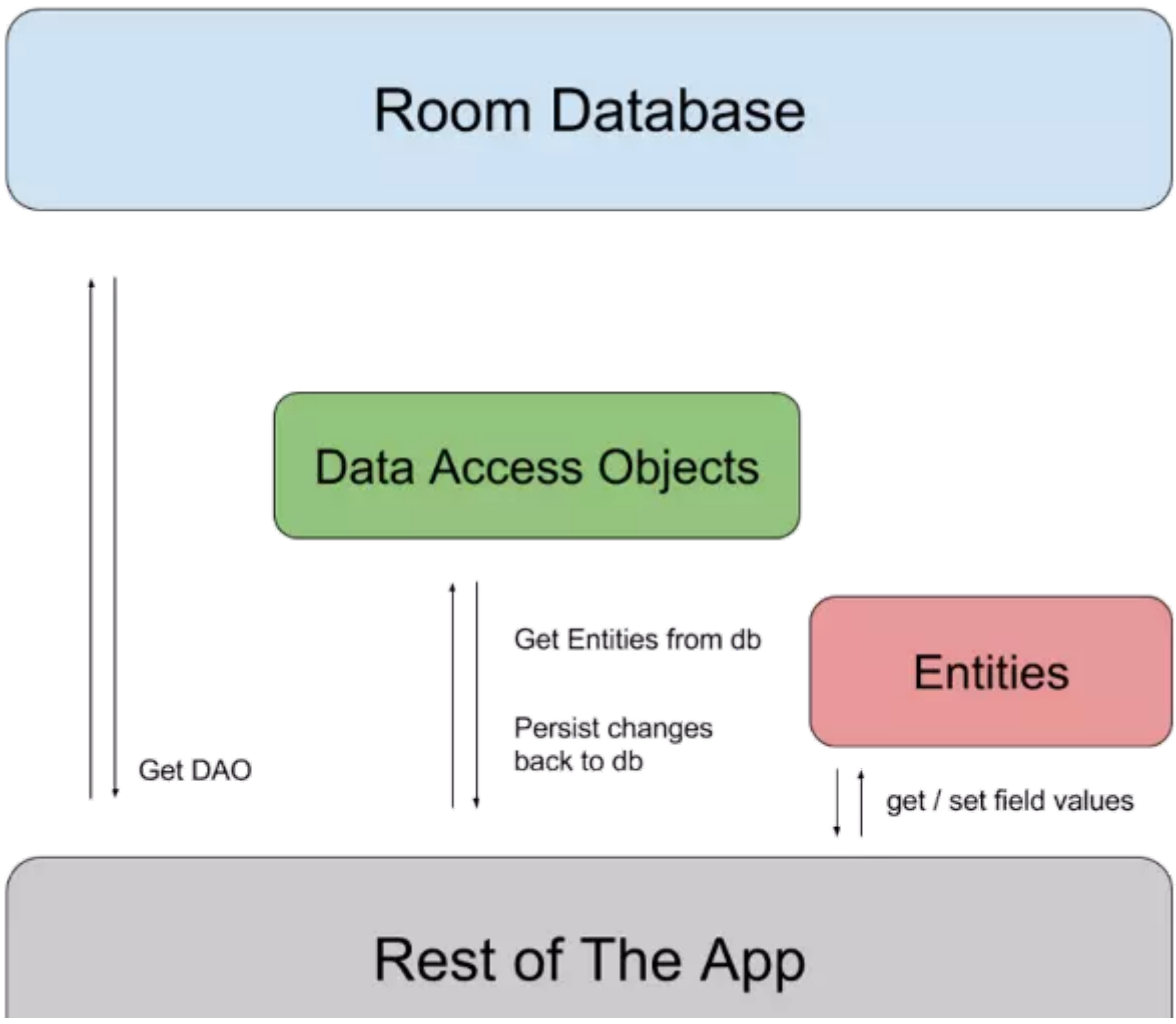
```
com.troy.androidrc I/DetailActivity:
 User{userId='9372622', name='邓楠', phone='15607043749'}

com.troy.androidrc I/DetailActivity:
 User{userId='6099877', name='文瑾慧', phone='13005794027'}
```

四、Room

4.1、Room 介绍

Room 持久层库提供了一个方便我们访问 SQLite 数据库的抽象层(an abstraction layer)，帮助我们更好的在 APP 上创建我们的数据缓存，能够让 APP 即使在没有网络的情况也能正常使用。

