Android前端代码规范

本规范结合阿里巴巴Android开发手册、Kotlin代码规范、Jetpack官方文档编写

- Android前端代码规范
- 一、项目架构
  - MVVM
  - LiveData
  - Jetpack Compose
  - Navigation-Compose导航库
- 一、目录结构
  - base
  - compose
  - db
  - entity
  - global
  - navigation
  - network
  - theme
  - ui
  - util
- 二、Kotlin 语言规范
  - 2.1 命名规则
    - 类、对象
    - Compose
    - 函数名
    - 属性名
    - 幕后属性的名称
  - 2.2缩进
  - 2.3留白
  - 2.4Unit
  - 2.5Lambda表达式
- 三、Android 资源文件命名与使用
  - drawable资源
  - color资源
- 四、Android 基本通信规则
- 五、进程、线程与消息通信
- 六、文件与数据库

一、项目架构

MVVM

> MVVM(Model-View-ViewModel)是一种代码架构模式，被广泛应用在Android程序设计领域。Mobius项目Android端使用Jetpack组件进行MVVM架构

M: 仓库层，仓库层要做的工作是自主判断接口请求的数据应该是从数据库中读取还是从网络中获取，并将数据返回给调用方。

V：UI控制层，Activity、Fragment、Compose组件。

VM：ViewModel层，ViewModel用于持有和UI元素相关的数据，以保证这些数据在屏幕旋转时不会丢失，以及负责和仓库之间进行通讯。

mvvm架构，对应关系是

- 一个Activity/Fragment持有一个或多个ViewModel
- 一个ViewModel持有一个或多个UseCase
- 一个UseCase持有一个或多个Repository

UseCase的作用是

- 切线程(例如把网络请求切换到非UI线程)
- Try-Catch处理(例如处理网络请求异常)
- 业务逻辑处理(业务逻辑放在UseCase里面可以实现复用，放在ViewModel里面不好复用)

LiveData

UseCase使用LiveData更新数据

> LiveData是一种可观察的数据存储器类。与常规的可观察类不同，LiveData具有生命周期感知能力，意指它遵循其他应用组件（如Activity，Fragment或Service）的生命周期

LiveData

- 确保界面符合数据状态：LiveData遵循观察者模式。当生命周期状态发生变化时，LiveData会通知Observer对象。
- 不会发生内存泄漏：观察者会绑定到Lifecycle对象，并在其关联的生命周期遭到销毁后进行自我清理。
- 不会因Activity停止而导致崩溃
- 不再需要手动处理生命周期
- 数据始终保持最新状态

- 适当的配置更改。
- 共享资源。

Jetpack Compose

> Jetpack Compose 是Google在2019年发布的一个Android原生现代UI工具包，它完全采用Kotlin编写，可以使用Kotlin语言的全部特性.

Navigation-Compose导航库

项目使用单Acitivity承载Compose，切换页面基于view级别

> Compose中每一个@Composable注解标注的方法就可以成为一个视图，所以导航就是用来处理这些视图之间的跳转操作。 Navigation在设计上高度抽象，只负责导航逻辑不关心页面的具体实现，无论是Activity、Fragment甚至是一个已定义View都可以基于Navigation实现导航。

