目录

[一、MVC、MVP、MVVM 1](#_Toc3125)

[MVC，Model + View + Controller。 1](#_Toc11179)

[MVP，Model + View + Presenter。 2](#_Toc1786)

[MVVM 即 Model-View-ViewModel 3](#_Toc744)

[二、MVVM使用总结： 3](#_Toc29888)

[三、ViewModel使用 4](#_Toc194)

[四、LiveData使用 7](#_Toc16589)

[五、 DataBinding 9](#_Toc31338)

[① 单向绑定 10](#_Toc11913)

[② 双向绑定 13](#_Toc17592)

[Observable和LiveData的区别 17](#_Toc10806)

[Observable动态更新的3种方式 18](#_Toc650)

[对Jetpack四大组件的理解 18](#_Toc7253)

[1、添加Jetpack库 18](#_Toc28429)

[2、LifeCycle 18](#_Toc15689)

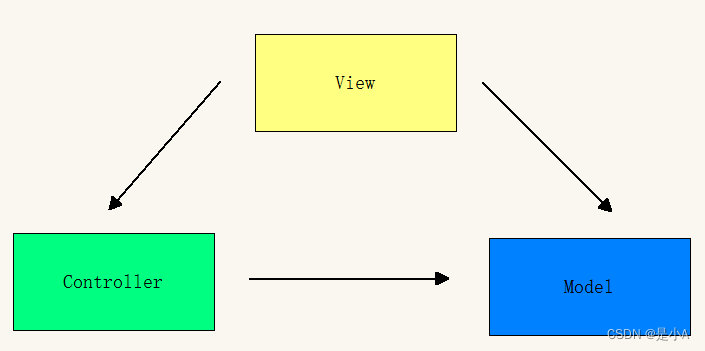
# 一、MVC、MVP、MVVM

# MVC，Model + View + Controller。

M——模型层（Model）负责处理数据的加载或者存储

V——视图层（View）负责界面数据的展示，与用户进行交互

C——控制器层（Controller）负责逻辑业务的处理



在[MVC模式](https://so.csdn.net/so/search?q=MVC%E6%A8%A1%E5%BC%8F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Ai1114/article/details/_blank)中，View层可以直接访问Model层和Controller层，所以View层包含Model层信息和Controller层的业务逻辑处理

在MVC模式中，Model层不依赖View层，但是View层依赖Model层，导致更改View层比较困难

Android中使用了Activity来充当Controller，但实际上一些UI也是由Activity来控制的，比如进度条等。因此部分视图就会跟Controller捆绑在同一个类了。同时，由于Activity的职责过大，Activity类的代码也会迅速膨胀。

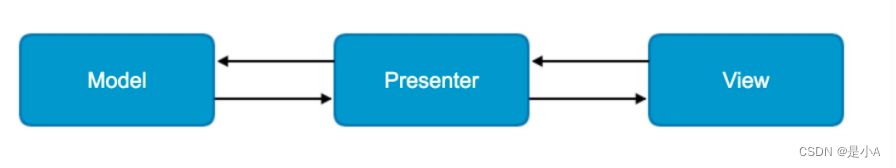
MVC还有一个重要的缺陷就是View跟Model是有交互的，没有做到完全的分离，这就会产生耦合。

# MVP，Model + View + Presenter。

M——数据层（Model）负责对数据的存取操作，例如对数据库的读写，网络的数据的请求等

V——视图层（View）负责对数据的展示，提供友好的界面与用户进行交互，Android通常将Activity或者Fragment作为View层

P——主持层（Presenter）连接View层与Model层的桥梁并对业务逻辑进行处理



**MVP执行流程：**

* View层收到用户的操作
* View层把用户的操作交给Presenter
* Presenter直接操作Model层进行业务逻辑处理
* Model层处理完毕后，通知Presenter
* Presenter收到通知后，去更新View层