Room 的架构如下：



4.2、Room 使用与主要注解

创建包含订单表的数据库如下步骤：

1、创建 Order.java:

```
@Entity(tableName = "orders")
public class Order {

    @PrimaryKey
    @ColumnInfo(name = "order_id")
    public long orderId;

    @ColumnInfo(name = "address")
    public String address;

    @ColumnInfo(name = "owner_name")
    public String ownerName;

    @ColumnInfo(name = "owner_phone")
    public String ownerPhone;

    //  指示 Room 需要忽略的字段或方法
    @Ignore
    public String ignoreText;

    @Embedded
    public OwnerAddress ownerAddress;
}
```

2、创建 OrderDao：

```
@Dao
public interface OrderDao {

    @Query("SELECT * FROM orders")
    List<Order> loadAllOrders();

    @Insert
    void insertAll(Order... orders);

    @Query("SELECT * FROM orders WHERE order_id IN (:orderIds)")
    List<Order> queryOrderById(long[] orderIds);
```

```
    @Update
    void updateOrder(Order... orders);
}
```

3、创建数据库

```
@Database(entities = {Order.class, AddressInfo.class}, version = 2)
public abstract class AppDatabase extends RoomDatabase{

   public abstract OrderDao getOrderDao();

}

// 实现类
public static void buildDb(){
DB_INSTANCE = Room.
    databaseBuilder(TroyApplication.getInstance(), AppDatabase.class, "troy_db")   //  指定数
    .addCallback(new RoomDatabase.Callback() {
        @Override
        public void onCreate(@NonNull SupportSQLiteDatabase db) {
            super.onCreate(db);  //  数据库创建回调;
            LogUtil.i(TAG, "onCreate");
        }

        @Override
        public void onOpen(@NonNull SupportSQLiteDatabase db) {
            super.onOpen(db);   //  数据库使用回调;
            LogUtil.i(TAG, "onOpen");
        }
    })
    .allowMainThreadQueries()   // 数据库操作可运行在主线程
    .build();

}
```

使用到的主要注解:

- @Entity(tableName = "orders") // 定义表名;
- @PrimaryKey // 定义主键;
- @ColumnInfo(name = "order_id") // 定义数据表中的字段名;
- @Ignore // 指示 Room 需要忽略的字段或方法;
- @Embedded // 指定嵌入实体

- @Delete // 定义删除数据接口；
- @Update // 定义更新数据接口；
- @Database // 定义数据库信息，表信息，数据库版本

3.3、增删改查实现

增：

```
//  1、插入接口声明
@Insert
void insertAll(Order... orders);

//  2、插入接口实现
@Override
  public void insertAll(Order... orders) {
    __db.beginTransaction();
    try {
      __insertionAdapterOfOrder.insert(orders);
      __db.setTransactionSuccessful();
    } finally {
      __db.endTransaction();
    }
}

//  3、插入接口调用
AppDatabase db = DbManager.getDbInstance();
OrderDao orderDao = db.getOrderDao();
Order order = Order.createNewOrder();
orderDao.insertAll(order);
```

删：

```
//      1、删除接口声明
@Delete
void deleteOrder(Order... orders);

//      2、删除接口实现
@Override
  public void deleteOrder(Order... orders) {
    __db.beginTransaction();
    try {
      __deletionAdapterOfOrder.handleMultiple(orders);
```

```
        }
    }

    //      3、删除接口调用
    AppDatabase db = DbManager.getDbInstance();
    OrderDao orderDao = db.getOrderDao();
    orderDao.deleteOrder(orderList.get(orderList.size() - 1));
```

改:

```
    //      1、修改接口声明
     @Update
     void updateOrder(Order... orders);

    //      2、修改接口实现
    @Override
      public void updateOrder(Order... orders) {
        __db.beginTransaction();
        try {
          __updateAdapterOfOrder.handleMultiple(orders);
          __db.setTransactionSuccessful();
        } finally {
          __db.endTransaction();
        }
    }

    //      3、修改接口调用
    AppDatabase db = DbManager.getDbInstance();
    OrderDao orderDao = db.getOrderDao();
    Order order = orderList.get(orderList.size() - 1);
    order.ownerName = "update - " + RandomUtil.getChineseName();
    orderDao.updateOrder(order);
```

