MVC、MVP、MVVM，谈谈我对Android应用架构的理解

近日，美国国务院公布了一条新规，要求外国人在申请美国签证时提供过去5年在指定社交媒体平台上使用的用户名。据美国《纽约时报》报道，新规针对的指定社交媒体平台有20家，其中大部分设在美国，包括脸书、推特、领英、优兔等。此外，新规还涉及设在中国的豆瓣、QQ、新浪微博、腾讯微博和优酷视频以及俄罗斯社交媒体平台VK、比利时交友平台Twoo和拉脱维亚社交问答平台Ask．fm。

周一早上好，清明节小长假即将来临咯，但是大家仍然要保持好好学习工作哦。

本篇来自 08_carmelo 的投稿，分享了他对 Android应用架构的理解 ，一起来看看！希望大家喜欢。

08_carmelo 的博客地址：

> https://www.jianshu.com/u/b8dad3885e05

android架构可能是论坛讨论最多的话题了，mvc mvp和mvvm不绝于耳，后面又有模块化和插件化。对此，关于哪种架构更好的争论从未停止。

我的观点：脱离实际项目比较这些模式优劣毫无意义，各种模式都有优点和缺点，没有好坏之分。越高级的架构实现起来越复杂，需要更多的学习成本更多的人力，所以说技术选型关键是在你自己项目的特点，团队的水平，资源的配备，开发时间的限制，这些才是重点！但是不少团队本末倒置，把mvvm往自己的项目硬套。

下面我从两大块讲下我理解的Android架构：代码层面，主要是MVC和MVP的理解。项目层面，主要是怎么搭建整个项目，怎么划分模块。

先上结论：

- MVC：Model-View-Controller，经典模式，很容易理解，主要缺点有两个：
    - View对Model的依赖，会导致View也包含了业务逻辑；
    - Controller会变得很厚很复杂。
- MVP：Model-View-Presenter，MVC的一个演变模式，将Controller换成了Presenter，主要为了解决上述第一个缺点，将View和Model解耦，不过第二个缺点依然没有解决。
- MVVM：Model-View-ViewModel，是对MVP的一个优化模式，采用了双向绑定：View的变动，自动反映在ViewModel，反之亦然。

MVC

简单的说：我们平时写的Demo都是MVC，controller就是我们的activity，model（数据提供者）就是读取数据库，网络请求这些我们一般有专门的类处理，View一般用自定义控件。

