夏风天气软件 V1.0 说明书

1. 程序框架结构

1.1. 程序架构

夏风天气软件采用MVVM框架进行编写，下图给出了MVVM设计的基本模型（图1-1）：

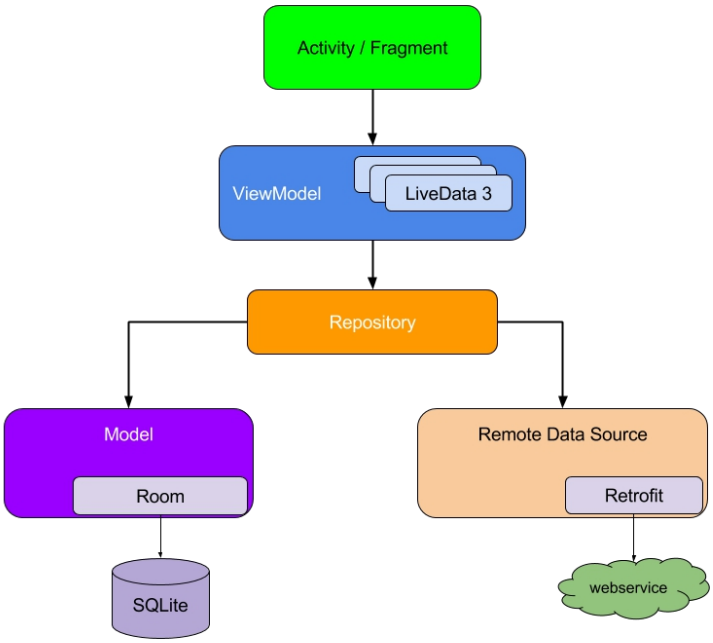


图 1 - 1 MVVM 框架

借助这个模型，通过结合 Retrofit+DataBinding+LiveData+ViewModel 来搭建 App 客户端。

1.2. 程序特点

基于以上设计模式，该软件主要具有以下优势：

1.2.1. LiveData 优势

当数据发生改变时，LiveData 的观察者能够及时捕捉到数据改变并将其反馈到界面上，保证了 UI 与数据实时一致。告别了在子线程中获取数据，通过 Message 传递数据来达到更新界面的不便之处。

1.2.2. ViewModel 优势

Activity 与 Fragment 之间的通信通过 ViewModel 来实现，不再使用 Intent 或者回调接口来交互数据，通过创建具有同一个 Owner 的 ViewModel 可以实现 Activity 里面的数据交流，这样做的好处在于只要有一个界面的数据发生了改变其他的界面都会及时刷新。下面是一个 Activity 与 Fragment 通信的示例：

1. Activity 与 Fragment 创建 ViewModel Object：

HomeActivity 创建 ViewModel

```
private val viewModel by lazy {
    ViewModelProvider( owner: this)[WeatherDataViewModel::class.java]
}
```

HomeFragment 创建 ViewModel

```
//获取ViewModel
private val viewModel by lazy {
    ViewModelProvider(requireActivity())[WeatherDataViewModel::class.java]
}
```

2. HomeFragment 设置当前主界面应该显示的 Fragment 对应的 index：

```
binding.seeTodayWeather.setOnClickListener { it: View!
    viewModel.setCurrentPage(1)
}
```

3. ViewModel 中将更新的数值 Value 传给 LiveData 数据 currentPage：

```
val currentPage = MutableLiveData<Int>()

fun setCurrentPage(index:Int){
    currentPage.postValue(index)
}
```

4. HomeActivity 观察到 current 发生改变，对界面进行刷新：

```
viewModel.currentPage.observe( owner: this){ it: Int!
    binding.viewPager.currentItem = it
    binding.bottomNavView.selectTabAt(it)
}
```

1.2.3. DataBinding 的优势

1.2.3.1. Item 与 item_layout

可以将 item 对象直接与对应的布局文件 item_layout 绑定，这样在 item 对象的属性发生改变后界面也可以及时反馈，我们以 City Model 为例。在 item_city.xml 中，我们首先创建了一个 variable，其名称为 item，类型为 City，这样我们就可以在 TextView 中使用以下语句：

```
android:text="@{item.city_id}"
```

来将 item 对象对应的属性呈现在 TextView 里面

```
<?xml version="1.0" encoding="utf-8"?>
<layout
xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable
            name="item"
            type="com.example.takisukaze.model.City" />
```

```
    </data>
    <androidx.cardview.widget.CardView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="@style/MyCardView"
        android:alpha="0.85">
        <LinearLayout
            android:id="@+id/rootView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:descendantFocusability="blocksDescendants"

android:background="?android:attr/selectableItemBackground"
            android:clickable="true"
            android:orientation="vertical"
            android:focusable="true"
            android:padding="5dp">
            <TextView
                android:textColor="@color/black"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@{item.city_id}"/>
            <TextView
                android:textColor="@color/black"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"

android:text="@{item.administrative_attribution}"/>
            <TextView
                android:textColor="@color/black"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@{item.city_abbreviation}"/>
            <TextView
                android:textColor="@color/black"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@{item.pinyin}"/>
        </LinearLayout>
    </androidx.cardview.widget.CardView>
</layout>
```

