

Android Architecture Components 开发架构



前行的乌龟 (/u/fb093dd92ed8) ✓ 已关注

2018.03.12 21:16* 字数 8440 阅读 787 评论 2 喜欢 15

(/u/fb093dd92ed8)

前言

Android Architecture Components, 简称 AAC, 是 Google IO 2017 大会新推出的 app 架构, 从使用感受上说是 MVVM 的官方加强, AAC 可以解决我们常见的一些内存泄露的场景使用, 添加了更易操作的 view 生命周期管理。

拓展一下, MVC / MVP / MVVM 这种代码架构组织方式都是基于: UI 驱动数据 (), 我们常用的 EventBus 框架则是: 数据驱动 UI ()。数据和 UI 谁驱动谁, 这其实就是我们架构的核心, 可以决定我们如何组织代码机构, 如何具体的书写代码。

仔细想了2天, 我说下自己的体会:

- UI 驱动数据 ()

我管这种方式叫: 主动持有型。大家仔细想想, 我们在页面中 findViewById 持有控件的对象, 然后给 UI 控件注册交互方法, 页面对象持有 Persenter 逻辑对象, Persenter 逻辑持有数据层对象, 这一层层对象之间都是通过上游对象调用下游对象的方法, 接受数据或是传入 callback 回调的方式, 实现这一层层上下游对象之间的交互的, 这一层层上下游对象拥有不同的生命周期, 相互支持, 就容易造成内存泄露, 最大的问题是对象类型的强依赖, 代码解耦不容易, 我们在做组件化很麻烦, 需要写大量的公共中间通讯接口, 有时候还不一定能解决强依赖的问题。我觉得不太适合如今越来越大型的 app 开发和公司内部多 app 联动开发的需求了。

- 数据驱动 UI ()

我管这种方式叫: 事件驱动型。我们在使用 EventBus 时发送一个 info 数据对象出去就行, 我们不管有多少对象会处理这个数据对象, 如果有10个对象会消费这个数据读写, 那么在 EventBus 架构中我们不用持有这10个对象, 就避免了对象类型的强耦合, 给我们使用组件化带来了巨大的便利, 这就是我们组件化, 甚至插件化改造当前项目的基石。一个 info 数据对象就是一个事件, 这样看我们就是用数据来驱动 UI 了。



之前我尝试过使用 EventBus 来搭建我的事件驱动型 app 架构，尝试过会发现问题很多，info 类爆炸，内存泄露等问题很难解决，不是我一个人恩那个解决的。但是今天我们带来的这个官方的 AAC 架构，就是基于事件驱动型的新的架构体系，大大解决了我们在组件化过程中的痛点，可以实现高度解耦。

大家可以试想，EventBus 是一个中间件，一个可以在所有 module 中通用的，EventBus 内部维护了所有的消费 info 数据的消费对象，EventBus.getInstance() 我们可以注册消费对象，这就简单的实现了 module 之间的解耦，module A 提供数据，不论游多少 module 去消费这个 info 数据，module A 都可以不关心，页不用持有这些消费 module 的引用。这就是基于事件的 app 架构的基础，之后不论游多复杂的架构，都是基于这个思想的，基于事件的架构核心痛点就是：高度解耦。具体应用应用就是为组件化，插件化扫平障碍

大家也可以看看我摘录别人博客的一些思路：

常见的构建原则

如果不可在应用程序组件中存储应用数据和状态，那么该如何构建应用呢？这儿有两条常见的构建原则：

- 关注点分离：一个常见的错误是在 Activity 和 Fragment 中编写所有的代码。任何和UI或者操作系统交互无关的代码都尽量不要出现在这些类中，尽量保持这些类的精简会帮助你避免很多和生命周期相关的问题。最好减少对它们的依赖以提供一个稳定的用户体验。
- 通过 Model 驱动 UI，最好是持久化的 Model。最好使用持久化的数据有两个原因：a. 如果系统销毁应用释放资源，用户也不用担心丢失数据；b. 即使网络连接不可靠或者断网，应用仍将继续运行。Model 是负责处理应用数据的组件，Model 独立运行于应用中的 View 和应用程序中的其他组件，因此 Model 和其他应用程序组件的生命周期无关。基于 Model 构建的应用程序，其管理数据的职责明确，所以更容易测试，而且稳定性更高

Snip20180313_1.png

我说的也是我自己个人的认识，有异议欢迎大家在下面喷我啊，一定要喷啊.....

