**Android MVVM**

2018年03月17日 11:33:45 [zzyawei](https://me.csdn.net/zzyawei) 阅读数：878 标签： [mvvm](http://so.csdn.net/so/search/s.do?q=mvvm&t=blog)[android mvvm](http://so.csdn.net/so/search/s.do?q=android%20mvvm&t=blog) 更多

个人分类： [android](https://blog.csdn.net/zzyawei/article/category/6814656)[mvvm](https://blog.csdn.net/zzyawei/article/category/7523464)

**1、MVVM 模式简介**

MVVM 软件设计模式由微软在2005年提出，下图及介绍总结自微软[The MVVM Pattern](https://msdn.microsoft.com/en-us/library/hh848246.aspx)和[Implementing the MVVM Pattern](https://msdn.microsoft.com/en-us/library/gg405484%28v=pandp.40%29.aspx)。上面两篇文章中和微软自家产品关联性很强，并很适用于Android，这里仅仅是介绍MVVM模式的概念及MVVM模式中各模块所承担的职责。



* **View**   
  就像在MVC和MVP模式中一样，视图是用户在屏幕上看到的结构、布局和外观（UI），决定如何呈现数据
* **ViewModel**   
  封装了View的显示逻辑和数据。不直接引用View。ViewModel实现来自View的命令（如点击事件）、处理（转换/聚合）View所需绑定的数据、通知View数据或状态的改变。ViewModel和数据和状态提供给View，但View决定了如何呈现。
* **Model**   
  封装了业务逻辑和数据（业务逻辑是指所有有关数据检索与处理的程序逻辑），并且保证数据的一致性和有效性。为了最大化重用机会，Model不应包含任何用于特定ViewModel的处理逻辑。
* **Binder 绑定器**   
  数据绑定技术的实现在MVVM中是必须的。Binder确保ViewModel中数据发生变化时能够及时通知View，使View呈现最新的数据。

**2 、Android MVVM 模式**

MVVM在不同的平台实现方式是有一定差异性的。在Google IO 2017 ，Google发布了一个官方应用架构库[Architecture Components](https://developer.android.google.cn/topic/libraries/architecture/guide.html)，这个架构库便是Google对Android应用架构的建议，也被称之为Android官方应用架构指南。Android Architecture Components在Google中国开发者网站中能找到。和[Data Binding Library](https://developer.android.google.cn/topic/libraries/data-binding/index.html)一样官方还没翻译为[中文](http://www.cnblogs.com/zqlxtt/p/6895717.html)。

下图是Architecture的应用架构图。结合Android程序特点，整体上与微软的MVVM类似，但是做了更细致的模块划分。



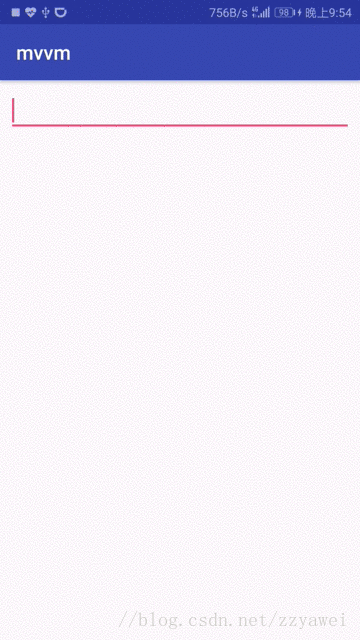
* **View**   
  显而易见 Activity/Fragment 便是MVVM中的View，当收到ViewModel传递来的数据时，Activity/Fragment负责将数据以你喜欢的方式显示出来。实际是View成还包括ViewDataBinding(根据xml自动生成)，上面中并没有体现。
* **ViewModel**   
  ViewModel作为Activity/Fragment与其他组件的连接器。负责转换和聚合Model中返回的数据，使这些数据易于显示，并把这些数据改变及时的通知给Activity/Fragment。   
  [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel.html)是具有生命周期意识的，当Activity/Fragment销毁时ViewModel的onClear方法会被回调，你可以在这里做一些清理工作。   
  [LiveData](https://developer.android.com/topic/libraries/architecture/livedata.html)是具有生命周期意识的一个可观察的的数据持有者，ViewModel中的数据由LiveData持有，并且只有当Activity/Fragment处于活动时才会通知UI数据的改变，避免无用的刷新UI；
* **Model**   
  Repository及其下方就是Model了。Repository负责提取和处理数据。数据可以来自本地数据库[(Room)](https://developer.android.com/topic/libraries/architecture/room.html)，也可以来自网络，这些数据统一有Repository处理，对应隐藏数据来源及获取方式
* **Binder 绑定器**   
  上图中并没有标出绑定器在哪里，其实在任何MVVM的实现中，数据绑定技术都是必须的。而上图仅仅是应用架构图。   
  Android中的数据绑定技术由 [DataBinding](https://developer.android.com/topic/libraries/data-binding/index.html)和[LiveData](https://developer.android.com/topic/libraries/architecture/livedata.html)共同实现。当Activity/Fragment接收到来自ViewModel中的新数据时(由LiveData自动通知数据的改变)，将这些数据通过DataBinding绑定到ViewDataBinding中，UI将会自动刷新，而不用书写类似setText的方法。

