MVP模式在携程酒店的应用和扩展

0

接口 (http://www.csdn.net/tag/接口/news) new (http://www.csdn.net/tag/new/news)

应用 (http://www.csdn.net/tag/应用/news) 模式 (http://www.csdn.net/tag/模式/news)

代码 (http://www.csdn.net/tag/代码/news)

前言

酒店业务部门是携程旅行的几大业务之一，其业务逻辑复杂，业务需求变动快，经过多年的研发，已经是一个代码规模庞大的工程，如何规范代码，将代码按照其功能进行分类，将代码写到合适的地方对项目的迭代起着重要的作用。

MVP模式是目前客户端比较流行的框架模式，携程在很早之前就开始探索使用该模式进行相关的业务功能开发，以提升代码的规范性和可维护性，积累了一定的经验。本文将探讨一下该模式在实际工程中的优点和缺陷，并介绍携程面对这些问题时的思考，解决方案以及在实践经验基础上对该模式的扩展模式MVCPI。

一、从MVC说起

MVC已经是非常成熟的框架模式，甚至不少人认为它过时陈旧老气，在实践中，很多同事会抱怨，MVC会使得代码非常臃肿，尤其是Controller很容易变成大杂烩，预期的可维护性变得很脆弱，由此导致一方面希望有新框架模式可以解决现在的问题，但同时对框架模式又有些怀疑，新的框架模式是否能真正解决现在的问题？会不会重蹈覆辙？会不会过度设计？会不会掉进一个更深的坑？总之，这些类似“一朝被蛇咬，十年怕井绳”的担忧显得不无道理。但不管如何，我们需要仔细耐心的做工作。

1.1、被误解的MVC

在MVP模式逐渐流行之前，不管我们有意识或无意识地，我们使用的就是MVC模式。以Android为例，我们来看看MVC是什么样子。

```
public class HotelActivity extends Activity {
        private TextView mNameView;
        private TextView mAddressView;
        private TextView mStarView;
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main2);

            mNameView = (TextView) findViewById(R.id.hotel_view);
            mAddressView = (TextView) findViewById(R.id.address_view);
            mStarView = (TextView) findViewById(R.id.star_view);

            HotelModel hotel = HotelLoader.loadHotelById(1000);

            mHotelNameView.setText(hotel.hotelName);
            mHotelAddressView.setText(hotel.hotelAdress);
            mHotelStarView.setText(hotel.hotelStar);
        }
}
```

上面的代码，概括了Android MVC的基本结构，从笔者的经验来看，很多应用都存在这样的代码风格，也就是大部分人认为的MVC：

- Model：

```
Hotel,HotelLoader
```

- Controller：

```
HotelActivity
```

- View：

```
mHotelNameView
mHotelAddressView
```

```
mHotelStarView
```

可以试想一下如果这个界面展示的数据非常的多话，MainActivity必然会变得非常庞大，就像大部分人所抱怨的那样。诚然，上面的demo是MVC模式，但是，它仅是从系统框架的角度来看，如果从应用框架来看，它不是。下面来看一下，从应用框架来看一下MVC正确的结构：

1.2、MVC的正确姿势

应用中的MVC应该在系统的MVC框架上根据业务的自身的需要进行进一步封装，也就是说，如果在我们宣称我们是使用MVC框架模式的时候，代表我们的主要工作是封装自己的MVC组件。它看起来应该是像下面的风格：

```
public class HotelActivity extends Activity {

    private HotelView mHotelView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);
        mHotelView = (HotelView) findViewById(R.id.hotel_view);
        HotelModel hotel = HotelLoader.loadHotelById(1000);
        mHotelView.setHotel(hotel);
     }
}
```

跟之前的代码相比，基本结构是相似的，如下：

- Model：

```
HotelHotelLoader
```

- Controller：

```
HotelActivity
```

- View：

```
mHotelView
```

仅仅View层发生了变化，这是因为，Model和Controller相对是大家容易理解的概念，在面临任何一个业务需求的时候，自然就能产生的近乎本能的封装（尽管Model的基本封装大部分工程师都可完成，但不可否认Model的设计是至关重要而有难度的）；而对View的看法，可能就是“能正确布局和展示就行”。但这正是关键所在：我们需要对界面进行全方位的封装，包括View。具体来说，一个真正的MVC框架应该具备下面的特点：

