## 序列化（Serializable）

序列化：将对象转化为一个字节序列，实现了对象的持久化，便于储存。

反序列化：将序列化的字节序列还原。

用到的对象为ObjectOutputStream和ObjectInputStream，用到的方法为writeObject()和readObject()。

Serializable接口是一个标记接口，不包含任何方法或者字段，仅用于标识可序列化的语义。

* 方法一：实现Serializable接口实现隐式序列化

该方法会自动序列化所有非static和transient关键字修饰的成员变量。（JVM会忽略transient变量的原始值并将默认值保存到文件中）

方法一使用字节将对象直接还原，虽然未调用构造方法，但必须确保读取程序的classpath中包含对象的class文件，否则会抛出异常ClassNotFoundException。

|  |
| --- |
| **class** Student **implements** Serializable {  **private int age**;  **private** String **name**;  **public static int** *qq* = 1234;  **private transient** String **address** = **"China"**;   **public** Student(**int** age, String name) {  **this**.**age** = age;  **this**.**name** = name;  }  **public void** setAge(**int** age) {  **this**.**age** = age;  }  **public** String toString() {  **return "age:"** + **age** + **" name:"** + **name** + **" qq:"** + *qq* + **" address:"** + **address**;  } }  **public class** SeriDemo {  **public static void** main(String[] args) {  Student stu = **new** Student(14, **"David"**);  System.***out***.println(stu);   ByteArrayOutputStream buff = **new** ByteArrayOutputStream();  **try** {  ObjectOutputStream out = **new** ObjectOutputStream(buff);  out.writeObject(stu);  out.close();  *// 改变相关值* Student.*qq* = 6666;  stu.setAge(16);   ByteArrayInputStream bais = **new** ByteArrayInputStream(buff.toByteArray());  ObjectInputStream in = **new** ObjectInputStream(bais);  Student student = (Student) in.readObject();   *// static成员不属于对象实例，不会被序列化；address被transient修饰不会被序列化，故为空* System.***out***.println(student);  } **catch** (Exception e) {  e.printStackTrace();  }  } } |

* 方法二：实现Externalizable接口实现显示序列化

该方法可以通过实现writeExternal()和readExternal()方法，手动进行序列化（包括static和transient属性）。

该方法获取对象时，Externalizable类首先会调用对象的公共无参构造方法初始化对象，其次在进行数据还原。因此若对象缺少公共无参构造方法会报错。

|  |
| --- |
| **class** Student1 **implements** Externalizable {  **private int age**;  **public static** String *name*;  **private int id**;   **public** Student1(**int** age, String name, **int** id) {  **this**.**age** = age;  **this**.*name* = name;  **this**.**id** = id;  }  *// 必须给无参构造方法* **public** Student1() {}   @Override  **public** String toString() {  **return "age:"** + **age** + **" name:"** + *name* + **" id:"** + **id**;  }   @Override  **public void** writeExternal(ObjectOutput out) **throws** IOException {  out.writeObject(*name*);  out.writeInt(**age**);  }   @Override  **public void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException {  *name* = (String) in.readObject();  **age** = in.readInt();  } }  **public class** ExterDemo {  **public static void** main(String[] args) {  Student1 stu = **new** Student1(14, **"David"**, 1);  System.***out***.println(stu);  **try** {  *// 保存对象* ByteArrayOutputStream buff = **new** ByteArrayOutputStream();  ObjectOutputStream out = **new** ObjectOutputStream(buff);  out.writeObject(stu);  out.close();   Student1.*name* = **"hh"**;   *// 获取对象* ObjectInputStream in = **new** ObjectInputStream(**new** ByteArrayInputStream(buff.toByteArray()));  Student1 newStu = (Student1) in.readObject();  System.***out***.println(newStu);  } **catch** (Exception e) {  e.printStackTrace();  }  } } |