学习资料

AAC 的是去年出的，第一时间获得了高关注，奈何本人小白一个，现在才刚刚看到这个 AAC 架构，网上的学习资料很多，基础的部分大家还是详细去看看我贴出来的资料，我就简单的总结一下知识点，然后会重点说一下我的认识。

基础学习资料：

- AAC(Android Architecture Components)
(<http://blog.csdn.net/user11223344abc/article/details/79364274>)
- 浅谈Android Architecture Components
(<http://blog.csdn.net/guijiaoba/article/details/73692397#room>)
- Android 项目最新架构 (<https://juejin.im/post/59526e18f265da6c3d6c0ac9>)
- Android 架构组件 Room 介绍及使用 (https://mp.weixin.qq.com/s?__biz=MzIxNzU1Nzk3OQ==&mid=2247486728&idx=1&sn=e9211eb04079ff6b666d54836b12c913&chksm=97f6b3bca0813aaa5f3bd192d6262af881f99c716c70ad478e3fd16cacc40ecbdeb5aec1435&mpshare=1&scene=1&srcid=03223t4Al3bv45HpplSu)



qsZH&key=b58032bc20cd466d4cd383c86b852adeb36d6a01e394fa7fa6734d7abf586e6d9a88899b3332e0f6318122e1d330587399cf56a09f96ffe2829cb1461c749f09fb5cf50f2379e06a5267a8d29047f115&scene=0&uin=NTI5MjcyMDk1&devicetype=iMac+MacBookPro13%2C1+OSX+OSX+10.12+build(16A2323a)&version=12020010&nettype=WIFI&lang=zh_CN&fontScale=100&pass_ticket=VL%2Frfq0mEoQ%2F6jgN8D8SB0ArE1i%2F5UrPishCI4oKGN2lt%2B%2BHipsFnIkt4EPG8Hse)

- Android 应用架构组件 (Architecture Components) 实践
(<http://lijiankun24.com/Android-%E5%BA%94%E7%94%A8%E6%9E%B6%E6%9E%84%E7%BB%84%E4%BB%B6%E5%BC%88Architecture-Components%E5%AE%9E%E8%B7%B5/>)

官方demo:

- BasicSample (<https://github.com/googlesamples/android-architecture-components/tree/master/BasicSample>)
演示如何使用a SQLite database 和 Room做持久化, 同时也使用了ViewModel和 LiveData。
- PersistenceContentProviderSample (<https://github.com/googlesamples/android-architecture-components/tree/master/PersistenceContentProviderSample>)
演示如何使用Room通过 Content Provider暴露数据。
- GithubBrowserSample (<https://github.com/googlesamples/android-architecture-components/tree/master/GithubBrowserSample>)
一个使用 Architecture component, Dagger以及Github API的高级示例, 需要Android Studio 2.4。

官方视频学习资源:

- 架构组件之 ViewModel | 中文教学视频 (https://mp.weixin.qq.com/s/RcupMal5d-N36rq42Xb0WQ?utm_source=androidweekly&utm_medium=website)
官方称 Lifecycles 这3个组件为生命周期管理组件, 我觉得已经不仅仅是生命周期管理了, 而是把 app 架构改造成响应式架构了, 整个 app 的事件和数据流动完全基于 Observable 和 Observer 的订阅来实现。观官方方法当所言, ViewModel 是解决 UI 和数据之间的逻辑, 但是不宜让 ViewModel 处理过多的页面逻辑, 若是页面逻辑和设置比较复杂, Persenter 还是有必要存在的。

