

一文带你全面了解MVC、MVP、MVVM模式（含实例讲解）



Carson_Ho



已关注



9 2019.08.15 08:32:26 字数 1,548 阅读 6,672



前言

- 在 Android 开发中，当你梳理完需求后，你要做的并不是马上写下你的第一行代码，而是需先设计好整个项目的技术框架
- 今天，我将全面介绍 Android 开发中主流的技术框架 MVC、MVP 与 MVVM 模式，并实例讲解 MVP 模式，希望您们会喜欢。

目录



示意图

1. 为什么要进行技术框架的设计

- 模块化功能
使得程序模块化，即：**内部的高聚合、模块之间的低耦合**
- 提高开发效率
开发人员只需专注于某一点（视图显示、业务逻辑 / 数据处理）
- 提高测试效率
方便后续的测试 & 定位问题

切记：不要为了设计而设计，否则反而会提高开发量



示意图

2. Android开发主流的技术框架

- 主要有 **MVC**、**MVP**、**MVVM** 3种模式
- 下面，我将详细 & 具体的介绍上述3种模式

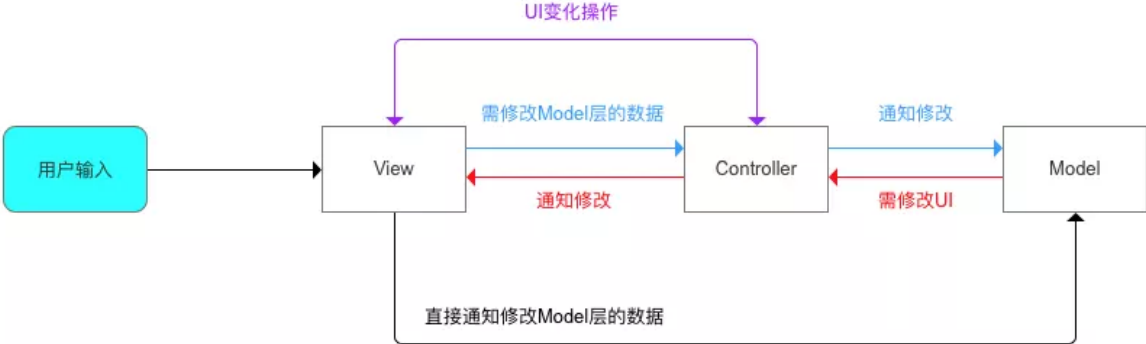
2.1 MVC模式

- 角色说明

类型	定义	作用	使用方式
M (Model)	模型层 (表示 数据模型)	业务逻辑 / 数据存储	<ul style="list-style-type: none">• 写成“外观类(facade class)”• 为一个复杂的子系统提供高层次的简单易用的访问接口

示意图

• 模式说明

模式示意图	
具体描述	<p>View层接收用户的输入：</p> <ol style="list-style-type: none">1. 需UI变化操作（不涉及业务数据调用），即直接与Controller交互即可。（紫色）2. 需修改Model层的数据，即需通过Controller才能修改。（蓝色） （View层也可直接更新Model的数据而不必通过Controller。（黑色））3. Model里数据发生变化时，若需修改UI，则需通过Controller对View通知更新UI。（红色）

示意图

• 该模式存在的问题：Activity**责任不明、十分臃肿**

Activity 由于其生命周期的功能，除了担任 **View** 层的部分职责（加载应用的布局、接受用户操作），还要承担 **Controller** 层的职责（业务逻辑的处理）

随着界面的增多 & 逻辑复杂度提高，**Activity** 类的代码量不断增加，越加臃肿

2.2 MVP模式

• 出现的原因

为了解决上述 **MVC** 模式存在的问题，把分离 **Activity** 中的 **View** 层和 **Controller** 层的职责，从而对Activity代码量进行优化、瘦身，所以出现了 **MVP** 模式

• 角色说明

类型	定义	作用	使用方式
M (Model)	模型层 (表示 数据模型)	业务逻辑 / 数据存储	<ul style="list-style-type: none">• 写成“外观类(facade class)”• 为一个复杂的子系统提供高层次的简单易用的访问接口
V (View)	视图层 (表示一种视图)	<ul style="list-style-type: none">• View的绘制• 用户交互	Activity
P (Presenter)	呈现层	<ul style="list-style-type: none">• 连接 V层、M层，完成View层与Model层的交互• 业务逻辑处理	接口