* 方法三：实现Serializable接口+添加writeObject()和readObject()方法

**注意点：**

1) writeObject()和readObject()方法必须被private修饰。（内部通过反射调用）

2) 这两个方法第一行必须调用默认的defaultRead()和defaultWriteObject()方法。

3) 显示化定义需要保存的static和transient成员变量。

|  |
| --- |
| **class** Student2 **implements** Serializable {  **private int id**;  **private** String **name**;  **private transient int age**;   **public** Student2(**int** id, String name, **int** age) {  **this**.**id** = id;  **this**.**name** = name;  **this**.**age** = age;  }   **private void** writeObject(ObjectOutputStream out) **throws** IOException {  out.defaultWriteObject();  out.writeInt(**age**);  }   **private void** readObject(ObjectInputStream in) **throws** IOException, ClassNotFoundException {  in.defaultReadObject();  **age** = in.readInt();  }   @Override  **public** String toString() {  **return "id:"** + **id** + **" name:"** + **name** + **" age:"** + **age**;  } }  **public class** SerExrDemo {   **public static void** main(String[] args) {  Student2 stu = **new** Student2(1, **"David"**, 14);  System.***out***.println(stu);   ByteArrayOutputStream buff = **new** ByteArrayOutputStream();  **try** {  ObjectOutputStream out = **new** ObjectOutputStream(buff);  out.writeObject(stu);  out.close();   *// 获取数据* ObjectInputStream in = **new** ObjectInputStream(**new** ByteArrayInputStream(buff.toByteArray()));  Student2 newStu = (Student2) in.readObject();  System.***out***.println(newStu);  } **catch** (Exception e) {  e.printStackTrace();  }  } } |

注意：

* 序列化时，只对对象的状态进行保存，而不管对象的方法。
* 当一个父类实现序列化时，子类自动实现序列化。
* 当一个对象的实例变量引用其他对象，序列化该对象时也把引用对象进行序列化。
* 因为java的枚举类型都会默认继承类java.lang.Enum，而该类实现了Serializable接口，所以枚举类型对象都是可以默认序列化的。
* 写入读取完后需要调用close()方法关闭流。
* 对于单例模式，若想要获得相同的对象，需要加入readResolve()方法，不论实现Serializable/Externalizable接口，都用调用readResolve()方法，用该方法返回的对象替换反序列化过程中创建的对象，而被创建的对象则会被垃圾回收掉。

|  |
| --- |
| **class** Person **implements** Serializable {  **private static class** InstanceHolder {  **private static final** Person ***instance*** = **new** Person();  }   **public static** Person getInstance() {  **return** InstanceHolder.***instance***;  }   **private** Person() {}   *// 该方法可是写入和读出的对象是一个对象* **private** Object readResolve() **throws** ObjectStreamException {  **return** InstanceHolder.***instance***;  } }  **public class** SingletonDemo {  **public static void** main(String[] args) {  Person person = Person.*getInstance*();  File file = **new** File(**"singletonDemo.txt"**);  ObjectOutputStream out = **null**;  ObjectInputStream in = **null**;  **try** {  out = **new** ObjectOutputStream(**new** FileOutputStream(file));  out.writeObject(person);   in = **new** ObjectInputStream(**new** FileInputStream(file));  Person newPerson = (Person) in.readObject();  **if** (person == newPerson) {  System.***out***.println(**"Yes"**);  } **else** {  System.***out***.println(**"No"**);  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if** (out != **null**) {  out.close();  }  **if** (in != **null**) {  in.close();  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  } } |

Rxjava

1. 响应式编程是一种面向数据流和变化传播的编程范式。

数据流：只能以事先规定好的顺序被读取一次的数据的一个序列。

变化传播：类似观察者模式，要将变化传播出去。

1. Rxjava是异步基于回调的事件分发、消息传递的库，是基于观察者模式和支持链式调用的。

四个基本概念：Observable（被观察者）、Observer（观察者）、subscribe（订阅）、事件

1. java中观察者与被观察者简单使用

|  |
| --- |
| **public class** SimpleObserver **implements** Observer {  **public** SimpleObserver(SimpleObservable observable) {  observable.addObserver(**this**);  }  @Override  **public void** update(Observable o, Object arg) {  System.***out***.println(**"data is changed"** + ((SimpleObservable)o).getData());  } } |
| **public class** SimpleObservable **extends** Observable {  **private int data**;  **public int** getData() {  **return data**;  }   **public void** setData(**int** data) {  **if** (**this**.**data** != data) {  **this**.**data** = data;  *// 被观察者发出通知前要调用setChanged()方法* setChanged();  notifyObservers();  }  } } |
| **public class** MyTest {  **public static void** main(String[] args) {  SimpleObservable observable = **new** SimpleObservable();  SimpleObserver observer = **new** SimpleObserver(observable);  observable.setData(1); *// 打印data is changed1*  observable.setData(2); *// 打印data is changed2*  observable.setData(2);  } } |