但这一切，只是看起来很美。

想象实际开发中，我们的activity代码其实是越来越多，model和controller根本没有分离，控件也需要关系数据和业务。

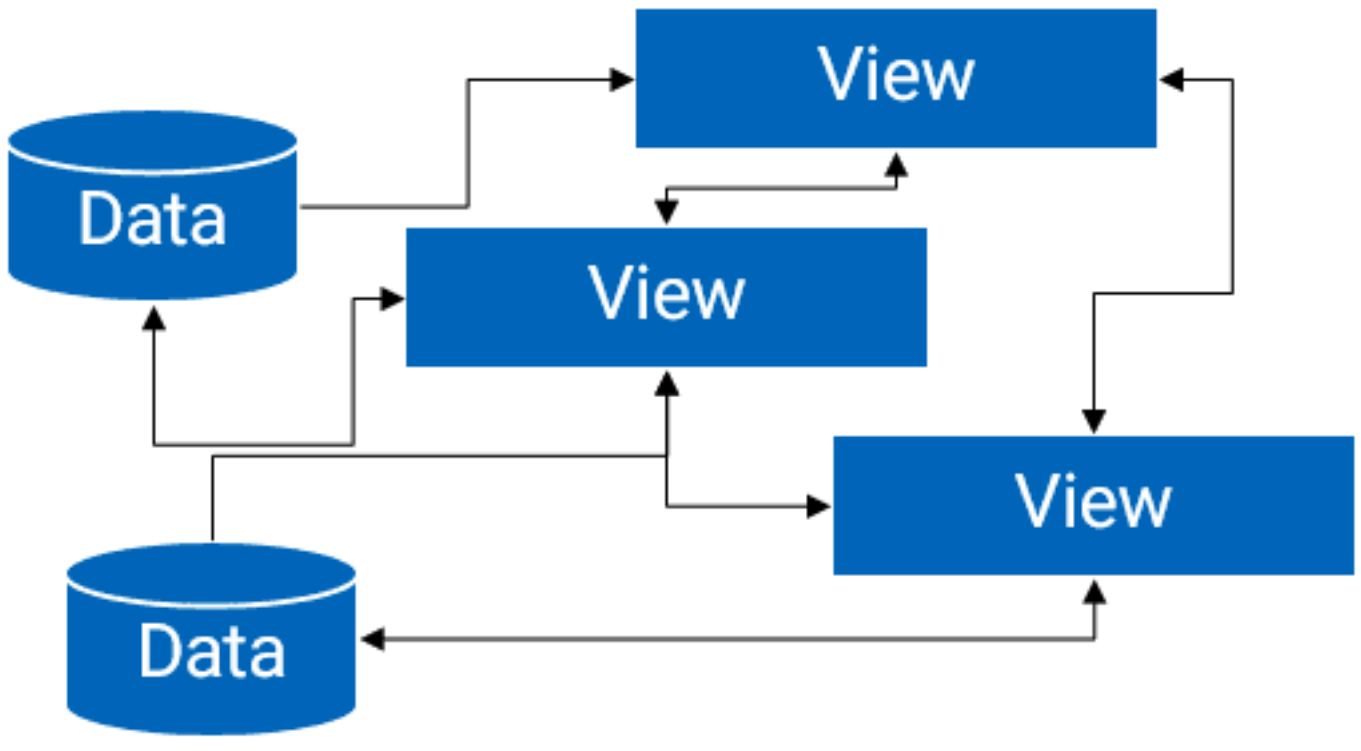

image.png

所以说，MVC的真实存在是MC（V），Model和Controller根本没办法分开，并且数据和View严重耦合。这就是它的问题。举个简单例子： 获取天气数据展示在界面上

image.png

- Model层

```
public interface WeatherModel {
    void getWeather(String cityNumber, OnWeatherListener listener);
}
```

```
.................
public class WeatherModelImpl implements WeatherModel {
        @Override
    public void getWeather(String cityNumber, final OnWeatherListener liste
        /*数据层操作*/
        VolleyRequest.newInstance().newGsonRequest(http://www.weather.com.c
                Weather.class, new Response.Listener<weather>() {
                    @Override
                    public void onResponse(Weather weather) {
                        if (weather != null) {
                            listener.onSuccess(weather);
                        } else {
                            listener.onError();
                        }
                    }
                }, new Response.ErrorListener() {
                    @Override
                    public void onErrorResponse(VolleyError error) {
                        listener.onError();
                    }
                });
    }
}
```

- Controllor（View）层

```
public class MainActivity extends ActionBarActivity implements OnWeatherLis
    private WeatherModel weatherModel;
    private EditText cityNOInput;
    private TextView city;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        weatherModel = new WeatherModelImpl();
        initView();
    }
    //初始化View
    private void initView() {
        cityNOInput = findView(R.id.et_city_no);
        city = findView(R.id.tv_city);
        ...
        findView(R.id.btn_go).setOnClickListener(this);
    }
    //显示结果
    public void displayResult(Weather weather) {
```

```
        WeatherInfo weatherInfo = weather.getWeatherinfo();
        city.setText(weatherInfo.getCity());
        ...
    }
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.btn_go:
                weatherModel.getWeather(cityNOInput.getText().toString().tr
                break;
        }
    }
    @Override
    public void onSuccess(Weather weather) {
        displayResult(weather);
    }
    @Override
    public void onError() {
        Toast.makeText(this, 获取天气信息失败, Toast.LENGTH_SHORT).show();
    }
    private T findView(int id) {
        return (T) findViewById(id);
    }
}
```

简单分析下这个例子：

- activity里面的控件必须关心业务和数据，才能知道自己怎么展示。换句话说，我们很难让两个人在不互相沟通的情况下，一人负责获取数据，一人负责展示UI，然后完成这个功能。

