MVP

Callback接口回调调用P方法

Model

Presenter

Activity/fragment

View抽象接口控制界面.如:showbackgroud()来控制activity界面变化方法

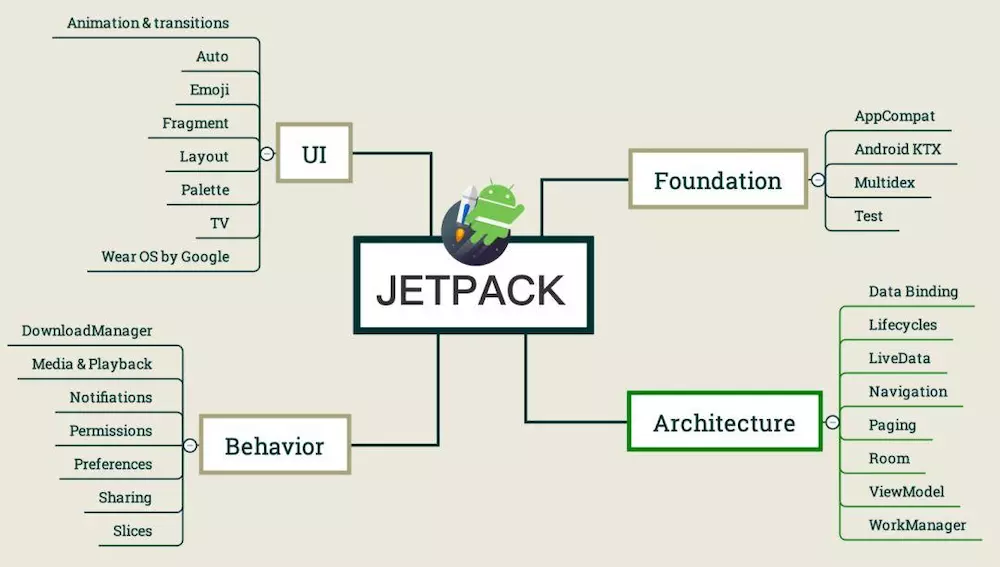
调用获得数据

调用让P去请求数据

为了避免一个activity对应一个presenter,presenter不能是内部属性形式,而应该是参数传入

MVP和MVVM区别

MVVM和MVP概念上的差别不是很大,只是实践上换了形式.使用数据驱动的方式和一些新的组件来优化,避免MVP繁杂的代码,提升拓展性



Android Jetpack是谷歌公司为了更快更好地开发 App，推出了一系列组件.Jetpack 不属于 Android Framework，不是 Android 开发的必需品，它只是应用层开发的一种辅助手段，帮我们解决了一些常见问题，比如版本兼容、API 易用性、生命周期管理等。

其中 Architecture 部分的组件（Android Architecture Components，以下简称 AAC）组合起来形成了一套完整的架构解决方案,也是目前(2019年)最好的Android 架构领域的最佳实践

**Lifecycle**

Lifecycle 组件指的是 android.arch.lifecycle 包下提供的各种类与接口，可以让开发者构建能感知其他组件（主要指Activity 、Fragment）生命周期（lifecycle-aware）的类。

在MVP中,我们要监控activity的生命周期,提供给数据,需要在presenter中写下activity生命周期的映射方法onCreate()/onResume()等,并在activity各个生命周期方法下调用P的这些映射方法

lifycycle 其实是用观察者模式实现的，当 Activity 生命周期变化的时候，通知相应的 Observers 即观察者。

使用Lifecycle

在26后可直接使用,但在此之前的版本,要在build.gradle中添加

implementation "android.arch.lifecycle:extensions:1.0.0-alpha4"

如果activity是继承至AppCompatActivity ,可直接使用getLifecycle().addObserver,当 Activity 的生命周期变化的时候，将会回调 onStateChanged 的方法，状态分别是一一对应的.

