浅谈MVC,MVP,MVVM架构模式

为什么要引用架构?

规范代码，减少代码的复用，解耦合，将一个复杂的工程问题，分解成很多小的问题，分而治之。做到模块内部的高聚合和模块之间的低耦合（面向对象原则）

1.MVC架构模式

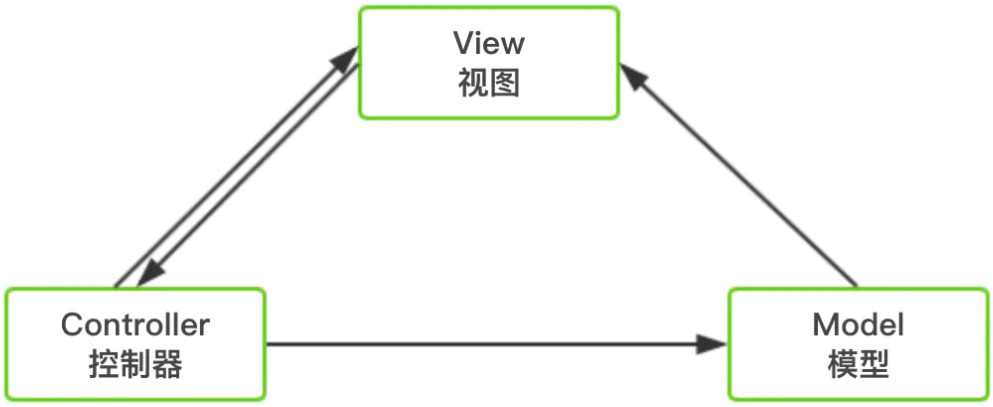
MVC分别是Model（模型），View（视图）、Controller（控制器）三个模块。

View（视图层）最主要完成前端的数据展示，

Controller（控制层）是对数据的接收和触发事件的接收和传递，

Model（模型层）则是对数据的储存和处理，再传递给视图层相应或者展示。

> 例如，现在有一个view需要设置点击事件。视图层首先需要这个view，然后再控制层处理点击逻辑，再把具体点击后做是什么交付给模型层去处理，然后把处理结果返回，只是Android不太方便直接从model传到view，只有通过控制层。



在安卓开发环境下，最主要是通过Controller层将数据返回给View层。基本的业务逻辑写在Controller层上，导致Controller层的臃肿复杂。

在Android的MVC中，View大多数情况是xml文件，我们通常对View所做的操作逻辑都是写在Activity或者Fragment中，比如点击事件，长按事件等，这样请求事件发送到Controller中，比如点击事件是下载，那么Controller就会将事件转发到Model层去请求数据，Model获取数据后就会通过消息订阅发布机制或者接口去更新View，更新View的操作也是在Activity或者Fragment中操作，这样一个闭环就形成了，View -> Controller -> Model - > View。

但是这里的View可以是自定义View，就是一个Java类的文件，这个类里是可以直接进行访问Model操作获取数据的，就是绕开了Controller，也是一个闭环，View -> Model - > View

Demo展示

这里有人可能会说为什么要定义接口做呢，这么麻烦，干脆写个公共的网络操作类，提供一个getUserMsg方法调用就行了呀！

假如这里网络操作是用原生HttpUrlConnection开发的，然后哪天需要改成OKHttp，难道去修改getUserMsg方法；就算改完后，假如哪天又改成了Retrofit去实现网络操作，难道又去修改getUserMsg方法；这样一连串的迭代，方法被改了多少遍，原来的代码不复存在，假如哪天需要用到以前的，那又得重新编写

可能有的朋友说，那就写多个getUserMsg方法，面临的问题就是一个类里写了太多这种同一种功能方法，繁杂，对以后的维护不是一个好消息

所以这里就采用面向接口的开发，不管以后的迭代中改成什么需求；比如使用OKHttp请求，那就再定义一个类UserModelWithOkHttpImpl去实现上面的UserModel接口，**因为接口是高度抽象的，只定义功能，具体实现因人而定；这样操作以后，不仅保留了以前的代码，而且实现类简洁易维护**

- XML是属于View层，对它的操作放在了Activity
- 然而Activity作为Controller层，持有UserModel引用（UserModel 作为Model层），接着点击按钮请求用户数据，这样Controller层就向Model层发起了请求
- 然后Model层进行网络操作，因为可能耗时，所以在真正执行网络前通过回调，通知View层显示加载的dialog，然后获取数据通过回调返回给View层显示出来

