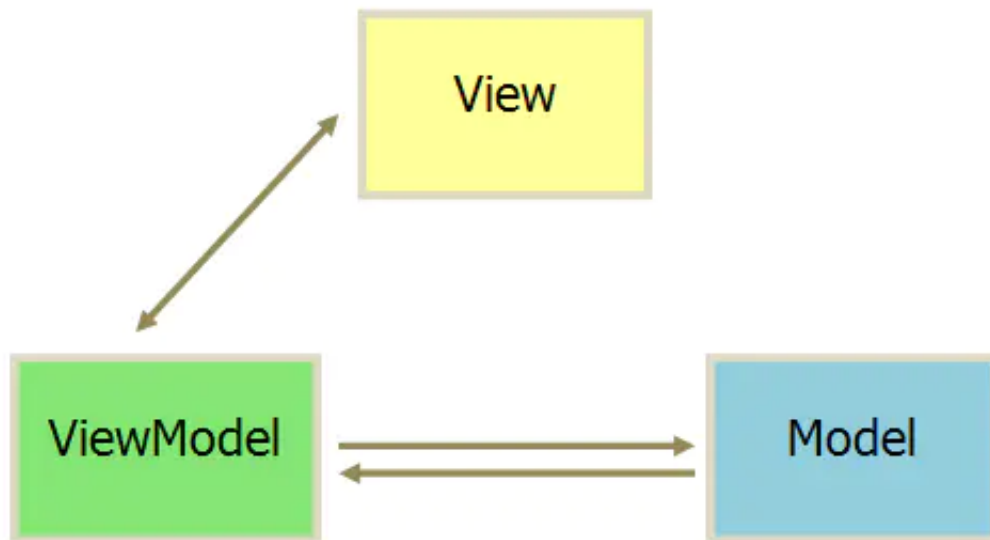


MVVM架构在项目中的落地和应用

一、Mvvm介绍(Mvvm+Databind+LiveData):

Mvvm:

- 与传统的Mvc, Mvp架构模式类似, Mvvm也采用三层结构: Model, View, ViewModel, 其中ViewModel的功能跟Mvp中的Presenter非常接近



Databinding:

- DataBinding 是谷歌官方发布的一个框架, 顾名思义即为数据绑定, 是 MVVM 模式在 Android 上的一种实现, 用于降低布局和逻辑的耦合性, 使代码逻辑更加清晰。

LiveData:

- LiveData是Jetpack的一个组件, 可以感知 Activity、Fragment生命周期的数据容器。当 LiveData 所持有的数据改变时, 它会通知相应的界面代码进行更新。同时, LiveData 持有界面代码 Lifecycle 的引用, 这意味着它会在界面代码 (LifecycleOwner) 的生命周期处于 started 或 resumed 时作出相应更新, 而在 LifecycleOwner 被销毁时停止更新。

二、Mvvm的使用场景(Mvvm+Databind+LiveData):

- 项目结构比较复杂, 界面数据频繁刷新。多用于中大型项目
- 配合新技术, 例如Jetpack(其中Room, lifecycle, livedata)等都比较偏向于MVVM的设计

三、新技术或开源库的优缺点：

- Mvvm

优点：

双向绑定技术，当Model变化时，View-Model会自动更新，View也会自动变化

重复的操作View的部分完全省略，代码量少，结构清晰

缺点：

数据绑定使得 Bug 很难被调试，上手难度稍稍偏大

- LiveData

优点：

UI和实时数据保持一致，

自动管理数据的更新机制，不需要再解决生命周期带来的问题

减少日常开发中因为数据持有View对象导致内存泄漏的问题

解决Configuration Change问题，在屏幕发生旋转或者被回收再次启动，立刻就能收到最新的数

据

缺点：

需要对正常使用的数据进行一层包装，设置数据时，稍稍要注意下线程问题(postValue和直接.value)