- 数据都由Model进行封装
- View绑定业务实体，view.setXXX
- Controller不管理与业务无关的View

1.3 MVC模式的问题所在

前面说到，很多人抱怨采用MVC模式使得Controller变得很臃肿，我相信，Controller变得臃肿是事实，但其归结于采用MVC模式是不正确的，这个锅不应该由MVC来背，因为，这个论点会导致我们走向错误的方向从而无法发现MVC真正的问题所在。为什么这么说呢，那是因为在本人了解到的很多情况下，大家并没有正确理解MVC框架模式，如采用前文中第一种模式，自然会使得Controller臃肿，但是如果采用第二种模式，Controller的代码和逻辑也会非常清晰，至少不至于如此多的抱怨。因此如果只是想解决Controller臃肿的话，MVC就够了，毋庸质疑。那MVC的问题是什么呢？我想只有深刻的理解了这个问题，我们才有必要考虑是否需要引入新的框架模式，以及避免新的模式中可能出现的问题。

View强依赖于Model是MVC的主要问题。由此导致很多控件都是根据业务定制，从Android的角度来看，原本可以由一个通用的layout就能实现的控件，由于要绑定实体模型，现在必须要自定义控件，这导致出现大量不必要的重复代码。因此有必要将View和Model进行解耦，而MVP的主要思想就是解耦View和Model。由此引入MVP就显得很自然。

二、 Android MVP

2.1、参考实现

Android 官方提供的MVP参考实现，大致思想如下：

请输入标签

1、抽象出IView接口，规范控件访问方法，而不限View具体来源

请输入标题

```
public interface IHotelView {
      public TextView getNameView();
      public TextView getAddressView();
      public TextView getStarView();
}
```

请输入链接地址

2、抽象出IPresenter接口，定义IView 和 Model的绑定接口

请输入推荐理由

```
public interface IHotelPresenter {
      public void setView(IHotelView hotelView);
      public void setData(HotelMotel hotel);
}
```

3、IPresenter的实现类，实施数据和IView的绑定，并负责相关的业务处理

```
public class HotelPresenter implements IHotelPresenter {
      private IHotelView hotelView;
      public void setView(IHotelView hotelView) {
             this.hotelView  = hotelView;
      }
      public void setData(HotelModel hotel) {
        hotelView.getNameView().setText(hotel.hotelName);
        hotelView.getAddressView().setText(hotel.hotelAddress);
        hotelView.getStarView().setText(hotel.hotelStart);
      }
}
```

4、Activity实现IView，角色转变为View，弱化Controller的功能

```
public class HotelActivity extends Activity  implements IHotelView {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);

        HotelModel hotel = HotelLoader.loadHotelById(1000);
        IPresenter presenter = new Presenter();
        presenter.setView(this);
        presenter.setData(hotel);
    }
    @Override
    public TextView getNameView() {
        return (TextView)findViewById(R.id.hotel_name_view);
    }
    @Override
    public TextView getAddressView() {
        return (TextView)findViewById(R.id.hotel_address_view);
    }
    @Override
    public TextView getStarView() {
        return (TextView)findViewById(R.id.hotel_address_view);
    }
}
```


上述代码，主要的特点可以概括为：

- 面向接口
- View - Model 解耦
- Activity角色转换

就目前了解到的情况来看，很多采用MVP模式的应用基本上和android参考实现方案差别不大，说明该模式的应用场景也是很广泛的。

2.2 Android MVP存在的问题

尽管已经有了大量的应用，但不可否认该模式的还是存在一些问题，这些问题在携程的使用过程中也得到了体现。比如，上下文丢失问题，生命周期问题，内存泄露问题以及大量的自定义接口，回调链变长等问题。可以归纳为：

- 业务复杂时，可能使得Activity变成更加复杂，比如要实现N个IView，然后写更多个模版方法。
- 业务复杂时，各个角色之间通信会变得很冗长和复杂，回调链过长。
- Presenter处理业务，让业务变得很分散，不能全局掌握业务，很难去回答某个业务究竟是在哪里处理的。
- 用Presenter替代Controller是一个危险的做法，可能出现内存泄漏，生命周期不同步，上下文丢失等问题。

以下面的这个需求来看几个具体的示例：

详情按钮的展示需要服务端下发标记位控制，展示时点击需要请求一个服务，服务返回时toast提示用户

```
public class HotelPresenter {
    private IHotelView mHotelView;
    private Handler handler = new  Handler(getMainLooper());
    public void setData(HotelModel hotelModel) {
        View button = mHotelView.getButtonView();
        int visibility = hotelModel.showButton ? .VISIBLE :GONE;
        button.setVisibility(visibility);
        if (hotelModel.showButton) {
            button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                sendRequest();
            }
        });
    }

    private void sendRequest() {
       new Thread() {
           public void run() {
               Thread.sleep(15*1000);
               handler.post(new Runnable() {
                       public void run() {
                             Toast.makeText(???) //Where is Context?
                       }
               });
         }
       }.start();
    }
}
```

