【Dev Club分享】基于RxJava的一种MVP实现

原创 2016-09-01 戴俊 腾讯Bugly

Dev Club 是一个交流移动开发技术，结交朋友，扩展人脉的社群，成员都是经过审核的移动开发工程师。每周都会举行嘉宾分享，话题讨论等活动。

本期，我们邀请了腾讯IEG Android 开发工程师——戴俊，为大家分享《基于RxJava的一种MVP实现》。

分享内容简介：

RxJava是一个实现Java响应式编程的库，让异步事件以序列的形式组织。MVP则通常用来将View业务层与Model层分离开来，两者结合起来可轻松实现业务解耦、线程控制、单元测试等等强大功能

内容大体框架:

1. Android开发框架的选择
2. 如何一步步搭建MVP分层框架
3. 使用RxJava来进行线程控制
4. 结语

下面是本期分享内容整理

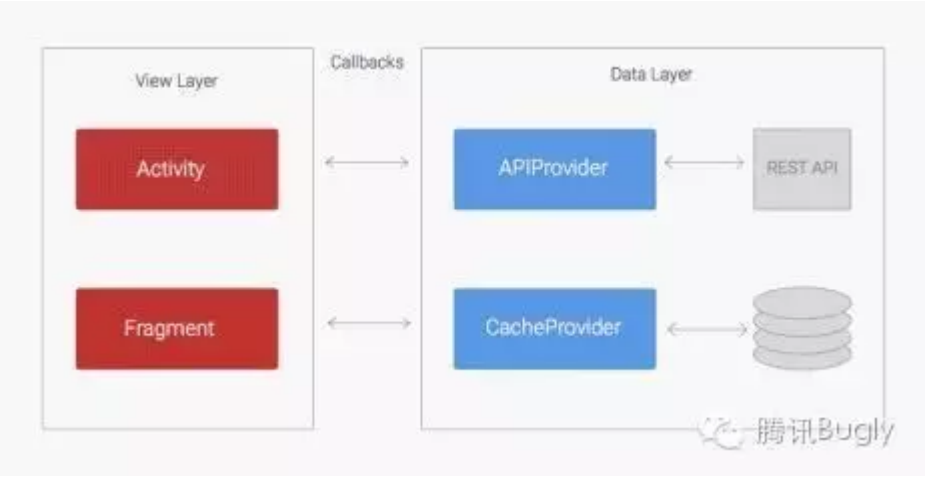
Hello，大家好，我是戴俊。目前在IEG腾讯动漫主要负责Android端的开发工作。

第一次进行这种微信群的分享，如果有任何疑问，欢迎大家在分享结束后提问。下面开始我们今天的分享。

1. Android开发框架的选择

我们知道原生Android开发已经是一个基础的MVC框架，所以在项目刚开始开发的时候并没有遇到太多问题。

对一个经典的Android MVC框架来讲，它的结构大概是下面这样（图片来自参考文献）



这样的结构下，Activity层既承担了View层的一部分工作（因为XML作为View层的一部分功能实在太弱了），又承担Controller层的工作，因此当业务变化时，Activity层会极剧膨胀。