示意图

• 模式说明

模式示意图	<p>The diagram illustrates the MVP (Model-View-Presenter) pattern. It shows four main components: '用户输入' (User Input), 'View', 'Presenter', and 'Model'. '用户输入' is represented by a blue rounded rectangle, while the others are white rectangles. An arrow points from '用户输入' to 'View'. Between 'View' and 'Presenter', there are two horizontal arrows: a top blue arrow pointing right labeled '需修改Model层的实例' (Need to modify the instance of the Model layer) and a bottom red arrow pointing left labeled '通知修改' (Notify modification). Between 'Presenter' and 'Model', there are also two horizontal arrows: a top blue arrow pointing right labeled '通知修改' (Notify modification) and a bottom red arrow pointing left labeled '需修改UI' (Need to modify UI).</p>
具体描述	<p>1. View层接收用户的输入</p> <p>2. View层 与 Model层交互必须经过Presenter层</p>

示意图

• 优点：（对比MVC模式）

1. 耦合度更低：通过 **Presenter** 实现数据和视图之间的交互，完全隔离了View层与Mode层，二者互不干涉

避免了 **View**、**Model** 的直接联系，又通过 **Presenter** 实现两者之间的沟通

- 2. **Activity** 代码变得更加简洁：简化了 **Activity** 的职责，仅负责UI相关操作，其余复杂的逻辑代码提取到了 **Presenter** 层中进行处理

2.3 MVVM

为了更加分离M、V层，更加释放Activity的压力，于是出现了MVVM模式

- 定义

VM 层：**ViewModel**，即 View的数据模型和Presenter的合体

基本上与 **MVP** 模式完全一致，将逻辑处理层 **Presenter** 改名为 **ViewModel**

- 模式说明

模式示意图	
具体描述	采用双向绑定（Data Binding）：View的变动，自动反映在 ViewModel；反之亦然，即： 1. ViewModel、View之间的交互通过Data Binding完成 2. Data Binding可实现双向的交互

示意图

- 优点

使得视图层（**View**） & 控制层（**Controller**）之间的耦合程度进一步降低，关注点分离更为彻底，同时减轻了 **Activity** 的压力

本文主要讲解MVC和MVP模式，不过多阐述MVVM模式.

3. MVC、MVP模式的区别

类型	层级含义	业务逻辑处理方式	View与Model的交互	View与业务逻辑层的交互
MVC	V对应的是：布局文件 & Activity	Controller层	• 直接 • 间接：Controller层	Activity (Controller层)
MVP	V对应的是：Activity (独立于Activity的助手)	Presenter层	间接：必须经过Presenter层 (完全隔离了View与Model)	接口 (Presenter层)

示意图

4. 三种模式出现的初衷

- MVC** 模式的出现
为解决**程序模块化**问题，于是MVC模式出现了：将业务逻辑、数据处理与界面显示进行分离来组织代码，即分成M、V、C层；
- MVP** 模式的出现
但M、V层还是有相互交叉、**隔离度不够**，同时写到Activity上使得Activity代码**臃肿**，于是出现了MVP：隔离了MVC中的 M 与 V 的直接联系，将M、V层更加隔离开来，并释放了Activity的压力；
- MVVM** 模式的出现
为了**更加分离M、V层**，更加释放Activity的压力，于是出现了MVVM：使得V和M层之间的耦合程度进一步降低，分离更为彻底，同时更加减轻了Activity的压力。

