# MyChat项目报告

姓名：俞若鹏

学号：U202115612

学院：华中科技大学计算机科学与技术学院

github url：https://github.com/921757623/MyChat

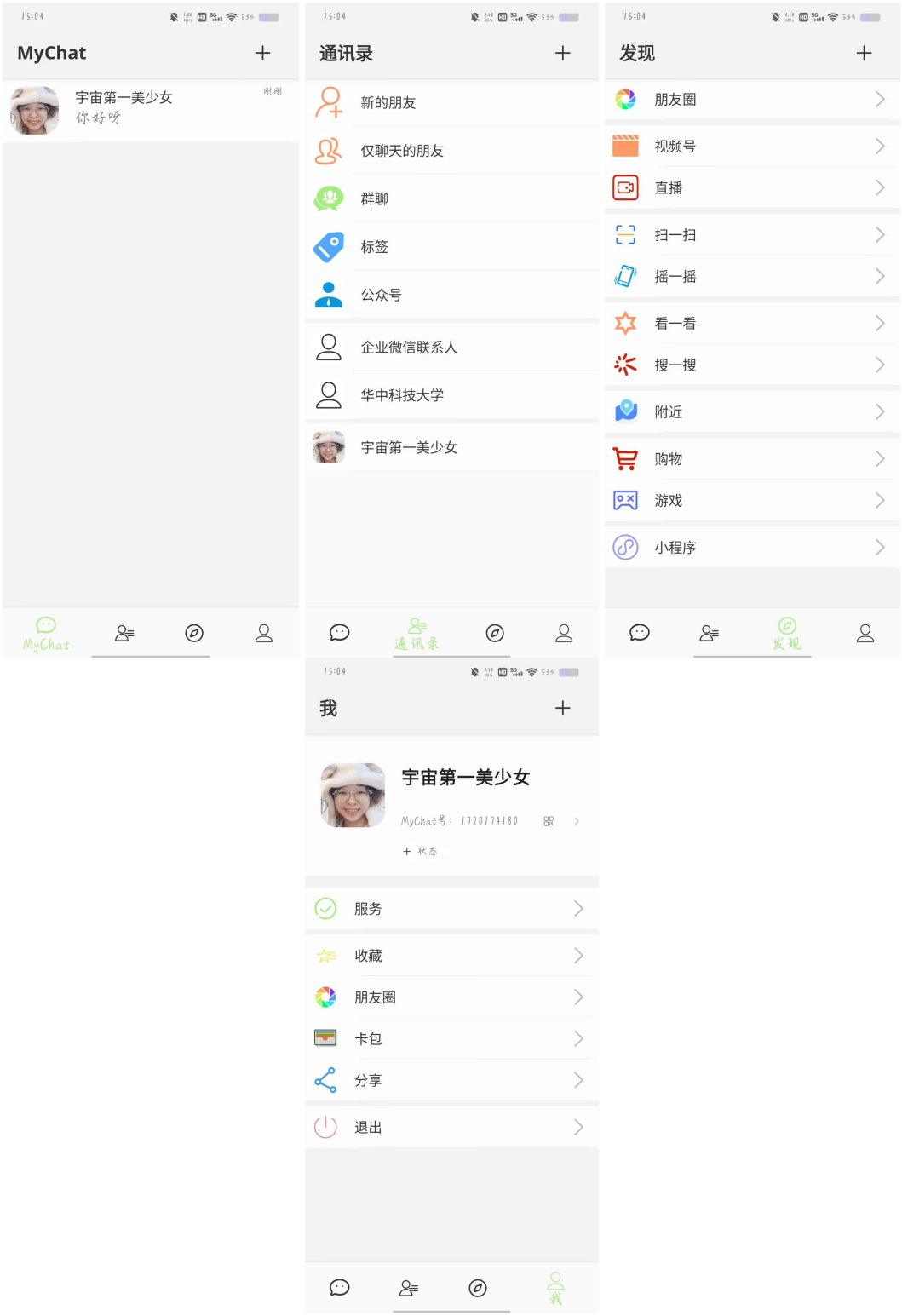
---

## 一、产品功能介绍

MyChat是一款仿微信的简易本地聊天软件，实现了账号注册登录、联系人添加、聊天、分享应用卡片等功能，但碍于个人能力，本产品没有实现远程后端，所有功能均基于本地持久化数据存储操作。

