自定义控件（尚学谷）

目录[(?)](http://blog.csdn.net/study_zhxu/article/details/52152895" \o "系统根据文章中H1到H6标签自动生成文章目录)[[-]](http://blog.csdn.net/study_zhxu/article/details/52152895)

1. [本篇文章已授权微信公众号 guolin\_blog 郭霖独家发布](http://blog.csdn.net/study_zhxu/article/details/52152895#t0)
2. [前言](http://blog.csdn.net/study_zhxu/article/details/52152895#t1)
3. [什么是MVP](http://blog.csdn.net/study_zhxu/article/details/52152895#t2)
   1. [MVP分为三层](http://blog.csdn.net/study_zhxu/article/details/52152895#t3)
   2. [MVP模式的核心思想](http://blog.csdn.net/study_zhxu/article/details/52152895#t4)
4. [MVP的使用](http://blog.csdn.net/study_zhxu/article/details/52152895#t5)
5. [MVP的优势](http://blog.csdn.net/study_zhxu/article/details/52152895#t6)
6. [总结](http://blog.csdn.net/study_zhxu/article/details/52152895#t7)

**本篇文章已授权微信公众号 guolin\_blog （郭霖）独家发布**

**前言**

随着MVP概念的兴起和发展，MVP使用越来越广泛，当然MVP的优势也越来越被认同，在合作开发功能模块细分中MVP有着得天独厚的优势。本篇文章就来简单的说说如何使用MVP。

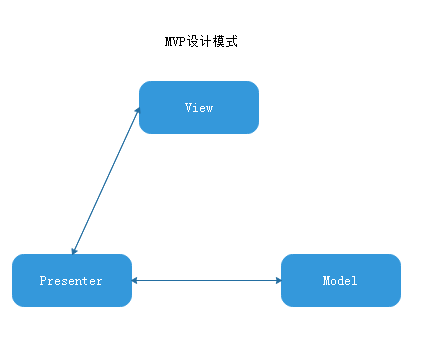
**什么是MVP**

MVP是MVC的变种，其实是一种升级。要说MVP就要说说MVC，在MVC中Activity其实是View层级，但是通常在使用中Activity即使View也是Controller，并没有将View层和Controller层进行分离，   
耦合度大大提高，非常不利于项目的管理。这时候MVP就应运而生了。

**1、MVP分为三层**

Model View Presenter

**2、MVP模式的核心思想**

MVP把Activity中的UI逻辑抽象成View接口，把业务逻辑抽象成Presenter接口，Model类还是原来的Model。   
在MVP模式中Activity的功能就是响应生命周期和显示界面，具体其他的工作都丢到了Presenter层中进行完成，Presenter其实是Model层和View层的桥梁。   
具体的MVP的好处还有很多，就不详细介绍了，下面看看如何使用MVP。   
 

上面一张简单的MVP模式的UML图，从图中可以看出，使用MVP，至少需要经历以下步骤：   
1、创建 IPresenter 接口，把所有业务逻辑的接口都放在这里，并创建它的实现 PresenterCompl（在这里可以方便地查看业务功能，由于接口可以有多种实现所以也方便写单元测试）。   
2、创建 IView 接口，把所有视图逻辑的接口都放在这里，其实现类是当前的Activity/Fragment。   
3、由UML图可以看出，Activity里包含了一个IPresenter，而PresenterCompl里又包含了一个IView并且依赖了Model。Activity里只保留对IPresenter的调用，其它工作全部留到PresenterCompl中实现。   
4、Model并不是必须有的，但是一定会有View和Presenter。   
通过上面的介绍，MVP的主要特点就是把Activity里的许多逻辑都抽离到View和Presenter接口中去，并由具体的实现类来完成。这种写法多了许多IView和IPresenter的接口，   
在某种程度上加大了开发的工作量，刚开始使用MVP的小伙伴可能会觉得这种写法比较别扭，而且难以记住。其实一开始想太多也没有什么卵用，只要在具体项目中多写几次，   
就能熟悉MVP模式的写法，理解意图，以及享受其带来的好处。