查:

```
    //      1、查询接口声明
    @Query("SELECT * FROM orders")
    List<Order> loadAllOrders();

    //      2、查询接口实现
    @Override
      public List<Order> loadAllOrders() {
        final String _sql = "SELECT * FROM orders";
        final RoomSQLiteQuery _statement = RoomSQLiteQuery.acquire(_sql, 0);
```

```
        final int _cursorIndexOfOrderId = _cursor.getColumnIndexOrThrow("order_id");
        final int _cursorIndexOfAddress = _cursor.getColumnIndexOrThrow("address");
        final int _cursorIndexOfOwnerName = _cursor.getColumnIndexOrThrow("owner_name");
        final int _cursorIndexOfOwnerPhone = _cursor.getColumnIndexOrThrow("owner_phone");
        final int _cursorIndexOfStreet = _cursor.getColumnIndexOrThrow("street");
        final int _cursorIndexOfState = _cursor.getColumnIndexOrThrow("state");
        final int _cursorIndexOfCity = _cursor.getColumnIndexOrThrow("city");
        final int _cursorIndexOfPostCode = _cursor.getColumnIndexOrThrow("post_code");
        final List<Order> _result = new ArrayList<Order>(_cursor.getCount());
        while(_cursor.moveToNext()) {
          final Order _item;
          final Order.OwnerAddress _tmpOwnerAddress;
          if (! (_cursor.isNull(_cursorIndexOfStreet) && _cursor.isNull(_cursorIndexOfState)
            _tmpOwnerAddress = new Order.OwnerAddress();
            _tmpOwnerAddress.street = _cursor.getString(_cursorIndexOfStreet);
            _tmpOwnerAddress.state = _cursor.getString(_cursorIndexOfState);
            _tmpOwnerAddress.city = _cursor.getString(_cursorIndexOfCity);
            _tmpOwnerAddress.postCode = _cursor.getInt(_cursorIndexOfPostCode);
          }  else  {
            _tmpOwnerAddress = null;
          }
          _item = new Order();
          _item.orderId = _cursor.getLong(_cursorIndexOfOrderId);
          _item.address = _cursor.getString(_cursorIndexOfAddress);
          _item.ownerName = _cursor.getString(_cursorIndexOfOwnerName);
          _item.ownerPhone = _cursor.getString(_cursorIndexOfOwnerPhone);
          _item.ownerAddress = _tmpOwnerAddress;
          _result.add(_item);
        }
        return _result;
      } finally {
        _cursor.close();
        _statement.release();
      }
    }

//      3、查询接口调用
AppDatabase db = DbManager.getDbInstance();
OrderDao orderDao = db.getOrderDao();
return orderDao.loadAllOrders();
```

3.4、内嵌实体

如果实体 Order 内部包含地址信息，地址信息分别包含 城市，邮政等信息，可以这样写，使用 @Embedded 注解：

```
static class OwnerAddress {

        public String street;
        public String state;
        public String city;

        @ColumnInfo(name = "post_code")
        public int postCode;
}

@Embedded
public OwnerAddress ownerAddress;
```

3.5、配合 LiveData

数据查询可以返回 LiveData 数据：

```
 @Query("SELECT * FROM orders")
 LiveData<List<Order>> loadAllOrderData();
```