拿我们项目早期的例子，一个Activity曾经最多达到了2000到3000行，重构的时候极其痛苦。

要解决这个问题，主要的办法有两种：

- 第一种是分层
- 第二种是模块化。

> 两个方法最终要实现的都是解耦。分层讲的是纵向层面上的解耦，模块化则是横向上的解耦。

我们今天要讨论的MVP就是一种通过分层来进行解耦的框架。

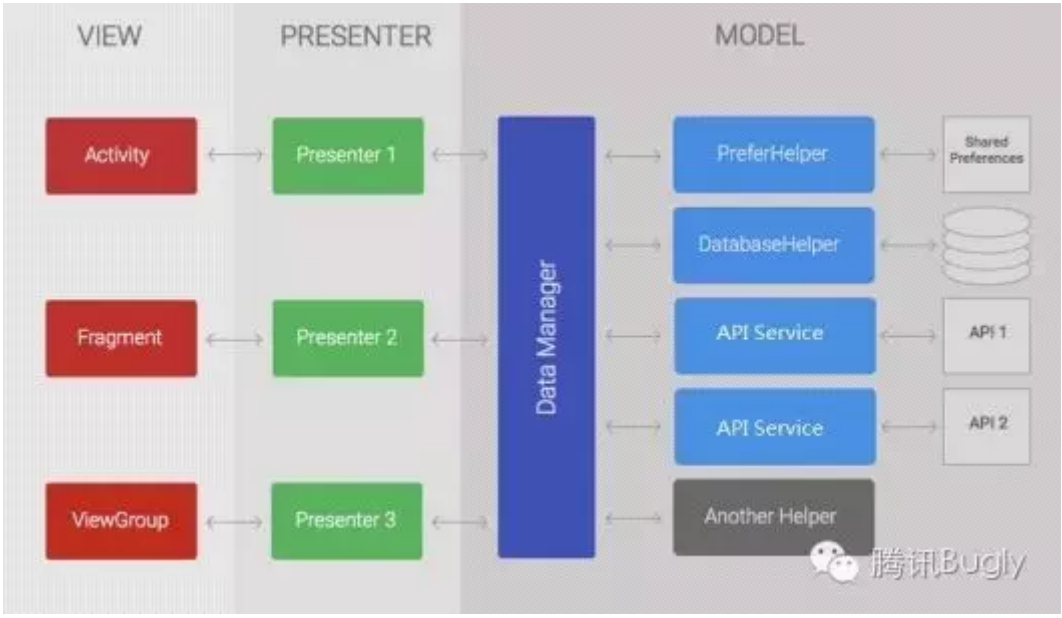
2. 如何一步步搭建MVP分层框架

如果你是个习惯了读文档的老司机，可以直接参考下面几篇文章

(Google文章名就可以了)

1. Android Application Architecture
2. Android Architecture Blueprints - Github
3. Google官方MVP示例之TODO-MVP - 简书
4. todo-mvp - github
5. dev-todo-mvp-rxjava - github

当然如果觉得看官方的示例太麻烦，那么下面我们就来讲解一下如何实现一个简单的MVP构架。



这是一个比较典型的MVP结构图(图片来自参考文献)，相比于第一张图，多了两个层，一个是Presenter和DataManager层。

多出这两个层到底有什么作用，下面我们来用代码说明。

首先我们假设有一个从服务端获取字符串并显示的手机上的简单功能。下面是主界面的代码

```java
//MainActivity.java 文件

public class MainActivity extends Activity implements MainView {

    MainPresenter presenter;
    TextView mShowTxt;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mShowTxt = (TextView)findViewById(R.id.text1);
        loadDatas();
    }

    public void loadDatas() {
        presenter = new MainPresenter().addTaskListener(this);
        presenter.getData();
    }

    @Override
    public void onShowString(String str) { mShowTxt.setText(str); }
}
```


Activity里面包含了几个文件，一个是View层的对外接口MainView,一个是P层的Presenter。

首先看View层的对外接口文件

```java
//MainView.java 文件

public interface MainView {
    void onShowString(String json);
}
```


因为这个功能比较简单，只需要在设备上显示一个字符串，所以只有一个接口方法onShowString()，再看P层代码

从上面三个文件可以看到，View层通过注册Listener将自己的接口MainView交给了Presenter，而Presenter层持有Model层的也只是一个接口。通过Presenter层将业务层与展现层隔离了开来，**这样的好处是什么？**

我们知道接口的一个作用通常是用来抽象行为，对外部屏蔽实现细节。所以对于View层来说，业务细节被屏蔽了，对业务层来说，展示细节被屏蔽了。而对于处于中间的Presenter层来说，它就像一个接口拼装器，把View层发出的请求传递给业务层，把业务层返回的数据又送还给View层展示，至于前后两端怎么实现的，它才不用关心。