1. 使用

a. 添加依赖

|  |
| --- |
| implementation **'io.reactivex.rxjava2:rxjava:2.1.13'** implementation **'io.reactivex.rxjava2:rxandroid:2.0.2'** |

b. 分别创建Observer、Observable、Subscribe

* 通过onCreate()方法创建Observable对象
* 通过fromArray(Integer[] items)方法创建Observable对象（该方法使用时被观察者返回的对象一般为数值类型）
* 通过interval()方法创建Observable对象（创建一个按固定时间间隔发射整数序列的Observable，可用作定时器）
* 通过just()方法处理数组集合，顺序输出数组元素。
* 通过range()方法输出某一范围的值。

|  |
| --- |
| **public static void** createObservable() {  *// 定义被观察者* Observable.*create*(**new** ObservableOnSubscribe<String>() {  @Override  **public void** subscribe(ObservableEmitter<String> emitter) **throws** Exception {  Log.*e*(***TAG***, **"currentThread"** + Thread.*currentThread*().getName());  emitter.onNext(**"1"**);  emitter.onNext(**"2"**);  emitter.onComplete();  }  }).subscribe(**new** Observer<String>() {  @Override  **public void** onSubscribe(Disposable d) {  Log.*e*(***TAG***, **"onSubscribe"**);  }   @Override  **public void** onNext(String s) {  Log.*e*(***TAG***, **"onNext"** + s);  }  @Override  **public void** onError(Throwable e) {  Log.*e*(***TAG***, **"onError"**);  }  @Override  **public void** onComplete() {  Log.*e*(***TAG***, **"onComplete"**);  }  }); } |

c. 示例

|  |
| --- |
| **public class** DownLoadUtils {   **private static** OkHttpClient *mClient*;  **public** DownLoadUtils() {  *mClient* = **new** OkHttpClient();  }  */\*\*  \* 声明一个观察者对象作为结果返回  \** ***@param path*** *\** ***@return*** *\*/* **public static** Observable<**byte**[]> downLoadImage(**final** String path) {  **return** Observable.*create*(**new** ObservableOnSubscribe<**byte**[]>() {  @Override  **public void** subscribe(**final** ObservableEmitter<**byte**[]> emitter) **throws** Exception {  **final** Request request = **new** Request.Builder().url(path).build();  *mClient*.newCall(request).enqueue(**new** Callback() {  @Override  **public void** onFailure(Call call, IOException e) {  emitter.onError(e);  }   @Override  **public void** onResponse(Call call, Response response) **throws** IOException {  **if** (response.isSuccessful()) {  **byte**[] data = response.body().bytes();  **if** (data != **null**) {  emitter.onNext(data);  }  emitter.onComplete();  }  }  });  }  });  } } |
| **public class** Main2Activity **extends** AppCompatActivity {   **private static final** String ***TAG*** = Main2Activity.**class**.getSimpleName();  **private** ImageView **mImgPic**;   **private** String **path** = **"https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1563729481095&di=c049552c44deea6c223942baa64a7a06&imgtype=0&src=http%3A%2F%2Fimg.9ku.com%2Fgeshoutuji%2Fsingertuji%2F1%2F16431%2F16431\_1.jpg"**;  **private** DownLoadUtils **mLoadUtils**;   @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.***activity\_main2***);  **mImgPic** = (ImageView) findViewById(R.id.***img\_pic***);  **mLoadUtils** = **new** DownLoadUtils();  }   **public void** download(View view) {  *// Schedulers.io()表示使用http访问数据* **mLoadUtils**.*downLoadImage*(**path**).subscribeOn(Schedulers.*io*())  *// 运行在UI主线程上* .observeOn(AndroidSchedulers.*mainThread*())  .subscribe(**new** Observer<**byte**[]>() {  @Override  **public void** onSubscribe(Disposable d) {   }   @Override  **public void** onNext(**byte**[] bytes) {  Bitmap bitmap = BitmapFactory.*decodeByteArray*(bytes, 0, bytes.**length**);  **mImgPic**.setImageBitmap(bitmap);  }   @Override  **public void** onError(Throwable e) {  Log.*i*(***TAG***, e.getMessage());  }   @Override  **public void** onComplete() {  *// 通常会使对话框消失* Log.*i*(***TAG***, **"onComplete"**);  }  });  } } |
| *// 示例*  *// 如果对于返回的数据进行类型装换，可以通过map方法*  **mUtils**.*pdfDownLoad*(url).subscribeOn(Schedulers.*io*())  .observeOn(AndroidSchedulers.*mainThread*())  .map(**new** Function<ResponseBody, Boolean>() {  @Override  **public** Boolean apply(ResponseBody responseBody) **throws** Exception {  **return** writeToDisk(filePath, responseBody);  }  }).subscribe(**new** Consumer<Boolean>() {  @Override  **public void** accept(Boolean aBoolean) **throws** Exception {  **if** (aBoolean) {  *// 处理下载成功的UI* }  } }, **new** Consumer<Throwable>() {  @Override  **public void** accept(Throwable throwable) **throws** Exception {  Log.*e*(***TAG***, throwable.getMessage());  } }); |