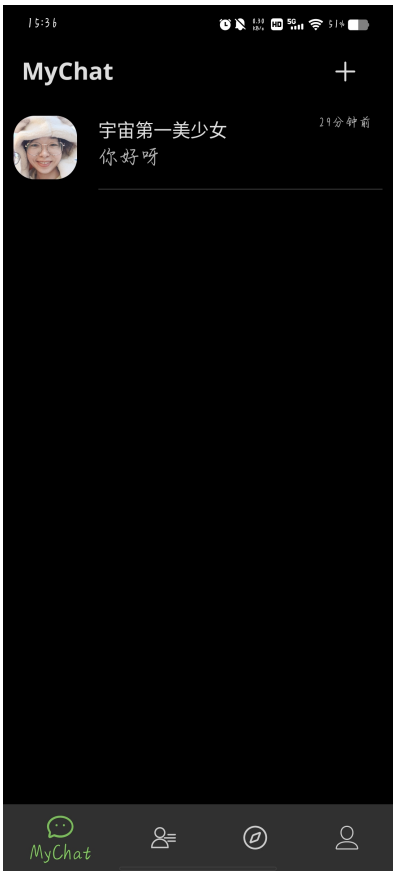
以下是产品界面展示：


下面是展示应用内的四个主要界面：




应用整体模仿腾讯微信的应用风格，同时也适配了夜间模式：


下面是展示应用内的四个主要界面的夜间模式：

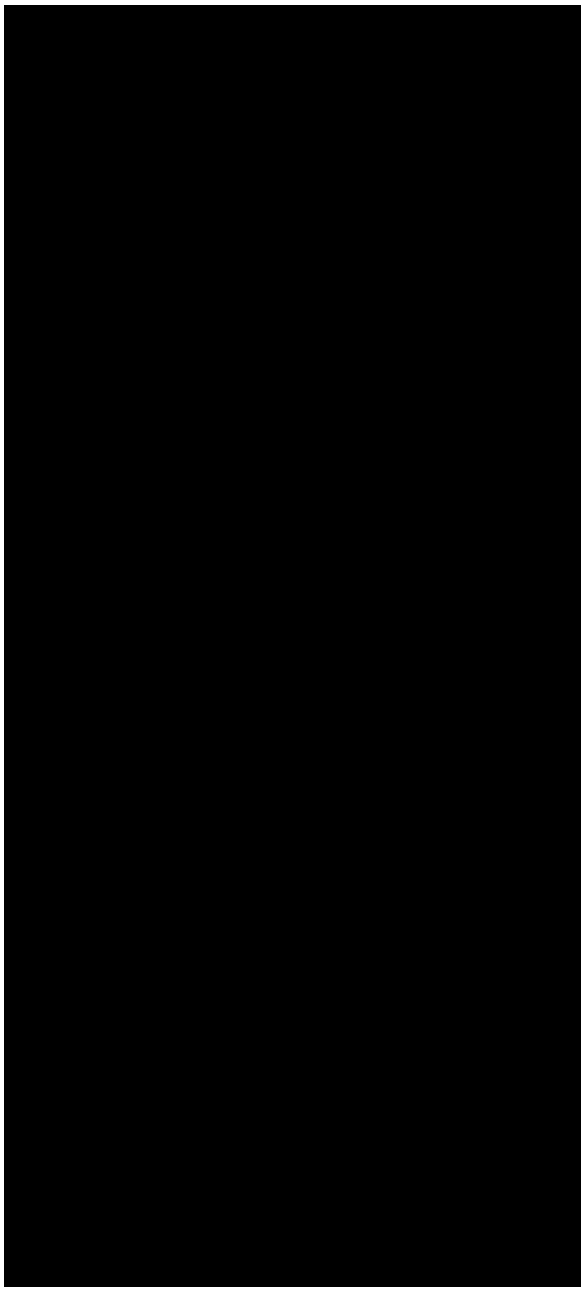

MyChat
宇宙第一美少女
你好呀
29分钟前
MyChat


通讯录
新的朋友
仅聊天的朋友
群聊
标签
公众号
企业微信联系人
华中科技大学
宇宙第一美少女
通讯录


发现
朋友圈
视频号
直播
扫一扫
摇一摇
看一看
搜一搜
附近
购物
游戏
小程序
发现


我
宇宙第一美少女
MyChat号：1720174180
+ 状态
服务
收藏
朋友圈
卡包
分享
退出
我

本程序的演示视频如下：



## 二、程序概要设计

MyChat完全基于Kotlin语言开发，采用Gradle-8.0开源工具自动化构建项目，IDE采用Android Studio Flamingo版本，

项目的minSDK为API 28，targetSDK为API 33。

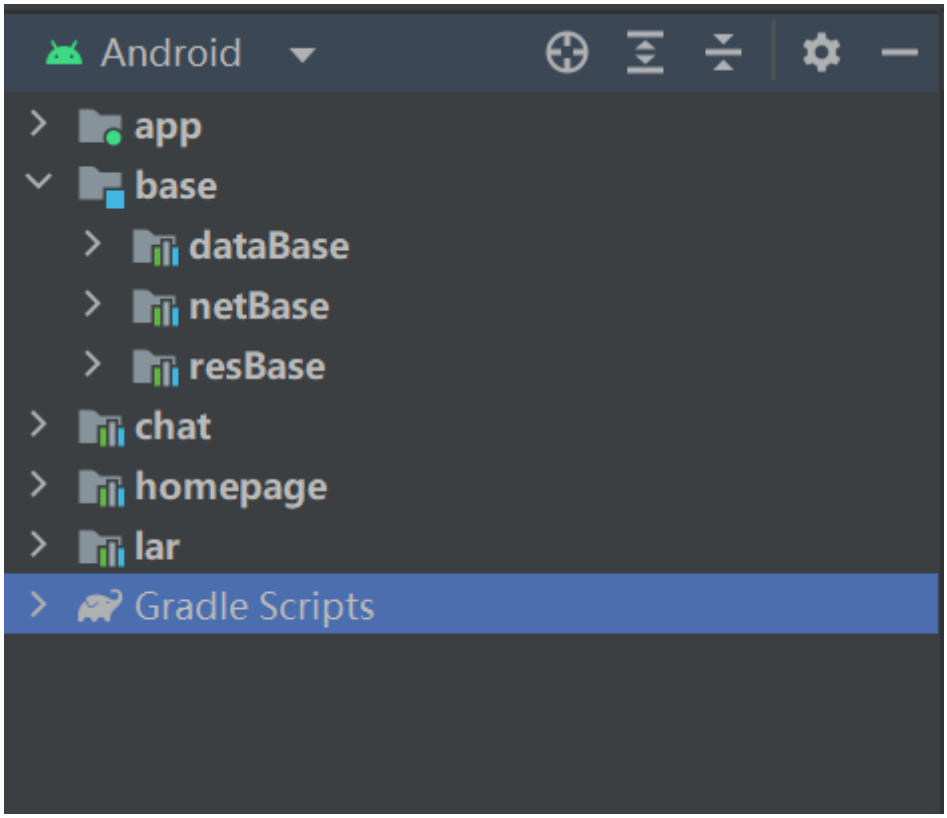
测试环境如下：

| 手机型号 | 安卓版本 | 系统 |
| --- | --- | --- |
| iQOO Neo6 SE | 13 | Vivo Origin OS 3 |
| MatePad 11 | 12 | Harmony OS 4 |

| 手机型号 | 安卓版本 | 系统 |
| --- | --- | --- |