如果activity是直接继承的activity,那么要implements  LifecycleOwner.而且还得实现LifecycleRegistry

1/我们自建的需要跟随activity生命周期的类

public class CycleObserver implements GenericLifecycleObserver {  
  
 @Override  
 public void onStateChanged(LifecycleOwner source, Lifecycle.Event event) {  
 Log.*d*("pjl++","LifecycleOwner:"+source+"\n"+"Lifecycle.Event:"+event);  
 }  
}

event输出:

Lifecycle.Event:ON\_CREATE

Lifecycle.Event:ON\_START

Lifecycle.Event:ON\_RESUME等

2/将我们自建的观察者,和activity建立订阅关系

Java:

getLifecycle().addObserver(new CycleObserver());

Kotlin:

*lifecycle*.addObserver(CycleObserver())

**LiveData**

LiveData具有的这种可观察性和生命周期感知的能力,且只有注册的组件处于active时才会接收通知,跟随组件的生命周期,且自动注册和解除注册,这些都优于eventbus和Rxbus

使用LiveData,先建立一个map集合来管理所有Livedata的通讯总线类

public final class LiveDataBus {  
 private final Map<String, BusMutableLiveData<Object>> bus;  
  
 private LiveDataBus() {  
 bus = new HashMap<>();  
 }  
  
 private static class SingletonHolder {  
 private static final LiveDataBus *DEFAULT\_BUS* = new LiveDataBus();  
 }  
  
 public static LiveDataBus get() {  
 return SingletonHolder.*DEFAULT\_BUS*;  
 }  
  
 public <T> MutableLiveData<T> with(String key, Class<T> type) {  
 if (!bus.containsKey(key)) {  
 bus.put(key, new BusMutableLiveData<>());  
 }  
 return (MutableLiveData<T>) bus.get(key);  
 }  
  
 public MutableLiveData<Object> with(String key) {  
 return with(key, Object.class);  
 }  
  
 private static class ObserverWrapper<T> implements Observer<T> {  
 private Observer<T> observer;  
  
 public ObserverWrapper(Observer<T> observer) {  
 this.observer = observer;  
 }  
  
 @Override  
 public void onChanged(@Nullable T t) {  
 if (observer != null) {  
 if (isCallOnObserve()) {  
 return;  
 }  
 observer.onChanged(t);  
 }  
 }  
  
 private boolean isCallOnObserve() {  
 StackTraceElement[] stackTrace = Thread.*currentThread*().getStackTrace();  
 if (stackTrace != null && stackTrace.length > 0) {  
 for (StackTraceElement element : stackTrace) {  
 if ("android.arch.lifecycle.LiveData".equals(element.getClassName()) && "observeForever".equals(element.getMethodName())) {  
 return true;  
 }  
 }  
 }  
 return false;  
 }  
 }  
  
 private static class BusMutableLiveData<T> extends MutableLiveData<T> {  
 private Map<Observer, Observer> observerMap = new HashMap<>();  
  
 @Override  
 public void observe(@NonNull LifecycleOwner owner, @NonNull Observer<T> observer) {  
 super.observe(owner, observer);  
 try {  
 hook(observer);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Override  
 public void observeForever(@NonNull Observer<T> observer) {  
 if (!observerMap.containsKey(observer)) {  
 observerMap.put(observer, new ObserverWrapper(observer));  
 }  
 super.observeForever(observerMap.get(observer));  
 }  
  
