Android 中的 MVC

Will Also

2019

目录

1 MVC

1.1 MVC 来源

MVC 模式（Model-view-controller）是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

MVC 模式最早由 Trygve Reenskaug 在 1978 年提出，是施乐帕罗奥多研究中心（Xerox PARC）在 20 世纪 80 年代为程序语言 Smalltalk 发明的一种软件架构。MVC 模式的目的是实现一种动态的程序设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外，此模式透过对复杂度的简化，使程序结构更加直观。软件系统透过对自身基本部分分离的同时也赋予了各个基本部分应有的功能。

1.2 组件

- 模型（Model）用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法。“Model”有对数据直接访问的权力，例如对数据库的访问。“Model”不依赖“View”和“Controller”，也就是说，Model 不关心它会被如何显示或是如何被操作。但是 Model 中数据的变化一般会通过一种刷新机制被公布（观察者模式）。

- 视图（View）能够实现数据有目的的显示。在 View 中一般没有程序上的逻辑。为了实现 View 上的刷新功能，View 需要访问它监视的数据模型（Model），因此应该事先在被它监视的数据那里注册。

- 控制器（Controller）起到不同层面间的组织作用，用于控制应用程序的流程。它处理事件并作出响应。“事件”包括用户的行为和数据 Model 上的改变。

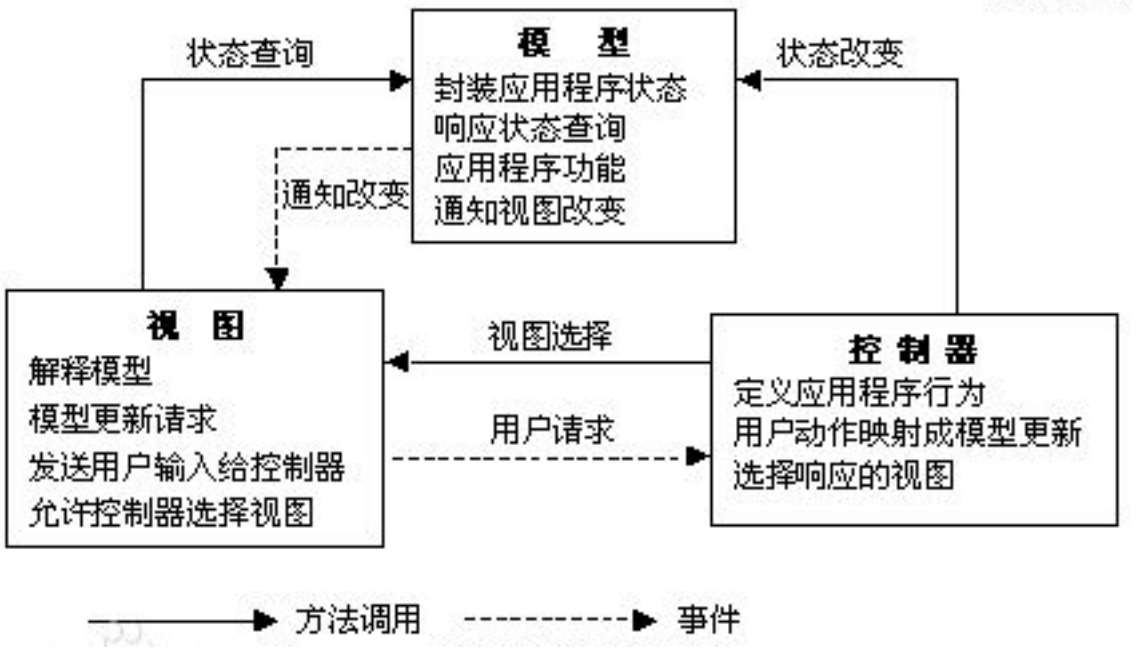

图 1: MVC 组件和行为

1.3 优点

- 耦合性低：视图层和业务层分离，这样就允许更改视图层代码而不用重新编译模型和控制器代码，同样，一个应用的业务流程或者业务规则的改变只需要改动 MVC 的模型层即可。
- 重用性高：MVC 模式允许使用各种不同样式的视图来访问同一个服务器端的代码，因为多个视图能共享一个模型，它包括任何 WEB（HTTP）浏览器或者无线浏览器（wap），由于模型返回的数据没有进行格式化，所以同样的构件能被不同的界面使用。
- 生命周期成本低：MVC 使开发和维护用户接口的技术含量降低。
- 部署快：使用 MVC 模式使开发时间得到相当大的缩减，它使程序员（Java 开发人员）集中精力于业务逻辑，界面程序员（HTML 和 JSP 开发人员）集中精力于表现形式上。
- 可维护性高：分离视图层和业务逻辑层也使得 WEB 应用更易于维护和修改。Controller 提高了应用程序的灵活性和可配置性。
- 有利于软件工程化管理：由于不同的层各司其职，每一层不同的应用具有某些相同的特征，有利于通过工程化、工具化管理程序代码。

1.4 缺点

