# **建立一个属于自己的框架**

### **1、拥有一套属于自己的框架的重要性**

拥有属于自己的一套框架和代码库是非常重要的事情。这里面有个词语“属于”，嗯，没错，只有属于自己的事物才能真正掌控着并把它玩转起来。

一套框架、一套成熟的解决方案、一套成熟的代码库是你去谈项目或者和产品经理交涉时候的筹码，它有多少，你心中就有多少底，即使你的学习能力再强，也是不能胜过已经有的代码，因为商业项目讲求的是质量 + 效率，只有经过多次实践而不败的代码才是拥有高质量的，而不是把 demo 实现了就拿来用这么简单。

由于我没有一套属于自己的框架（或称成熟的架构）。所以在做这外包之前就开始写一属于自己的框架。既然是自己的东西，就无需强求高度集成丰富的组建，而是根据实际的业务需求而定制一套框架，所以，框架并不是能万能套用，而是要看需求、看情况而定，盲目使用“漂亮”的框架的后果之一就是造成项目的臃肿。

注：本文说的框架，指的是一应用的体系结构，如：应用骨架 + 网络访问 + 数据库存取 + 消息通知 +UI 显示。

##### **本次构建的框架主要用到以下的组建：**

1、Activity + Fragment

单个 Activity 控制多个 fragment，每个 fragment 充当控制器的角色，每个页面对应自己的控制器，控制器控制自己的 view（界面），view 用 java 实现，view 中也绑定自己对应的控制器，页面的跳转使用用 eventbus 通知 activity 进行跳转。

2、ORM 使用 Activeandroid

操作数据库还是使用 ORM 比较好，相对传统的 SqlHelper 更方便简单，不易出错，减少代码工作量。

3、网络请求使用 retrofit，中间自己做了个解析器

网络请求返回的数据自动用 gson 解析并赋入对应的数据结构 bean 类，和 ORM 的性质有点相似，因为 retrofit 只是网络请求并解析，并没有封装好回调到 UI 线程的消息通知机制，所以自己实现了一个。

如果这一切都只是放在 demo 的角度来实现的话，都是很简单的事情。但如果放在一个商业项目里面去实践，便会发现好多问题，这里指的不是单单我上面提到的，而是每一种技术或者解决方案，真的要拿到真实场景去实践才能测试到问题的存在。

在这里面主要探讨一下页面跳转的结构体，我已经快给多个 fragment、fragment 嵌套的问题搞死了。也终于明白到为什么 square 建议不要用 fragment。不过致命的 bug 都解决得七七八八。不过还是存在一些奇怪现象，这得再另外一篇博文去探讨了。用 fragment 始终比用多个 activity 跳转这个老方法好多了。

### **2、使用 Fragment 实现页面跳转方法**

单一 Activity，布局仅需要一个 FrameLayout。

页面切换的代码如下:

// 最底层的页面  
private void setFragment(Fragment mTargetFragment) {  
 FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();  
 transaction  
 .replace(mMainLayout.getId(), mTargetFragment, mTargetFragment.getClass().getName())  
 .setTransitionStyle(FragmentTransaction.TRANSIT\_FRAGMENT\_FADE)  
 .commit();  
}  
  
// 弹出页面  
public void popFragmant(Fragment from, Fragment to) {  
 FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();  
 transaction.hide(from)  
 .add(mMainLayout.getId(), to, to.getClass().getName())  
 .addToBackStack(to.getClass().getName())  
 .commit();  
}  
  
// 弹出页面， dialog 形式  
public void popFragmant(Fragment to) {  
 FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();  
 if (!to.isAdded()) {  
 transaction  
 .add(mMainLayout.getId(), to, to.getClass().getName())  
 .addToBackStack(to.getClass().getName())  
 .commit();  
 }  
}  
  
// 关闭页面  
public void closeFragment(Fragment mTargetFragment) {  
 FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();  
 transaction  
 .remove(mTargetFragment)  
 .commit();  
 getSupportFragmentManager().popBackStack();  
}  
  