 @Override  
 public void removeObserver(@NonNull Observer<T> observer) {  
 Observer realObserver = null;  
 if (observerMap.containsKey(observer)) {  
 realObserver = observerMap.remove(observer);  
 } else {  
 realObserver = observer;  
 }  
 super.removeObserver(realObserver);  
 }  
  
 private void hook(@NonNull Observer<T> observer) throws Exception {  
 //get wrapper's version  
 Class<LiveData> classLiveData = LiveData.class;  
 Field fieldObservers = classLiveData.getDeclaredField("mObservers");  
 fieldObservers.setAccessible(true);  
 Object objectObservers = fieldObservers.get(this);  
 Class<?> classObservers = objectObservers.getClass();  
 Method methodGet = classObservers.getDeclaredMethod("get", Object.class);  
 methodGet.setAccessible(true);  
 Object objectWrapperEntry = methodGet.invoke(objectObservers, observer);  
 Object objectWrapper = null;  
 if (objectWrapperEntry instanceof Map.Entry) {  
 objectWrapper = ((Map.Entry) objectWrapperEntry).getValue();  
 }  
 if (objectWrapper == null) {  
 throw new NullPointerException("Wrapper can not be bull!");  
 }  
 Class<?> classObserverWrapper = objectWrapper.getClass().getSuperclass();  
 Field fieldLastVersion = classObserverWrapper.getDeclaredField("mLastVersion");  
 fieldLastVersion.setAccessible(true);  
 //get livedata's version  
 Field fieldVersion = classLiveData.getDeclaredField("mVersion");  
 fieldVersion.setAccessible(true);  
 Object objectVersion = fieldVersion.get(this);  
 //set wrapper's version  
 fieldLastVersion.set(objectWrapper, objectVersion);  
 }  
 }  
}

注册订阅(观察者和被观察者关系建立):

前景:

MainActivity中的TextView的显示内容,依据SecondActivity中的editText的输入而改变

Java形式:

LiveDataBus.*get*().with("text\_str", String.class)  
 .observe(this, new Observer<String>() {  
 @Override  
 public void onChanged(@Nullable String aStr) {  
 //拿到key为"text\_str"的LiveData,强制携带String型数据,observe注册观察者,onChanged是收到被观察的LiveData改变时的响应,aStr是改变的数据  
 tv.setText(aStr);

}  
 });

Kotlin形式:

LiveDataBus.get().with("text\_str", String::class.*java*).observe(this, *Observer* **{**str **->** tv.*text*=str **}**)

发送消息:

LiveDataBus.*get*().with("text\_str").setValue(text);

完整版:https://github.com/JeremyLiao/LiveEventBus

**ViewModel**

ViewModel说是用来改善presenter的,但它更像是以前那些写在application下的配置元素.

ViewModel的生命周期,从第一次onCreate开始到finish()引起的销毁而结束,不会因为其期间因为屏幕翻转或其他原因导致的多次onCreate而重新创建.简单来说,就是防止系统回收app造成的数据丢失.

Viewmodel和livedata应该一起使用

Viewmodel,应该完全剥离界面,不应该出现android.\*这样的包,viewmodel不持有view的引用,通过livedata,让activity们来观察viewmodel的变化.

Viewmodel的使用

1/自建一个viewmodel类

public class MyViewModel extends ViewModel {  
  
 private String second\_text="this is second text";  
  
 public String getSecond\_text() {  
 return second\_text;  
 }  
}

2/在activity中使用viewmodel

viewModel=ViewModelProviders.*of*(this).get(MyViewModel.class);  
ed.setText(viewModel.getSecond\_text());

结合前面的LiveData操作,viewmodel写法

public class MyViewModel extends ViewModel {  
  
 public LiveData initLiveDataTestStr(){  
 return LiveDataBus.*get*().with("liveDataTestStr", String.class);  
 }  
  
 public void updataLiveDataTestStr(String liveDataTestStr){  
 LiveDataBus.*get*().with("liveDataTestStr").setValue(liveDataTestStr);  
 }  
}

在activity中使用viewmode

viewModel=ViewModelProviders.*of*(this).get(MyViewModel.class);  
viewModel.initLiveDataTestStr().observe(this,new Observer<String>() {  
 @Override  
 public void onChanged(@Nullable String s) {  
 //这里是观察者们的响应操作  
 }  
});

**Room**