**MVP的使用**

理论说了这么多其实没有什么卵用，下面就来在代码中看看如何使用吧。   
先来看看项目中代码结构，结构图如下   
   
通过代码结构图可以看到看出MVP结构层级分明（在项目中使用MVP最好通过模块进行分层这样便于管理且结构清晰），

先来看看View层代码   
ILoginView接口代码如下

1 public interface ILoginView {

2 public void onClearText();

3 public void onLoginResult(Boolean result, int code);

4 }

可以看出ILoginView只是一个接口，简单的定义了两个方法。

看看ILoginView的实现类也就是我们的LoginActivity   
LoginActivity代码如下

[复制代码](javascript:void(0);)

1 public class LoginActivity extends AppCompatActivity implements ILoginView,View.OnClickListener{

2

3 private Button mLogin ;

4 private Button mClear ;

5 private EditText mName ;

6 private EditText mPassWord ;

7

8 ILoginPresenter loginPresenter ;

9

10 @Override

11 protected void onCreate(Bundle savedInstanceState) {

12 super.onCreate(savedInstanceState);

13 setContentView(R.layout.activity\_main);

14

15 mLogin = (Button) findViewById(R.id.btn\_login);

16 mClear = (Button) findViewById(R.id.btn\_clear);

17 mName = (EditText) findViewById(R.id.et\_name);

18 mPassWord = (EditText) findViewById(R.id.et\_password);

19

20 mLogin.setOnClickListener(this);

21 mClear.setOnClickListener(this);

22

23 loginPresenter = new LoginPresenterCompl(this) ;

24

25 }

26

27 @Override

28 public void onClearText() {

29 mName.setText("");

30 mPassWord.setText("");

31 Toast.makeText(this,"clear",Toast.LENGTH\_SHORT).show();

32 }

33

34 @Override

35 public void onLoginResult(Boolean result, int code) {

36 mLogin.setEnabled(true);

37 mClear.setEnabled(true);

38

39 if(result){

40 Toast.makeText(this,"登陆成功: "+code,Toast.LENGTH\_SHORT).show();

41 }else{

42 Toast.makeText(this,"登陆失败："+code,Toast.LENGTH\_SHORT).show();

43 }

44 }

45

46 @Override

47 public void onClick(View v) {

48 int id = v.getId() ;

49 String name = mName.getText().toString() ;

50 String password = mPassWord.getText().toString() ;

51

52 switch (id){

53 case R.id.btn\_login :

54 loginPresenter.doLogin(name,password);

55 break ;

56 case R.id.btn\_clear :

57 loginPresenter.clear();

58 break;

59 }

60 }

61 }

[复制代码](javascript:void(0);)

布局文件如下

按 Ctrl+C 复制代码



按 Ctrl+C 复制代码

在LoginActivity中我们可以看到，LoginActivity实现了ILoginView接口，实现了未实现的方法，在代码中可以看出LoginActivity并没有做一些逻辑处理工作，数据处理的工作都是调用ILoginPresenter   
完成的。下面就来看看ILoginPresenter

1 public interface ILoginPresenter {

2 public void clear() ;

3 public void doLogin(String name, String password) ;

4 }

也是简单的接口定义两个未实现的方法。   
看看其实现类LoginPresenterCompl

[复制代码](javascript:void(0);)

1 public class LoginPresenterCompl implements ILoginPresenter {

2

3 private ILoginView loginView ;

4 private User user ;

5

6 @Inject

7 public LoginPresenterCompl(ILoginView view){

8 loginView = view ;

9 user = new User("张三","123456") ;

10 }

11

12 @Override

13 public void clear() {

14 loginView.onClearText();

15 }

16

17 @Override

18 public void doLogin(String name, String password) {

19 boolean result = false ;

20 int code = 0 ;

21 if(name.equals(user.getName()) && password.equals(user.getPassword())){

22 result = true ;

23 code = 1 ;

24 }else{

25 result = false ;

26 code = 0 ;

27 }

28

29 loginView.onLoginResult(result,code);

30 }

31 }

[复制代码](javascript:void(0);)

该实现类也比较简单，定义了用户名是张三，密码是123456的一个登陆用户，然后进行登陆和清除的操作。   
OK这就完成了最简单的MVP模式了。

看看model层的代码   
User的代码如下

[复制代码](javascript:void(0);)