- 没有明确的定义：使用 MVC 需要精心的计划，由于它的内部原理比较复杂，模型和视图要严格的分离，这样也给调试应用程序带来了一定的困难。每个构件在使用之前都需要经过彻底的测试。
- 不适合小型，中等规模的应用程序，增加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：视图与控制器是相互分离，但却是联系紧密的部件，。
- 视图对模型数据的低效率访问：依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。
- 一般高级的界面工具或构造器不支持模式：改造这些工具以适应 MVC 需要和建立分离的部件的代价是很高的，会造成 MVC 使用的困难。

1.5 实现

1.5.1 Java

- 视图（View）可能由 Java Server Page(JSP) 担任。生成 View 的代码则可能是一个 servlet 的一部分，特别是在客户端服务端交互的时候。
- Controller 可能是一个 servlet。
- Model 则是由一个实体 Bean 来实现。

1.5.2 JavaScript

```
1 var M = {}, V = {}, C = {};
2 //Model
3 M.data = "hello world";
4 //View
5 V.render = (M) => { alert(M.data); }
6 //Controller
7 C.handleOnload = () => { V.render(M); }
8 window.onload = C.handleOnload;
```

2 Android 中的 MVC

2.1 MVC class

In a world where the user interface logic tends to change more often than the business logic, the desktop and Web developers needed a way of separating user interface functionality.

2.2 MVC 变体

This(Figure 2) means that both the Controller and the View depend on the Model: the Controller to update the data, the View to get the data. But, most important for the desktop and Web devs at that time: the Model was separated and could be tested independently of the UI. Several variants of MVC appeared. The best-known ones are related to whether the Model is passive or is actively notifying that it has changed.