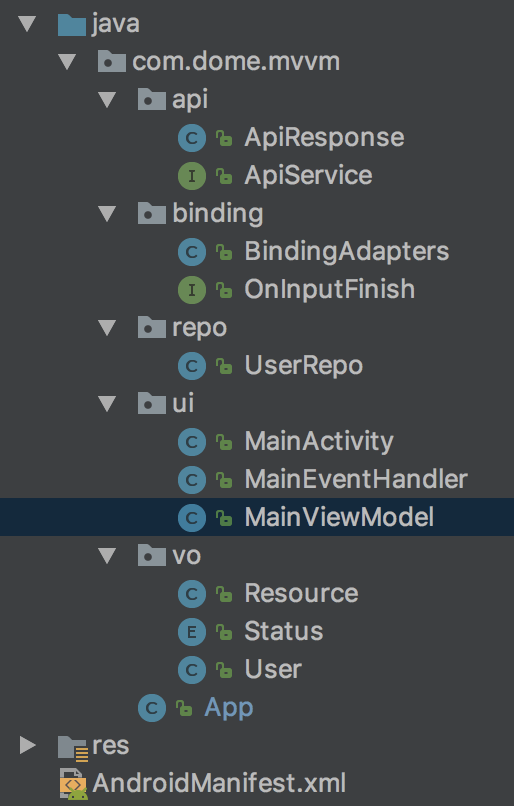
**3、Android MVVM 实战**

上面都是一些理论，下面开始的按照Android Architecture Components写一个的MVVM Demo。这个Dome会加入DataBinding、ViewModel、LiveData、retrofit并且使用java8。不准备添加Room(数据库)和Dagger2(依赖注入)。

**现在我们来写这个Dome**

我们将在这个Dome里面通过Github用户的用户名，来获取具体的用户信息详情。其实Github返回很多，我们这里为了方便只显示用昵称，头像，公开库数量，最后修改时间。

**效果图：**   


**项目结构：**   


**依赖：**

首先，Android Studio 3.0 是必须的。然后添加依赖..

android {

...

*//添加DataBinding支持*

dataBinding {

enabled = true

}

*//添加java8支持*

compileOptions {

sourceCompatibility JavaVersion.VERSION\_1\_8

targetCompatibility JavaVersion.VERSION\_1\_8

}

}

dependencies {

...

*//LiveData，ViewModel*

implementation "android.arch.lifecycle:extensions:1.1.0"

implementation "android.arch.lifecycle:common-java8:1.1.0"

*//网络请求*

implementation "com.squareup.retrofit2:retrofit:2.3.0"

implementation "com.squareup.retrofit2:converter-gson:2.3.0"

*//图片加载*

implementation "com.github.bumptech.glide:glide:3.7.0"

...

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

**XML:**

*<!--为了方便，删掉了xml中一些不重要的属性，仅保留了DataBinding相关的属性。-->*

<?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">

<data>

*<!--导包，类似java导包。下面要用到这个枚举进行判断-->*

<import type="com.dome.mvvm.vo.Status" />

*<!--事件处理-->*

<variable

name="eventHandler"

type="com.dome.mvvm.ui.MainEventHandler" />

<variable

name="user"

type="com.dome.mvvm.vo.User" />

*<!--当前加载状态，上面导包了，这里就不用写全包名了-->*

<variable

name="loadStatus"

type="Status" />

<variable

name="resource"

type="com.dome.mvvm.vo.Resource" />

</data>

<LinearLayout>

*<!--app:onInputFinish,这个是自定义的接口，当输入完成后回调eventHandler.onTextSubmit(text)。-->*

*<!--BindingAdapter相关知识-->*

<android.support.v7.widget.AppCompatEditText

android:imeOptions="actionDone"

android:inputType="text"

android:lines="1"