1 public class User {

2

3 private String name ;

4 private String password ;

5

6 public User(String name,String password){

7 this.name = name ;

8 this.password = password ;

9 }

10

11 public String getName() {

12 return name;

13 }

14

15 public void setName(String name) {

16 this.name = name;

17 }

18

19 public String getPassword() {

20 return password;

21 }

22

23 public void setPassword(String password) {

24 this.password = password;

25 }

26 }

[复制代码](javascript:void(0);)

看完上述MVP的代码后可能有的同学会说一个简单的登陆就是简单的验证用户名和密码为什么要设计的这么复杂，可能有人会觉得还没有以前的写法简单呢。但是这只是一个简单的实现MVP的例子。   
当项目越来越大，代码越来越多的时候就能够真正的体现出MVP模式的优势了。

知道了什么是MVP也知道了如何在项目中使用MVP，那MVP的优势到底有哪些呢？

**MVP的优势**

1、使Activity中的代码更加简洁   
在传统的项目中Activity兼顾着Controller和View，这使得其代码分分钟上千行（本人深受其害），这使得代码难以理解难以维护，看到这样的Activity就想吐。   
使用MVP之后，Activity就能瘦身许多了，基本上只有FindView、SetListener以及Init的代码。其他的就是对Presenter的调用，还有对View接口的实现。这种情形下阅读代码就容易多了，   
而且你只要看Presenter的接口，就能明白这个模块都有哪些业务，很快就能定位到具体代码。Activity变得容易看懂，容易维护，以后要调整业务、删减功能也就变得简单许多。

2、方便进行单元测试   
一般单元测试都是用来测试某些新加的业务逻辑有没有问题，如果采用传统的代码风格，我们可能要先在Activity里写一段测试代码，测试完了再把测试代码删掉换成正式代码，   
这时如果发现业务有问题又得换回测试代码，咦，测试代码已经删掉了！好吧重新写吧……   
MVP中，由于业务逻辑都在Presenter里，我们完全可以写一个PresenterTest的实现类继承Presenter的接口，现在只要在Activity里把 Presenter的创建换成PresenterTest，   
就能进行单元测试了，测试完再换回来即可。万一发现还得进行测试，那就再换成PresenterTest吧。

3、避免Activity的内存泄露   
APP发生OOM的最大原因就是出现内存泄露造成APP的内存不够用，而造成内存泄露的两大原因之一就是Activity泄露（Activity Leak）（另一个原因是Bitmap泄露（Bitmap Leak））。   
Java一个强大的功能就是其虚拟机的内存回收机制，这个功能使得Java用户在设计代码的时候，不用像 C++ 用户那样考虑对象的回收问题。然而，Java用户总是喜欢随便写一大堆对象，   
然后幻想着虚拟机能帮他们处理好内存的回收工作。可是虚拟机在回收内存的时候，只会回收那些没有被引用的对象，被引用着的对象因为还可能会被调用，所以不能回收。   
Activity 是有生命周期的，用户随时可能切换Activity，当APP的内存不够用的时候，系统会回收处于后台的Activity的资源以避免OOM。   
采用传统的模式，一大堆异步任务和对UI的操作都放在Activity里面，比如你可能从网络下载一张图片，在下载成功的回调里把图片加载到Activity的ImageView里面，   
所以异步任务保留着对Activity的引用。这样一来，即使Activity已经被切换到后台（onDestroy 已经执行），这些异步任务仍然保留着对Activity实例的引用，   
所以系统就无法回收这个Activity实例了，结果就是Activity Leak。Android 的组件中，Activity 对象往往是在堆（Java Heap）里占最多内存的，所以系统会优先回收Activity对象，   
如果有Activity Leak，APP很容易因为内存不够而 OOM。   
采用MVP模式，只要在当前的Activity的onDestroy里，分离异步任务对Activity的引用，就能避免Activity Leak。

**总结**

以上就是MVP的简单实现，可能示例代码太简单无法体现MVP的优势，但是理解了MVP的思路在项目中使用MVP就才能够真正体验到MVP带来的好处优势。