接口的第二个作用是可以用来切换实现。我们先看下面的代码。

```java
//DataSource.java文件

public interface DataSource {
    String getStringFromRemote();
    String getStringFromCache();
}
```



```java
//DataSourceImpl.java文件

public class DataSourceImpl implements DataSource {
    @Override
    public String getStringFromRemote() {
        String ss = "remote Hello";
        /*
         * 从服务端获取到数据的代码
         * ..............................
         */
        return ss;
    }

    @Override
    public String getStringFromCache() { return "World"; }
}
```



```java
//DataSourceTestImpl.java文件

public class DataSourceTestImpl implements DataSource {

    @Override
    public String getStringFromRemote() { return "Hello "; }

    @Override
    public String getStringFromCache() { return " World Test "; }
}
```


从上面三个文件可以看到，业务层对外的只有一个接口，实现却有两个（DataSourceImpl和DataSourceTestImpl)。从名字大家就能看出来有什么作用了，一个是正常环境的业务层实现，一个是测试环境的业务层实现。

这里我们设想一个场景：

> 开发同学接到一个新的需求，设计稿也输出完成了，然而后台的接口却迟迟没到，怎么办？现在通过MVP，我们把业务层实现切换到DataSourceTestImpl，是不是可以先自己假写数据，调好一切前端和交互，然后泡一杯咖啡等后台同学把接口写完联调？或者有时候为了重现一个bug，要在线上写一条脏数据，测试完再删除？

类似的应用场景其实有非常非常多，这里我们就看到了使用接口解耦的一个好处了，不仅把业务层和展示层解耦开来，还把Android开发同其它的一切的外部数据依赖都解耦开来。

这里我想提到之前讨论过的单元测试问题，很多同学反馈项目开发过程中没有做过，或者没有时间精力去做单元测试，或者因为业务变化太大导致无法做单元测试。其实在我们项目中也遇到过样的问题，但其实通过这样分层之后，才发现单元测试其实是完全可以推进的，也完全不用再担心测试的时候会把脏数据写到线上的问题了。

到现在为止一个基于MVP简单框架就搭建完成了，但其实还遗留了一个比较大的问题。

很多同学可能已经发现了，Presenter层在调用业务层的时候是直接调用的，而Android规定，主线程是无法直接进行网络请求,会抛出NetworkOnMainThreadException异常。

所以在presenter层，我们需要进行一项线程切换的工作，这样才能保证“所有的IO操作都应当在线程中完成，主线程只负责页面渲染的工作”这一优化准则。

当然，Android本身提供一些方案，比如下面这种：

```
//MainPresenter.java文件

public void getData() {
    final Handler mainHandler = new Handler(Looper.getMainLooper());
     new Thread() {
         @Override
         public void run() {
             super.run();
             final String str = dataSource.getStringFromRemote();
             mainHandler.post(new Runnable() {
                 @Override
                 public void run() {
                     mainView.onShowString(str);
                 }
             });
         }
     }.start();


}
```