1.2.3.2. 抛弃 findViewById

我们通常获取页面中的控件会使用 findViewById()方法，使用此方法的弊端有两点：

1.如果控件数量较多，需要书写大量的 findViewById()方法，使得代码显示臃肿

2.如果不同的布局文件中存在相同 id 的控件，很容易产生不可预知的错误

而采用 DataBinding，我们可以通过 binding 对象来获取对应页面的控件，避免了上述两点麻烦，以下是示例：

1.2.3.2.1. 创建 Binding 对象并赋值

```
private lateinit var binding: ActivityHomeBinding
```

```
binding = DataBindingUtil.setContentView( activity: this,R.layout.activity_home)
```

1.2.3.2.2. 通过 Binding 对象获取界面内控件

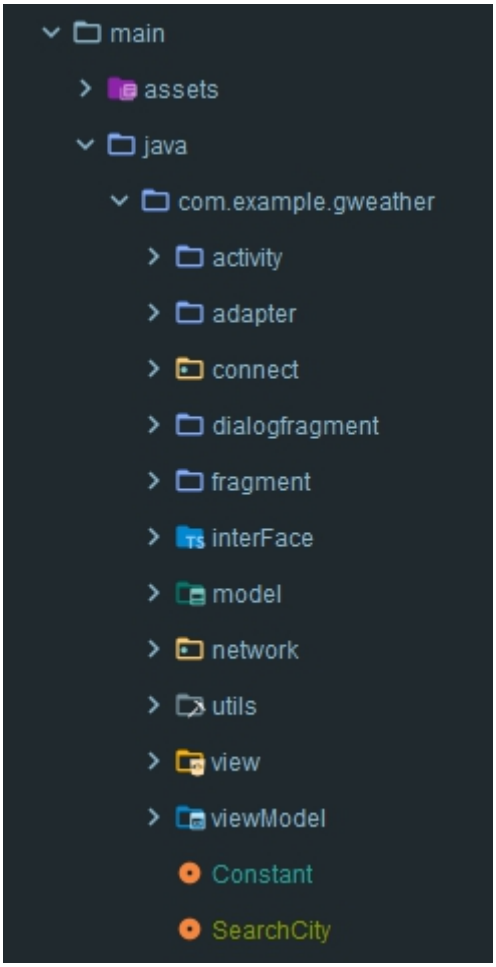
ViewPager 为 actvity_home.xml 内定义的控件

```
<androidx.viewpager.widget.ViewPager
    android:id="@+id/viewPager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_below="@id/homeToolbar"
    android:layout_above="@id/bottomNavView" />
```

```
binding.viewPager.apply{  this: ViewPager
    this.adapter = adapter
    /**
     * view_pager滑动事件
     */
    addOnPageChangeListener(object : OnPageChangeListener {
        override fun onPageScrolled(i: Int, v: Float, i1: Int) {}
        override fun onPageSelected(i: Int) {
            // Selecting a tab at a specific position
            binding.bottomNavView.selectTabAt(i)
            if (i == 1 && isFirstUpdate) {
                val dataFragment: DataFragment =
                    fragments[i] as DataFragment
                dataFragment.getSelectedMessage()
                isFirstUpdate = false //设置为false
            }
        }

        override fun onPageScrollStateChanged(i: Int) {}
    })
}
```

2. 程序代码结构



3. UI 界面部分

3.1. UI 界面结构

App 的 UI 界面结构图如下图所示（图 3-1）：

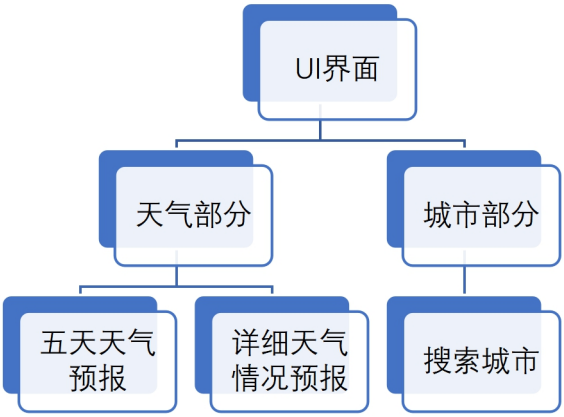


图 3- 1 App 客户端 UI 界面结构

3.2. 五天天气预报

在五天天气预报部分，夏风天气软件为用户提供了未来五天的天气预测，包括最高温、最低温、湿度和风速，同时还在界面上方显示了现在的天气实况方便用户浏览，如下图所示：

在用户不搜索城市的时候，程序获取的天气数据是根据手机 GPS 定位获取的定位来获得定位从而获取用户所处位置的天气，如果此时用户没有打开定位软件则会以弹窗的形式进行提示：

3.3. 详细天气情况预报

在详细天气情况预报部分，夏风天气软件向用户展示当前天气状况、包括室外温度、体感温度、能见度、湿度、风速，近五天天气质量预测等详细数据，保证用户对当前天气与较为详尽的了解，如下图所示：