下面，我将详细讲解一下最常用的 **MVP** 模式的核心思想 & 使用

5. MVP模式详解

此处主要详细分析MVP模式的核心思想，并实例说明。

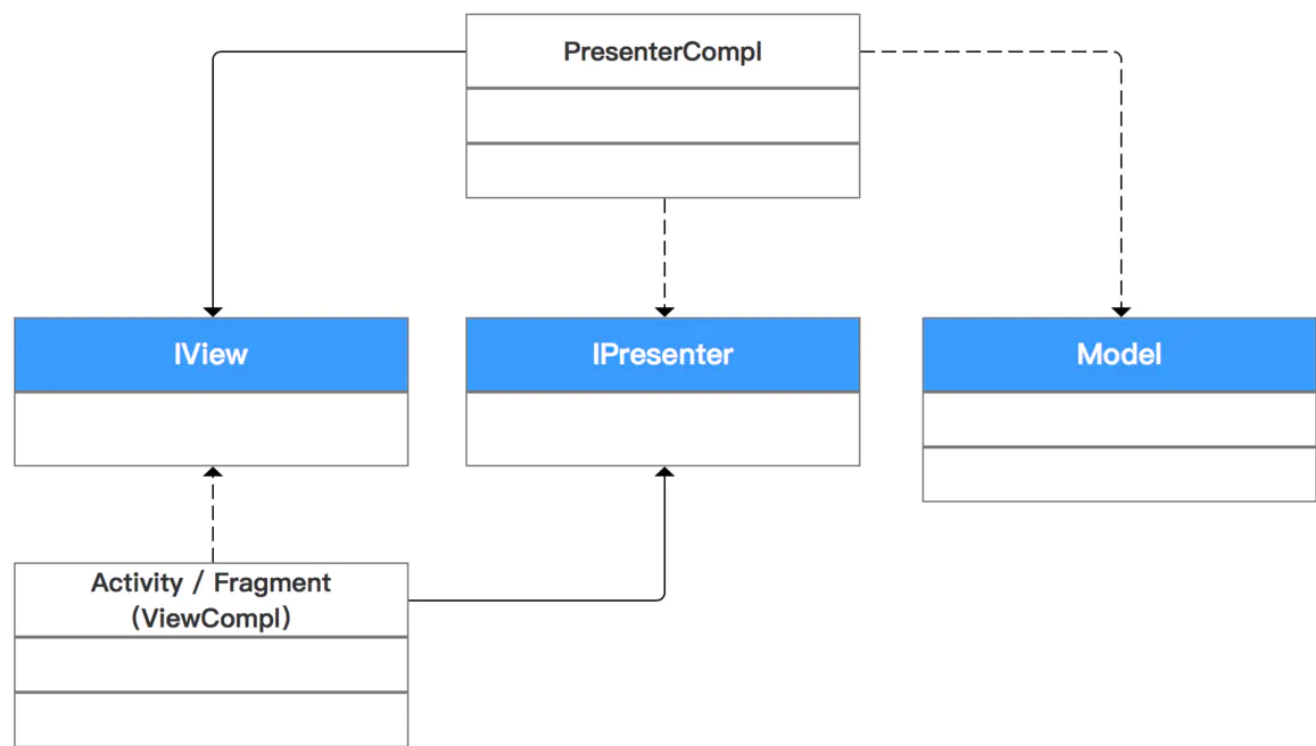
5.1 核心思想

把Activity里的逻辑都抽离到 **View** 和 **Presenter** 接口中去 & 由具体的实现类来完成。具体实现思路如下：

- 把 **Activity** 中的 **UI** 逻辑抽象成 **View** 接口 & 由具体的实现类来完成
- 把业务逻辑抽象成 **Presenter** 接口 & 由具体的实现类来完成
- Model** 类还是原来 **MVC** 模式的 **Model** 层

5.2 实现步骤

MVP 模式的 UML 图



示意图

通过 UML 图可看出，使用 MVP 模式的步骤如下：

步骤	具体描述	备注
1. 设置View层	1. 创建IView接口 & 放置所有视图逻辑的接口 2. 其实现类是当前的Activity/Fragment	<ul style="list-style-type: none">• Activity里包含了一个IPresenter接口• 而PresenterCompl实现类里又包含了一个IView & 依赖了Model层• Activity里只保留对IPresenter接口的调用，其它工作全部留到PresenterCompl实现类中实现
2. 设置Presenter层	1. 创建IPresenter接口 & 放置所有业务逻辑的接口 2. 创建它的实现类PresenterCompl	<ul style="list-style-type: none">• 由于接口可以有多种实现，方式多样且灵活• 故可方便地定位到相应的业务功能 & 单元测试
3. 设置Model层	放置业务逻辑 & 数据存储	Model并不是必须有，但必须有Presenter层 & View层

示意图

5.3 实例讲解

本节通过一个 英语词典 app 实例 讲解 MVP 模式具体的实现

前言：工程项目的列表架构

MVP 技术架构的项目结构非常清晰：把 M、V、P 层分别分为三个文件夹：Model、View、Presenter，每个文件下分别是对应的接口和实现的类

其中 Model 层的 fanyi 类是作为实现用 GSON 解析 JSON 信息的一个 JavaBean

步骤1：设置View层（ IView接口 & 实现类 ）

```
/**
 * View接口: IfanyiView
 * 需定义在实现类中需要用的方法
 */

public interface IfanyiView {

    void init();//初始化
    void SetInfo(String str); //输出翻译信息
    void SetError(); //输出出错信息

}

/**
 * View实现类: MainActivity类
 * 注：由于MainActivity是对应View层的实现类,所以要实现View层的接口
 */

public class MainActivity extends AppCompatActivity implements IfanyiView {

    private EditText et;
    private TextView tv;
    CidianPresenter cidianPresenter; // 声明了Presenter对应类

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 实例化P对应类的对象和findView
        init();
        // 接受用户的输入
        findViewById(R.id.btnfanyi).setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View v) {
                //将View层获得的数据传入Presenter层 ,注意还要传递MainActivity
                cidianPresenter.InputToModel(et.getText().toString(), MainActivity.this);
            }
        });
    }
}
```

```

@Override
public void init(){
//实例化P类的对象和findView
    cidianPresenter = new CidianPresenter(this);
    et = (EditText) findViewById(R.id.editText);
    tv = (TextView) findViewById(R.id.tv);
}

@Override
//输出出错信息
public void SetError() {
    tv.setText("查询不成功，请检查网络");
}

//输出翻译信息
@Override
public void SetInfo(String str){
    tv.setText(str);
}
}