通过新建子线程进行IO读写获取数据，然后通过主线程的Looper将结果通过传回主线程进行展示，这种方案是勉强也行得通的。

但问题也有，一是线程需要额外管理，不可能每次发请求都要开启一个线程；二是适应性差，假如数据请求有先后依赖，有并行的情况，这样的写法变得脏乱无比。

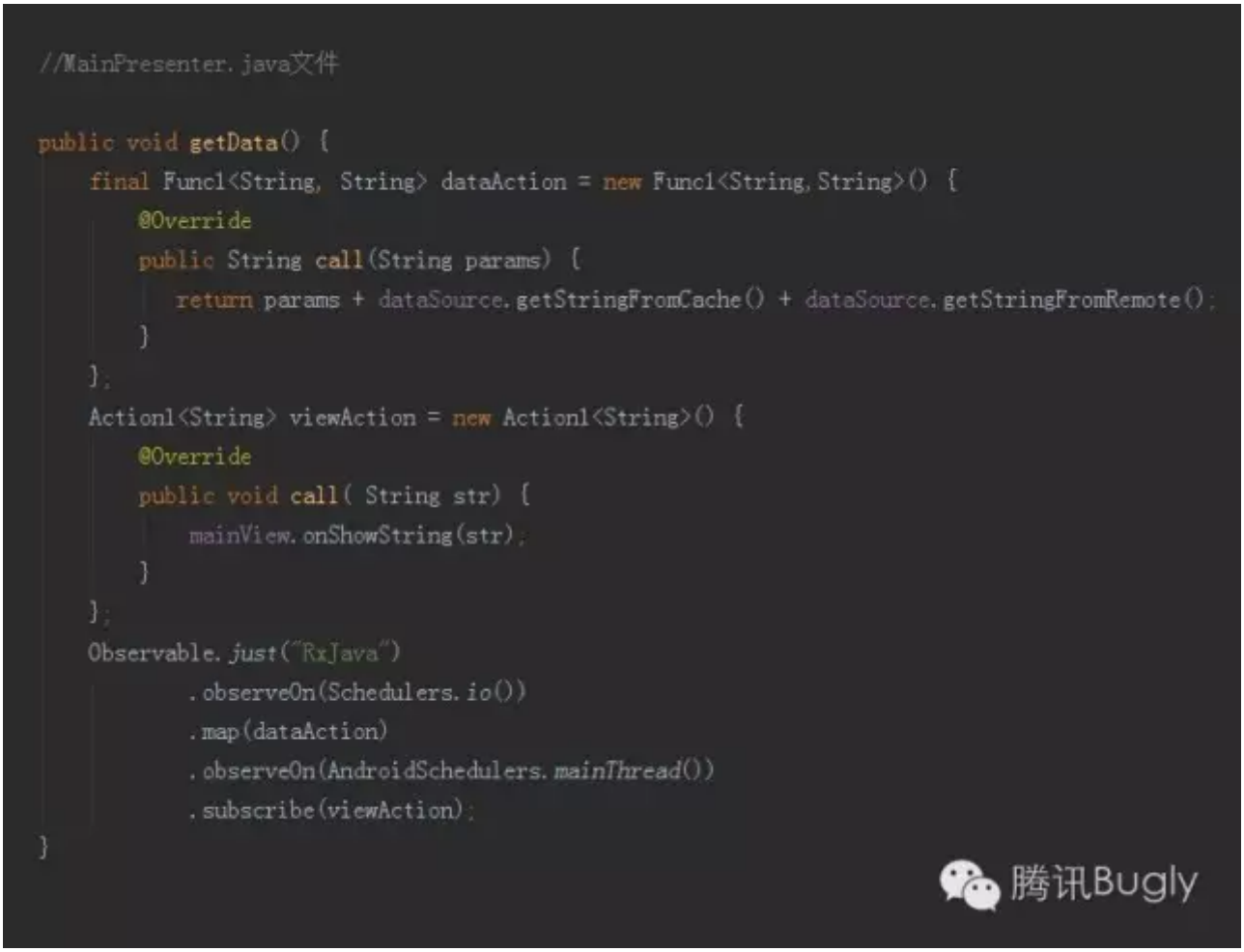
好在有了RxJava ,可以比较方便的解决这个问题。

3. 使用RxJava来进行线程控制

RxJava是一个天生用来做异步的工具，相比AsyncTask,Handler等，它的优点就是简洁，无比的简洁。在Android中使用RxJava需要加入下面两个依赖。

```
compile 'io.reactivex:rxjava:1.0.14'
compile 'io.reactivex:rxandroid:1.0.1'
```

这里我们直接介绍如何使用RxJava解决这个问题，在presenter中修改方法getData()。



简单解释一下，dataAction是我们的数据业务逻辑，viewAction是界面的显示逻辑，通过RxJava的传递和变换，dataAction会在由RxJava管理的IO线程—Schedulers.io() 中执行，而viewAction则会在UI线程—AndroidSchedulers.mainThread()中执行。

RxJava当然不止这么简单，还有别的玩法，比方说进入一个界面的时候，需要先加载缓存的数据，然后再从网络获取更新的数据进行刷新。有的时候，可能还需要处理IO过程中的异常情况，加入RxJava的异常处理参数。

```
//MainPresenter.java 文件中的getData方法

    public void getData() {
        Func1<Boolean, String> dataAction = new Func1<Boolean,String>() {
            @Override
            public String call(Boolean isCache) {
                if(isCache) {
                    return dataSource.getStringFromCache();
                } else {
                    return dataSource.getStringFromRemote();
                }
            }
        };
        Action1<String> viewAction = new Action1<String>() {
            @Override
            public void call( String str) {
                mainView.onShowString(str);
            }
        };
        Observable.just(false,true)
                .observeOn(Schedulers.io())
                .map(dataAction)
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(viewAction,getDefaultErrorAction());
    }

    private Action1<Throwable> getDefaultErrorAction() {
        return new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                throwable.printStackTrace();
            }
        };
    }
```