一、目录结构

项目的目录结构如下

base

基类目录

compose

存放用于复用Compose组件

db

持久层，数据库等相关的组件

entity

存放实体类，Kotlin的data类

global

存放全局静态常量

navigation

存放navigation导航库相关内容

network

存放网络请求服务相关的内容

theme

MaterialDesign主题相关内容，color、shape、theme、type

ui

View+ViewModel层，存放时遵循Screen+ViewModel

util

存放单例工具类

```
java
   |---edu
       |---fzu
           |---mobius
               |--- base
               |   |--- BaseAcitvity.kt
               |   |--- xxx.kt
               |
               |--- compose
               |   |--- xxxCompose.kt
               |   |--- xxx.kt
               |
               |--- db
               |   |--- DbManager.kt
               |   |--- xxx.kt
               |
               |--- entity
               |   |--- User.kt
               |   |--- Letter.kt
               |   |--- xxx.kt
               |
               |--- global
               |   |--- xxx.kt
               |
               |--- navigation
               |   |--- Navigation.kt
               |   |--- xxx.kt
               |
               |--- network
               |   |--- HttpClient.kt
               |   |--- xxx.kt
               |
               |--- theme
               |   |--- Color.kt
               |   |--- Shape.kt
               |   |--- Theme.kt
               |   |--- Type.kt
               |   |--- xxx.kt
               |
               |--- ui
               |   |--- capsule
               |   |   |--- CapsuleScreen.kt
               |   |   |--- CapsuleViewModel.kt
               |   |   |--- xxx.kt
               |   |
               |   |--- mailbox
               |   |   |--- MailBoxScreen.kt
               |   |   |--- MailBoxViewModel.kt
               |   |   |--- xxx.kt
               |   |
```

```
|   |--- mine
|   |   |--- MineScreen.kt
|   |   |--- MineViewModel.kt
|   |   |--- xxx.kt
|   |
|   |--- penpal
|   |   |--- PenPalScreen.kt
|   |   |--- PenPalViewModel.kt
|   |   |--- xxx.kt
|
|--- usecase
|   |--- xxxUseCase.kt
|   |--- xxx.kt
|
|--- util
    |--- xxxUtil.kt
    |--- xxx.kt
```

二、Kotlin 语言规范

2.1 命名规则

在 Kotlin 中，包名与类名的命名规则非常简单：
包的名称总是小写且不使用下划线（org.example.project）。 通常不鼓励使用多个词的名称，但是如果确实需要使用多个词，可以将它们连接在一起或使用驼峰风格（org.example.myProject）。

类、对象

类与对象名称以大写字母开头并使用驼峰风格：

```
open class DeclarationProcessor { /*……*/ }

object EmptyDeclarationProcessor : DeclarationProcessor() { /*……*/ }
```

Compose

为了区分Compose与函数，Compose组件使用大写字母开头的驼峰

```
@Compose
fun TextCompose(){
    Text(
        text = "xxx"
    )
}
```

函数名

函数、属性与局部变量的名称以小写字母开头、使用驼峰风格而不使用下划线：

```
fun processDeclarations() { /*……*/ }
var declarationCount = 1
```

例外：用于创建类实例的工厂函数可以与抽象返回类型具有相同的名称：

```
interface Foo { /*……*/ }

class FooImpl : Foo { /*……*/ }

fun Foo(): Foo { return FooImpl() }
```

属性名

常量名称（标有 `const` 的属性，或者保存不可变数据的没有自定义 `get` 函数的顶层/对象 `val` 属性）应该使用大写、下划线分隔的名称:

```
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

保存带有行为的对象或者可变数据的顶层/对象属性的名称应该使用驼峰风格名称：

```
val mutableCollection: MutableSet<String> = HashSet()
```

保存单例对象引用的属性的名称可以使用与 object 声明相同的命名风格：

```
val PersonComparator: Comparator<Person> = /*...*/
```

对于枚举常量，可以使用大写、下划线分隔的名称（`enum class Color { RED, GREEN }`）也可使用首字母大写的常规驼峰名称，具体取决于用途。

幕后属性的名称

如果一个类有两个概念上相同的属性，一个是公共 API 的一部分，另一个是实现细节，那么使用下划线作为私有属性名称的前缀(这种方式借鉴了Flutter私有属性的命名方式)：

```
class C {
    private val _elementList = mutableListOf<Element>()

    val elementList: List<Element>
         get() = _elementList
}
```

2.2缩进

使用 4 个空格缩进。 Tab设置为四空格

2.3留白

- 在二元操作符左右留空格（a + b）。例外：不要在“range to”操作符（0..i）左右留空格。
- 不要在一元运算符左右留空格（a++）
- 在控制流关键字（if、 when、 for 以及 while）与相应的左括号之间留空格。
- 在 // 之后留一个空格：
- 不要在用于指定类型参数的尖括号前后留空格：class Map<K, V> { …… }
- 不要在 :: 前后留空格：Foo::class、 String::length
- 不要在用于标记可空类型的 ? 前留空格：String?

2.4Unit

如果函数返回 Unit 类型，该返回类型应该省略：

```
fun foo() { // 省略了 ": Unit"

}
```

2.5Lambda表达式

在lambda表达式中, 大括号左右要加空格，分隔参数与代码体的箭头左右也要加空格 。lambda表达应尽可能不要写在圆括号中

list.filter { it > 10 }.map { element -> element * 2 }
在非嵌套的短lambda表达式中，最好使用约定俗成的默认参数 it 来替代显式声明参数名 。在嵌套的有参数的lambda表达式中，参数应该总是显式声明。

在多行的 lambda 表达式中声明参数名时，将参数名放在第一行，后跟箭头与换行符：