请输入标签

请输入标题

请输入链接地址

请输入推荐理由

上述代码表明，HotelPresenter可以处理大部分的业务，但是在最后需要使用上下文的时候，出现了困难，因为脱离了上下文，展示一个Toast都不能实现

为了避免这样的尴尬，因此改进方案如下：

```
public class HotelPresenter {
     private IHotelView mHotelView;
     private Fragment mFragment;
     private HotelPresenter(Fragment fragment) {
       this.mFragment = fragment;
     }
     private Handler handler = new Handler(Looper.getMainLooper());

     public void setData(HotelModel hotelModel) {
         View button = mHotelView.getButtonView();
         button.setVisibility(hotelModel.showButton ? VISIBLE :GONE);
         if (hotelModel.showButton) {
               button.setOnClickListener(new View.OnClickListener() {
                      @Override
                      public void onClick(View v) {
                             sendRequest();
                      }
               });
         }
     }

     private void sendRequest() {
        new Thread() {
            public void run() {
                Thread.sleep(15*1000);
                handler.post(new Runnable() {
                    public void run() {
                        Context context = mFragment.getActivity();
                        int duration = LENGTH_SHORT;
                        //NullPointerException will occur
                                    Toast.makeText(context," 成 功 ",duration).show();

                    }
                });
        }.start();
     }
}
```

改进的方案中，考虑到需要使用上下文，因此新增了接口传入Fragment作为上下文，在Presenter需要时可以使用，但是，由于Fragment生命周期会了变化，可能会导致空指针问题。

于是新的问题又需要解决。主要是两个思路，一个是为Presenter增加生命周期方法，在Fragment的生命周期方法里调用Presenter对应的生命周期函数，但这就让Presenter看起来像Fragment的孙子；另外一个就是承认Presenter其实不太合适承担Controller的职责，从而提供接口给外部处理；如下：

```
public class HotelPresenter {
        private IHotelView mHotelView;
        private Handler handler = new Handler(Looper.getMainLooper());
        public void setData(HotelModel hotelModel) {
            View button = mHotelView.getButtonView();
            button.setVisibility(hotelModel.showButton ? VISIBLE :GONE);
            if (hotelModel.showButton) {
                 button.setOnClickListener(new View.OnClickListener() {
                 @Override
                 public void onClick(View v) {
                      if (mCallback != null) {
                           mCallback.onSendButtonClicked();
                      }
                 });
            }
        }

        public interface Callback {
           public void onSendButtonClicked();
        }

        private Callback mCallback;
        public  void setCallback(Callback  callback) {
             mCallack = callback;
        }
}
```


三、MVP的扩展模式MVCPI

由于前面的分析，MVP参考实现并不是万能的，携程酒店并没有完全采用参考实现方案，而是结合自身的实践经验思考之后设计出来的扩展方案。我们主要考虑了一下的几个问题：

- 如何定义View接口？
- 如何定位Presenter ？
- 如何对待Controller？
- 如何解决长长的回调链？

通过对上述问题的思考，提出对应的解决方法，规避前面论述的各种问题，形成了携程酒店的MVCPI框架模式，并在多个业务场景运行，取得了较为满意的效果。下面，详细介绍MVCPI模式。

3.1、 IView

和Android 参考实现不一样的是，我们并没有采用强类型的接口作为表达View的方式，而是采用弱类型的接口来定义View。具体定义方式如下：

```
public interface IView {
    //用于展示酒店名称的控件
    int NAME_VIEW = R.id.name_view;
    //用于展示酒店地址的控件
    int ADDRESS_VIEW = R.id.address_view;
    //用于展示酒店星级的控件
    int STAR_VIEW = R.id.star_view;
    //用于展示酒店详情入口的的控件
    int DETAIL_BUTTON = R.id.detail_button;
}
```

上面的接口简洁的描述了作为业务控件的View需要具备的子控间ID，并不需要具体的实现类。因此也不需要Activity去实现这个接口，只需要在layout中申明这几个ID的即可，极大的简化了代码。