- 所以的逻辑都在activity里面。

完美的体现了MVC的两大缺点，下面看看MVP怎么解决第一个缺点的

MVP

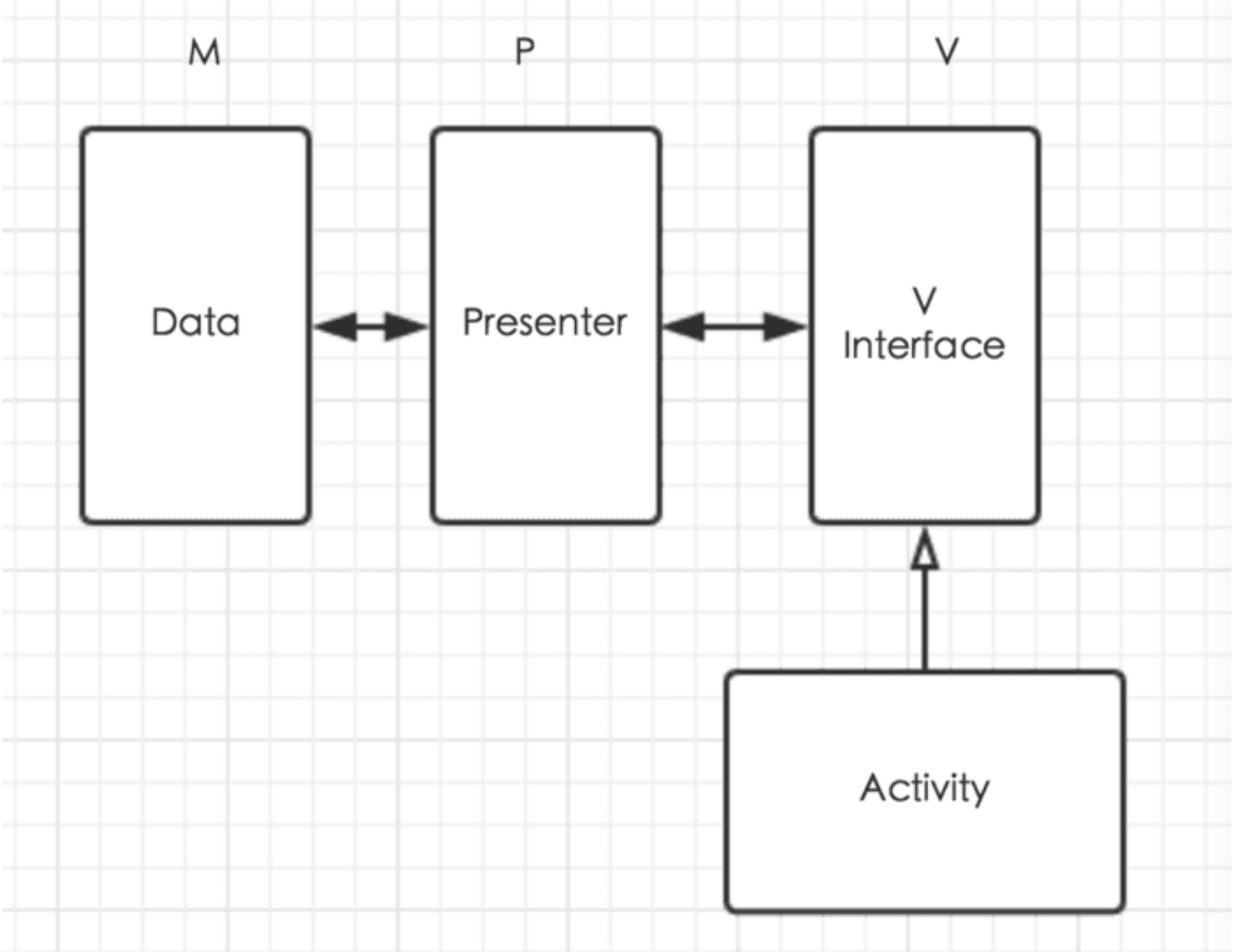

image.png

看上图可以看出，从MVC中View被拆成了Presenter和View，真正实现了逻辑处理和View的分离。下面写一个实例：模拟一个登录界面，输入用户名和密码，可以登录以及清除密码

- Model层

```
/**
定义业务接口
*/
public interface IUserBiz {
    public void login(String username, String password, OnLoginListener log
}
/**
结果回调接口
*/
public interface OnLoginListener {
    void loginSuccess(User user);
    void loginFailed();
```

```
}
/**
具体Model的实现
*/
public class UserBiz implements IUserBiz {
    @Override
    public void login(final String username, final String password, final O
    {
        //模拟子线程耗时操作
        new Thread()
        {
            @Override
            public void run()
            {
                try
                {
                    Thread.sleep(2000);
                } catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
                //模拟登录成功
                if ("zhy".equals(username) && "123".equals(password))
                {
                    User user = new User();
                    user.setUsername(username);
                    user.setPassword(password);
                    loginListener.loginSuccess(user);
                } else
                {
                    loginListener.loginFailed();
                }
            }
        }.start();
    }
}
```

- **View**

上面说到View层是以接口的形式定义，我们不关心数据，不关心逻辑处理！只关心和用户的交互，那么这个登录界面应该有的操作就是（把这个界面想成一个容器，有输入和输出）。

获取用户名，获取密码，现实进度条，隐藏进度条，跳转到其他界面，展示失败dialog，清除用户名，清除密码。接下来定义接口：

```
public interface IUserLoginView  {
    String getUserName();
    String getPassword();
    void clearUserName();
    void clearPassword();
    void showLoading();
    void hideLoading();
    void toMainActivity(User user);
    void showFailedError();
}
```