| Pixel 2（虚拟机） | 9 | 不知道 |

## 2.1 模块设计

项目总体采用模块化设计的方式，将各个功能模块抽象成物理上不同的安卓library，各个功能模块间处于互不依赖的状态，项目的模块如下图所示：




包app是程序的主模块，也是程序入口StartActivity存在的地方的，StartActivity负责展示启动页，同时处理跳转逻辑，通过判断是否有登录信息而选择跳转到lar模块或者homepage模块。

包base是程序最底层的模块，主要职能是存放上层模块所需的各种基础代码，有依据各个类的功能主要分为了三个library，分别为存放数据库基础类、Dao类、Table类等有关数据存储功能代码的dataBase包、负责网络基础框架等功能代码的netBase包以及存放各类drawable、layout等资源文件的resBase。

包lar是程序负责登录部分的模块，包含注册和登录等功能的相关实现代码，并通过base包将用户信息持久化存储在本地当中。

包homepage是本程序的核心功能模块，是负责展示程序四个主界面并处理相应逻辑功能的模块，包括展示聊天列表、通讯录联系人、发现、个人资料等相关功能。

包chat是负责用户跟用户交流的模块，即实现了两个用户进行对话所需各类功能的模块，并且通过数据库持久化存储用户间交流的信息。

在各个模块的依赖关系上，app是作为顶层模块凌驾所有模块之上的，base是作为底层模块可以被各个模块所依赖并使用的，lar、homepage、chat三个模块作为程序的中间模块，彼此之间互不依赖，均能依赖下层的base模块，同时也会被上层的app模块依赖。