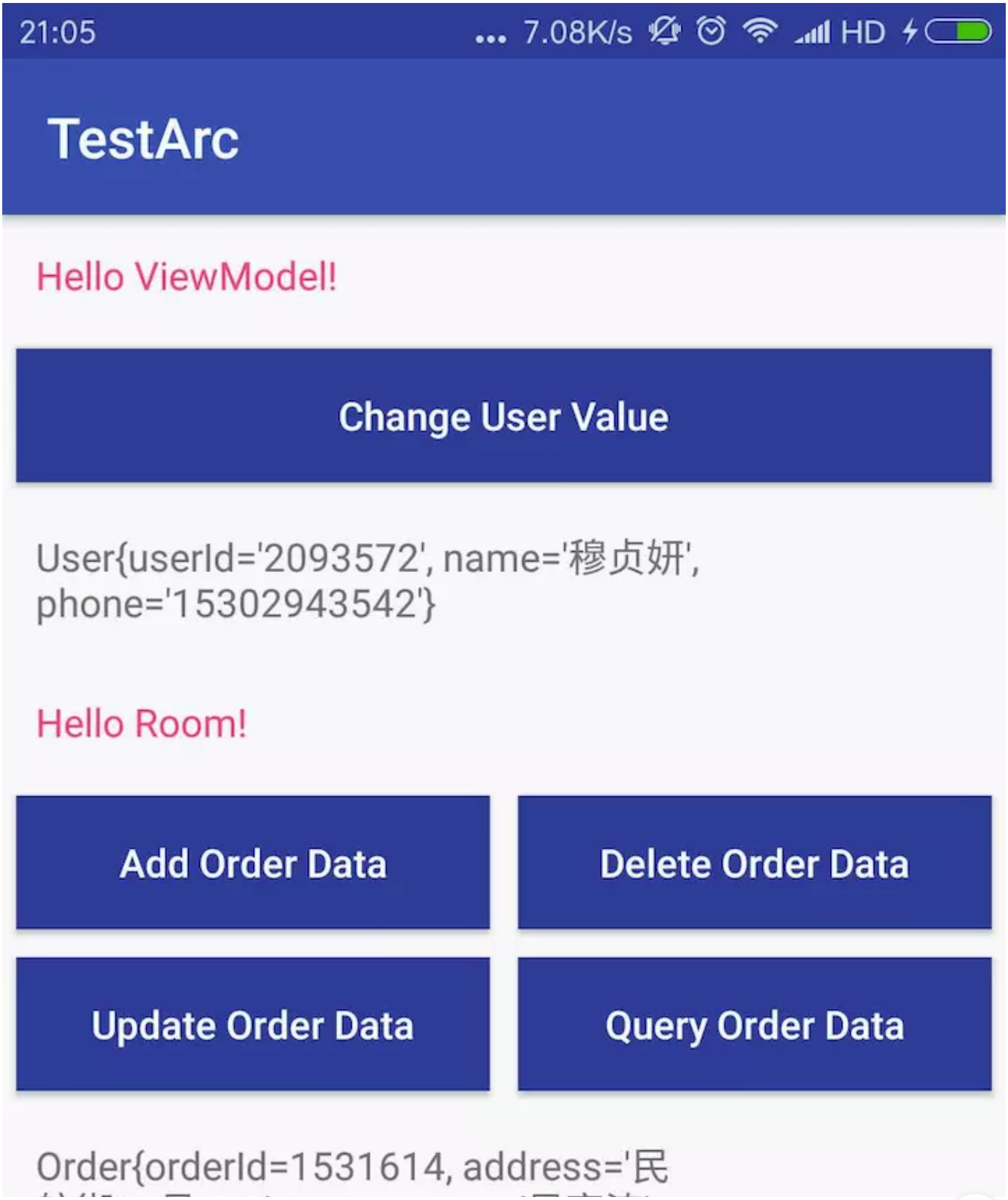
3.6、配合 RxJava

通过 query 查询返回的实体，可以封装成 对应RxJava 的操作符封装对象，例如 Flowable，Maybe 等：

```
//      接口声明
@Query("SELECT * from orders where order_id = :id LIMIT 1")
Flowable<Order> queryOrderByIdV2(long id);

@Query("SELECT * from orders where order_id = :id LIMIT 1")
Maybe<Order> queryOrderByIdV3(long id);

// 接口调用
private Maybe<Order> queryOrderV3(){
      AppDatabase db = DbManager.getDbInstance();
      OrderDao orderDao = db.getOrderDao();
      return orderDao.queryOrderByIdV3(10001);
}

Maybe<Order> orderMaybe = queryOrderV3();
```

```
                @Override
                public void accept(@NonNull Order order) throws Exception {

                }
        });
```