然后Activity实现这个这个接口：

```
public class UserLoginActivity extends ActionBarActivity implements IUserLo
    private EditText mEtUsername, mEtPassword;
    private Button mBtnLogin, mBtnClear;
    private ProgressBar mPbLoading;
    private UserLoginPresenter mUserLoginPresenter = new UserLoginPresenter
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_user_login);
        initViews();
    }
    private void initViews()
    {
        mEtUsername = (EditText) findViewById(R.id.id_et_username);
        mEtPassword = (EditText) findViewById(R.id.id_et_password);
        mBtnClear = (Button) findViewById(R.id.id_btn_clear);
        mBtnLogin = (Button) findViewById(R.id.id_btn_login);
        mPbLoading = (ProgressBar) findViewById(R.id.id_pb_loading);
        mBtnLogin.setOnClickListener(new View.OnClickListener()
        {
            @Override
            public void onClick(View v)
            {
                mUserLoginPresenter.login();
            }
        });
        mBtnClear.setOnClickListener(new View.OnClickListener()
        {
            @Override
            public void onClick(View v)
            {
                mUserLoginPresenter.clear();
```

```
        }
    });
}
@Override
public String getUserName()
{
    return mEtUsername.getText().toString();
}
@Override
public String getPassword()
{
    return mEtPassword.getText().toString();
}
@Override
public void clearUserName()
{
    mEtUsername.setText("");
}
@Override
public void clearPassword()
{
    mEtPassword.setText("");
}
@Override
public void showLoading()
{
    mPbLoading.setVisibility(View.VISIBLE);
}
@Override
public void hideLoading()
{
    mPbLoading.setVisibility(View.GONE);
}
@Override
public void toMainActivity(User user)
{
    Toast.makeText(this, user.getUsername() +
            " login success , to MainActivity", Toast.LENGTH_SHORT).sho
}
@Override
public void showFailedError()
{
    Toast.makeText(this,
            "login failed", Toast.LENGTH_SHORT).show();
}
}
```

- Presenter

Presenter的作用就是从View层获取用户的输入，传递到Model层进行处理，然后回调给View层，输出给用户！

```
public class UserLoginPresenter {
    private IUserBiz userBiz;
    private IUserLoginView userLoginView;
    private Handler mHandler = new Handler();
//Presenter必须要能拿到View和Model的实现类
    public UserLoginPresenter(IUserLoginView userLoginView)
    {
        this.userLoginView = userLoginView;
        this.userBiz = new UserBiz();
    }
    public void login()
    {
        userLoginView.showLoading();
        userBiz.login(userLoginView.getUserName(), userLoginView.getPasswor
        {
            @Override
            public void loginSuccess(final User user)
            {
    //需要在UI线程执行
    mHandler.post(new Runnable()
    {
        @Override
        public void run()
        {
            userLoginView.toMainActivity(user);
            userLoginView.hideLoading();
        }
    });
}
@Override
public void loginFailed()
{
    //需要在UI线程执行
    mHandler.post(new Runnable()
    {
        @Override
        public void run()
        {
            userLoginView.showFailedError();
            userLoginView.hideLoading();
        }
    });
```

```
            }
        });
    }
    public void clear()
    {
        userLoginView.clearUserName();
        userLoginView.clearPassword();
    }
}
```

分析下这个例子:

- 我们有了IUserLoginView 这个接口（协议），activity里面的控件根本不需要关心数据，只要实现这个接口在每个方法中“按部就班”的展示UI就行了。换句话说，我们让两个人一起开发这个功能，一人要处理数据并且制定接口（协议），另一人直接用activity实现这个接口，闭着眼睛就可以在每个回调里展示UI，合作很愉快。

- MVP成功解决了MVC的第一个缺点，但是逻辑处理还是杂糅在Activity。

MVC到MVP简单说，就是增加了一个接口降低一层耦合。那么，用样的MVP到MVVM就是再加一个接口呗。实际项目我建议用MVP模式，MVVM还是复杂了对于中小型项目有点过度设计，这里就不展开讲。