开源学习 demo:

- awaker (<https://github.com/ruzhan123/awaker>)
一个挺不错的 demo, 很详细的使用了 AAC 框架, 代码封装的不错。个人感官, AAC 框架使用的有些畏手畏脚, 很中规中矩, AAC 参与获取数据, 刷新页面显示, 没有探讨更深入的用 AAC 把 app 架构向数据, 事件流方向修改。另外数据层中有些职责不清晰, 在 respositroy 在最外层类内封装了 respositroyHelp 具体页面实现层的情况下, respositroy 最外层类还是涉及到一些具体的业务逻辑, 另外 SQL 数据库层



不是和网络层同一个级别的，页面有管理层对象。但是总体来说，还是不错的，尤其是数据层适合一看，查漏补缺，看别人的实现完善自己的实现。

- ArchitecturePractice (<https://github.com/lijiankun24/ArchitecturePractice>)
这个 demo 比较简单，数据层页很简单，但是数据层次很清晰，适合看过上面的 demo 再来看看这个反思一下数据层到底应该怎么写。
- 使用Android Architecture Component开发应用（附demo）
(<https://www.jianshu.com/p/8333768481c3>)

本文 demo：

- BW_AAC_Demo (https://github.com/zb25810045/BW_AAC_Demo)
看点是用 LiveData 改造了一下 repository 数据层，UI 层通过 ViewModule 获取这个 LiveData，建立通道联系刷新数据。简单用 RxJava 延迟 2 秒发送一条数据，模拟一下网络请求。

补充资料：

- [译] Architecture Components 之 Guide to App Architecture
(<https://www.jianshu.com/p/c6e452537b2f>)
翻译的 AAC 官方文档，详细讲解了官方的学习 demo，里面是一个系列博客，详细分节讲解了 AAC 的每个部分。各位要是看我的文章哪里看糊涂了，不妨去这里寻找答案。
- 安卓架构组件-分页库
([http://chuckiefan.com/2017/06/07/%E7%BF%BB%E8%AF%91-%E5%AE%89%E5%8D%93%E6%9E%B6%E6%9E%84%E7%BB%84%E4%BB%B6\(7\)-%E5%88%86%E9%A1%B5%E5%BA%93.html](http://chuckiefan.com/2017/06/07/%E7%BF%BB%E8%AF%91-%E5%AE%89%E5%8D%93%E6%9E%B6%E6%9E%84%E7%BB%84%E4%BB%B6(7)-%E5%88%86%E9%A1%B5%E5%BA%93.html))
这个有意思啊，使用 AAC 组件开发列表分页加载，作者使用 PagerList 来管理分页状态，隔壁根据分页状态从 DataSource 中获取数据，作者使用数据库举例子的，remote 远程资源同样也可以，值得一看。
- 浅析MVP中model层设计【从零开始搭建android框架系列（7）】
(<https://www.jianshu.com/p/d299153ff853>)
- Android 生命周期架构组件与 RxJava 完美协作
(<https://juejin.im/entry/5987022df265da3e161a86f3>)

AAC 主要内容

- Lifecycle ()
生命周期管理，把原先Android生命周期的中的代码抽取出来，如将原先需要在 onStart()等生命周期中执行的代码分离到Activity或者Fragment之外。
- LiveData ()
一个数据持有类，持有数据并且这个数据可以被观察被监听，和其他Observer不同的是，它是和Lifecycle是绑定的，在生命周期内使用有效，减少内存泄露和引用问题。



- ViewModel ()