- DataBinding

优点：

数据与界面可以双向绑定，减少日常开发中的各种View的设置属性，监听等等代码的编写

View的功能进一步的强化，具有控制的部分功能

减少View层和Model层的部分操作View属性的逻辑代码，再也不必为View层臃肿的代码而烦恼

缺点：

因为xml文件中引入了部分简单逻辑，出现语法错误有可能检测不出，导致编译或者运行出错，调试代码难度偏大。

MVVM封装流程：

- 技术引入前配置依赖

- Databinding： 在使用到了databinding的模块中的build.gradle文件中的android节点下加入

```
buildFeatures {  
    dataBinding = true  
}
```

- LiveData:在使用到了LiveData的模块中的build.gradle文件中的dependencies中加入livedata, lifecycle

的依赖

```
androidx.lifecycle:lifecycle-livedata-ktx:2.2.0  
androidx.lifecycle:lifecycle-extensions:2.2.0
```

- 具体封装流程以及使用方式

例如BaseMvvmActivity,BaseMvvmFragment,BaseVmViewMoel,简化具体业务类的公共逻辑

V : ViewDataBinding 一般都是Activity, Fragment或者layout文件对应的V层对象

VM: 保存了V层的数据的对象, 其中基础数据类型或者Object类型使用ObservableField, 列表数据类型使用

MutableLiveData类型(LiveData是抽象类, ObservableField和MutableLiveData实现了LiveData)

Repository: VM层的数据仓库, 一般由数据库, 网络接口, 本地缓存等对象组成

整体加载数据流程: 当V层创建时, 会去获取V层对应的VM, 而VM层会从Repository获取V层的数据来源, 然后赋值给VM中声明的ObservableField或者MutableLiveData对象, 而LiveData对象一旦值发生改变后, 会自动通知与其关联的lifecycleOwner对象(也就是V层, XML文件中引用了LiveData对象的View去刷新)

```
/**  
 * desc    :包含网络状态, 数据状态的基类Activity T:对应界面布局的dataBind对象 VM:对应界  
 *          面的ViewModel对象  
 * date    : 2020/08/04  
 * version: 1.0  
 */  
@Suppress("UNCHECKED_CAST")  
abstract class BaseVMActivity<V : ViewDataBinding, VM : BaseVMViewModel> :  
    BaseActivity() {
```

```

/**
 * 初始化base的dataBind对象，并且注册lifecycle
 */
private val mBaseBinding: ActivityBaseVmBinding by lazy {
    DataBindingUtil.setContentViews<ActivityBaseVmBinding>(this,
        R.layout.activity_base_vm)
        .apply {
            lifecycleOwner = this@BaseVMActivity
        }
}

/**
 * 延迟加载ViewModel
 */
protected val mViewModel by lazy {
    val clazz =
this.javaClass.kotlin.supertypes[0].arguments[1].type?.classifier as
KClass<VM>
    getViewModel(clazz) //koin 注入
}

/**
 * 定义子类的view，用于跟子类的dataBind进行绑定
 */
private var mChildView: View? = null

/**
 * 定义子类的dataBind对象
 */
protected lateinit var mDataBind: V

/**
 * 设置dataBind为true，让父类不要setContentViews
 * @return Boolean
 */
override fun isDataBind() = true

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    //获取子类初始化的ViewModel
    mBaseBinding.viewModel = mViewModel
    //开启base的liveData的数据变化监听
    startBaseObserve()
    //初始化子类的布局
    initChild()
    //开启子类的LiveData监听

```

```

        startObserve()
        //初始化子类的view
        initView()
        //初始化子类的数据
        initData()
        //添加通用的提示框
    }

    /**
     * 初始化子类布局
     */
    private fun initChild() {
        if (!mBaseBinding.baseChildView.isInflated) {
            mBaseBinding.baseChildView.viewStub?.layoutResource =
getLayoutId()
            mChildView = mBaseBinding.baseChildView.viewStub?.inflate()
            if (mChildView != null) {
                mDataBind = DataBindingUtil.bind(mChildView!!)!!
                mDataBind.setVariable(initModelBrId(), mViewModel)
            }
        }
    }

    fun getBaseContainer(): ConstraintLayout =mBaseBinding.clBaseContainer

    /**
     * 开始监控baseViewModel的数据变化，包含网络状态，标题栏，以及错误类的布局加载
     */
    private fun startBaseObserve() {

        mViewModel.baseStatusModel.observe(this, Observer {
            setStatus(it)
        })

        mViewModel.baseTitleModel.observe(this, Observer {
            setTitle()
        })
        mViewModel.baseToastModel.observe(this, Observer {
            if (it.contentId > 0) {
                ToastUtil.show(this@BaseVMActivity, getString(it.contentId),
it.isCenter)
            } else {
                ToastUtil.show(this@BaseVMActivity, it.content, it.isCenter)
            }
        })

        mViewModel.baseIntentModel.observe(this, Observer {