模块化

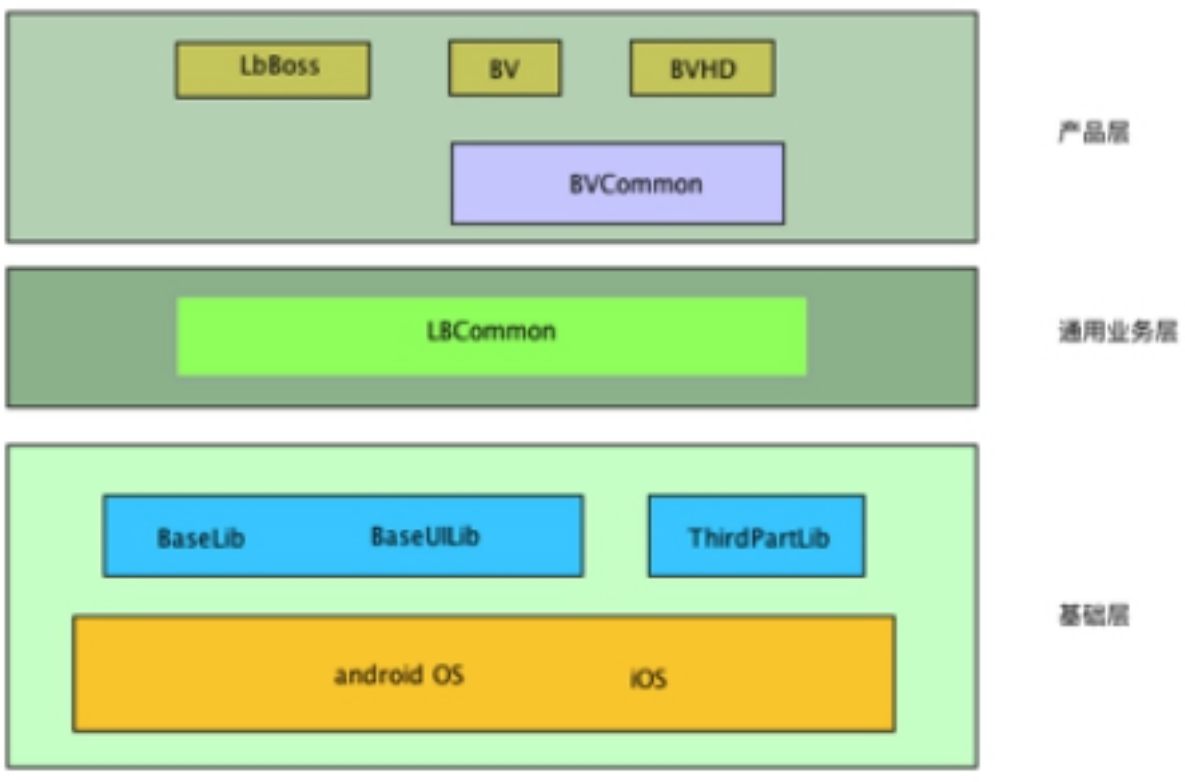

image.png

上图是一个项目常见的架构方式

- 最底层是基础库，放置与业务无关的模块：比如基础网络请求，图片压缩等等，可以按需分为逻辑模块，通用UI模块和第三方库。（建议采用独立的svn分支）
- 中间层是通用业务层，放置公司多个android项目的通用业务模块（和业务相关的），比如登录流程，文件上传/下载等。
- 最上层就是应用层了，比如公司有三个android项目：LbBoss,BV和BVHD。我们还可以针对相似的项目再抽取通用层（比如这里的BV和BV PAD版，通用层为BVCommon）。

新建一个app，我们往往有两种模块划分方法：

- 按照类型划分

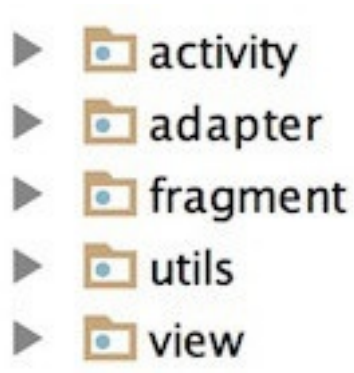

image.png

- 按照业务划分

每一个包都是一个业务模块，每个模块下再按照类型来分。

- 怎么选

我建议中小型的新项目按照类型比较好，因为开始代码量不多按照业务来分不切实际，一个包只放几个文件？？ 况且前期业务不稳定，等到开发中期业务定型了，再进行重构难度也不大。

上面讲的模块划分既不属于模块化也不属于插件化，仅仅是一个简单package结构不同而已,app还是一个app并没有产生什么变化。通常讲的模块

化，是指把业务划分为不同的moduler（类型是library），每个moduler之间都不依赖，app(类型是application)只是一个空壳依赖所有的moduler。