优点

代码耦合性低，减小模块之间的相互影响；比如同一个数据模型，你可以用柱形图来展示，也可以用圆形图展示，也就是说修改View层的显示效果不用修改数据

可扩展性好，模块之间影响小，加上面向接口的开发，当你新增一个功能或者新增一种功能实现的时候，只需要定义接口和实现接口，就不需要修改以前的代码

模块职责划分明确，如上面所说，每个模块做自己的事代码维护性好，修改模型不会影响到视图，反过来，修改视图，也不会影响到模型

缺点

增加了系统结构和实现的复杂性，对于简单的界面，严格遵循MVC，使模型、视图与控制器分离，会增加结构的复杂性，并可能产生过多的更新操作，降低运行效率
视图与控制器间的过于紧密的连接，也就是耦合性高，视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用

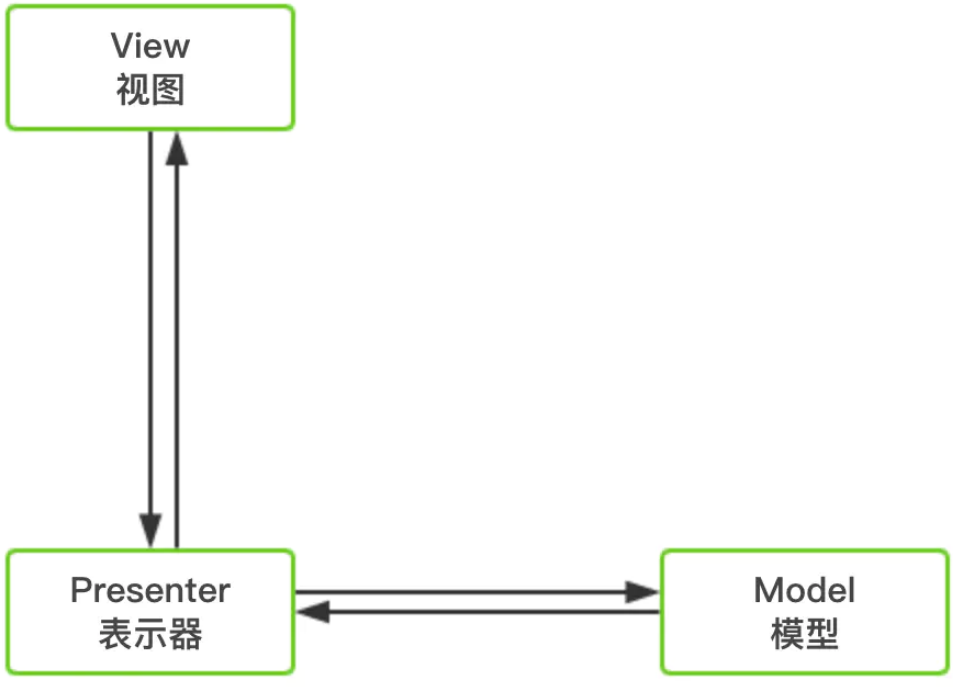
2.MVP架构模式

MVC的架构模式中，Model层未和

- Model：这里跟MVC中的Model基本一样
- View：这里就跟MVC中的View不太一样了，不仅仅是XML文件，还包括Activity，Fragment，自定义View等java文件，这样**Activity等就可以只负责View的绘制，显示，响应用户的操作**
- Presenter：作为View与Model之间联系的纽带，**让View和Model解耦**；同时它与View层是**通过接口通信**，这又与View层解耦；将原来Activity属于Controller的操作移到了Presenter中

与MCV的区别

- MVP中**View与Model并不直接交互**，而是通过与Presenter交互来与Model间接交互。而在**MVC中View可以与Model直接交互**
- 通常View与Presenter是一对一的，但**复杂的View可能绑定多个Presenter来处理逻辑。**而**Controller是基于行为的，并且可以被多个View共享，Controller可以负责决定显示哪个View**
- Presenter与View的交互是通过接口来进行的，更有利于添加单元测试；而MVC中**Activity包含了View和Controller代码，很难做测试**



在Android中的MVP，XML，Activity，Fragment等都是属于View层，不再执行具体的逻辑，只是纯粹的操作View；具体逻辑交给Presenter处理，View与Presenter通过接口通信，达到解耦；Presenter收到View层请求后转发给相应的Model层，Model层处理完数据后将数据返回给Presenter，然后Presenter再通知View层更新View

MVP的核心思想就是一切皆为接口，把Activity中的UI逻辑抽象成View接口，把业务逻辑抽象成Presenter接口，把数据操作抽象成Model接口

Demo展示

没了View，Presenter依然可以工作，在单元测试的时候，比如想测试Presenter中的业务逻辑是否正确，只需要写个脚本模拟Activity，然后在相应的方法中提供数据，最后调用Presenter接口的login方法，看能否得到预期结果，而且两者完成解耦；而Presenter只专注于业务逻辑，至于页面是什么样的，不关心，所以同一个Presenter可以对应多个View；