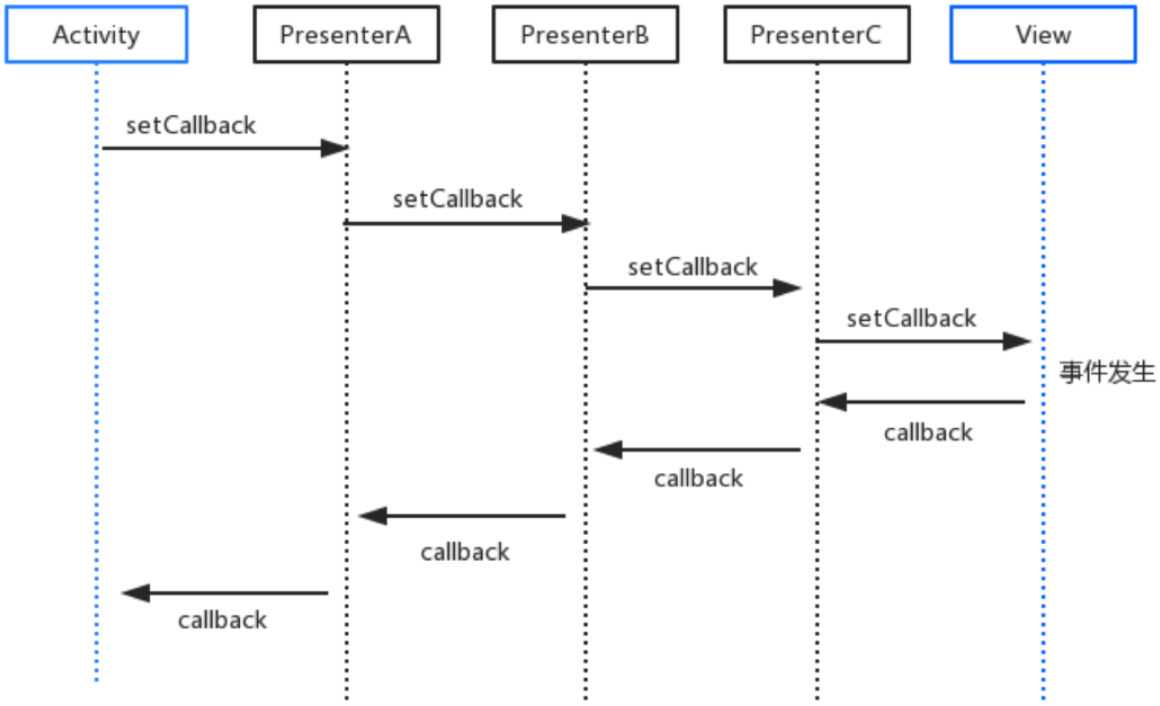
3.2、 Presenter

与参考实现的定位不一样，我们认为由Presenter取代Controller并不是一个好的做法，Presenter应是Controller的补充，主要起到View和Model解耦和数据绑定的作用，所负责的控件的上的业务还是有Controller决定如何去处理。另外setView接受的参数是一般的View，而非一个接口类型，内部根据IView定义的ID去查找子控件。如下：


```
public class CtripHotelPresenter {
    TextView mNameView;
    TextView mAddressView;
    TextView mStarView;
    Button mDetailButton;
    public void setView(View view) {
        mNameView = (TextView)mView.findViewById(IView.NAME_VIEW);
        mAddressView = (TextView)mView.findViewById(IView.ADDRESS_VIEW);
        mStarView = (TextView) mView.findViewById(IView.STAR_VIEW);
        mDetailButton = (Button) mView.findViewById(IView.DETAIL_BUTTON);
    }
    public void setData(HotelModel hotel) {
        mNameView.setText(hotel.hotelName);
        mAddressView.setText(hotel.hotelAdress);
        mStarView.setText(hotel.hotelStar);
        int v = hotel.showButton ? View.VISIBLE : View.GONE;
        mDetailButton.setVisibility(v);
    }
}
```

3.3、 Interactor

Interactor是我们定义出来的扩展元素，在MVP和MVC中都没有对应的角色。为了阐述它的含义，我们先来看看两个非常常见的场景。



回调链过长

请输入标签

在前面介绍过，Presenter自定义接口是很多候选方案中较为合理的选择，但相比MVC而言，MVP更容易出现如上图的一种调用和回调关系（甚至更长）。维护这种回调链通常来说是一件非常头痛的事情，从View的角度来看，很难知道某个事件到最后究竟完成了什么业务，Acitivity也不知道到要装配哪些回调。某个未知的新需求可能需要将该链条上的每个环节都增加回调。

请输入链接地址

下面来是另外一种场景，大家可以脑补一下采用上面的回调方案，回调链会是什么情况。

请输入推荐理由

交互集中型界面

请输入标签

请输入标题

请输入链接地址

请输入推荐理由

在该界面有几个特点：

- 几十种动态交互需求，
- 分布于不同的模块
- 分布于不同深度的嵌套层次中

经过大量版本迭代后，无论产品经理，研发或者测试，都不清楚到底有哪些需求，业务逻辑是什么，写在什么地方等等.....