app:onInputFinish="@{(text)->eventHandler.onTextSubmit(text)}" />

*<!--visibleGone,自定义的BindingAdapter，处理View的显示和隐藏-->*

*<!--当loadStatus为SUCCESS时显示此LinearLayout，绑定具体的用户信息-->*

<LinearLayout visibleGone="@{loadStatus==Status.SUCCESS}">

*<!--imgUrl,自定义的BindingAdapter，绑定ImageView的url，由Glide处理-->*

<ImageView app:imgUrl="@{user.avatarUrl}" />

*<!--@string,引用字符串，格式化user.name-->*

<TextView android:text="@{@string/format\_name(user.name)}" />

<TextView android:text="@{@string/format\_repo(user.repoNumber)}" />

<TextView android:text="@{@string/format\_time(user.lastUpdate)}" />

</LinearLayout>

*<!--当loadStatus为ERROR时显示此View，text绑定错误信息-->*

<TextView

visibleGone="@{loadStatus==Status.ERROR}"

android:text="@{resource.message}" />

*<!--当loadStatus为LOADING时显示此View，表示正在请求-->*

<ProgressBar

style="?android:attr/progressBarStyleHorizontal"

visibleGone="@{loadStatus==Status.LOADING}"

android:indeterminate="true" />

</LinearLayout>

</layout>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57

**可以看到View的显示逻辑完全由数据驱动**。 Activity只需要把相关的数据对象绑定到xml中，Data Binding 会自动把这些数据显示到相关的View。

事实上，Databinding会根据当前xml自动生成一个ViewDataBinding的**.java**文件。上面写的有关属性与绑定都会在这个ViewDataBinding中实现。生成的ViewDataBinding在/app/build/generated/source/apt/debug/\*包名\*/databinding/目录下，感兴趣可以看看。如果你对The mvp这个框架有了解的话，就会发现它和DataBinding的相似处，都是把View的显示逻辑放到Activity之外。接下来我们看MainEventHander.java:

**MainEventHander**

public class MainEventHandler {

private MainActivity mainActivity;

MainEventHandler(MainActivity mainActivity) {

this.mainActivity = mainActivity;

}

*/\**

*\* 这个方法由xml中的app:onInputFinish="@{(text)->eventHandler.onTextSubmit(text)}"调用。*

*\*/*

public void onTextSubmit(String text) {

mainActivity.onSearchUser(text);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

这个java文件并不是必须的，你可以把点击事件直接放到Activity中去。之所以这样写，是不想让Activity去处理复杂的点击事件，简化Activity。

**MainActivity**

public class MainActivity extends AppCompatActivity {

*//自动生成的ViewDataBinding ，类名是根据xml名称自动生成*

private ActivityMainBinding mainBinding;

*//ViewModel*

private MainViewModel mainViewModel;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

*// 替换setContentView()*

mainBinding = DataBindingUtil.setContentView(this, R.layout.activity\_main);

*// 注意：这里不可以直接new MainViewModel()*

mainViewModel = ViewModelProviders.of(this).get(MainViewModel.class);

*//设置事件处理器*

mainBinding.setEventHandler(new MainEventHandler(this));

*//获取userLiveData*

LiveData<Resource<User>> userLiveData = mainViewModel.getUser();

*//观察userLivedata中的数据(User)变化*

userLiveData.observe(this, userResource -> {

*//绑定到DataBinding,set\*\*()方法根据xml中的<var.. >标签自动生成.*

mainBinding.setLoadStatus(userResource == null ? null : userResource.status);

mainBinding.setUser(userResource == null ? null : userResource.data);

mainBinding.setResource(userResource);

});

}

*//eventHander调用这个*

void onSearchUser(String text) {

*//通知ViewModel*

mainViewModel.setUserName(text);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31

**Activity**没有通过自身去**获取数据**，当数据返回时Activity也没有去**处理数据**，也没有处理简单**显示逻辑**，也没有处理**点击事件**（*监听软件盘的输入完成+获取输入文字,在这里已经变成了onSearchUser*）。这样Activity就被大大简化，没有动辄几百行的代码。

**Activity的职责是:在数据更改时更新视图，或将用户操作通知给ViewModel**；

* **为什么不可以new MainViewModel ?**

前面有说过ViewModel是具有生命周期意识的，但这并不是与生俱来的。直接new会让ViewModel的失去对生命周期的感知。   
上述方式实际上是通过反射生成MainViewModel.class的对象，然后创建一个没有视图的Fragment添加到Activity，把这个viewModel对象交由Fragment持有，因为Fragment和Activity的生命周期是同步的，所以当Activity销毁时ViewModel的onClear()会被回调并且销毁这个ViewModel。   
上述写法使用的是默认的创建工厂(反射方式创建)。我们可以使用自定义的工厂来创建对象，我们可以在工厂里传入参数（一般都需要传参，这个简单而已）。而当我们使用了依赖注入（如dagger2）后，就不需要传参了。