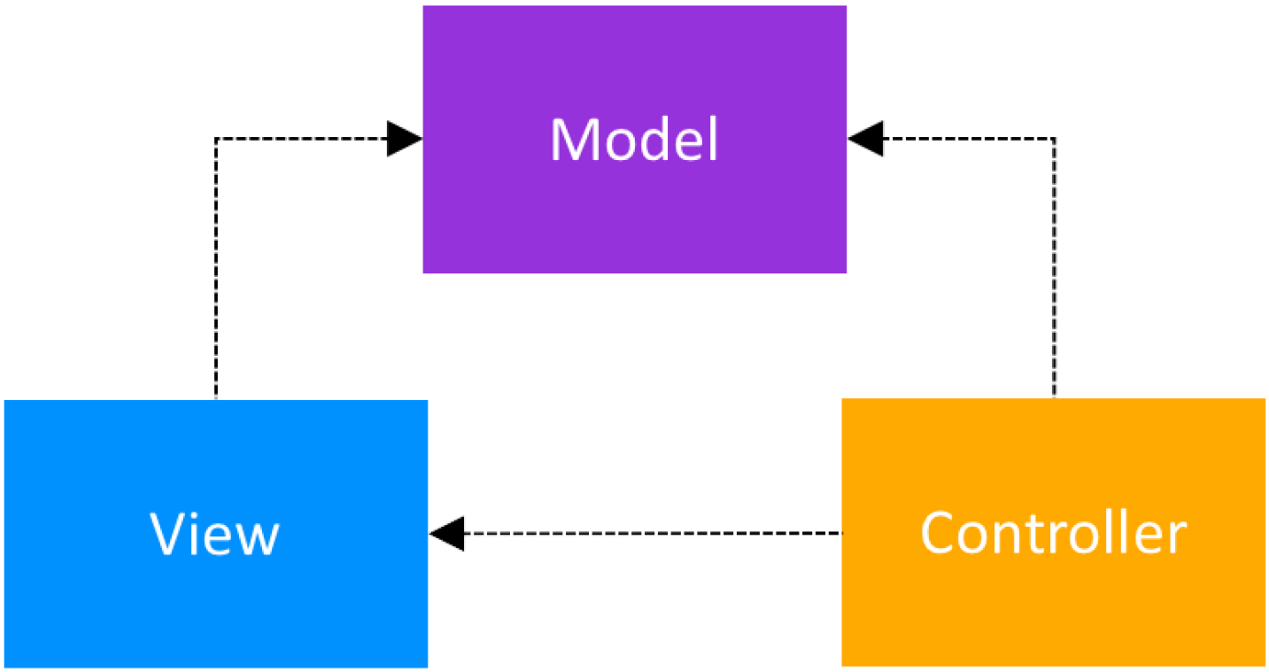

图 2: Model-View-Controller class structure

2.2.1 Passive Model

In the Passive Model version, the Controller is the only class that manipulates the Model. Based on the user’ s actions, the Controller has to modify the Model. After the Model has been updated, the Controller will notify the View that it also needs to update. At that point, the View will request the data from the Model.

2.2.2 Active Model

For the cases when the Controller is not the only class that modifies the Model, the Model needs a way to notify the View, and other classes, about updates. This is achieved with the help of **the Observer pattern**. The Model contains a collection of observers that are interested in updates. The View implements the observer interface and registers as an observer to the Model. Every time the Model updates, it will also iterate through the collection of observers and call the update method. The implementation of this method in the View will then trigger the request of the latest data from the Model.

2.3 应用

The Activities, Fragments and Views should be the Views in the MVC world. The Controllers should be separate classes that don’ t extend or use any Android class, and same for the Models.