```
appendCommaSeparated(properties) { prop ->
    val propertyValue = prop.get(obj)  // ……
}
```

如果参数列表太长而无法放在一行上，请将箭头放在单独一行：

```
foo {
    context: Context,
    environment: Env
    ->
    context.configureEnv(environment)
}
```

三、Android 资源文件命名与使用

drawable资源

drawable 资源名称以小写单词+下划线的方式命名，根据分辨率不同存放在不同的 drawable 目录下，建议只使用一套,例如 drawable-xhdpi

> 业务功能描述_控件描述_控件状态限定词
> eg: login_btn_pressed,tabs_icon_home_normal

大分辨率图片（单维度超过 1000）大分辨率图片建议统一放在 xxhdpi 目录下管理，否则将导致占用内存成倍数增加。

> eg: 将 144*144 的应用图标 PNG 文件放在 drawable-xxhdpi 目录

color资源

color 资源使用#AARRGGBB 格式，存放在Color.kt，命名采用如下规则：

> 主题名+逻辑名称+颜色
> eg: MobiusThemeButtonBackgroundColor

四、Android 基本通信规则

(此部分和之后的部分缺少jetpack的相关规范，使用阿里巴巴的java规范作为参考)

- 不要向ViewModel中传入Activity，如有使用需要，使用Application中的Activity
- 隐式 Intent 的跳转，在发出 Intent 之前必须通过 resolveActivity 检查，避免找不到合适的调用组件，造成 ActivityNotFoundException 的异常。

eg:

```
public void viewUrl(String url, String mimeType) {
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setDataAndType(Uri.parse(url), mimeType);
    if (getPackageManager().resolveActivity(intent, PackageManager.MATCH_DEFAULT_ONLY) != null)
        try {
            startActivity(intent);
        } catch (ActivityNotFoundException e) {
            if (Config.LOGD) {
                Log.d(LOGTAG, "activity not found for " + mimeType + " over " +Uri.parse(url). g
            }
        }
    }
}
```

- 避免在 Service#onStartCommand()/onBind()方法中执行耗时操作，如果确实有需求，应改用 IntentService 或采用其他异步机制完成.

> 由于该方法是在主线程执行，如果执行耗时操作会导致 UI 不流畅。可以使用IntentService 、 创 建 HandlerThread 或者调用 Context#registerReceiver(BroadcastReceiver, IntentFilter, String, Handler)方法等方式，在其他 Wroker 线程执行 onReceive 方法。BroadcastReceiver#onReceive()方法耗时超过 10 秒钟，可能会被系统杀死。

eg:

```
IntentFilter filter = new IntentFilter();
filter.addAction(LOGIN_SUCCESS);
this.registerReceiver(mBroadcastReceiver, filter);
mBroadcastReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Intent userHomeIntent = new Intent();
        userHomeIntent.setClass(this, UseHomeActivity.class);
        this.startActivity(userHomeIntent);
    }
};
```

- 避免使用隐式 Intent 广播敏感信息，信息可能被其他注册了对应BroadcastReceiver 的 App 接收

> 通过 Context#sendBroadcast()发送的隐式广播会被所有感兴趣的 receiver 接收，恶意应用注册监听该广播的 receiver 可能会获取到 Intent 中传递的敏感信息，并进行其他危险操作。如果发送的广播为使用 Context#sendOrderedBroadcast()方法发送的有序广播，优先级较高的恶意 receiver 可能直接丢弃该广播，造成服务不可用，或者向广播结果塞入恶意数据。如果广播仅限于应用内，则可以使用 LocalBroadcastManager#sendBroadcast()实现，避免敏感信息外泄和 Intent 拦截的风险

- 不要在 Activity#onDestroy()内执行释放资源的工作，例如一些工作线程的销毁和停止，因为onDestroy()执行的时机可能较晚。可根据实际需要，在Activity#onPause()/onStop()中结合isFinishing()的判断来执行
- 总是使用显式 Intent 启动或者绑定 Service，且不要为服务声明 Intent Filter，保证应用的安全性。如果确实需要使用隐式调用，则可为 Service 提供 Intent Filter并从 Intent 中排除相应的组件名称，但必须搭配使用 Intent#setPackage()方法设置Intent 的指定包名，这样可以充分消除目标服务的不确定性
- 不要在 Android 的 Application 对象中缓存数据。基础组件之间的数据共享请使用 Intent 等机制，也可使用 SharedPreferences 等数据持久化机制。
- 使用 Toast 时，建议定义一个全局的 Toast 对象，这样可以避免连续显示Toast 时不能取消上一次 Toast 消息的情况(如果你有连续弹出 Toast 的情况，避免使用 Toast.makeText)

五、进程、线程与消息通信

(此部分和之后的部分缺少jetpack的相关规范，使用阿里巴巴的java规范作为参考)

- 在 Application 的业务初始化代码加入进程判断，确保只在自己需要的进程初始化。特别是后台进程减少不必要的业务初始化。
- 新建线程时，必须通过线程池提供（AsyncTask 或者 ThreadPoolExecutor或者其他形式自定义的线程池），不允许在应用中自行显式创建线程。
- 子线程中不能更新界面，更新界面必须在主线程中进行，网络操作不能在主线程中调用。
- 禁止在多进程之间用SharedPreferences共享数据
- 谨慎使用 Android 的多进程，多进程虽然能够降低主进程的内存压力，但会遇到如下问题
    - 1. 不能实现完全退出所有 Activity 的功能；
    - 2. 首次进入新启动进程的页面时会有延时的现象（有可能黑屏、白屏几秒，是白屏还是黑屏和新 Activity 的主题有关）；
    - 3. 应用内多进程时，Application 实例化多次，需要考虑各个模块是否都需要在所有进程中初始化
    - 4. 多进程间通过 SharedPreferences 共享数据时不稳定。

六、文件与数据库

(此部分和之后的部分缺少jetpack的相关规范，使用阿里巴巴的java规范作为参考)

- 任何时候不要硬编码文件路径，请使用 Android 文件系统 API 访问。

> Android 应用提供内部和外部存储，分别用于存放应用自身数据以及应用产生的用户数据。可以通过相关 API 接口获取对应的目录，进行文件操作

```
android.os.Environment.getExternalStorageDirectory()

android.os.Environment.getExternalStoragePublicDirectory()

android.content.Context.getFilesDir()

android.content.Context.getCacheDir
```

> 正例：

```
public File getDir(String alName) {
    File file = new File(Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PIC
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
        }
    return file;
}
```

反例：

```
public File getDir(String alName) {
 // 任何时候都不要硬编码文件路径，这不仅存在安全隐患，也让 app 更容易出现适配问题
    File file = new File("/mnt/sdcard/Download/Album", alName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
```

- 当使用外部存储时，必须检查外部存储的可用性

```
// 读/写检查
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
// 只读检查
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||Environment.MEDIA_MOUNTED_READ_ONLY.equals(sta
        return true;
    }
    return false;
}
```

- 应用间共享文件时，不要通过放宽文件系统权限的方式去实现，而应使用 FileProvider。
- SharedPreference 中只能存储简单数据类型（int、boolean、String 等），复杂数据类型建议使用文件、数据库等其他方式存储。
- 数据库 Cursor 必须确保使用完后关闭，以免内存泄漏。

> 说明：
> Cursor 是对数据库查询结果集管理的一个类，当查询的结果集较小时，消耗内存不易察觉。但是当结果集较大，长时间重复操作会导致内存消耗过大，需要开发者在操作完成后手动关闭 Cursor。
> 数据库 Cursor 在创建及使用时，可能发生各种异常，无论程序是否正常结束，必须在最后确保 Cursor 正确关闭，以避免内存泄漏。同时，如果 Cursor 的使用还牵涉多线程场景，那么需要自行保证操作同步。

- 多线程操作写入数据库时，需要使用事务，以免出现同步问题。

> Android 的通过 SQLiteOpenHelper 获取数据库 SQLiteDatabase 实例，Helper 中会自动缓存已经打开的 SQLiteDatabase 实例，单个 App 中应使用 SQLiteOpenHelper的单例模式确保数据库连接唯一。由于 SQLite 自身是数据库级锁，单个数据库操作是保证线程安全的（不能同时写入），transaction 时一次原子操作，因此处于事务中的操作是线程安全的。
> 若同时打开多个数据库连接，并通过多线程写入数据库，会导致数据库异常，提示数据库已被锁住。

- 大数据写入数据库时，请使用事务或其他能够提高 I/O 效率的机制，保证执行速度。