```

```

        val startIntent = Intent(this@BaseVMActivity, it.clazz)
        if (it.dataMap != null && it.dataMap.size > 0) {
            it.dataMap.forEach { (key, value) ->
                startIntent.putAnyExtras(key, value)
            }
        }
        startActivityResult(startIntent, it.requestCode)
    })

    mViewModel.baseFinishModel.observe(this, Observer {
        if (it.finish) {
            if (it.dataMap != null && it.dataMap.size > 0) {
                val finishIntent = Intent()
                it.dataMap.forEach { (key, value) ->
                    finishIntent.putAnyExtras(key, value)
                }
                setResult(it.resultCode, finishIntent)
            } else {
                setResult(it.resultCode)
            }
            finish()
        }
    })

    mViewModel.baseResultModel.observe(this, Observer {
        if (it.dataMap != null && it.dataMap.size > 0) {
            val finishIntent = Intent()
            it.dataMap.forEach { (key, value) ->
                finishIntent.putAnyExtras(key, value)
            }
            setResult(it.resultCode, finishIntent)
        } else {
            setResult(it.resultCode)
        }
    })
})

}

/**
 * 初始化子类viewModel的BrId
 * @return Int
 */
abstract fun initModelBrId(): Int

/**

```

```

        * 开启子类的LiveData观察者
        */
abstract fun startObserve()

/**
 * 初始化网络错误的View
 * @return Int 化网络错误View的layoutId
 */
protected open fun initErrorLayout(): Int {
    return R.layout.base_layout_error
}

/**
 * 初始化加载中的view
 * @return Int 加载中View的layoutId
 */
protected open fun initLoadLayout(): Int {
    return R.layout.base_layout_loading
}

/**
 * 初始化空数据的view
 * @return Int 空数据View的layoutId
 */
protected open fun initEmptyLayout(): Int {
    return R.layout.base_layout_empty
}
}

```

```

<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:bind="http://schemas.android.com/apk/res-auto">

    <data>
        <!-- 声明BaseNetActivity所需要的ViewModel -->
        <variable
            name="viewModel"
            type="com.rm.baselisten.viewmodel.BaseVMViewModel" />
    </data>

```

```

<androidx.constraintlayout.widget.ConstraintLayout
    android:id="@+id/clBaseContainer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/base_activity_bg_color">

    <!-- BaseNetActivity的TitleBar -->
    <ViewStub
        android:id="@+id/baseTitleLayout"
        android:layout_width="match_parent"
        android:layout_height="48dp"
        android:layout="@layout/base_layout_title"
        bind:title="@{viewModel.baseTitleModel}"
        tools:ignore="MissingConstraints" />

    <!-- BaseNetActivity的子类布局容器 -->
    <ViewStub
        android:id="@+id/baseChildView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintTop_toBottomOf="@+id/baseTitleLayout"
        />

    <!-- BaseNetActivity的正在加载中布局容器 -->
    <ViewStub
        android:id="@+id/baseLoad"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        bind:status="@{viewModel.baseStatusModel}"
        />

    <!-- BaseNetActivity的网络错误布局容器 -->
    <ViewStub
        android:id="@+id/baseError"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"