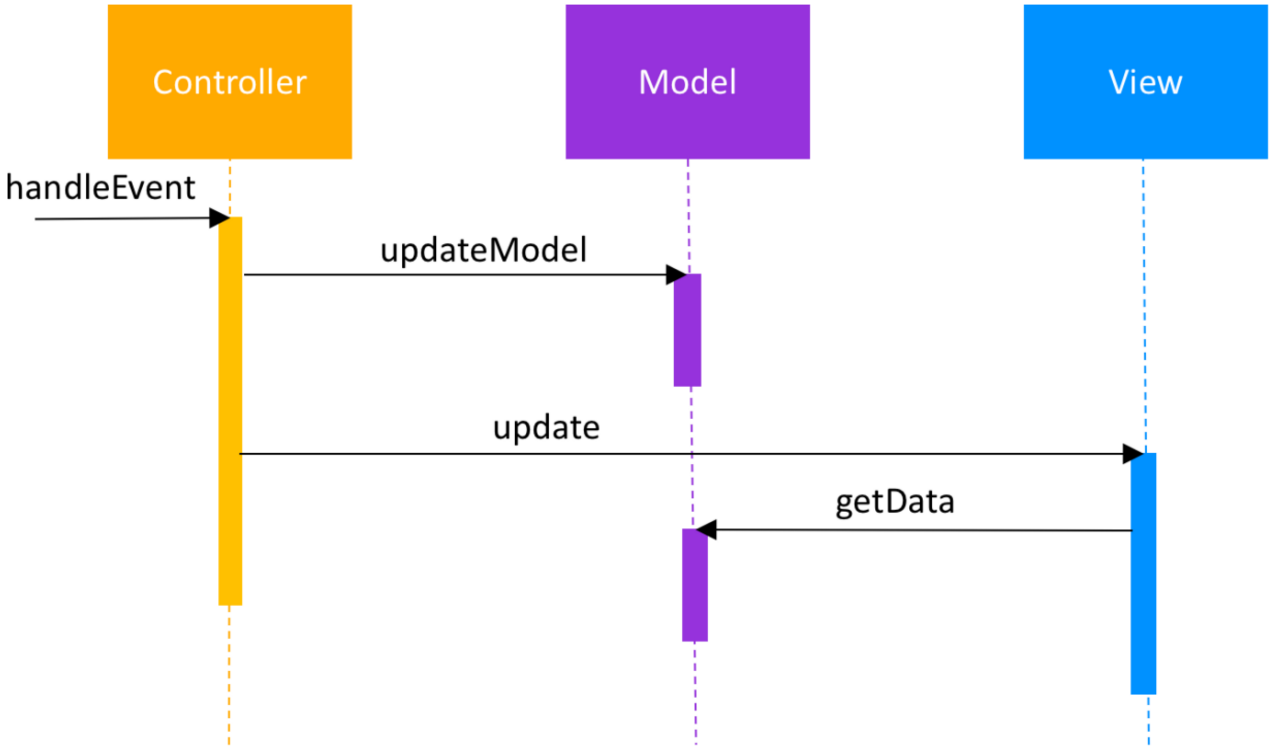

图 3: Model-View-Controller —passive Model —behavior

将 Controller 连接到 View 时会出现一个问题，因为 Controller 需要告知 View 更新。在被动 Model MVC 架构中，Controller 需要保留对 View 的引用。在专注于测试的同时，最简单的方法是拥有一个 BaseView 接口，该接口可以扩展 Activity / Fragment / View。因此，控制器将具有对 BaseView 的引用。

优点：

Model-View-Controller 模式高度支持关注点分离。这一优点不仅提高了代码的可测试性，而且还使扩展变得更容易，从而可以轻松实现新功能。

Model 类没有对 Android 类的任何引用，因此可以直接进行单元测试。Controller 不会扩展或实现任何 Android 类，并且应具有对 View 接口类的引用。这样，也可以对控制器进行单元测试。

If the Views respect **the single responsibility principle** then their role is just to update the Controller for every user event and just display data from the Model, without implementing any business logic. In this case, UI tests should be enough to cover the functionalities of the View.

缺点：

The View Depends On The Controller And On The Model

为了最小化 View 中的逻辑，模型应该能够为要显示的每个元素提供可测试的方法。在主动的 Model 实现中，这将成倍增加类和方法的数量，因为需要每种类型的数据的观察者。

Given that the View depends on both the Controller and the Model, changes in the UI