## 2.2 数据模型设计

针对本应用所需的功能，数据模型主要依靠数据库的表结构以及使用kotlin的data class实现，相关数据定义存放在dataBase的**Table包**和netBase的**Response.kt**文件当中。

针对用户信息的存储，设计了user表，对用data class **User**，其中包含如下成员变量：

| 变量名 | 数据类型 | 其他 |
| --- | --- | --- |
| id | Int | 存储用户id，由用户代码随机生成 |
| userName | String | 存储用户账号 |
| password | String | 密码 |
| createAt | Long | 用户创建的时间戳 |
| nickname | String | 昵称 |
| profilePicPath | String | 头像的Uri**字符串**信息 |

由于需要展示某一个用户的联系人列表，所以也需要设计用户关系表来存储用户关系，对应data class **UserToUser**，其中包含如下成员变量：

| 变量名 | 数据类型 | 其他 |
| --- | --- | --- |
| id（autoGenerate） | Int | 唯一标识用户对关系 |
| selfId | Int | 当前用户Id |
| friendNickname | String | 当前用户的朋友的昵称 |
| friendProfilePicPath | String | 对应朋友头像的Uri**字符串**信息 |
| chatId | String | 二人间唯一的聊天Id |

因为仍需要实现朋友间的对话，并将对话持久化存储，因此设计聊天记录表，对应data class **ChatRecord**，其中包含如下成员变量：

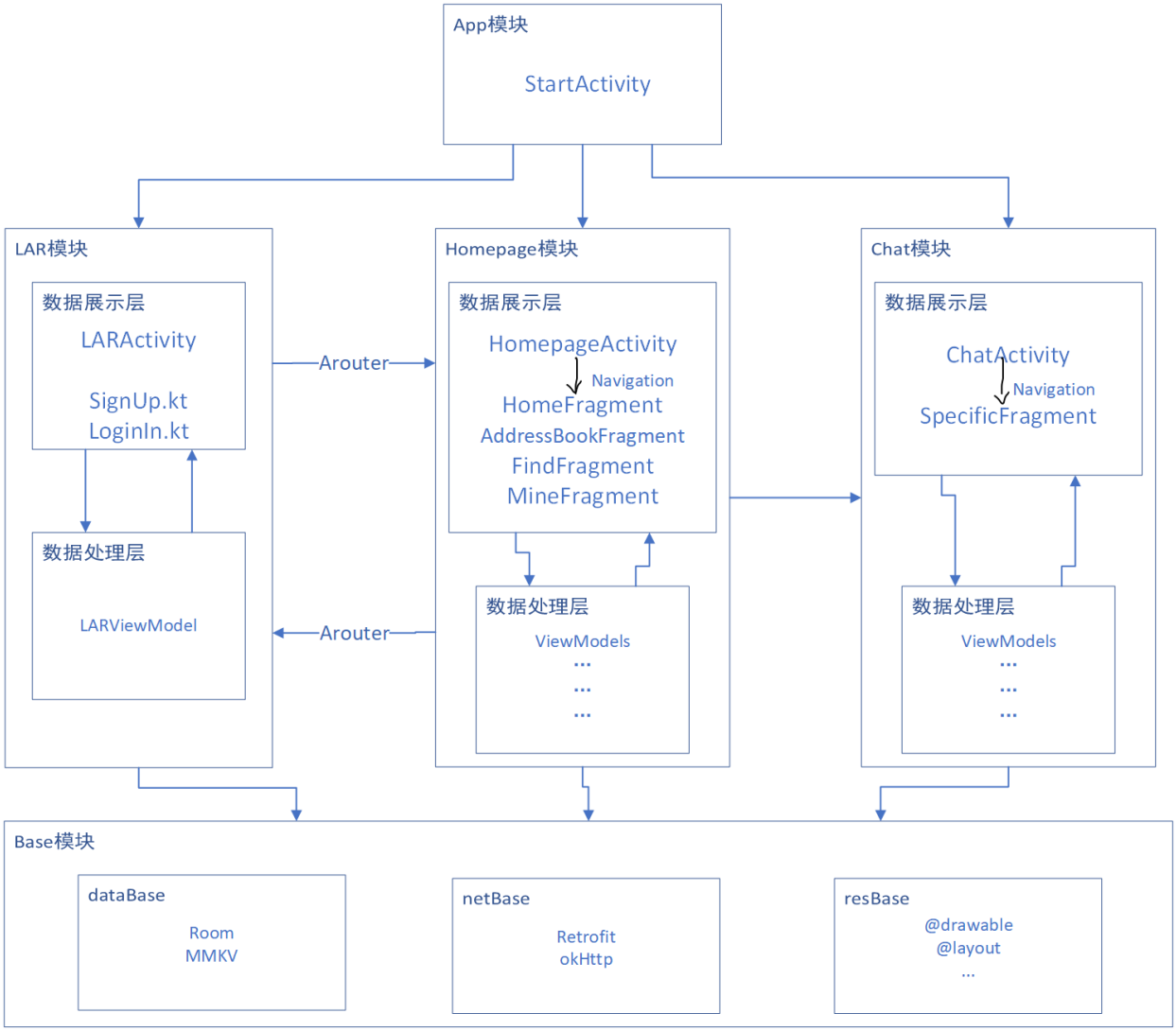
| 变量名 | 数据类型 | 其他 |
| --- | --- | --- |
| msgSeq（atuoGenerate） | Int | 唯一标识一条信息，并可用于确定信息顺序 |
| ownerId | Int | 谁发出的这条消息 |
| content | String? | 消息内容 |
| createAt | Long | 发送时的时间戳 |