相当于对MVC框架的进一步升级，解耦。不过也有缺点，额外增加了大量的接口、类，不方便进行管理，所以关于MVP的话就还有一个Contract要去处理。

Presenter层中包含了一个View接口，并且依赖于Model接口，从而将Model层与View层联系在一起

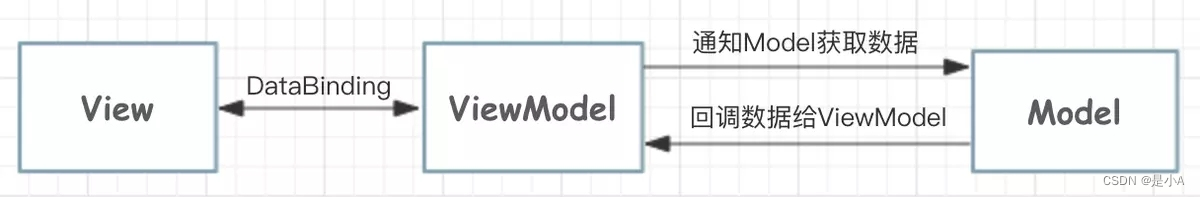
View层会持有一个Presenter成员变量并且只保留对Presenter接口的调用，具体业务逻辑全部交由Presenter接口实现类中处理。

## MVVM 即 Model-View-ViewModel

M——Model（模型）数据源（本地数据和网络数据等）实体模型，定义实体类，获取业务数据模型，如通过数据库或者网络来操作数据等

V——View（视图）xml、Activity、Fragment、Adapter和View等，主要进行控件的初始化设置

VM——ViewModel（控制器）：ViewModel 与 Model 直接交互处理数据以及逻辑处理，通过DataBinding将数据变化反应给View



**注：**ViewModel可以理解为View与Presenter的合成体

在 MVP 的基础上，MVVM 把 View 和Presenter也进行了解耦

MVVM 模式中，数据处理逻辑是独立于 UI 层的

ViewModel 只负责提供数据和处理数据，不会持有 View 层的引用

View 层只负责对数据变化的监听，不会处理任何跟数据相关的逻辑

View 层的 UI 发生变化时，也不需要像 MVP 模式那样修改对应接口和方法实现，一般情况下ViewModel 不需要做太多的改动

UI 的展现是依赖于数据的，数据的变化会自然的引发 UI 的变化，而 UI 的改变也会使数据 Model 进行对应的更新

ViewModel的优势在于生命周期和数据持久化，那么它就适用于Activity和Fragment，其次就是异步回调，不会造成内存泄漏，再次就是对View层和Model层进行隔离，两者不存在耦合性

# 二、MVVM使用总结：

一：准备工作

1创建MainActivity

2创建MainVM继承于AndroidViewModel

3在app的build.gradle中写dataBinding.enabled=true

4创建布局并加入<data><variable name= type=>

二：开始

1在MainActivity中绑定布局和VM（获取布局绑定实例binding和VM实例并绑定）

2

(1)绑定TextView或EditText:

在MainVM中用MutableLiveData<>创建变量，其中泛型是要监听绑定的数据的类型。同时在XML中将要监听的TextView中添加android:text=@{vm.text}若是EditText，则为@={vm.text}。此时完成双向绑定。

1. 绑定点击事件：

在MainVM中加入普通点击事件click（必须静态）。同时在XML文件的button上写入android:onClick="@{vm::click}"

1. 绑定其他控件：

在MainVM中用MutableLiveData<>创建变量，并新增方法（名随意，可以和后面的注释内容一样）和注释@BindingAdapter("bindImage(和xml中相同)"),方法有两个参数（1控件实例， 2变量）。然后在xml中需要绑定的控件下写入app:方法名="@{vm.变量}"

工作顺序是一旦imgdir.setValue()或者imgdir.postValue()更新了数据，那么bindImage方法便会执行，就实现了数据的监听，在bindImage方法中设置布局内容就相当于更新后的数据同步布局，达到高度解耦效果。

**Tips：**

Model从网络获取数据

ViewModel通过回调将数据接收

View会持有ViewModel实例，ViewModel持有Model实例

在MVVM中，Model是负责数据的获取、处理和存储的部分，它通常包含了业务逻辑和数据模型。ViewModel是负责将Model中的数据转化为View可以直接使用的形式，并且提供给View进行展示。ViewModel还负责处理用户与界面的交互，并将用户的操作反馈给Model进行数据的更新。

所以，在MVVM中，Model并不直接持有ViewModel实例，而是通过数据绑定或者观察者模式将数据传递给ViewModel，让ViewModel来处理数据的展示和交互逻辑。这样可以实现View和Model之间的解耦，提高代码的可维护性和可测试性。