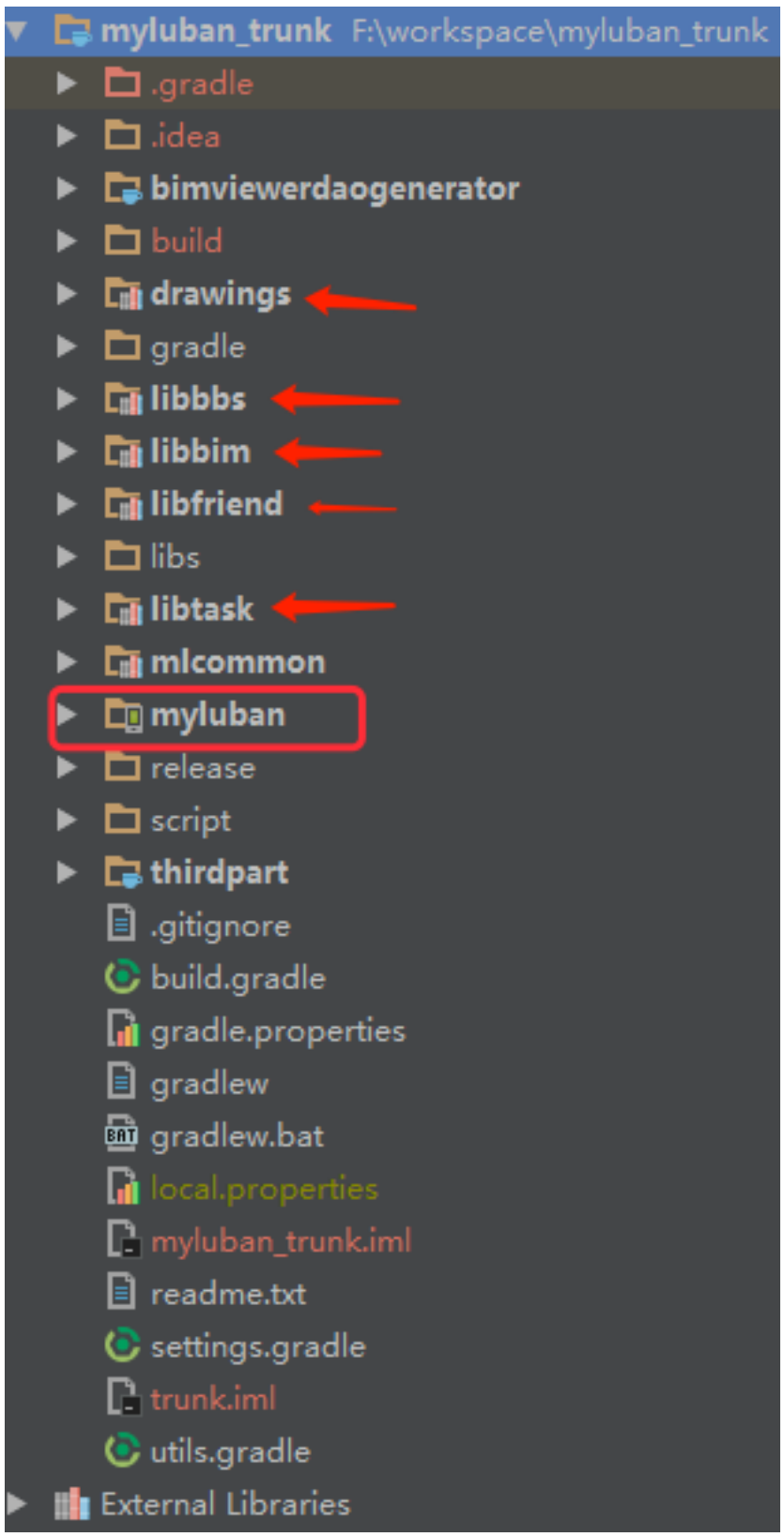

image.png

每个红色箭头都是一个业务模块，红色框是我们的app里面只包含简单的业务：自定义Application，入口Activity，build.gradle编译打包配置。看下项目的依赖关系：