```



```

        bind:status="@{viewModel.baseStatusModel}"
    />

    <!-- BaseNetActivity的空数据布局容器 -->
    <ViewStub
        android:id="@+id/baseEmpty"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        bind:status="@{viewModel.baseStatusModel}"

    />

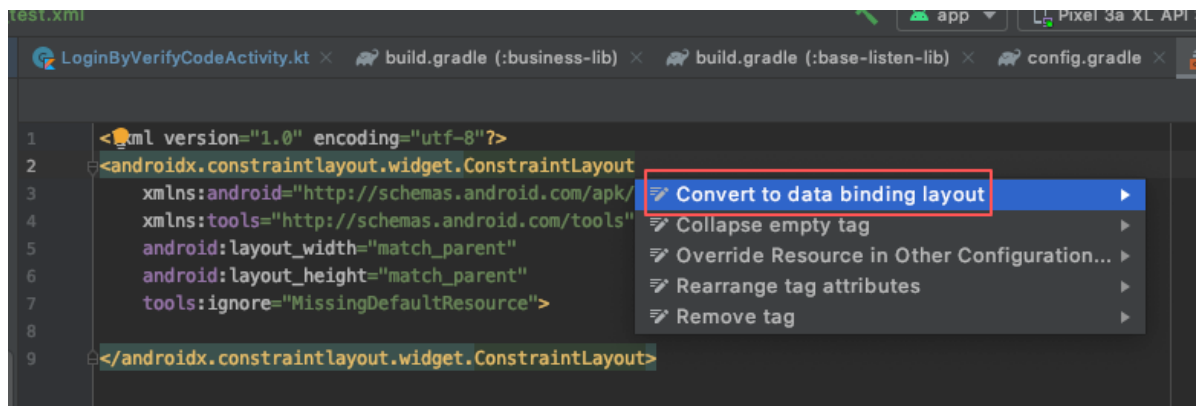
</androidx.constraintlayout.widget.ConstraintLayout>

</layout>

```

使用流程(以Activity为例子)

- 新建一个Activity，并且将Activity对应的布局文件按照提示修改为databinding类型的，



- 定义该Activity对应的ViewModel(类似MVP模式中的Presenter，可以认为是对数据的操作者，包含获取，修改，删除等等)，该ViewModel中所有声明的数据类型和方法，大部分都会跟Activity中的xml文件绑定，ViewModel的作用就是声明对应View层所需的数据类型，然后通过和repository(可以为是数据的存储仓库，例如网络接口，数据库集合，本地缓存等等)的交互，对页面上的数据进行一系列操作，然后通过数据绑定，将变化后的数据直接显示在界面上。

```

14
15  /**
16   * desc : 登陆viewModel
17   * date : 2020/08/26
18   * version: 1.0
19  */
20  class LoginByVerifyViewModel(private val repository: LoginRepository) : BaseVMViewModel() {
21      var phoneInputViewModel = PhoneInputViewModel()
22
23      // 用户协议和隐私协议是否选择
24      var isChecked = ObservableField<Boolean>(value: false)
25
26      // 错误提示信息
27      var errorTips = ObservableField<String>(value: "")
28
29      var loginStatus = ObservableField(value: 0)
30
31      // // 回调到Activity中的方法块
32      // var callBackToActivity: (Int) -> Unit = {}
33
34      var testDialogData = MutableLiveData<MutableList<LoginDialogModel>>(mutableListOf())
35
36      /** 获取验证码 ...*/
37      fun getCode() {...}
38
39      /** 密码登录入口 ...*/
40      fun loginByPassword() {...}
41
42      fun testClick() {...}
43
44      fun getDialogData() {...}
45
46
47
48
49
50