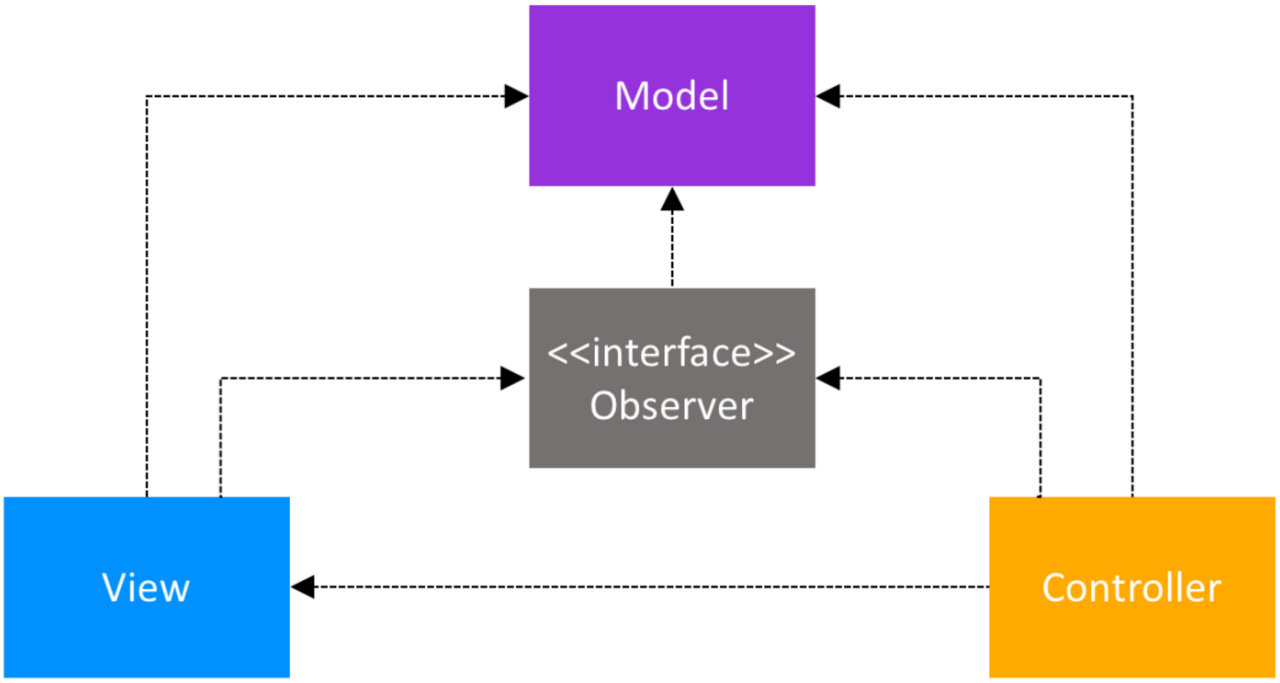

图 4: Model-View-Controller —active Model —class structure

logic might require updates in several classes, decreasing the flexibility of the pattern.

2.3.1 问题：Who Handles The UI Logic?

根据 MVC 模式，控制器更新模型，视图从模型中获取要显示的数据。但是谁决定如何显示数据？是模型还是视图？下面是在 View 中展示”Lastname,Firstname”

如果 Model 只提供原始数据，意味着 View 处理 UI 逻辑，这使得将会很难测试 UI 逻辑，code 如下：

```
1 String firstName = userModel.getFirstName();
2 String lastName = userModel.getLastName();
3 nameTextView.setText(lastName + ", " + firstName)
```

另一种方法是让模型仅公开需要显示的数据，从而从视图中隐藏任何业务逻辑。但是，然后，我们得到处理业务和 UI 逻辑的模型。它可以进行单元测试，但是模型最终隐含地依赖于 View,code 如下：

```
1 String name = userModel.getDisplayName();
2 nameTextView.setText(name);
```