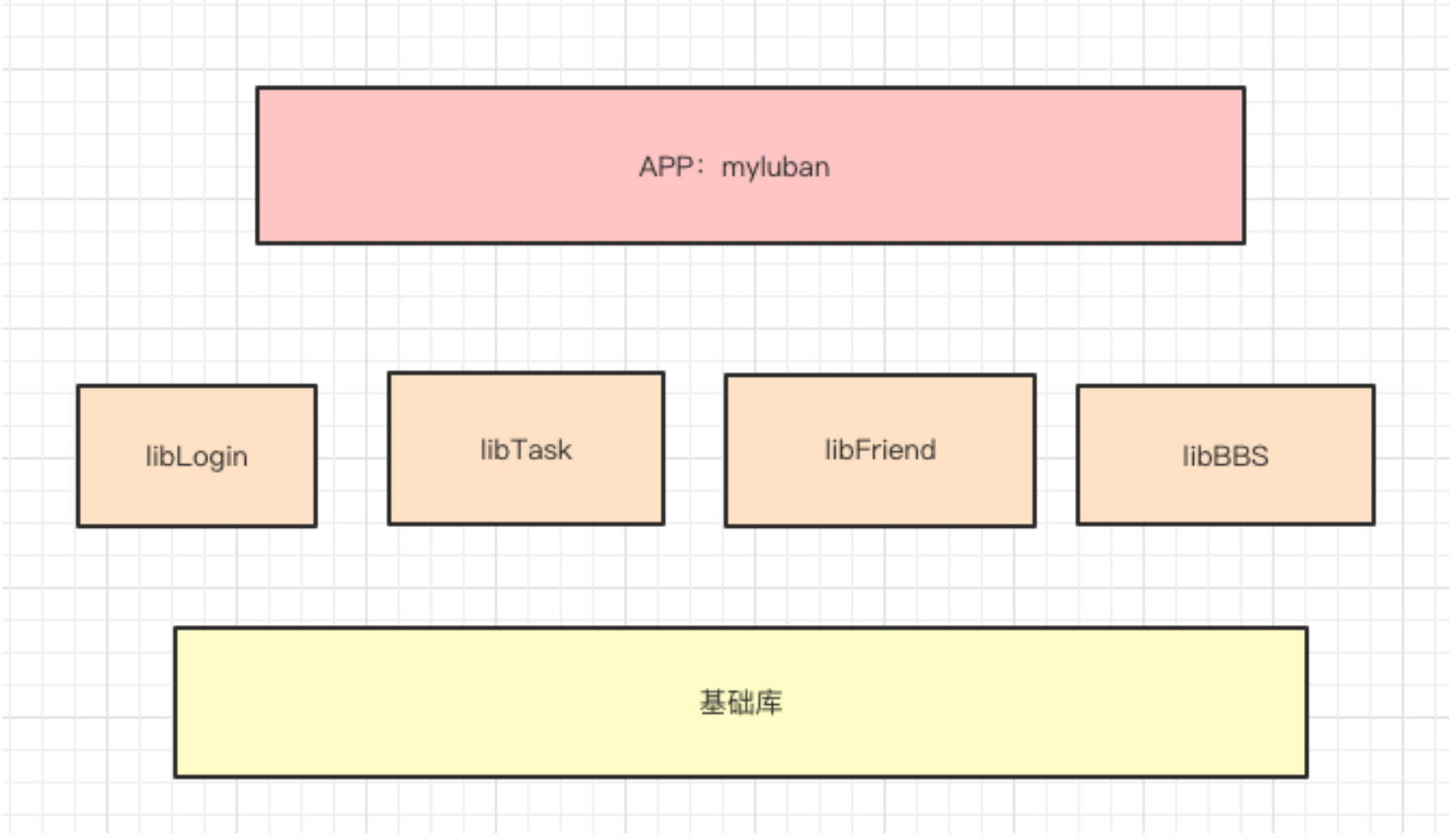

image.png

这样架构后，带来最大的不同就是：不同业务模块完全分离，好处就是不同模块的开发绝对不会互相耦合了，因为你在模块A 根本访问不到模块B的API。此时模块间通信急需解决，Intent隐式跳转可以处理部分Activity的跳转，但真正的业务场景远不止两个界面跳一跳。你之前封装的业务通用方法，工具类，数据缓存现在其他模块都拿不到了，本本来可以复用的控件，fragment都不能共享，而这些都是和业务耦合没办法拿到底层基础库。

模块间通信

针对上面问题有两个解决办法，根据自己项目实际情况，如果项目的前期搭建已经很优秀，有完善的基础库，不同模块间的通信不是很多，可以自己实现。如果项目比较庞大，不同业务间频繁调用建议使用阿里巴巴的开源库。