# 三、ViewModel使用

ViewModel 是androidx包中抽象类，ViewModel 是一个以生命周期感知的方式来保存和管理 UI 相关数据的类。

ViewModel 还提供了一种在 Activity 和 Fragment 之间共享数据的机制。

一个ViewModel关联到一个ViewModelStoreOwner（Activity或Fragment），如果ViewModelStoreOwner由于配置改变而被销毁时（比如屏幕旋转），ViewModel并不会被销毁，新的ViewModelStoreOwner实例会重新连接到已经存在的ViewModel。

ViewMode存在的目的，就是为Activity/Fragment 获得以及保留 必要信息。 Activity / Fragment 应该可以观察到VIewModel的变化，ViewModel通常通过LiveData或者DataBinding 或者其它观察的结构者框架暴露信息。

ViewModel仅有的职责，就是为UI管理数据，它不应该访问到你任何的View层级 或者 持有Activity 、Fragment的引用。

**ViewModel**

启用方式

//添加ViewModel的引用

implementation 'androidx.appcompat:appcompat:1.0.2'

implementation 'androidx.lifecycle:lifecycle-viewmodel:2.2.0'

### ① 绑定Activity

在MVVM的框架中，每一个Activity都应该对应一个ViewModel

新建一个viewmodels包，包下新建一个MainViewModel类继承ViewModel抽象类，表示与MainActivity进行绑定。在ViewModel中定义两个变量 ，账号和密码

public class MainViewModel extends ViewModel {

private MutableLiveData<Integer> count = new MutableLiveData<>();

public void setCount(int count) {

this.count.setValue(count);

}

public LiveData<Integer> getCount() {

return count;

}

}

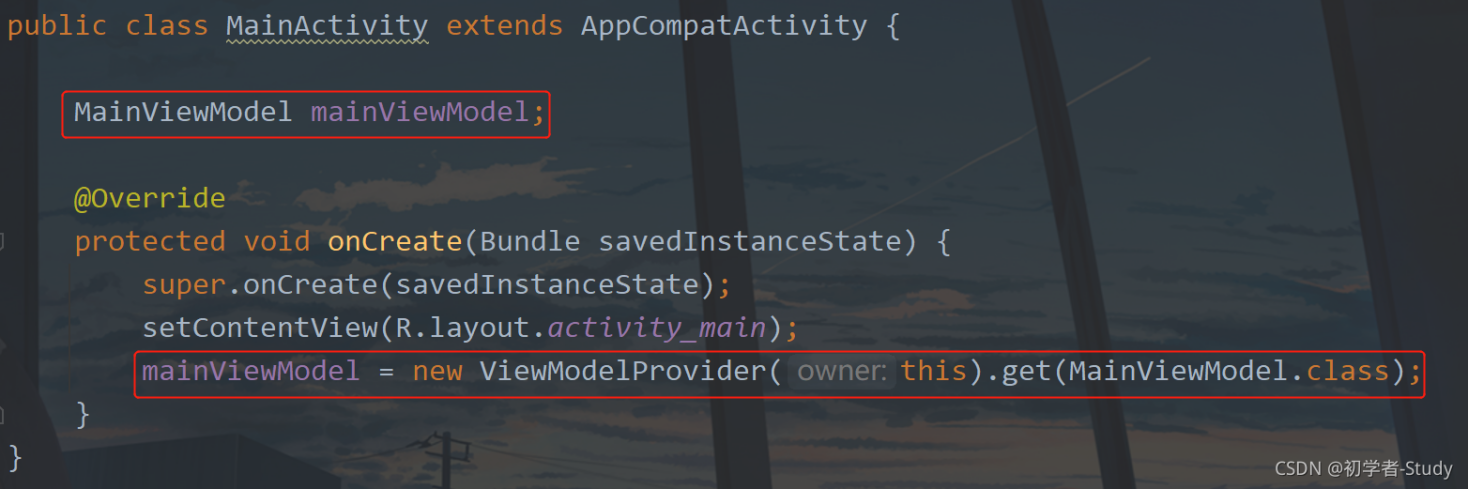
Kotlin：

public String account;

public String pwd;

kotlin会自动实现getter 和 setter

回到MainActivity中，修改代码如下



现在我们的MainActivity和MainViewModel就绑定起来了。ViewModel是数据持久化的，对于一些变量就可以直接放在ViewModel当中，而不再放在Activity中，可以根据一个实际的需求来进行。

### ② 页面布局绘制

**假设实现登录系统**

修改一下activity\_main.xml中的布局

### ③ 实现登录

回到MainActivity中

public class MyActivity extends AppCompatActivity {

private MyViewModel viewModel;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

viewModel = ViewModelProviders.of(this).get(MyViewModel.class);

viewModel.getCount().observe(this, count -> {

// 更新 UI

});