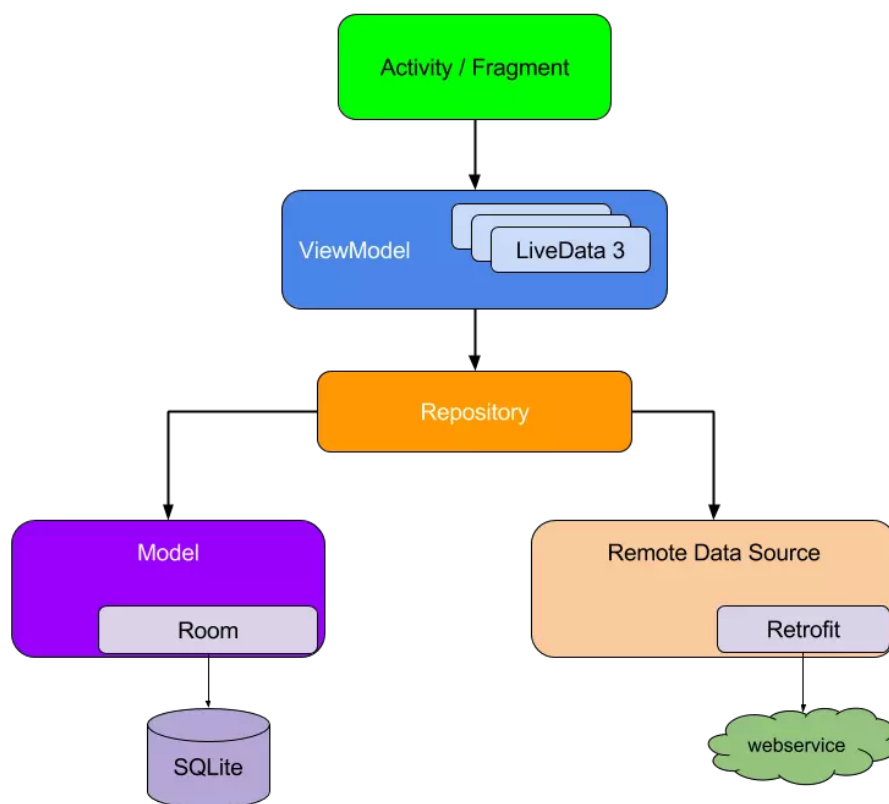
代替 persenter 的角色，生命周期长于 activity / fragment ，一种新的保存数据的方式。同时是与Lifecycle绑定的，使用者无需担心生命周期。可以在多个Fragment之间共享数据，比如旋转屏幕后Activity会重新create，这时候使用ViewModel还是之前的数据，不需要再次请求网络数据。

- Room ()

Google 推出的一个Sqlite ORM库，不过使用起来还不错，使用注解，极大简化数据库的操作，有点类似Retrofit的风格。

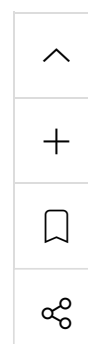
上面4个就是这次 AAC 架构的核心 API 了，通过这几个 API 我们可以搭建一套基于事件的 app 架构出来，LiveData 和其他一些 API 可以简单的看做是 Google 版的 RXJAVA ，只不过是针对 android 系统的，功能上也是很简单，没有 RXJAVA 那么强大的变换和线程控制。但是这正是我们所需要的，因为有了 RXJAVA ，我们在复杂的业务场景中用 RXJAVA 就好了，LiveData 使用简单，学习成本低，我们用来搭建基于数据流的响应式架构体系的 app 是最合适的。

AAC 架构图如下：



AAC 架构图

添加依赖



```
allprojects {
    repositories {
        jcenter()
        maven { url 'https://maven.google.com' } //添加此行
    }
}
```

```
//For Lifecycles, LiveData, and ViewModel
compile "android.arch.lifecycle:runtime:1.1.0"
compile "android.arch.lifecycle:extensions:1.1.0"
annotationProcessor "android.arch.lifecycle:compiler:1.1.0"

//For Room
compile "android.arch.persistence.room:runtime:1.1.0-alpha1"
annotationProcessor "android.arch.persistence.room:compiler:1.1.0-alpha1"

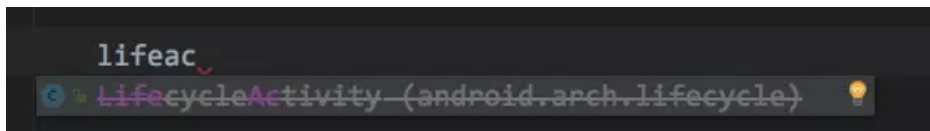
// For Room RxJava support, add:
compile "android.arch.persistence.room:rxjava2:1.1.0-alpha1"
```