自己实现

首先每个moduler有个目录叫include，里面有三个类，此处以一个bbs论坛模块为例说明，

- IBBSNotify：里面是一堆interface，作用是该模块对外的回调，只能被动被触发。

- IBBService：里面是一堆interface，作用是对外暴露的方法，让别的模块来主动调，比如enterBbsActivity

- IBBSServiceImpl：很明显是IBBService的实现，比如enterBbsActivity就是具体怎么跳转到论坛界面，传递什么数据。

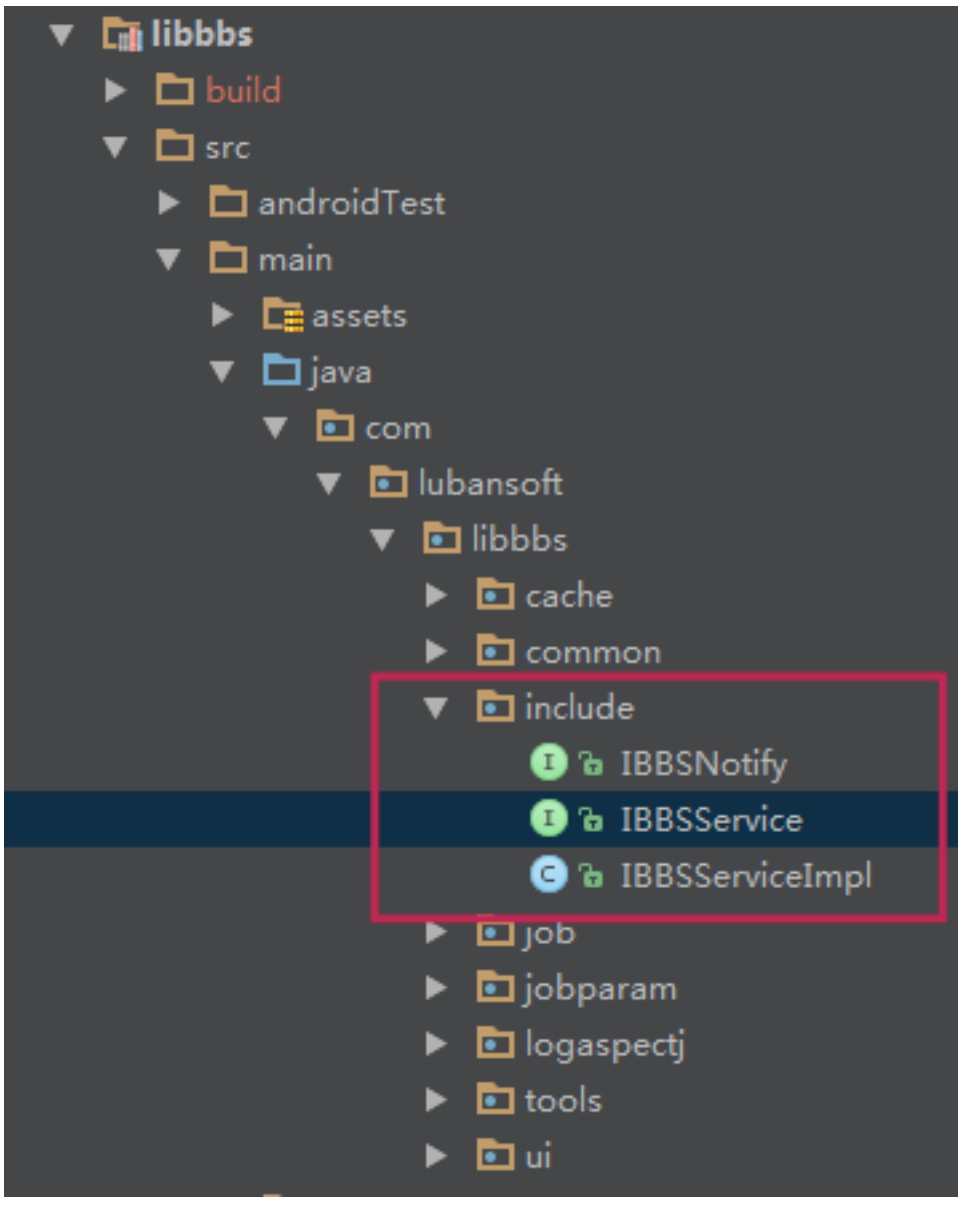

image.png

每个模块方法和回调都有了，in和out都具备了，别的模块怎么使用呢？就该app该上场了，app不能只是一个壳里面要定义一个ModulerManager implements 所有模块的对外interface，作为每个模块的中转站，A模块告诉ModulerManager我想跳转到论坛模块，接着ModulerManager调用IBBService.enterBbsActivity，IBBSServiceImpl是IBBService的具体实现（多态）然后调用IBBSServiceImpl.enterBbsActivity跳转到BBS界面。

通信是解决了，其实踩坑才刚刚开始：

- 这里的app是我们新建的，那么之前项目的app模块要降为library：

```
apply plugin: 'com.android.library'
```

app的build.gradle配置：

```
apply plugin: 'com.android.application'
```

性质发生巨大变化。里面的自定义application，build.gradle，代码混淆配置等全部移到app

- R.java在Lib类型的moduler中不是final的，所有switch case语句全部替换成if else
- 一定要再建一个common模块，放置通用数据，缓存等
- 还有很多通用功能，例如分享，推送，尽量剥离业务放到common
- 其他与项目相关的细节

插件化其实最后发布的产品也是一个apk，只不过大小可以控制（可以随意去掉某些模块），支持用户动态加载子apk。因此，插件化就是动态加载apk。有人说我用intent隐式可以直接跳转到另一个apk啊，干嘛还要插件化。

其实是两码事，intent只是指定一个Activity跳过去，后面的交互完成不受你控制，2个apk也是运行在独立的进程数据无法共享。而插件化可以让两个apk运行在一个进程，可以完全像同一个apk一样开发。不过，我觉得插件化只适合需要多部门并行开发的那种，比如支付宝这种超级app，一般的app开发除非特殊需要，否则用不到。

插件化也有成熟的框架，在此不详细说了。另外，每个人的习惯不一样，组件化，模块化在我看来差不多，没必要纠结两个名词。

欢迎长按下图 -> 识别图中二维码

或者 扫一扫 关注我的公众号