}

public void onButtonClicked(View view) {

int count = viewModel.getCount().getValue() + 1;

viewModel.setCount(count);

}

}

在这个 Activity 中，我们使用 ViewModelProviders.of() 方法获取 MyViewModel 的实例，并将其与该 Activity 绑定。传入什么就是创建和什么绑定的实例，要共享就传入同一个。

然后，我们可以使用 getCount() 方法获取 count 变量的值，并在 setCount() 方法中设置其值。

通过调用 getCount().observe() 方法，我们可以在 LiveData 数据变化时更新 UI。

**Tips：**

1. 不适合处理长时间运行的任务：

ViewModel 主要用于管理 UI 相关的数据，如果需要处理长时间运行的任务，需要在 ViewModel 中使用异步任务或者单独使用 Service。

1. 数据持久化需要额外处理：

ViewModel 中保存的数据只是暂时性的，如果需要长期保存数据，需要将数据保存到数据库或者 SharedPreferences 中。

1. 需要了解生命周期：

ViewModel 是以生命周期感知的方式来保存和管理数据的，因此需要了解 Activity 或 Fragment 的生命周期以便正确使用 ViewModel。

1. ViewModel 的生命周期不同于 Activity 和 Fragment，它是被系统缓存的，因此可能会出现数据被清除的情况。在使用 ViewModel 时应该注意它的生命周期，及时保存数据并恢复数据。
2. 在Fragment中想使用相同的ViewModel实例的话，假设ViewModelProvider是在BaseActivity中获取的：

可以在onAttach中初始化viewModelProvider 的实例。

if (context instanceof BaseActivity) {

BaseActivity baseActivity = (BaseActivity) context;

viewModelProvider = baseActivity.getViewModelProvider();

} else {

viewModelProvider = new ViewModelProvider(requireActivity());

}

# 四、LiveData使用

LiveData是用来做什么的？

1. 一种用于管理应用程序界面和数据交互的组件。
2. LiveData 是一种可观察的数据持有者，用于在应用程序组件（如 Activity、Fragment 和 Service）之间共享数据，并在数据发生更改时通知观察者。
3. LiveData 可以确保 UI 与数据的同步更新，避免了一些常见的错误，如内存泄漏和 UI 组件无法正确更新的问题。
4. LiveData 具有生命周期感知功能，可以自动感知应用程序组件的生命周期，并在组件处于活动状态时更新 UI，而在组件处于非活动状态时停止更新，从而有效地减少了资源消耗。
5. LiveData 还提供了线程安全的访问数据的机制，避免了多线程并发访问的问题。

**1数据变化感知**

依然是之前那个登录页面，不过需要修改一下MainViewModel中的变量，如下：

public MutableLiveData<String> account = new MutableLiveData<>();

public MutableLiveData<String> pwd = new MutableLiveData<>();

请注意这里使用的是MutableLiveData，表示值的内容开变动，而LiveData是不可变的。<>中的是泛型，你可以直接将一个对象放进去，当对象的内容有改动时，通知改变就可以了，现在这么写是为了方便理解。

改变一下布局activity\_main.xml 添加TextView

### 2 数据观察

然后回到MainActivity中，修改代码如下图所示：

account.setValue("admin")

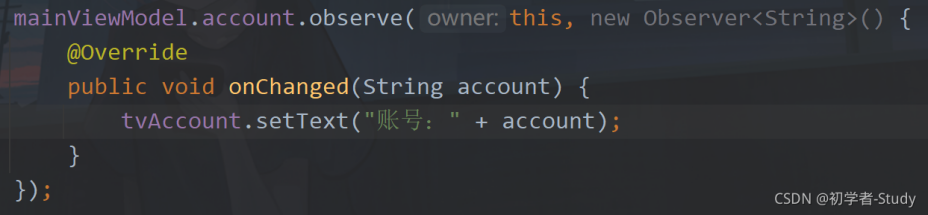
pwd.setValue("123456")



一个要注意的点是setValue（）中的参数是数据源，一般来源于Model从网络或别的地方获取。

下面的两处标注来看，首先在给MainViewModel中的account赋值时，采用了MutableLiveData的setValue()的方式，还有一种方式是postValue()，这里要注意一点setValue()只能在主线程中调用，postValue()可以在任何线程中调用。pwd也是一样的。