上述两个场景可以得出两个结论：

- 排查问题非常耗时
- 增加功能成本高，容易引致其他问题

为了解决上述两个比较棘手的问题，我们引入了Interactor，用于描述整个界面的交互，一举解决上述两个问题。我们认为交互模型是一个功能模块的重要逻辑单元，相对于实体模型来说，交互模型更加抽象，在大多数的情况，并不能引起大家的注意，但它确实是如实体一样的存在，正是因为没有对交互进行系统的描述，才导致上面两种突出的问题。尽管抽象，但是交互模型本质非常简单，它有着和实体模型有相似的结构，示例如下：

```
public class HotelOrderDetailListeners {
    public View.OnClickListener mBackListener; // 返回按钮点击事件监听者
    public View.OnClickListener mShareClickListener;//分享按钮事件监听者
    public View.OnClickListener mConsultClickListener;//咨询按钮事件监听者
    ……
}
```

通过对界面整体分析后，我们建立如上的交互模型，所有的交互都在交互模型进行注册，由交互模型统一管理，进而可以对整个界面的交互进行宏观把控；然后在页面的所有元素中共享同一个交互模型，进而各个元素不再需要自定义接口和避免建立回调链。最后由Controller负责组装，进一步加强Controller的控制能力。

3.4、 MVCPI全貌

最后请辗转介绍一下MVCPI的代码结构

1、首先定义整个界面中有哪些用户交互，本例中就一个详情按钮交互

```
public class HotelInteractor {
    //点击详情的事件处理器
    public View.OnClickListener mDetail;
}
```

2、Presenter构造时需要传入交互模型，内部定义了IView接口，传入的View中需要包含它定义的ID的控件，在bindData时，详情按钮的点击不是通过匿名内部类去处理，而是直接引用交互模型中定义的mDetail

```
public class HotelPresenter {
    private View hotelView;
    private HotelInteractor mInteractor;
    private Button mDetailButton;

    public HotelPresenter(HotelInteractor interactor) {
          this.mInteractor = interactor;
    }

    private interface IView {
          int DETAIL= R.id.detail_button;
           ……
    }
    public void setView(View hotelView) {
          this.hotelView  = hotelView;
          mDetailButton= (Button)findViewById(IView. DETAIL );
    }
    public void setData(HotelModel hotel) {
         if (hotel.showButton) {
              mDetailButton.setVisibility(View.Visibile);
              mDetailButton.setOnClickListener(mInteractor.mDetail);
         }
    }
}
```

3、*Controller负责界面各个元素（包括交互模型）的初始化和装配*

请输入标签

```
public class HotelActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_main2);
         HotelInteractor interactor = new HotelInteractor();
         interactor.mDetail = new View.OnClickListener() {
              public void onClick(View view) {
                   viewHotelDetail();//处理详情业务;
              }
         };
         HotelModel model= HotelLoader.loadHotelById(1000);
         HotelPresenter presenter = new HotelPresenter (interactor);
         View view= findViewById(R.id.hotel_view);
         presenter.setView(view);
         presenter.setData(hotel);
       }
}
```

请输入链接地址

请输入推荐理由

四、结论

通过对MVC、MVP的介绍和研究，我们发现二者的关系并不是相互取代的关系，而是一种演化和改进的关系。经实践证明，MVC仍然具有强大的生命力，试图用MVP取代MVC几乎都会失败。携程在MVC模式基础上，结合MVP思想，加入Interactor元素搭建的MVCPI框架模式，一方面将数据

绑定逻辑从Controller（或者View）中分离出去，另一方面将交互模型的控制纳入进来，进一步加强了Controller的控制能力。无论从代码的简洁性，维护性，扩展性来看，都具有较大优势，具有一定的实践推广价值。

当然，任何框架模式都不是全能的，MVCPI也存在它不足，如果有好的意见和建议，欢迎加入，一起讨论推进框架模式的发展。

作者：赵伟麟，2011年就职于创新工场旗下点心OS，2014年加入携程酒店事业部，从事Android研发工作。擅长基于组件的业务架构，系统架构，建模，性能优化和重构，关注应用系统的扩展性和耦合性，追求简洁的代码。

声明：本文来自携程技术中心（ID：ctriptech）原创投稿。

请输入标签

请输入链接地址

(http://geek.csdn.net/user/publishlist/qiansg123)

钱曙光 (http://geek.csdn.net/user/publishlist/qiansg123)

发布于 架构 (http://geek.csdn.net/forum/83) 22小时前

请输入推荐理由

评论

已有0条评论

最新