// 关闭所有页面  
public void closeAllFragment() {  
 int backStackCount = getSupportFragmentManager().getBackStackEntryCount();  
 for (int i = 0; i < backStackCount; i++) {  
 int backStackId = getSupportFragmentManager().getBackStackEntryAt(i).getId();  
 getSupportFragmentManager().popBackStack(backStackId, FragmentManager.POP\_BACK\_STACK\_INCLUSIVE);  
 }  
}  
  
// 最底层的页面  
public void onEvent(Event.SetFragmentEvent event) {  
 setFragment(event.mFragment);  
}  
  
// 弹出页面  
public void onEvent(Event.OpenFragmentEvent event) {  
 popFragmant(event.fromFragment, event.toFragment);  
}  
  
// 关闭页面  
public void onEvent(Event.CloseFragmentEvent event) {  
 closeFragment(event.mFragment);  
}  
  
// 关闭所有页面  
public void onEvent(Event.CloswAllFragmentEvent event) {  
 closeAllFragment();  
}  
  
// 弹出页面， dialog 形式  
public void onEvent(Event.PopFragment event) {  
 popFragmant(event.toFragment);  
}

在 Activity 的 onCreate 方法里面初始化并显示第一个页面：

mMainLayout = new FrameLayout(this);  
mMainLayout.setId(1);  
setContentView(mMainLayout);  
mWelcomeFragment = new WelcomeFragment();  
setFragment(mWelcomeFragment);

设置最底层页面，显示第一个 welcome 页面，以及 welcome 页面之后跳转到的的主页，这种情况就要把底层页面替换掉，则要用上面的setFragment(Fragment mTargetFragment)，这里面FragmentTransaction调用了它里面的replace方法，所以被替换的页面会被销毁，新换上的页面则是最底层的页面。

在底层页面 A 上面弹出一个页面 B，则需要popFragmant(Fragment from, Fragment to)方法。

transaction.hide(from)  
 .add(mMainLayout.getId(), to, to.getClass().getName())  
 .addToBackStack(to.getClass().getName());

这里只是把页面 A 隐藏掉（hide），并且加入到回退栈里。

但如果想弹出一个类似于弹窗这样的页面，即有透明部分的一个页面，则不能调用hide方法，否则页面 A 会被隐藏掉，这样的话透明就不存在意义了。

##### **这样做的好处有：**

##### **1、页面可实例化**

##### **2、可设置回调**

##### **3、更易于测试**

### **3、页面，逻辑分离**

我在 Fragment 里面我做了哪些事情？

Fragment 在这里充当的角色是一个控制器。

由于我是用 JAVA 布局界面，所以我会在 Fragment 里面实例化我的界面类，并在onCreateView return 这个界面，如：

private WelcomeView mWelcomeView;  
  
@Override  
public void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 mWelcomeView = new WelcomeView(this);  
}  
  
@Override  
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {  
 return mWelcomeView;  
}

这样我就可以在WelcomeView里面做布局界面的工作，并同时实现界面的逻辑，这样就不像在 xml 里面纯实现布局，而界面逻辑还需要在 Activity 去 findviewbyid，再去写界面逻辑。

注，上面说的是界面逻辑，而不是数据获取、处理、展示、以及跳转的逻辑。这些逻辑是放在 fragment 里面去做的。如 Fragment 里面实现一个从网络获取数据的方法，获取、处理后，把要展示的数据通过 View 的实例（在这里我们不是有 View 的实例吗）设置到相关的方法，这样就很好做到逻辑和界面分离。

### **4、遇到的问题**

由于多 fragment 嵌套导致的一个问题，onActivityResult不能成功被调用，这里我也没找到真正的原因，只是找了个解决方法。

@Override  
public void onActivityResult(int requestCode, int resultCode, Intent data) {  
 mCurrentFragment.onActivityResult(requestCode, resultCode, data);  
 super.onActivityResult(requestCode, resultCode, data);  
}

在父 fragment 实现onActivityResult方法，然后在里面调用子 fragment 的onActivityResult即可。