然后在最后一处标注的地方，对MainViewModel中的account和pwd进行数据观察observe()，在方法中处理数据改变的逻辑。当这两个值有改变时通知页面最新的值，这里用了lambda表达式进行了一次简化，实际的代码是这样的



Tips:

1. 如果在主线程执行发布之前，多次调用postValue方法，则只会分派最后一个值
2. API 限制：LiveData 是 Android Jetpack 组件，只能在支持 Jetpack 的 Android 版本上使用，对于一些较老的 Android 版本可能需要使用其他技术方案。
3. 数据共享：LiveData 可以在不同的组件之间共享数据，例如，当一个 Activity 和一个 Fragment 需要共享数据时，可以将 LiveData 对象设置为一个公共的数据持有者。
4. 资源释放：LiveData 可以在 UI 组件不再处于活动状态时自动释放资源，避免出现内存泄漏等问题。
5. 让liveData观察不受生命周期的影响：

observeForever()

要记得在相应的生命周期中注册观察，一般是onDestroy，看情景想在哪里用就在哪里用

cmViewModel.getCM\_locBeans\_NR().removeObservers(this);

# DataBinding

DataBinding 提供了一种声明性的方式将布局文件中的 UI 组件和应用程序的数据模型绑定在一起。

通过 DataBinding，开发者可以将 UI 组件的值绑定到数据模型中的属性，使得在更新 UI 时不需要手动更新每个组件的值。

**使用：**

配置：

首先在module的build.gradle中配置如下

//早期配置

defaultConfig{

dataBinding{

enabled=true

}

}

// 新版替代方案

buildFeatures {

dataBinding = true

}

Android的DataBinding在已经内置了，因此只需要在app模块的build.gradle中开启就可以使用了。DataBinding，顾名思义就是数据绑定，可以看到现在的三个组件都与数据有关系，ViewModel数据持有，LiveData数据观察、DataBinding数据绑定。

单向绑定：数据源->UI

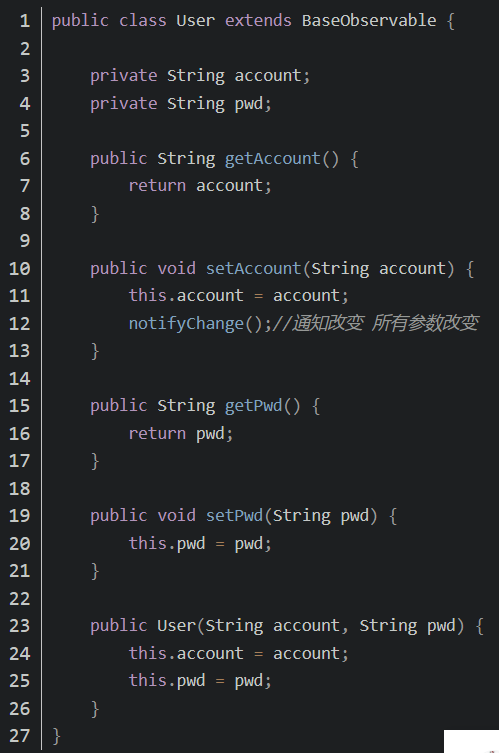
双向绑定：数据源<->UI，

## ① 单向绑定

举个例子：比如我手机上收到一个通知，我需要显示通知的文字内容在页面上，这就是单向绑定，而我页面上的文字内容改变也重新发一个通知出去，这就是双向绑定。

最常用的就是当我Model中的数据改变时，改变页面上的值。这个是单向绑定。

新建一个用户User对象



这里我继承了BaseObservable，注意它在androidx.databinding包下。然后我们的数据是需要显示在页面上的，而之前是通过Activity获取xml中的控件，然后显示数据在控件上，而现在有了DataBinding，可以直接和xml中的数据进行绑定

鼠标指向根布局Layout，按Alt+Enter键即可将布局xml变为DataBinding模式

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

xmlns:tools="http://schemas.android.com/tools">

<!--绑定数据-->

<data>

<variable

name="user"

type="com.llw.mvvm.User" />

</data>

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:gravity="center"

android:orientation="vertical"

android:padding="32dp"

tools:context=".MainActivity">

<com.google.android.material.textfield.TextInputLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content">