优点

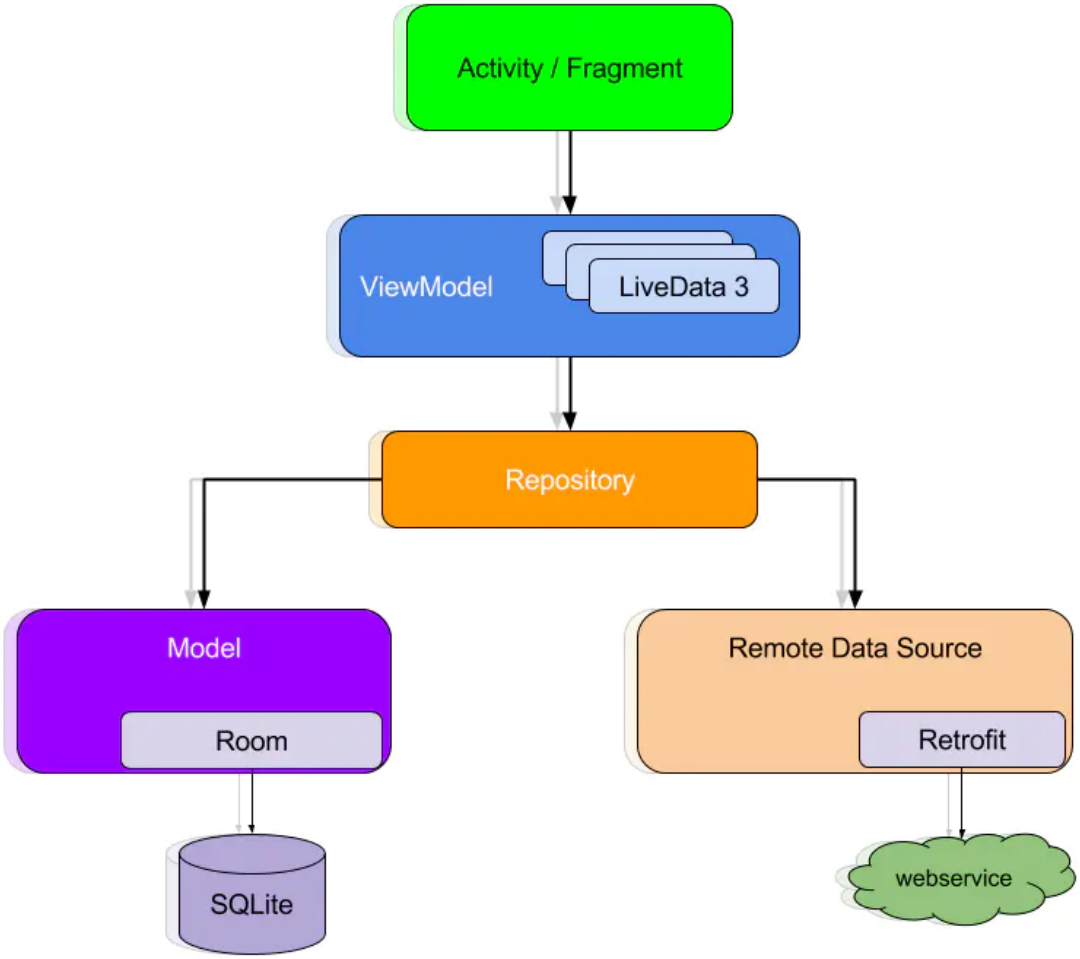
- 几乎所有的思想都是为了解耦，更加利于维护和开发，把大工程化整为零，每个团队负责一小部分，相互独立
- 从上面的样例也可以看出，这种模式下，Activity等View层类的代码相当简洁，基本上只有对View的操作；还有想了解一个模块有哪些业务，就看这个模块的Presenter接口就行了，不管是定位代码还是后续修改业务都很方便
- 由于面向接口的开发，非常便于单元测试代码复用程度提高，一个Presenter可以适用于多个View

缺点

- 从上面的例子和工程目录图明显可以看出来，使用MVP后，类的数量能增加一倍以上，也给维护工作增加了难度，这应该是最大的一个缺点了
- 维护接口的难度并不低，特别是业务逻辑越来越复杂以后，维护工作更难
  Presenter有很多连接View与Model之间的操作，造成Presenter的臃肿
- **Presenter和View通过接口通信太繁琐**,一旦View层需要的数据变化,那么对应的接口就需要更改

MVVM架构

谈到MVVM架构，必然逃不出两个东西（**dataBinding和观察者模式**）



Databinding

DataBinding是谷歌官方推出的一个库，DataBinding库来写声明的layouts文件，可以用最少的代码来绑定你的app逻辑和layouts文件。（我个人理解就是不用通过回调去更新UI，而Ui直接和数据绑定在一起，直接通过改变数据的值，ui也就自动更新了。）