3.4. 城市搜索

城市搜索部分用户可以在搜索栏搜索任意城市，以便了解当地天气情况。如下图，用户在搜索框内输入北京，下方就会显示出与北京有关的地区，用户点击对应的地区即可查看对应地区的天气情况，我们以点击石景山为例：

4. 使用 API 接口调用天气及城市更新

在图 1-1 中，可以了解软件中 Activity/Fragment 是从 ViewModel 中获取到数据的，而 ViewModel 是借助 Repository 来获取数据，而 Repository 则是使用 Retrofit 从 webService 获取到数据，以此过程为参照，我们以获取天气数据并更新为例子进行介绍。

4.1. Repository 的创建

[1]. 创建用于描述网络请求的接口

```
interface PlaceWeatherService {
    /**
     * 查询当日天气
     * @param location String
     * @return Call<DataResponse>
     */
    @GET( value: "v3/weather/now.json?key=${AppUtils.TOKEN}&language=zh-Hans&unit=c")
    fun searchPlaceWeather(@Query( value: "location") location:String):Call<NowDataResponse>

    /**
     * 调查最近几天的天气
     * @param location String
     * @return Call<DataResponse>
     */
    @GET( value: "v3/weather/daily.json?key=${AppUtils.TOKEN}&language=zh-Hans&unit=c&start=0&days=5")
    fun searchDailyWeather(@Query( value: "location") location:String):Call<DailyDataResponse>
}
```

[2]. 创建 Retrofit 实例

```
//天气请求根路径
private const val WEATHER_BASE_URL = "https://api.seniverse.com/"

private val weatherRetrofit = Retrofit.Builder()
    .baseUrl(WEATHER_BASE_URL)
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

[3]. 创建天气信息动态代理对象并创建 suspend 函数

```
//天气信息动态代理对象
private val placeWeatherService = ServiceCreator.weatherCreate(PlaceWeatherService::class.java)
```

```
suspend fun searchPlaceWeather(
    location:String
) = placeWeatherService.searchPlaceWeather(location).await()
```

```
private suspend fun <T> Call<T>.await():T{
    return suspendCoroutine { continuation ->
        enqueue(object :Callback<T>{
            override fun onResponse(call: Call<T>, response: Response<T>) {
                val body = response.body()
                if(body!=null) continuation.resume(body)
                else continuation.resumeWithException(
                    RuntimeException("response body is null")
                )
            }

            override fun onFailure(call: Call<T>, t: Throwable) {
                continuation.resumeWithException(t)
            }
        })
    }
}
```

[4]. 创建单例类 Repository，并添加请求天气数据函数

```
object Repository {
    fun searchPlaceWeather(location:String) = liveData(Dispatchers.IO) {  this: LiveDataScope<Result<List<Place>>>
        val result = try {
            val dataResponse = Network.searchPlaceWeather(location)
            if(dataResponse.results.isNotEmpty()){
                val place =  dataResponse.results
                Result.success(place)
            }else{
                Result.failure(RuntimeException("response data array is empty is ${dataResponse.results.isEmpty()}"))
            }
        }catch (e:Exception){
            Result.failure(e)
        }
        emit(result)
    }
```

4.2. ViewModel 获取数据并更新

[1]. 声明 LiveData 类型的变量_location 并创建 searchPlaces()方法

```
/**
 * location是请求天气等数据的核心
 */
private val _location = MutableLiveData<String>()
```

```
fun searchPlaces(location:String){
    _location.value = location
}
```

[2]. 当 Fragment 需要获取天气数据时，调用 viewModel 对象的 searchPlaces 方法，将查询地的的信息（即 location）传入进去，如下图所示。其中 AmapUtils 是一个自定义的高德地图定位工具类，getLocation()方法用于获取手机定位的经纬度坐标

```
viewModel.searchPlaces(AmapUtils.getLocation())
```

[3]. ViewModel 观察到_location 的数值发生变化，使用 Transformations.switchMap()方法调用 Repository.searchPlaceWeather(location)，将_location 对象转换为请求结果对象

```
val placeLiveData = Transformations.switchMap(_location){location->
    Repository.searchPlaceWeather(location)
}
```

[4]. Fragment 观察者发现 placeLiveData 发生了变化，将获取到的结果数值更新到界面上，至此 Fragment 便将获取到的数据更新成功

```
placeLiveData.observe(viewLifecycleOwner){ result ->
    //places 请求到的数据
    val places = result.getOrNull()
    if (places != null) {
        //更新ViewModel内数据
        places[0].now.apply { this: Now
            viewModel.updateWeatherData(
                temperature.toFloat(),
                feelLike.toInt(),
                humidity.toFloat(),
                windDirection,
                windSpeed.toFloat(),
                windDirectionDegree.toFloat(),
                text,
                visibility.toFloat()
            )
        }
        Log.d( tag: "updateWeather success", msg: "数据更新完成")
    } else {
        PopupWindowUtils.showShortMsg(requireContext(), msg: "查询数据错误")
        result.exceptionOrNull()?.printStackTrace()
    }
}
```