<com.google.android.material.textfield.TextInputEditText

android:id="@+id/et\_account"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:background="@color/white"

android:hint="账号" />

</com.google.android.material.textfield.TextInputLayout>

<com.google.android.material.textfield.TextInputLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:layout\_marginTop="12dp">

<com.google.android.material.textfield.TextInputEditText

android:id="@+id/et\_pwd"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:background="@color/white"

android:hint="密码"

android:inputType="textPassword" />

</com.google.android.material.textfield.TextInputLayout>

<com.google.android.material.button.MaterialButton

android:id="@+id/btn\_login"

android:layout\_width="match\_parent"

android:layout\_height="48dp"

android:layout\_margin="24dp"

android:insetTop="0dp"

android:insetBottom="0dp"

android:text="登 录"

app:cornerRadius="12dp" />

<TextView

android:id="@+id/tv\_account"

android:text="@{user.account}"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"/>

<TextView

android:id="@+id/tv\_pwd"

android:text="@{user.pwd}"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"/>

</LinearLayout>

</layout>

在最外层加了一个layout标签，然后将原来的布局放在layout里面，再增加一个数据源，也就是user对象，然后再底部的两个tv\_account和tv\_pwd两个TextView中的text属性中绑定了user对象中的属性值。

最后一步是在MainActivity中去进行绑定的。

进入MainActivity。在onCreate方法中修改代码如下图所示



1. 设置dataBinding的数据源为user对象
2. 更改user对象的值，会自动更新上去

初始化的修改时admin、123456。然后再通过输入框去修改。我将会输入study、666，然后点击登录按钮，也会将输入框的数据显示在TextView上

**Tips:**

1. Fragment要改成

binding = DataBindingUtil.inflate(inflater, R.layout.fragment\_all, container, false);

return binding.getRoot();

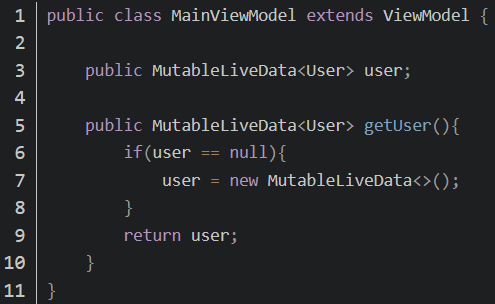
1. DataBinding中使用include：

要通过include的id去获取里面的控件

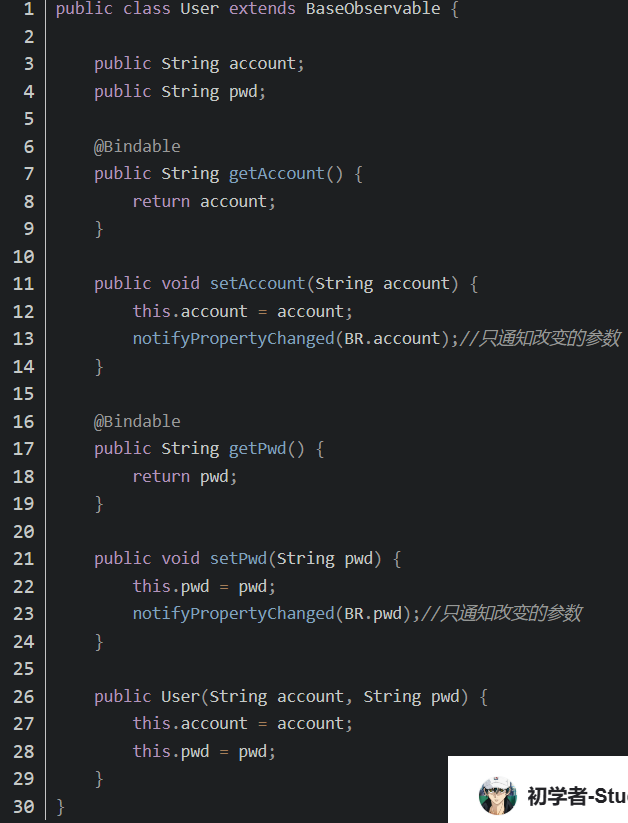
## ② 双向绑定

双向绑定是建立在单向绑定的基础上，实际的开发中用到双向绑定的地方并没有单向绑定多，双向绑定举一个例子，在输入框输入数据时候直接将数据源中的数据进行改变，这里会用到ViewModel和LiveData。