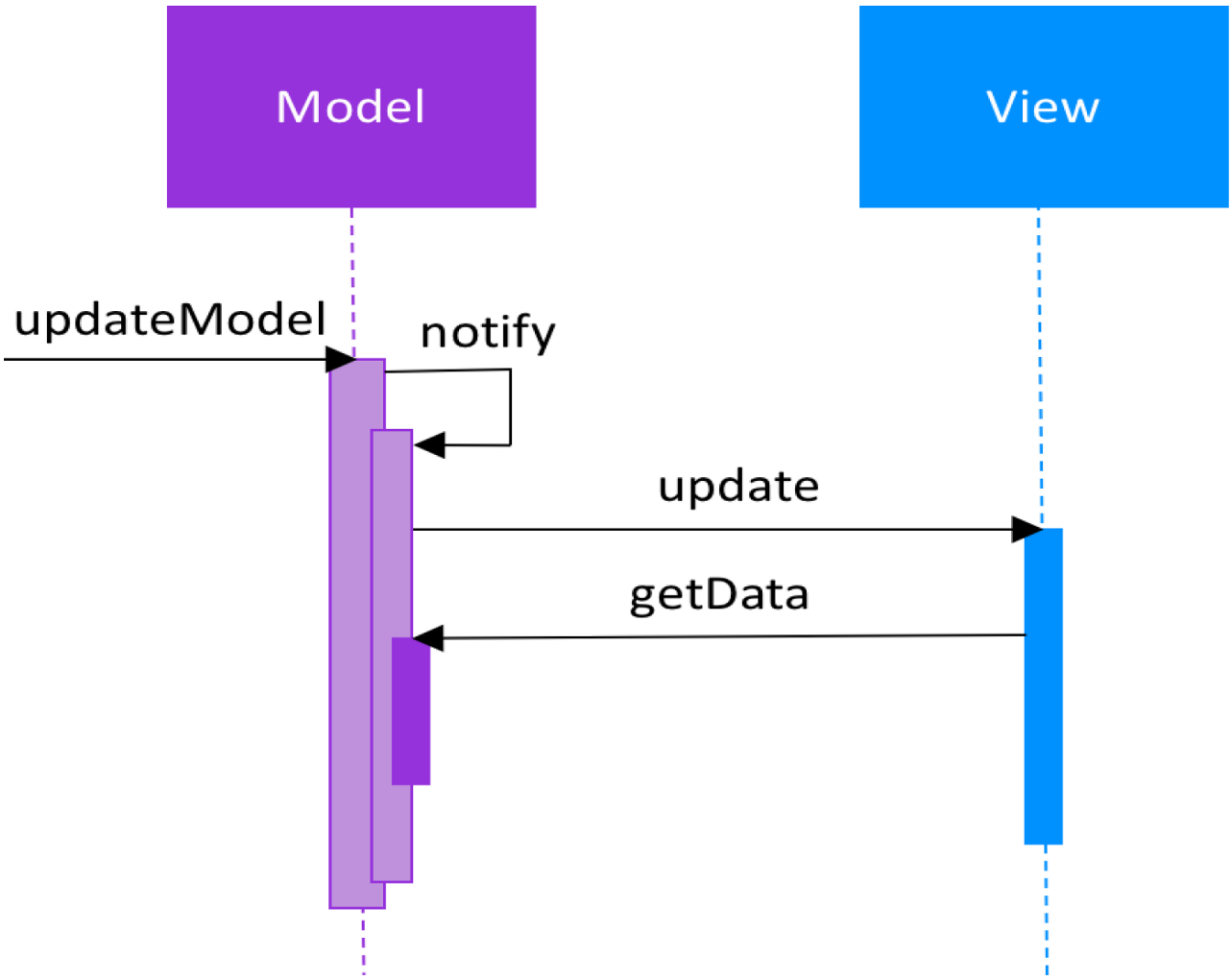

图 5: Model-View-Controller —active Model —behavior

3 代码例子

代码结构：

- Bean:

  User: Java Bean

- Model

  UserModel: Model 接口，提供给 Controller 使用

  UserModelImpl: Model 实现，业务逻辑实现，Http 请求和数据库交互

- MainActivity: 实际的 Controller 层，View 层是 xml 文件，处理用户行为

- MyObserver: MainActivity 的接口（实际为 View 接口），提供给 Model 层实现观察者模式，用来更新数据显示

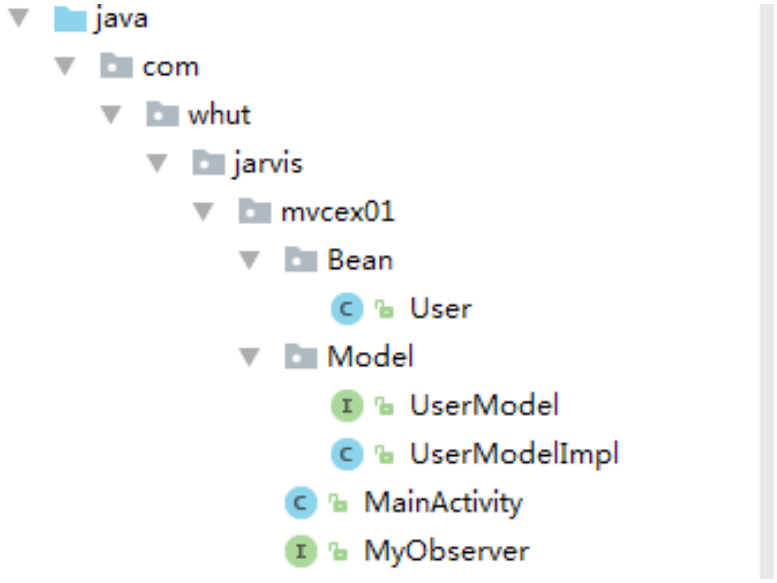

图 6: 代码目录

```
<!--activity_main.xml-->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
    android:id="@+id/text_username"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="点击按钮获取用户名"
    android:textColor="#000"
    android:textSize="30sp"
    android:gravity="center_horizontal"/>
    <Button
    android:id="@+id/btn_getusername"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="获取"/>
</LinearLayout>
```

```
//Bean.User
public class User {
    private String firstname;
    private String lastname;
    //构造器, setter and getter
}
```

```
//Model.UserModel
public interface UserModel {
    void getDisplayName(String userid, MyObserver myObserver);
}

//Model.UserModelImpl
public class UserModelImpl implements UserModel {

    private User user;
    private final String baseurl = "http://yourbaseurl";

    public UserModelImpl() {
        user = new User();
    }

    @Override
    public void getDisplayName(String userid, final MyObserver
       myObserver) {
        OkHttpClient client = new OkHttpClient();
        final Request request = new Request.Builder().url(baseurl).
           method("GET",null).build();
        Call call = client.newCall(request);
        call.enqueue(new Callback() {
        @Override
        public void onFailure(Call call, IOException e) {
            myObserver.onError();
        }
```

```
26
27          @Override
28          public void onResponse(Call call, Response response) throws
                IOException {
29              String displayname = response.body().string();
30              //在Ui线程更新，这里简写
31              myObserver.onSuccess(displayname);
32          }
33          });
34      }
35  }
```

```
1   //MyObserver
2   public interface MyObserver {
3       void onSuccess(String displayname);
4       void onError();
5   }
6
7   //MainActivity
8   public class MainActivity extends AppCompatActivity implements
        MyObserver {
9
10      private Button btn;
11      private TextView textView;
12
13      private UserModel userModel;
14
15      @Override
16      protected void onCreate(Bundle savedInstanceState) {
17          super.onCreate(savedInstanceState);
18          setContentView(R.layout.activity_main);
19
20          btn = findViewById(R.id.btn_getusername);
21          textView = findViewById(R.id.text_username);
22
```

```
23          userModel = new UserModelImpl();
24
25          btn.setOnClickListener(new View.OnClickListener() {
26              @Override
27              public void onClick(View v) {
28                  String userid = "yourid";
29                  userModel.getDisplayName(userid, MainActivity.this)
                        ;
30              }
31          });
32      }
33
34      private void setView(String displayname){
35          textView.setText(displayname);
36      }
37
38      @Override
39      public void onSuccess(String displayname) {
40          setView(displayname);
41      }
42
43      @Override
44      public void onError() {
45          setView("获取信息失败！");
46      }
47  }
```