RxJava的使用场景远不止这些，线程变换、数据变换、接口顺序依赖、接口并发请求这些要求对它来说都是小菜一碟。当然，有些同学可能觉得RxJava入手有些困难，代码也会变得不那么直观，但相信只要大家慢慢熟悉它之后，它就会变得无比讨人喜欢。

下面列出了一些常见的RxJava的常用场景，其实还有更多的其它功能等待着大家去挖掘。

1. 取数据先检查缓存的场景
2. 需要等到多个接口并发取完数据，再更新
3. 一个接口的请求依赖另一个API请求返回的数据

4. 界面按钮需要防止连续点击的情况
5. 响应式的界面
6. 复杂的数据变换

上面这些功能都可以通过RxJava来轻松完成。具体的使用就不再多讲了，大家可以参考下面的文章：(Google文章名就可以了)

1．给 Android 开发者的 RxJava 详解
2．RxJava 与 Retrofit 结合的最佳实践
3．RxJava使用场景小结
4．How To Use RxJava

结语

至此为止，通过MVP+RxJava的组合，我们已经构建出一个比MVC更灵活的Android项目开发框架，好处大概有以下几点:

1. 每层各自独立，通过接口通信
2. 实现与接口分离实现，不同场景（正式，测试）挂载不同的实现，方便测试写假数据
3. 所有的业务逻辑都在非UI线程中进行，最大限度减少IO操作对UI的影响
4. 使用RxJava可以将复杂的调用进行链式组合，解决多重回调嵌套问题

以上就是我今天的分享，内容上可能还有不足甚至不够好的地方，欢迎大家指出，一起讨论学习。

这里也顺便打个广告，欢迎大家下载腾讯动漫App，这里有最新最热的国漫日漫，支持正版，你我共享。

问答环节

Q1：对于这样的一个MVP项目，它里面的包结构怎样规划比较清晰合理呢？

> 包结构的通常分法有两种：一种是按功能模块分，把某一个功能的presenter, activity，view层接口放到一起；一种是按类型分，P层M层和V层分成三个包。实际项目应用，我个人倾向于第一种，这种无论是开发过程，还是排查问题都会方便很多。当然，不同的项目还是有不同的分法的，不一而论。

Q2：耗时操作可能引起的内存泄露问题，请问是如何处理的。
Q3：用mvp时，请问你们在哪里释放一些引用，防止内存泄露的
Q4：p持有v的引用，请问怎么解决Activity的内存泄露问题？
Q5：网特别慢的时候，应用退出，但网络请求还没结束，p层回调持有上下文造成内存泄露，一般怎么解决啊。

> 这几个问题其实比较类似，我们在实际项目中，presenter会随着activity的生命周期进行销毁，比如在onDestroy方法中对presenter进行置空和引用解绑， 当然我们可以给所有的Presenter写一个共有父类BasePresenter，专门来处理这个问题。

Q6：需求包含列表页的时候，列表项也是按照mvp的思想来分层，还是封装成模块比较合适

> 目前我们的做法是直接封装成模块，简单的问题不宜过度设计

Q7：想问一下腾讯动漫这个app目前用的就是您讲的这个架构吗，在实际用的过程中有遇到什么问题吗

> 是的，我们已经使用了这个架构。实际使用过程中，经常会纠结的问题是业务逻辑层要不要再次独立分层。

Q8：项目中做测试是好事，但我觉得建议去掉TestImpl测试文件。如果项目打包时，打到包里，会导致包变大，这种测试建议用node写个简单的服务，不知道嘉宾你咋看？