针对主页展示聊天列表时需要同时展示朋友的相关个人信息，以及彼此之间的最新一条聊天记录，所以设计如下数据模式**ChatUnit**用于存放相关数据：

| 变量名 | 数据类型 | 其他 |
| --- | --- | --- |
| use_id | Int | 朋友的Id |
| profilePicPath | String | 朋友的头像Uri字符串信息 |
| nickname | String | 朋友的昵称 |
| message | ChatRecord | 最新的一条聊天记录 |

存在的一些问题：一开始在设计数据模型时，考虑到要用户之间个人数据的相对隔离，所以考虑包user_id作为保护对象不暴露给其他用户，所以在UserToUser表中是通过userName查询朋友关系的，但是由于缺乏相关数据库设计经验，所以最后也不知道到底该如何保护用户个人数据的相对安全（直白来说就是为了避免被盗号），所以后续设计的数据模型中例如ChatUnit又把user_id暴露了出来，这里有些前后矛盾。

# 三、程序架构图




# 四、技术亮点

1. 采用模块化设计，各模块之间互相独立，通过Arouter在主模块生成不同类之间的映射表，从而实现不同模块间的跳转

2. LAR模块尝试采用Compose编写UI以及处理相应的逻辑
3. 采用Activity+Fragment模式，通过Navigation实现一个Activity管理多个Fragment，每一个模块有且仅有一个Activity，模块间的跳转也是不同Activity的跳转。
4. 单个模块内，采用MVVM架构，并且使用databinding完成数据、逻辑、视图之间的解耦。
5. 在Homepage各种列表的展示中，使用RecycleView提升性能，并且数据的获取使用了Kotlin的Flow以及协程等相关技术，VIewModel内的LiveData替换为Kotlin自带的StateFlow，同时创建repository专门负责创建相应的挂起函数，以HomeFragment的getLocalChatList方法为例：

   首先在viewModel内声明相应的StateFlow用于存储数据：

```
private val _chatList = MutableStateFlow<List<ChatUnit>>(listOf())
val chatList = _chatList.asStateFlow()
```

   随后在repository内定义如下挂起函数用于从数据库中取出相应聊天列表：

```
fun getLocalChatList(sortMode: String): Flow<ApiResult> = flow {
    emit(ApiResult.Loading())
    val friends : List<UserToUser>? = appRoomDataBase.userToUserDao().queryFriends(BaseApplica
    val chatList : MutableList<ChatUnit> = mutableListOf<ChatUnit>()
    friends?.let {
        it.forEach { friend ->
            val oneRecord : ChatRecord = appRoomDataBase.chatRecordDao().queryOneRecord(frie
            chatList.add(
                ChatUnit(
                    BaseApplication.currentUseId,
                    friend.friendProfilePicPath,
                    friend.friendNickname,
                    oneRecord
                )
            )
        }
    }
    emit(ApiResult.Success(data = chatList))
}.flowOn(dispatcher).onEach {
    offset = 0
}.catch {
    it.printStackTrace()
    emit(ApiResult.Error(code = -1, errorMessage = it.message))
}
```