下面进行双向绑定的使用，修改一下MainViewModel

、

下面修改User类，这里面做了一些改变



不同于notifyChange()改变某一个参数，某一个对象都会通知，现在notifyPropertyChanged()就具有针对性，只通知对应属性改变。之前在activity\_main.xml中的data标签中是使用的User，现在我们改成ViewModel，顺便把布局调整一下，代码如下

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

xmlns:tools="http://schemas.android.com/tools">

<!--绑定数据-->

<data>

<variable

name="viewModel"

type="com.llw.mvvm.viewmodels.MainViewModel" />

</data>

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:gravity="center"

android:orientation="vertical"

android:padding="32dp"

tools:context=".MainActivity">

<TextView

android:id="@+id/tv\_account"

android:text="@{viewModel.user.account}"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"/>

<TextView

android:layout\_marginBottom="24dp"

android:id="@+id/tv\_pwd"

android:text="@{viewModel.user.pwd}"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"/>

<com.google.android.material.textfield.TextInputLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content">

<com.google.android.material.textfield.TextInputEditText

android:id="@+id/et\_account"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:background="@color/white"

android:text="@={viewModel.user.account}"

android:hint="账号" />

</com.google.android.material.textfield.TextInputLayout>

<com.google.android.material.textfield.TextInputLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:layout\_marginTop="12dp">

<com.google.android.material.textfield.TextInputEditText

android:id="@+id/et\_pwd"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:background="@color/white"

android:text="@={viewModel.user.pwd}"

android:hint="密码"

android:inputType="textPassword" />

</com.google.android.material.textfield.TextInputLayout>

<com.google.android.material.button.MaterialButton

android:id="@+id/btn\_login"

android:layout\_width="match\_parent"

android:layout\_height="48dp"

android:layout\_margin="24dp"

android:insetTop="0dp"

android:insetBottom="0dp"

android:text="登 录"

app:cornerRadius="12dp" />

</LinearLayout>

</layout>

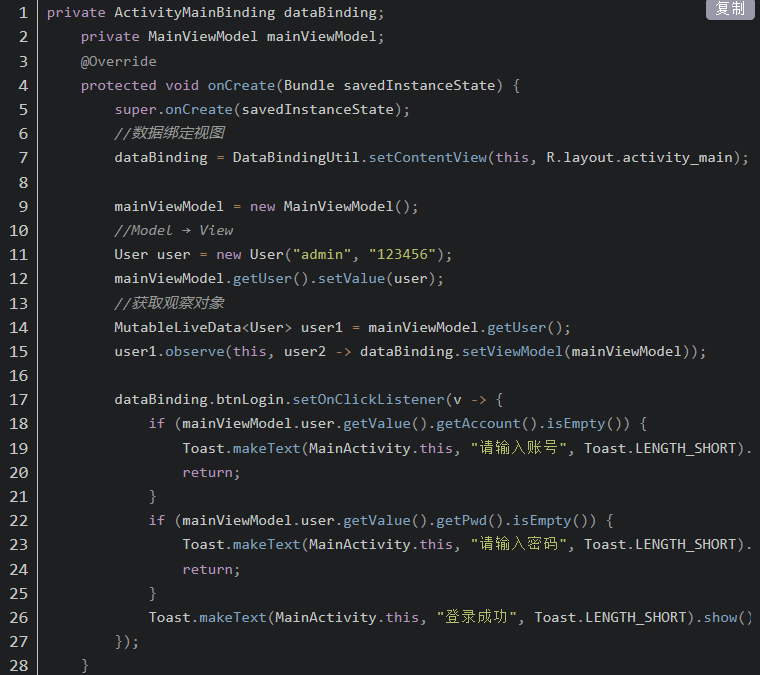
这里要注意点的地方有几个，

第一个是数据源，这里绑定的是ViewModel，那么相对应的ViewModel中的数据都可以拿到。

第二个就是响应的地方，通过这种方式去显示ViewModel中对象的变量数据在控件上。这里我把这两个TextView放到了输入框的上方

第三个地方，也是双向绑定的意义，就是UI改变数据源。我们都知道当输入框输入时，text属性值会改变为输入的数据，而@={viewModel.user.account}就是将输入的数据直接赋值给数据源。这样在Activity中我们将不需要去进行输入框的处理，减少了耦合。

下面让我们回到MainActivity中。修改代码后如下：



//19 .show

//23 .show

# Observable和LiveData的区别

前面讲到使用ViewModel+Data Binding解决内存泄漏问题时用到了ObservableInt/ObservableField，但其实还可以使用LiveData，它不仅可以实现与ObservableField相同的功能，而且有以下好处