```

```
LoginByVerifyCodeActivity.kt x login_activity_login_by_verify_code.xml x build.gradle (:business-lib) x build.gradle (:base-listen-lib) x
1 <?xml version="1.0" encoding="utf-8"?>
2 <layout xmlns:android="http://schemas.android.com/apk/res/android"
3       xmlns:app="http://schemas.android.com/apk/res-auto"
4       xmlns:bind="http://schemas.android.com/apk/res-auto"
5       xmlns:tools="http://schemas.android.com/tools">
6
7     <data>
8
9         <variable
10            name="viewModel"
11            type="com.rm.module_login.viewmodel.LoginByVerifyViewModel" />
12     </data>
13
14     <androidx.constraintlayout.widget.ConstraintLayout
15         android:layout_width="match_parent"
16         android:layout_height="match_parent"
17         tools:visibility="visible">
18
19         <androidx.appcompat.widget.AppCompatImageView...>
20
21         <include
22             layout="@layout/login_include_layout_phone_input"
23             android:layout_width="315dp"
24             android:layout_height="56dp"
25             android:layout_marginTop="50dp"
26             app:layout_constraintLeft_toLeftOf="parent"
27             app:layout_constraintRight_toRightOf="parent"
28             app:layout_constraintTop_toBottomOf="@id/login_by_verify_code_icon"
29             bind:phoneInputViewModel="@{viewModel.phoneInputViewModel}" />
30
31         <TextView
32             style="@style/BusinessTextStyleErrorTips"
33             bindText="@{viewModel.errorTips}"
34             android:layout_marginStart="22dp"
35             android:layout_marginTop="6dp"
36             app:layout_constraintLeft_toLeftOf="@id/login_include_phone_input_lay"
37             app:layout_constraintTop_toBottomOf="@id/login_include_phone_input_lay"
38             tools:text="手机号有误" />
39
40
41
42
43
44
45
46
47
```

- 让该Activity继承BaseVMActivity，并指定相应的泛型类，实现抽象的方法

```
/**
 * desc    : 验证码登陆界面
 * date    : 2020/08/26
 * version: 1.0
 */
class LoginByVerifyCodeActivity :
    BaseVMActivity<LoginActivityLoginByVerifyCodeBinding,
    LoginByVerifyViewModel>() {
    companion object {
        fun startActivity(context: Context) {
            context.startActivity(Intent(context,
                LoginByVerifyCodeActivity::class.java))
        }
    }
}

/**
```

```

    * 获取Avtivity的布局文件
    * @return Int
    */
    override fun getLayoutId(): Int =
        R.layout.login_activity_login_by_verify_code

    /**
     * 获取dataBinding模式下Activity布局文件中声明的viewModel的BR的id,BR类是
     dataBind特有的资源ID集合类, 类似于Android的R文件
     * @return Int
     */
    override fun initModelBrId() = BR.viewModel

    /**
     * MVVM模式开启LiveData的数据监听
     */
    override fun startObserve() {
        mViewModel.testDialogData.observe(this, Observer {
            dialogAdapter.setList(mViewModel.testDialogData.value!!)
        })
    }
}