Lifecycle

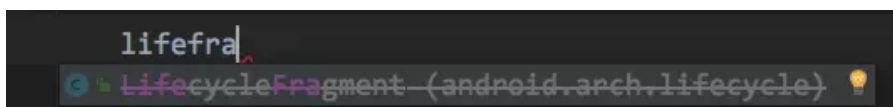
很多文章说的废话多，不直接，Android 页面生命周期加强，把页面的生命周期变成一个 Observable，这个 Observable 我们无法直接使用，而是配合注解写在需要 hook 生命周期的位置，最大的好处就是脱离了 callback 或是 Proxy 代理类，在代码上灵活太多了，Lifecycle 会扫描所有代码中有标记的方法，然后注册给生命周期的 Observable 对象里面去，这种写法是不是和 EventBus 一样啊。最直接的在写代码时这就是基于事件的魅力，我们不用再耗费心神设计代码结构，写那些 callback / Proxy 代理类了。

所以 AAC 这个基于事件的架构是学起来最通透的，一切都是 Observable / Observer，都是观察者和被观察者的关系，一切都是基于注册的方式运行。

既然一切都是 Observable / Observer 的，那么管理页面生命周期的 Observable / Observer 哪里来呢，这就要说到 LifecycleActivity / LifecycleFragment 了，我是用 API 26 做基准编译代码的，AppCompatActivity / V4 包下的 Fragment 都已经兼容了 AAC 架构了，所以 LifecycleActivity / LifecycleFragment 过时了，大家注意一下：



LifecycleActivity 过时了



LifecycleFragment 过时了

然后我们通过这个 API 可以获取这个 Observable Lifecycle

```
Lifecycle lifecycle = getLifecycle();
```

那么这个 Observer 呢，我们直接 new 一个 LifecycleObserver 对象出来也行，或是某个类实现这个接口也行，最后注册到 Observable 就行，下面看代码：

MyLifecycleTextView 实现 LifecycleObserver 接口

```
@SuppressWarnings("AppCompatCustomView")
public class MyLifecycleTextView extends TextView implements LifecycleObserver {

    public MyLifecycleTextView(Context context) {
        super(context);
    }

    public MyLifecycleTextView(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
    }

    public MyLifecycleTextView(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }

    public MyLifecycleTextView(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
    public void creat() {
        Log.d("AAA", "oncreat...");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    public void start() {
        Log.d("AAA", "onstart...");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void resume() {
        Log.d("AAA", "onresume...");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void pasue() {
        Log.d("AAA", "onpause...");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    public void stop() {
        Log.d("AAA", "onstop...");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
    public void destroy() {
        Log.d("AAA", "ondestroy...");
    }

}
```

综合使用



```
public class MainActivity extends AppCompatActivity {

    private MyLifecycleTextView tx_lifecycle;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        tx_lifecycle = findViewById(R.id.tx_lifecycle);

        getLifecycle().addObserver(tx_lifecycle);

        getLifecycle().addObserver(new LifecycleObserver() {

            @OnLifecycleEvent(Lifecycle.Event.ON_START)
            public void start() {
                Log.d("AAA", "new_onstart...");
            }

            @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
            public void resume() {
                Log.d("AAA", "new_onresume...");
            }

            @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
            public void pause() {
                Log.d("AAA", "new_onpause...");
            }

            @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
            public void stop() {
                Log.d("AAA", "new_onstop...");
            }

            @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
            public void destroy() {
                Log.d("AAA", "new_ondestroy...");
            }

        });
    }
}
```