   ApiResult起初是用于网络请求时表示不同请求状态的类，在本地请求数据时也可以适用，相关定义如下：

```
sealed class ApiResult {
    yuruop
    data class Success<T>(
        val status: ApiStatus = ApiStatus.SUCCESSFUL,
        var data: T? = null
    ) : ApiResult()

    yuruop
    data class Error(
        val status: ApiStatus = ApiStatus.ERROR,
        val code: Int,
        val errorMessage: String? = null
    ) : ApiResult()

    yuruop
    data class Loading(
        val status: ApiStatus = ApiStatus.LOADING
    ) : ApiResult()
}
```

请求到数据后，在viewModel中就可以根据不同的ApiResult选择不同的应对策略：

```
fun getLocalChatList(sortMode: String = _sortMode.value!!) {
    viewModelScope.launch {
        _loadingState.emit(ApiStatus.LOADING)
        repository.getLocalChatList(sortMode).collect {
            when (it) {
                is ApiResult.Success<*> -> {
                    _loadingState.emit(ApiStatus.SUCCESSFUL)
                    _sortMode.value = sortMode
                    _chatList.emit(it.data as List<ChatUnit>)
                    if (_chatList.value.isEmpty()) {
                        _showingPlaceholder.value = PlaceholderType.PLACEHOLDER_NO_CONTEN
                    }
                }

                is ApiResult.Error -> {
                    _loadingState.emit(ApiStatus.ERROR)
                    tip.value = it.code.toString() + it.errorMessage
                    _showingPlaceholder.value = PlaceholderType.PLACEHOLDER_NETWORK_ERROR
                }

                else -> {}
            }
        }
    }
}
```

随后在对应的Fragment当中就可以直接通过监听流数据的变化来反馈到视图：

```
lifecycleScope.launch {
    viewModel.chatList.onEach {
        finishRefreshAnim()
    }.collectLatest {
        adapter.submitList(it.reversed())
        if (it.isEmpty()) {
            binding.rvChatlist.visibility = View.GONE
            binding.minePlaceholder.visibility = View.VISIBLE
        } else {
            binding.rvChatlist.visibility = View.VISIBLE
            binding.minePlaceholder.visibility = View.GONE
        }
    }
}
```

StateFlow相较于LiveData更好的运用了Kotlin语言的特性，也更好地适用于数据驱动视图的变化模式。

6. 设定了Constant类统一存储管理各类常量。
7. 适配了夜间模式。
8. 通过创建各种drawable资源个性化改变view控件的形状，按下松开的颜色反馈等……

9. 虽然没有后端用于进行网络请求，但是网络请求相关的基础类全都实现了，引入Retrofit和OkHttp技术实现相关网络请求，并且在**HomeFragmentViewModel**中也定义了相关实际的网络请求方法，可以参阅相关类。
10. 引入Glide用于头像图片的加载。
11. Compose的头像拾取功能虽然是参考网上某位大佬的写法，但是收益颇多，具体参考PhotoComponent.kt文件，当初自己实现该功能的难点在于不知道如何将viewModel中的非Composable函数的结果返回到Composable的函数当中，大佬的写法提供了一种全新的思路，直接定义Composable函数并添加在组件当中，并且也是利用kotlin流数据的变化将非Composable函数的变化传递到Composable函数当中。
12. 引入Zxing库用于生成应用分享的二维码，具体移步我-分享体验，主要涉及xml文件转化为bitmap的实现，以及将bitmap文件存储到本地相册的操作，学习了文件存储以及视图等相关的操作。
13. PopUpWindow添加了进出进入的相关动画，并且增加暗背景。
14. 登录模块以及聊天模块，点击文本框外可以收起键盘，可通过重写dispatchTouchEvent实现。
15. 使用MMKV存储一些基本的配置信息，比如是否登录过，个人的一些信息等等。
16. 本地数据的持久化存储使用了Room数据库，第一次用，挺好用的。
17. 使用Android Studio自带的Profiler工具检查了无内存泄漏。
18. 同时也将从相册选出的图片复制存储到应用的私有目录中。

# 五、总结

这个项目实际就开发了一周多一点的时间吧，看github的commit记录也可以看得出来，可以说这一小段时间又加深了自己对于安卓开发的一点小小的经验，也遇到了大大小小的bug，比如机型适配这种问题，鸿蒙系统好像Toast的内容不能为空，用vivo测试就没有问题……

同时因为时间比较紧张，所以功能实现的其实并不全面，find页面就没有实现任何功能，聊天也就仅限于发，不能删……但是一个好的应用是需要时间慢慢打磨的，更多地也是需要依靠个人的热情以及经验支持。