JetPack

Navigation

1. 一个可简化Android导航的库和插件，是用来管理Fragment切换的工具类。
2. 优点：处理Fragment的切换，为过度和动画提供标准化的资源，实现和处理深层连接、可以绑定Toolbar、BottomNavigationView和ActionBar等、safeArgs（Gradle插件）数据传递时提供类型安全性、支持ViewModel。
3. 关键三元素：

NavigationGraph：新的资源文件，用户在可视化界面可以看出他能够到达的Destination。

NavHostFragment：当前Fragment的容器。

NavController：导航的控制者。

1. 使用

**a.** 添加依赖

模块层的build.gradle文件

|  |
| --- |
|  |
|  |

当使用safeArgs插件时，项目目录下的build.gradle文件

|  |
| --- |
|  |

**b.** 创建navigation导航

res/navigation/

分页库的设计美学：Paging

1. Paging可以使开发者更轻松在RecyclerView中分页加载数据。
2. 当一条新的item插入到数据库，DataSource会被初始化，LiveData后台线程就会创建一个新的PagedList。这个新的的PagedList会被发送到UI线程的PagedListAdapter中，PagedListAdapter使用DiffUtil对比现在item和新建item的差异，对比结束后，PagedListAdapter通过RecycleView.Adapter.notifyItemInserte()插入新的item到适当位置。
3. Paging封装了三种不同的抽象类PageKeyedDataSource、ItemKeyedDataSource、PositionDataSource
4. PagedList通过DataSource加载数据，通过配置Config属性可以改变如下加载数据方式：

setPageSize：设置每一页加载的数量；

setInitialLoadSizeHint：设置首次加载的数量；

setPrefetchDistance：设置距离最后还有多少个item时，即开始加载下一页的数据；

setEnablePlaceholders：表示是否设置null占位符；

1. PagedListAdapter是RecyclerView.Adapterd 的实现，用于展示PagedList数据。
2. 数据加载：我们可以通过LivePagedLIstBuilder来创建LiveData为UI提供数据。如果数据源有多种，我们需要为PagedList设置BoundaryCallback来监听本地数据是否加载完成，当本地数据加载完成后进行网络数据加载。

杂七杂八

1. attachBaseContext()方法

* ContextWrapper中有一个attachBaseContext()方法，该方法会将传入的context对象赋值给mBase。Application类继承了ContextWrapper类，其中方法执行顺序：构造方法 -> attachBaseContext() -> onCreate()
* 一个应用程序可以认为是一个工作环境，用户在这个环境中会切换到不同的场景（Context）
* Context为一个抽象类，其的两个子类分工明确，其中ContextImpl是Context的具体实现类，ContextWrapper是Context的包装类，Application、Service直接继承自ContextWrapper，Activity通过ContextThemeWrapper继承自ContextWrapper，但是他们的初始化过程中都会创建ContextImpl对象。
* 由于Context的具体实例是由ContextImpl类去实现的，因此在绝大多数场景下，Activity、Service和Application这三种类型的Context都是可以通用的。但是当进行界面跳转、Dialog弹出、布局渲染时推荐使用Activity类的Context。
* 获取Context