> 是的。正式项目中，可以通过注解，或者proguard或者gradle的配置将这些测试文件不打到包里。Node写服务的话是不是又要搭环境，这里的做法就是不使用任何外部环境依赖。

Q9：mvp一般都是activity和Fragment加入presenter层，那么列表adapter里的逻辑是否也要加上presenter层呢

> Adapter其实跟View更接近的一个东西，它是用来处理重复显示问题。一般来说，我们传给adapter的数据完好能直接显示的，建议在业务逻辑层将数据拼装好再传进去。
> 答：Adapter其实跟View更接近的一个东西，它是用来处理重复显示问题。一般来说，我们传给adapter的数据完好能直接显示的，建议在业务逻辑层将数据拼装好再传进去。

Q10：我们项目中采用了MVP但是没有用RxJava，m与p层采用回调方式，这样m通过回调间接引用p，p层有v的引用。如果在网络情况不好频繁打开关闭页面在网络请求结束前是否会有内存泄漏问题。rx是否能解决这个问题。还有当网络结束回调时v对应activity destory了怎么办。每次都去判断activity状态吗？

> Rx不能解决内存泄漏的问题，前面2.3.7问题都提到了，通常的做法是在activity层销毁的时候进行解绑。回调时activity destory的话，我们现在的做法是对view层接口进行一次空值判定。如果有更好的办法，也欢迎大家提出来讨论

Q11：有时候例如自定义view依赖于服务器返回的model，里面也有很多根据model属性去绘制的过程，这种情况怎么处理？在P层抛出一个model的get方法吗？

> 自定义的View跟Activity一样，我们统称为View层。上面的例子中View层只有一个接口MainView，实际项目中，View层可能会实现好几个接口。对一个经常会被利用的自定义View，会额外给它新建一个接口。

Q12：你的例子中p层实现中getDate()方法对数据进行了处理，是否m层只是单纯的获取原始数据，对于数据上的业务也放入到p层中处理，有没有好的方式能够复用有关数据业务的这块逻辑

> 嗯，这个问题我们确实也遇到了。在项目实际操作过程中，如果有比较复杂业务流程，我会单独再分离出一层业务层，业务层再去调用dataSource取数据。如果只是单纯的取数据展示，现在这样就够了，尽量避免过度设计。

Q13：为了更好的解偶每一层，你们用MVP时 是否每层都有自己的数据结构，如果有的话，层与层之间的数据结构转换开销大不大？

> 目前来讲，大部分的业务都是一个数据结构穿透使用的，偶尔会有数据结构重新封装， 影响不大。我个人判断的话，相比IO处理，数据结构的转换开销还是小的，而且，如果有很多复杂转换的话，保证不要在UI线程中做，也不会太大问题。

Q14：activity与p层用接口的方式衔接的价值在哪？另外如何界定展现方法在哪调用？比如页面需要显示一个标题，内容是从之前页面传过来的，那是在activity接收后就直接显示？还是先传递到p层再回调activity的显示方法？感谢

> 价值在于，把presenter 与activity解耦之后，我可以在别的activity使用这个presenter层逻辑，也可以在这个activity 里调用其它页面的presenter方法。如果是前页传过来的，直接显示就好，不做过度设计。

Q15：rxJava使用lamaba的语法格式的话貌似会将代码缩减很多，请问嘉宾有试过这种方式吗？这个对项目的性能会有什么影响吗？因为我试用过几次后一直出现oom的问题

> lambda表达式会让语法看起来更简洁，非常推荐使用。但我们的项目目前只能使用jdk 7，悲伤。如果后面我们有机会切换的话，可以再一起分享一下。

Q16：rxjava怎么实现队列像handler message那样，就是队列执行，不是并发执行？

> rxJava中的just方法和from方法都是以队列形式发出事件。我猜你想问的问题可能是:一个接口的请求依赖另一个API请求返回的数据，这就是嵌套回调问题。可以找下大头鬼Bruce的一篇文章，《RxJava使用场景小结》,里面有介绍的，这里不详细讨论了。

如果您觉得我们的内容还不错，就请扫描二维码赞赏作者并转发到朋友圈，和小伙伴一起分享吧~

阅读原文