Databinding基本用法：

在app.gradle下，需要添加Data Binding到gradle构建文件里如下：

```
android {
    ....
    dataBinding {
        enabled = true
    }
}
```

在布局中就可以这样写：

```
<?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"
    >
```

```
    <data>
        <variable name="user" type="demo.com.databindingdemo.User"/>
    </data>

    <android.support.constraint.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">


        <TextView
            android:id="@+id/user_name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.mUserName}"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/user_age"
            android:text='@{user.mUserage+""}'
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toBottomOf="@id/user_name" />
    </android.support.constraint.ConstraintLayout>
</layout>
```

在data内描述了一个名为user的变量属性，使其可以在这个layout中使用

```
<variable name="user" type="demo.com.databindingdemo.User"/>
```

在layout的属性表达式写作@{}，下面是一个TextView的text设置为user的mUserName属性：

```
android:text="@{user.mUserName}"
```

2)Data对象

```
public class User {

    public final String mUserName;
    public final int mUserage;

    public User(String userName, int userAge) {
        this.mUserName = userName;
        mUserage = userAge;
    }


}
```

3）Binding数据

默认情况下，一个Binding类会基于layout文件的名称而产生,上述的layout文件是activity_main.xml，因此生成的类名是ActivityMainBinding

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityMainBinding binding =
 DataBindingUtil.setContentView(this, R.layout.activity_main);
        User user = new User("sam",11);
        binding.setUser(user);
    }
}
```

这就是Databinding的最基本的用法，剩下的一些深入的用法例如：

1.import 2.(alias)当类名有冲突时，其中一个类名可以重命名 3.导入的类型还可以在表达式中使用static属性和方法（一般可以用于给button设置监听事件啥的）.下面我可以通过一个例子来梳理一遍Databinding的流程。

1.View层就是展示数据的，以及接收到用户的操作传递给viewModel层，通过dataBinding实现数据与view的单向绑定或双向绑定

2.Model层最重要的作用就是获取数据了，当然不止于此，model层将结果通过接口的形式传递给viewModel层

3.ViewModel 层通过调用model层获取数据，以及业务逻辑的处理。

LoginActivity

```
public class LoginActivity extends AppCompatActivity {

    LoginActivityBinding binding;
    LoginViewModel viewModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = DataBindingUtil.setContentView(this,R.layout.login_activity);
        viewModel = new LoginViewModel();

        binding.loginButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                login();
            }
        });

    }
```

```
    private void login() {

        if (TextUtils.isEmpty(binding.nameEdit.getEditableText().toString())) {
            Toast.makeText(getApplication(), "请输入账号！",
Toast.LENGTH_SHORT).show();
            return;
        }

        if (TextUtils.isEmpty(binding.passEdit.getEditableText().toString())) {
            Toast.makeText(getApplication(), "请输入密码！",
Toast.LENGTH_SHORT).show();
            return;
        }


 viewModel.login(binding.nameEdit.getEditableText().toString(),binding.passEdit.
getEditableText().toString());
    }
```

LoginViewModel

```
public class LoginViewModel extends ViewModel {

    private static final String TAG = "LoginViewModel";
    private LoginModel model;
    public LoginViewModel() {
        model = new LoginModel();
    }

    public void login(String name, String pass) {
        model.appLogin(name, pass, new INetWorkCallback<LoginResult>() {
            @Override
            public void onCallApiSuccess(LoginResult loginResult) {

                android.util.Log.e(TAG,"---onCallApiSuccess---");
            }

            @Override
            public void onCallApiFailure(Throwable throwable) {
                android.util.Log.e(TAG,"---onCallApiFailure---");
            }

            @Override
            public void onCompleted() {
                android.util.Log.e(TAG,"---onCompleted---");
            }
        });
    }
}
```

LoginModel

```
public class LoginModel {
```

```
    public void appLogin(final String loginName, final String pass, final
INetWorkCallback<LoginResult> callback) {

        final RequestBodyEntity.Login loginRequest = new
RequestBodyEntity.Login();
        loginRequest.loginName = loginName;
        loginRequest.loginPass = pass;
        final Gson gson = new Gson();
        String gsonStr = gson.toJson(loginRequest);
        RequestBody body = RequestBody.create(MediaType.parse("application/json;
charset=utf-8"), gsonStr);

        RetrofitUtil.getInstance().getRetrofitService()
                .appLogin(body)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new DisposableObserver<LoginResult>() {
                    @Override
                    public void onNext(LoginResult loginResult) {
                        callback.onCallApiSuccess(loginResult);
                    }

                    @Override
                    public void onError(Throwable e) {
                        callback.onCallApiFailure(e);
                    }

                    @Override
                    public void onComplete() {
                        callback.onCompleted();
                    }
                });
    }
}
```