View.getContext()

Activity.getApplicationContext()：获得当前应用进程的Context对象

ContextWrapper.getBaseContext()：不推荐使用

Acticity.this

* getApplication()和getApplicationContext()得到的为同一个对象。
* 正确使用Context：

1：当Application的Context能搞定的情况下，并且生命周期长的对象，优先使用Application的Context。

2：不要让生命周期长于Activity的对象持有到Activity的引用。

3：尽量不要在Activity中使用非静态内部类，因为非静态内部类会隐式持有外部类实例的引用，如果使用静态内部类，将外部实例引用作为弱引用持有。

1. HttpLoggingInterceptor消息拦截器

* 有时候在排错的时候需要先从接口的相应中作出分析，利用消息拦截器可以清楚的看到接口返回的所有内容，不需要重新使用fidder等抓包工具来做分析。
* 创建对象时可以传一个HttpLoggingInterceptor.Logger(){}的匿名内部类，用来更改日志保存的位置。
* setLevel()方法用来设置日志打印级别，包括四个级别：

BASEIC：请求/相应行

HEADER：请求/相应行 + 头

BODY：请求/相应行 + 头 + 体

NONE：不打印

打印消息：interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);

不打印消息：interceptor.setLevel(HttpLoggingInterceptor.Level.NONE);

### 反射

1. 获得名字的方法

getCanonicalName() 获取类从java语言规范定义的格式输出。

getName() 返回实体类型名称

getSimpleName() 返回源代码中返回实例的名称。

注：数组中getName()通过[L表示数组

|  |
| --- |
| *// 内部类*  **public static void** main(String[] args) {  Class<MySingleton> clazz = MySingleton.**class**;  Class<?>[] classes = clazz.getDeclaredClasses();  **for** (Class cla : classes) {  System.***out***.println(cla.getSimpleName()); *// InstanceHolder* System.***out***.println(cla.getCanonicalName());  *// com.singleton.MySingleton.InstanceHolder* System.***out***.println(cla.getName());  *// com.singleton.MySingleton$InstanceHolder* } } |
| *// 匿名类*  Class clazz = **new** Object(){}.getClass(); System.***out***.println(clazz.getSimpleName()); *// 没有输出* System.***out***.println(clazz.getCanonicalName()); *// null* System.***out***.println(clazz.getName()); *// com.singleton.Test$1* |
| *// 数组*  Class clazz = **new** Object[2].getClass(); System.***out***.println(clazz.getSimpleName()); *// Object[]* System.***out***.println(clazz.getCanonicalName()); *// java.lang.Object[]* System.***out***.println(clazz.getName()); *// [Ljava.lang.Object;* |