ObservableField只有在数据发生改变时UI才会收到通知，而LiveData不同，只要你postValue或者setValue，UI都会收到通知，不管数据有无变化

LiveData能感知Activity的生命周期，在Activity不活动的时候不会触发，例如一个Activity不在任务栈顶部

# Observable动态更新的3种方式

# 对Jetpack四大组件的理解

Lifecycle：用来让其他的类感知生命周期

LiveData：用来感知数据变化

ViewModel：用来保存数据

DataBinding：用来简化更新UI操作，类似于在Xml文件内新增一个数据源，之后只需要操作数据源就可以实现更新，而不需要去设置具体的控件。

## 1、添加Jetpack库

build.gradle：

dependencies {

def lifecycle\_version = "2.4.1"

implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle\_version"//ViewModel

implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle\_version"// LiveData

implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle\_version"// Lifecycle

}

## 2、LifeCycle

Lifecycle 提供了一种轻松管理组件（如 Activity 和 Fragment）生命周期的方式，同时也支持自定义组件的生命周期。

Jetpack Lifecycle 提供了一组类和接口，使得开发者可以在组件的生命周期各个阶段执行相应的操作。

LifecycleOwner

: 拥有生命周期的对象，通常是 Activity 和 Fragment。

LifecycleObserver

: 监听组件的生命周期事件的观察者对象。

Lifecycle

: 组件的生命周期，包括 CREATED、STARTED、RESUMED、PAUSED、STOPPED、DESTROYED 等状态。

LiveData

: 一个可观察的数据容器，可以在组件生命周期的不同阶段更新数据。

例如，可以在组件被销毁时自动释放资源、取消网络请求等操作。

此外，Jetpack Lifecycle 还提供了一种方式来创建自定义的生命周期状态，以更好地满足 App 的需求。

**LifeCycle使用**

1. gradle添加组件库依赖

dependencies {

implementation "androidx.lifecycle:lifecycle-extensions:2.4.0"

}

1. 创建Timer 的 Java 类，并实现 LifeCycleObserver 接口。

public class Timer implements LifecycleObserver {

private Handler handler;

private int seconds = 0;

@OnLifecycleEvent(Lifecycle.Event.ON\_START)

public void startTimer() {

handler = new Handler();

handler.post(new Runnable() {

@Override

public void run() { //这里run不是start，其实是主线程可以更新UI，相当于是将代码放入runnable内发送给主线程

Log.d("Timer", "Seconds: " + seconds);

seconds++;

handler.postDelayed(this, 1000);

}

});

}

@OnLifecycleEvent(Lifecycle.Event.ON\_STOP)

public void stopTimer() {

handler.removeCallbacksAndMessages(null);

handler = null;

}

}

3. 在 MainActivity 中添加 LifecycleOwner，并在 onCreate 方法中添加 Observer。

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

// 获取LifecycleOwner对象

LifecycleOwner lifecycleOwner = this;

// 将Timer实例添加为Observer

getLifecycle().addObserver(new Timer());

// ...

}

// ...

}

这样，当 Activity 处于前台时，Timer 实例中的 startTimer 方法会被调用，计时器会开始递增；

当 Activity 被销毁时，Timer 实例中的 stopTimer 方法会被调用，计时器会停止。

Tips：

1. 需要添加注释：

在使用 LifeCycle 组件的时候，需要添加一些注释来指示方法是在什么时候被调用，否则可能会出现一些难以诊断的问题。

1. 不要在 onCreate() 方法中使用 Lifecycle 组件：

Lifecycle 组件在 onCreate() 方法中尚未初始化完成，因此在该方法中使用它们可能会导致崩溃或不可预测的行为。建议在 onStart() 方法中使用 Lifecycle 组件。

1. 不要手动调用 onDestroy() 方法：

手动调用 onDestroy() 方法会破坏 Lifecycle 组件的生命周期，从而导致应用程序行为异常。Lifecycle 组件应该由系统自动管理，应该避免手动干预。

1. 避免在 Fragment 中使用多个 LifecycleOwner：

Fragment 自身就是一个 LifecycleOwner，因此不应该在 Fragment 中创建其他的 LifecycleOwner。这样会导致多个 LifecycleOwner 之间的状态不同步，从而导致应用程序出现问题。