以上代码Demo 实现：

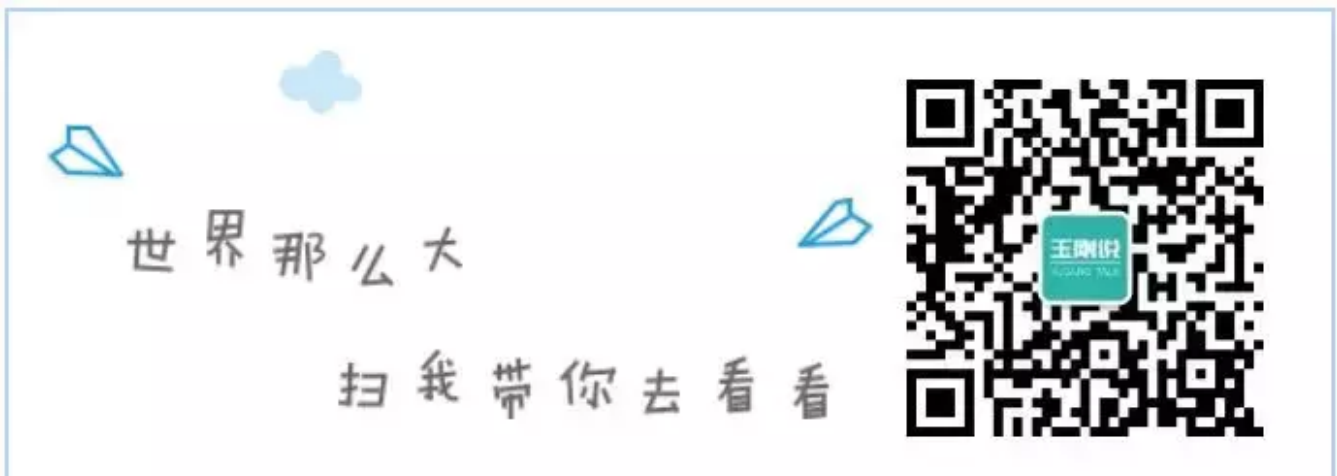


五、总结

学会使用 Android Architecture Components 提供的组件简化我们的开发，能够使我们开发的应用模块更解耦更稳定，视图与数据持久层分离，以及更好的扩展性与灵活性，

参考致谢：

1. Architechture
2. Lifecycle package class
3. Save data in a local database using Room
4. Android Room with a View
5. Google Samples



关注下面的标签，发现更多相似文章

Android 架构 数据库 gradle

评论

输入评论...

Vljolie 安卓工程师
demo的github地址能发下么?
3月前 回复

Vljolie 安卓工程师
我想问下Activity为例，注册观察者像这样：
mLifecycleRegistry.addObserver(mActivityLifeObserver);
那么移除观察者，在什么时候调用啊
mLifecycleRegistry.removeObserver(mActivityLifeObserver);
3月前 回复

未央 Android开发、RN开发...
Android Architecture Components 和 MVP怎么能够优雅的结合呢？只是简单的将MVP中的M由ViewModel代替吗?
4月前 回复

CodeLin
简化了么，我怎么感觉一般的应用场景用起来都复杂化了
4月前 回复

Alvince咩咩 高级Android工程师 @...
emmm...Presenter 不是这么用的
4月前 回复

singwhatiwanna 专家工程师 @ 滴...
回复Alvince咩咩: 可以详细一点么，我给作者反馈下
4月前

回复singwhatiwanna: presenter里面写的生命是没有什么用的，我认为是用不上的，只有attachView和detachView注入view对象有用。而且MVP或者MVVM，presenter都是要解耦的，直接在activity里new肯定是耦合的，一般都是用dagger2这种实例化presenter。

4月前

相关推荐

专栏 · 云水木石 · 15小时前 · 人工智能 / Android

移动设备上的多位数字识别

2

荐 · liutao · 1天前 · GitHub / 数据库

GitHub 服务中断 24 小时 11 分钟事故分析报告

175 3

专栏 · patience在掘金 · 14小时前 · PHP / Redis

php订单延时处理–延时队列

专栏 · Keelvin · 23小时前 · JavaScript / iOS

JSBridge实战

76

专栏 · 网易云社区 · 1天前 · 命令行 / 数据库

结合jenkins以及PTP平台的性能回归测试

1

专栏 · 你如世间春秋 · 1天前 · GitHub / Android

优雅地实现Android主流图片加载框架封装，可无侵入切换框架

56 7

专栏 · ScottSong · 17小时前 · Kotlin / Java

Kotlin的特点及各版本新特性

24 2

专栏 · 贝聊科技 · 1天前 · 后端 / 架构

专栏 · 顾家进 · 1天前 · 后端 / 数据库

单手撸了个springboot+mybatis+druid

31 6

民工哥技术之路 · 1天前 · 数据库 / MySQL / 服务器

这十个MySQL经典错误，老司机一定遇到过！你呢?

31