**Retrofit**

* 注解参数

@Query：会以?num=1&page=1的方式添加到url后边（常用于GET请求）

@QueryMap：类似@Query，不过其可以通过一个map数组往接口里注入参数（常用于GET请求）

@Path：会替换掉url中被‘{}’包括的部分（常用于POST请求）

@Body：可以用来注解大部分东西，包括map、实体类等（常用于POST请求）

@Headers：添加固定的请求头，作用于方法

@Header：添加不固定的请求头，作用于方法的参数

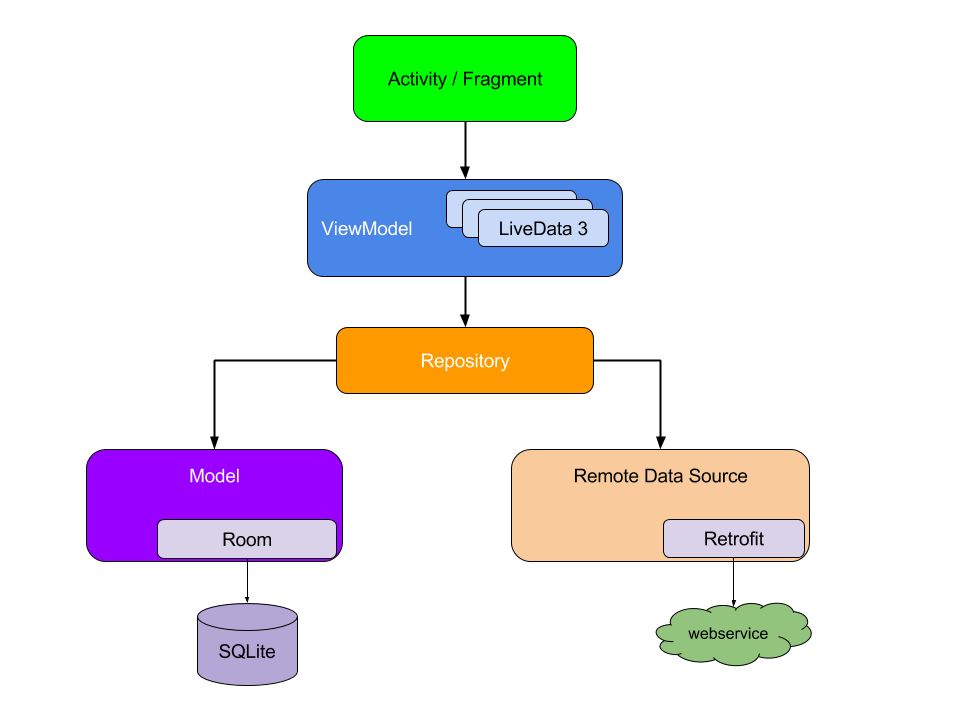
* Retrofit返回包括两部分，第一部分为Call或者Observable，Call类型是Retrofit默认支持的（Retrofit内部有一个DefaultCallAdapterFactory），故若想要支持Rxjava，就要添加如下代码：new Retrofit.Builder().addCallAdapterFactory(RxJavaCallAdapterFactory.create())

|  |
| --- |
| Retrofit retrofit = new Retrofit.Builder()  .addCallAdapterFactory(RxJavaCallAdapterFactory.create())  *//设置数据解析器*  .addConverterFactory(GsonConverterFactory.create())  *//设置网络请求的Url地址*  .baseUrl("http://apis.baidu.com/txapi/")  .build();  *// 创建网络请求接口的实例*  mApi = retrofit.create(APi.class); |

* 下载文件

|  |
| --- |
| public interface IServiceApi { *//PDF文件Retrofit下载*  @Streaming *// 注解大文件，大文件若无此注解，会出现OOM*  @GET  *// 返回的原数据，不需要数据解析*  Observable<ResponseBody> retrofitDownloadFile(@Url String fileUrl);  } |

## MVVM



1. 各层职能

* View层：继承自 LifecycleActivity/LifecycleFragment，是UI控件的宿主，核心职责为：由ViewModel驱动更新UI控件，监听UI事件及生命周期驱动ViewModel。
* ViewModel层：只做数据和业务逻辑操作，不持有任何UI控件的引用。（通过LiveData）
* Model层：数据层，由Repository整合各路来源的数据，再统一暴露给ViewModel层使用。

1. 层级间交互

## Android Architecure Components

https://www.jianshu.com/p/84a3442955ae

1. 一个帮助构建稳定、易于测试和易于维护的App架构的库。
2. 设计App架构的基本原则：

Soc原则（Separation of concerns分离关注点原则）：模块化，低耦合。即不要将和UI操作及调用系统组件无关的代码放在Activity或者Fragment中，应该将其作为App和系统交互的中间层。（即MVP）

Model驱动UI：最好采用持久化数据的Model，因为数据是持久化的，当系统因为各种原因回收资源时，并不会造成数据的丢失，且当网络不稳定或者无法连接的情况下，App仍然可以继续工作。