* **为什么userLiveData不用removeObserve ?**

和ViewModel一样，LiveData也能感知Activity的生命周期。当Activity销毁时，LiveData会自动的remove调，不用我们担心。

**MainViewModel**

public class MainViewModel extends ViewModel {

private final UserRepo userRepo = UserRepo.getInstance();

private final MutableLiveData<String> userNameLiveData = new MutableLiveData<>();

private final LiveData<Resource<User>> userEntityLiveData;

public MainViewModel() {

*//switchMap:当userNameLiveData中的数据发生变化时 触发input事件，*

userEntityLiveData = Transformations.switchMap(userNameLiveData, input -> {

if (input == null) {

return new MutableLiveData<>();

} else {

*//如果收到新的input（userName），那么就去UserRepo获取这个用户的信息*

*//返回值将赋值给userEntityLiveData；*

return userRepo.getUser(input);

}

});

}

public LiveData<Resource<User>> getUser() {

return userEntityLiveData;

}

public void setUserName(String userName) {

*//将userName设置给userNameLiveData*

userNameLiveData.postValue(userName);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

首先，ViewModel没有持有Activity对象或View对象，也必须不能持有这些对象。   
其次，ViewModel不负责提取数据（如网络请求）。   
而且，ViewModel不依赖特定的View。他对所有引用它的对象提供相同的数据支持，也是是说同一个数据来源，我们可以有不同的展现方式。

**ViewModel的职责是:1.处理数据逻辑，但是却不获取数据。2.作为Activity/Fragment 和其他组件之间的连接器**；

**Repo**

public class UserRepo {

private static UserRepo userRepo = new UserRepo();

public static UserRepo getInstance() {

return userRepo;

}

public LiveData<Resource<User>> getUser(String userId) {

MutableLiveData<Resource<User>> userEntityLiveData = new MutableLiveData<>();

userEntityLiveData.postValue(Resource.loading(null));

*//请求网络*

ApiService.INSTANCE.getUser(userId).enqueue(new Callback<User>() {

@Override

public void onResponse(Call<User> call, Response<User> response) {

ApiResponse<User> apiResponse = new ApiResponse<>(response);

if (apiResponse.isSuccessful()) {

userEntityLiveData.postValue(Resource.success(response.body()));

} else {

userEntityLiveData.postValue(Resource.error(apiResponse.errorMessage, null));

}

}

@Override

public void onFailure(Call<User> call, Throwable t) {

userEntityLiveData.postValue(Resource.error(t.getMessage(), null));

}

});

return userEntityLiveData;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28

虽然repo模块看上去没有必要，但他起着重要的作用。它为App的其他部分抽象出了数据源。现在我们的ViewModel并不知道数据是通过WebService来获取的，这意味着我们可以随意替换掉获取数据的实现。

**ApiService**

public interface ApiService {

ApiService INSTANCE = new Retrofit.Builder()

.baseUrl("https://api.github.com/")

.addConverterFactory(GsonConverterFactory.create())

.build()

.create(ApiService.class)*;*

@GET("users/{login}")

Call<User> getUser(@Path("login") String login)*;*

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

超级简单的写法..   
这里我们获取网络请求返回的是Call<User>对象，其实我们可以自定义一个转化器使retrofit直接返回给我们LiveData<?>对象。这个并不是mvvm的重点，所以这个dome里并没有这么做。

**BindingAdapters**

public class BindingAdapters {

@BindingAdapter("visibleGone")

public static void showHide(View view, boolean show) {

view.setVisibility(show ? View.VISIBLE : View.GONE);

}

@BindingAdapter("imgUrl")

public static void imgUrl(ImageView view, final String url) {

Glide.with(view.getContext()).load(url).into(view);

}

@BindingAdapter("onInputFinish")

public static void onInputFinish(TextView view, final OnInputFinish listener) {

if (listener == null) {

view.setOnEditorActionListener(null);

} else {

view.setOnEditorActionListener((v, actionId, event) -> {

if (actionId == EditorInfo.IME\_ACTION\_DONE) {

listener.onInputFinish(v.getText().toString());

}

return false;

});

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

上面xml里面所使用的app:visibleGone / app:imgUrl / app:onInputFinish属性都是这里定义的。前面两个很好理解，如果对onInputFinish的参数理解不了，可以了解了java8 lambda表达式相关知识。

4、最后

Dome 地址

Dome Github 地址：https://github.com/zyawei/DomeMvvm

Dome With Dagger2 :还没写..

Google Architecture Sample : https://github.com/googlesamples/android-architecture-components

参考链接:

The MVVM Pattern : https://msdn.microsoft.com/en-us/library/hh848246.aspx

Implementing The MVVM Pattern : https://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx

Android Architecture Components : https://developer.android.com/topic/libraries/architecture/index.html

Android官方应用架构指南(中文) : http://www.cnblogs.com/zqlxtt/p/6895717.html

---------------------

作者：zzyawei

来源：CSDN

原文：https://blog.csdn.net/zzyawei/article/details/79590453

版权声明：本文为博主原创文章，转载请附上博文链接！