```

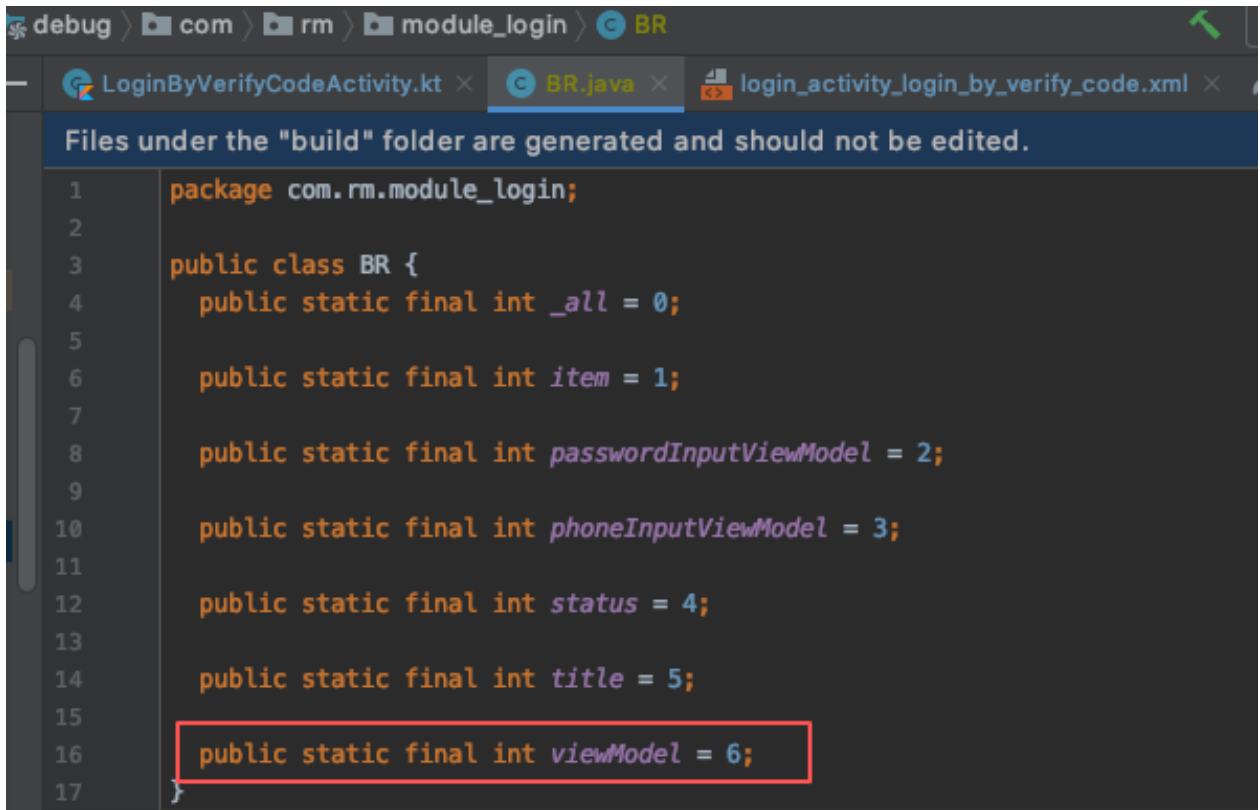
- 注意事项

- Activity的第一个泛型参数, 是DataBinding框架自动根据每个xml文件对应生成的类(规则xml是每个单词首字母大写, 去掉下划线然后以Binding结尾), 例如LoginByVerifyCodeActivity对应的xml文件是login_activity_login_by_verify_code, 那么就会有一个LoginActivityLoginByVerifyCodeBinding, 如果对应的xml文件是activity_test,那么就是ActivityTestBinding
- BR文件也是DataBinding框架的一个自动生成的类, 里面存放了, 所有DataBinding形式的xml文件中声明的variable变量,例如下图中在xml中声明了一个viewModel的变量, 那么在BR文件中就会有有一个viewModel的ID

```

<data>
    <!-- xml中逻辑需要用的类, 需要在data节点倒入, 类似java中的导包 -->
    <import type="android.text.TextUtils"/>
    <!-- xml中逻辑需要用的变量, 需要声明为variable, name属性会在BR文件中生成对应的
    ID type对应该变量的类的导包 -->
    <variable
        name="viewModel"
        type="com.rm.module_login.viewmodel.LoginByVerifyViewModel" />
</data>

```



```
debug > com > rm > module_login > BR
LoginByVerifyCodeActivity.kt x BR.java x login_activity_login_by_verify_code.xml x
Files under the "build" folder are generated and should not be edited.
1 package com.rm.module_login;
2
3 public class BR {
4     public static final int _all = 0;
5
6     public static final int item = 1;
7
8     public static final int passwordInputViewModel = 2;
9
10    public static final int phoneInputViewModel = 3;
11
12    public static final int status = 4;
13
14    public static final int title = 5;
15
16    public static final int viewModel = 6;
17 }
```

- DataBinding框架提供了@BindAdapter注解给开发者使用，可以根据业务需求扩展View的功能，举个例子，我们需要对View的onClickListener做扩展，先使用@BindAdapter声明属性名字和具体方法执行的逻辑，然后在xml中直接使用该属性，如果xml文件与viewModel中的属性是单向绑定，使用@{}，如果是双向绑定，请使用@={}，例如EditText输入用户手机号码

```
/**
 * BindingAdapter注解里面的属性名字，是可以直接用在xml中的，方法定义声明的
 View.bindClick
 * 代表了是对view的功能扩展，接受的参数是一个代码块，也就是方法的引用
 * @receiver View 扩展对象为View
 * @param action Function0<Unit>? 方法体(或者叫代码块)
 */
@BindingAdapter("bindClick")
fun View.bindClick(action: (() -> Unit)?) {
    if (action != null) {
        setOnClickListener { action() }
    }
}
```