1. ViewModel是为特定的UI提供数据的类，同时它也承担和数据相关的业务逻辑处理功能，比如网络请求等，因为ViewModel独立于View，故其不会被View层的事件影响。
2. LiveData是一个包含可以被观察的数据载体。其是基于观察者模式去做的，当其的数据发生改变时，所有对这个LiveData变化敏感的类都会收到变化的更新。同时其还能感知组件的声明周期，防止内存泄露。【优于rxjava的特点：其能够在组件生命周期结束后自动阻断数据流的传播，防止产生空指针等异常】
3. 由于ViewModel是与Activity/Fragment绑定的，所以当Activity/Fragment正常退出时，相应的ViewModel就会释放掉相关的数据，当用户再次进入该页面就需要重新进行网络请求，这显然是不合理的。同时在进行网络请求前通常需要在本地数据库查询是否有相关信息，导致ViewModel过于臃肿，不符合Soc原则，因此将这部分功能抽象为一个新的Repository类进行代理，该类的作用便是获取各种来源的数据，并且在来源数据更新时通知数据获取方。

## 持久化数据Room

1. Room是一个对象映射库，可以在编译时检测出SQL语句的异常，还能够在数据库内容发生改变时通过LiveData的形式发出通知。
2. Room主要分为三个部分，分别是Database（数据库）、Entity（实体）、DAO（数据访问对象）

* Database（数据库）

数据库指的就是一个数据库对象，它继承于 RoomDataBase 这个类，并且需要用 @DataBase 注解，获取这个数据库对象的方法是通过调用 Room.databaseBuilder() 或者 Room.inMemoryDatabaseBuilder()，后者表示在内存中存储数据，如果程序结束了数据也就消失了，所以一般还是使用前者。

* Entity（实体）

a. 实体相当于数据库中的一个表。

b. 必须在Database类中的entities数组中引用entity类。entity中的每个属性都会被持久化到数据库，除非使用了@Ignore注解。

* DAO（数据访问对象）

查询组件，Dao数据访问对象是一个面向对象的数据库接口，其中定义了操作数据库的方法。

1. 使用：

* 建表

当一个类用@Database注解中的entitles属性引用时，Room会在数据库中为此创建一张表。eg. @Database(entities = {User.class}, version = 1)

* 表名

Room默认把类名作为数据库的表名，自定义表名需要使用@Entity注解的tableName属性。

* 列名

默认使用字段名作为列名，如果想指定列名，可以使用 @ColumnInfo(name = "your\_name")

* 持久化忽略

如果一个实体类中有不想持久化的字段，那么可以使用@Ignore来注释它们。

* 主键

每个实体类Entity都必须至少定义一个主键，主键自增要使用@PrimaryKey(autogenerate = true)

* 组合主键

@Entity(primaryKeys = {"firstName", "lastName"})

* 索引

为entity添加索引，需要在@Entity注解中加入indices属性。

* 唯一性

指定某个字段或者某几个字段的值是唯一的，可以通过把@Index注解的unique属性设置为true

* 对象嵌套

当一个实体嵌入另一个实体类时，若使用@Embedded注解，则表示外实体中将拥有内实体的所有字段。为防止多个实体嵌套造成字段重复，可以通过设置prefix属性来保持每一列的唯一性。

|  |
| --- |
| *// 创建Entity实体类*  @Entity(tableName = **"user"**, indices = {@Index(value = **"id"**, unique = **true**)}) **public class** User {  @PrimaryKey(autoGenerate = **true**)  **private int uid**;    @ColumnInfo(name = **"first\_name"**)  **private** String **firstName**;    @Ignore  Bitmap **picture**; } |
| *// 创建Dao*  @Dao **public interface** UseDao {  *// 当返回LiveData<User>时，Room数据库发生改变时可以通知对其敏感的类* @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 id = :userId"**)  LiveData<User> load(String userId); } |
| *// 创建Database*  @Database(entities = {User.**class**}, version = 1) **public abstract class** AppDatabase **extends** RoomDatabase {  **public abstract** UserDao userDao(); } |
| *// 获取database实例，获取实例时应该保持单例模式*  AppDatabase db = Room.databaseBuilder(getApplicationContext(),AppDatabase.class, "database-name").build(); |