```

// 从上述代码可看出，MainActivity只做了FindView、setOnClickListener的工作（包含了cidianPresenter），简洁清晰

步骤2：设置Presenter层（创建IPresenter接口&实现类）

```

/**
 * Presenter接口： ICidianPresenter
 * 需定义在实现类中需要用到的方法
 */

public interface ICidianPresenter {

    void InputToModel(String input,Context context); // 将View层获得的数据传入Model层

}

/**
 * Presenter层的实现类： CidianPresenter类
 * 注： 由于CidianPresenter是对应Presenter层的实现类,所以要实现Presenter层的接口
 */

public class CidianPresenter implements onfanyiListener,ICidianPresenter {
    // 1. 声明View层对应接口、Model层对应的类
    IfanyiView fyV;
    fanyimodel fanyimodel;

    // 2. 重构函数，初始化View接口实例、Model实例
    public CidianPresenter(IfanyiView fyV){
        this.fyV = fyV;
        fanyimodel = new fanyimodel();
    }
}

```

```
// 3.将View层获得的数据传入Model层,注意要传递this.当前类
@Override
public void InputToModel(String input, Context context){

    fanyimodel.HandleData(input, context, this);

}
// 回调函数, 调用UI更新
@Override
public void onSuccess(String str) {
    fyV.SetInfo(str);    }
// 回调函数, 调用UI输出出错信息
@Override
public void onError() {
    fyV.SetError();    }
}

// 注:
// a. 保留IfanyiView的引用, 就可直接在CidianPresenter当前类进行UI操作而不用在Activity操作
// b. 保留了Model层的引用就可以将View层的数据传递到Model层
```

步骤3 : Model层 (Model层接口 & 实现类)

```
/**
 * Model层接口: Ifanyi
 * 需定义在实现类中需要用到的方法
 */
public interface Ifanyi {

    void HandleData(String input,Context context,final onfanyiListener listener);
    String fanyiToString(fanyi fy);

}

/**
 * Model层的实现类: fanyiModel类
 * 注: 由于fanyiModel是对应Model层的实现类,所以要实现Model层的接口
 */

public class fanyimodel implements Ifanyi {

    private fanyi fy = new fanyi();

    public void HandleData(String input,Context context,final onfanyiListener listener){

        // 使用Volley框架来实现异步从网络的有道API获取翻译数据
        RequestQueue mQueue = Volley.newRequestQueue(context);
        StringRequest stringRequest = new StringRequest("http://fanyi.youdao.com/openapi.do?key-

        @Override
        public void onResponse(String s) {

            // 用Gson方式解析获得的json字符串
            Gson gson = new Gson();
```

```
        fy = gson.fromJson(s.trim(),fy.getClass());

        // 回调监听器的函数把处理数据后的结果（翻译结果）返回给Presenter层
        listener.onSuccess(fanyiToString(fy));
    }
}, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError volleyError) {
        listener.onError();
    }
});
mQueue.add(stringRequest);
}

public String fanyiToString(fanyi fy){
    // 处理解析后的json数据，转成UI输出的字符串
    String strexplain = "解释: ";
    String strphonetic = "发音: ";
    String strweb = "网络释义: ";
    if (fy.basic == null){return "你所查找的还没有准确翻译";}
    for (int i = 0; i<fy.basic.explains.length; i++){
        strexplain +=fy.basic.explains[i]+"\\n";
        if (i != fy.basic.explains.length-1 )
            {strexplain += "\\t\\t\\t\\t";}
    }
    strphonetic += fy.basic.phonetic +"\\n";
    for (int i = 0; i<fy.web.size(); i++){
        for(int j = 0; j<fy.web.get(i).value.length;j++)
        {
            strweb += fy.web.get(i).value[j]+",";
        }
        strweb += fy.web.get(i).key+"\\n";
        strweb += "\\t\\t\\t\\t\\t\\t\\t\\t";
    }
    return strexplain+"\\n"+strphonetic+"\\n"+strweb;
}
}
```

至此，关于 MVP 模式的实例讲解，讲解完毕。