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <layout xmlns:android="http://schemas.android.com/apk/res/android"
3       xmlns:app="http://schemas.android.com/apk/res-auto"
4       xmlns:bind="http://schemas.android.com/apk/res-auto"
5       xmlns:tools="http://schemas.android.com/tools">
6
7     <data>
8         <!-- xml中逻辑需要用的类，需要在data节点倒入，类似java中的导包 -->
9         <import type="android.text.TextUtils"/>
10        <!-- xml中逻辑需要用的变量，需要声明为variable，name属性会在BR文件中生成对应的ID type对应应该变量的类的导包 -->
11        <variable
12            name="viewModel"
13            type="com.rm.module_login.viewmodel.LoginByVerifyViewModel" />
14    </data>
15
16    <androidx.constraintlayout.widget.ConstraintLayout
17        android:layout_width="match_parent"
18        android:layout_height="match_parent"
19        tools:visibility="visible">
20
21        <androidx.appcompat.widget.AppCompatImageView...>
22
23        <include...>
24
25        <TextView...>
26
27        <androidx.appcompat.widget.AppCompatButton
28            android:id="@+id/login_by_verify_code_get_btn"
29            style="@style/ButtonStyle"
30            bindClick="@{viewModel.getCode}"
31            android:layout_width="0dp"
32            android:layout_height="56dp"
33            android:layout_marginTop="52dp"
34            android:enabled="@{viewModel.phoneInputViewModel.isInputText}"
35            android:text="获取验证码"
36            app:layout_constraintLeft_toLeftOf="@id/login_include_phone_input_lay"
37            app:layout_constraintRight_toRightOf="@id/login_include_phone_input_lay"
38            app:layout_constraintTop_toBottomOf="@id/login_include_phone_input_lay" />
39    </androidx.constraintlayout.widget.ConstraintLayout>
40
41    <TextView...>
42
43    <androidx.appcompat.widget.AppCompatButton
44        android:id="@+id/login_by_verify_code_get_btn"
45        style="@style/ButtonStyle"
46        bindClick="@{viewModel.getCode}"
47        android:layout_width="0dp"
48        android:layout_height="56dp"
49        android:layout_marginTop="52dp"
50        android:enabled="@{viewModel.phoneInputViewModel.isInputText}"
51        android:text="获取验证码"
52        app:layout_constraintLeft_toLeftOf="@id/login_include_phone_input_lay"
53        app:layout_constraintRight_toRightOf="@id/login_include_phone_input_lay"
54        app:layout_constraintTop_toBottomOf="@id/login_include_phone_input_lay" />
55</layout>
```

使用BindAdapter声明的属性
对应传入viewmodel中的方法

- 通用的Activity, Fragment, Adapter(再也不用定义RecyclerView的Adapter和Holder了)
 - 纯展示的Activity(自带数据源, 且不会发生改变)继承BaseActivity, 其他的Activity应该都统一继承MVVM模式的BaseVMActivity
 - 纯展示的Fragment(自带数据源, 且不会发生改变)继承BaseFragment, 其他的Activity应该都统一继承MVVM模式的BaseVMFragment
 - 纯展示的Dialog请使用CommonMvFragmentDialog, 其他都使用CommonMvFragmentDialog
 - 纯展示的(没有点击事件)单Item的RecyclerView请使用CommonBindAdapter, 涉及到逻辑操作的单Item的RecyclerView请使用CommonBindVMAdapter
 - 纯展示的(没有点击事件)多Item的RecyclerView请使用CommonMultiAdapter, 涉及到逻辑操作的多Item的RecyclerView请使用CommonMultiVMAdapter
- 示例Demo

地址: http://192.168.11.214:8087/listen_book/android/tree/develop_v1.0

下载项目后, 分支切到develop_1.0,按照下图操作即可运行demo



数据绑定

单ITEMTYPE的RV

单ITEMTYPE的RV(带点击)

多ITEMTYPE的RV

多ITEMTYPE的RV(带点击)