## Fragment中传参

## Android中的事件分发

1. 在ViewGroup中，事件分为dispatchTouchEvent（事件的分发）、onInterceptTouchEvent（事件拦截）、事件处理（onTouchEvent）
2. 在View中，事件分为dispatchTouchEvent、onTouchEvent
3. onInterceptTouchEvent()是用于处理事件并改变事件的传递方向，也就是决定是否允许Touch事件继续向下传递，一旦返回true，表示事件将会在当前被处理，向下传递之路就会被截断，同时把事件传递给当前的onTouchEvent()方法进行处理；返回false则继续向子控件传递。
4. onTouchEvent()用于处理事件，返回值决定当前控件是否消费了这个事件，也就是说在当前控件处理完Touch事件后，是否还向父控件传递，若返回false，则上传给父控件。
5. ViewGroup的down事件首先会经过ViewGroup的onInterceptTouchEvent()方法，若该方法返回false，后续的move，up事件将继续先传递给ViewGroup，之后才和down事件一起传递给最终的目标View的onTouchView()处；若返回true，则后续的move，up将不会再传递给onInterceptTouchEvent()，而是和down事件一样传递给该ViewGroup的onTouchEvent()处理，最终View将不会接收到任何事件。
6. 如果最终需要处理事件的view的onTouchEvent()返回false，则该事件将被传递至其上一层次的view的onTouchEvent()；如果返回true，则后续事件将可以继续传递给该View的onTouchEvent()处理。

## Android中的DataSetObservable和DataSetObserver

1. Observable的直接继承类有两个：DataSetObservable和ContentObservable。
2. DataSetObservable使用DataSetObserver实例化了Observable。
3. DataSetObserver中包含了两个方法，当有数据集改变时，会调用其中的onChanged()方法；当数据集失效时，会调用其中的onInvalidated()方法。

|  |
| --- |
| **public abstract class** DataSetObserver {**public void** onChanged() {}**public void** onInvalidated() {} } |

## Drawable

1. Drawable是一个可绘制的对象，里面保存可绘制的数据，其可能是位图（BitmapDrawable）、可能是图形（ShapeDrawable）、可能是颜色（ColorDrawable）等。
2. Drawable对象的主要功能是通过draw()方法渲染canvas；也可为View类对象设置背景，如果Drawable类对象的属性修改了，Drawable类对象可以通知关联的View类对象重新绘制View，该功能通过invalidateSelf()方法实现，该方法会调用关联的Drawable.Callback类对象（即View）的invalidateDrawable()方法。
3. 每个Drawable对象都关联一个ConstantState对象，该对象用来保存一些恒定不变的数据，如果从同一个res中创建的Drawable对象，他们会共享一个ConstantState对象。故在修改Drawable的属性时，需要调用mutate()方法，让Drawable复制一个新的ConstantState对象关联。根据此特点，也可通过判断ConstantState判断是否是同一个drawable。

## 依赖注入（DI）和控制反转（Ioc）

Ioc（Inversion of Control）控制反转

1. Ioc是一种设计思路，将设计好的对象交给容器控制，而不是对象内部直接控制。

DI（Dependency Injection）依赖注入

## 枚举Enum

1. ordinal()方法返回枚举常量的序数。

## Butterknife

1. 默认情况下，@bind和@OnClick绑定都必须有一个目标View，当找不到这个View时会抛出异常，为了防止这种异常情况的发生，可以在绑定的字段前面使用@Nullable注解，在绑定的方法前面使用@Option注解。

## Kotlin

## CountDownLatch

1. CountDownLatch是一个多线程控制工具。
2. 常用方法：

CountDownLatch(int count) 实例化一个倒计时器，count指定计数个数

countDown() 计数减一

await() 等待，当计数减到0时，所有线程并行执行

## 杂项学习

1. **UI的深度理解**

- 自定义控件

- 系统组件 源码分析

- 屏幕适配

- 绘制流程

- 事件传递

- 属性动画

- 复杂绘制

1. **底层开发**

- NDK/JNI

- linux

- shell

- 热修复

- FFmpeg音视频编解码

- openCV 图像处理

- openGL 图形处理

- Nginx 流媒体

- 智能家居设备串口通信

1. **Framework**

- 系统源码

- UML建模

- 插件化

- 组件化

- 数据库框架

- 网络访问框架

- 图片加载框架

- 响应式编程

- MVC、MVP、MVVM

1. **优化问题**

- 内存泄露

- 内存消耗管理

- 过度绘制

- apk打包流程，大小压缩，启动优化，耗电管理，app加固，防反编译

1. **新技术**

- jetpack

- kotlin

- flutter

- android系统版本差异