上面的代码基本包含了 Lifecycle 的常用应用场景了，不管你事让某个自定义 view hook 某个生命周期也行，还是直接注册一个观察者对象也行，在代码上我们完全脱离了页面生命周期函数里，页面或是 presenter，viewmodel 都不用再保存相关的 callback，proxy，listener 在页面的生命周期函数里执行了，在 app 代码层面实现了基于观察者模式的代码思路。减少强制持有属性，功能，代理对象，这也是一种解耦啊，和以前对象方法比起来，代码是越写越灵活啊，这就是趋势啊。

好了，上面 XBB 了一下，Lifecycle 还没说完呢，我们了解了 Lifecycle 如何使用，代码里我们可以看到有很多带注解的函数

```
@OnLifecycleEvent(Lifecycle.Event.ON_STOP)
public void stop() {
    Log.d("AAA", "new_onstop...");
}
```

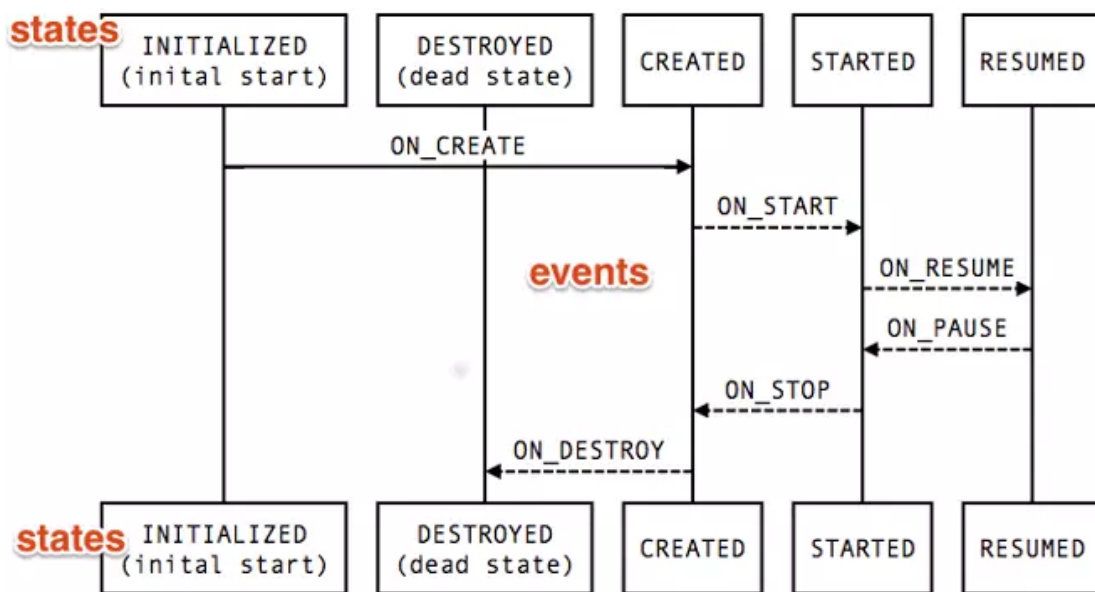


既然说了 Lifecycle 像 EventBus ，那么大家就不会对这注解函数陌生了吧，我来说说这个 Lifecycle.Event.ON_STOP 。API 的名字起来很规矩，大家一看应该就知道了，ON_STOP 就是代表了 onStop() 这个生命周期函数了，以此类推就不用多重复了

另外通过 Lifecycle 我们还可以获取当前 view 的生命周期状态，这在之前是需要自己去维护的，有了这个 API 我们在很多时候会方便很多啊。

```
Lifecycle.State currentState = getLifecycle().getCurrentState();
if( currentState == Lifecycle.State.STARTED ){
    xxxxxxxxx
}
```

另外官方有 Lifecycle 对应的生命周期图，大家看看



lifecycle-states.png

Lifecycle 注册的生命周期回调方法需要说一下是即使生效的，这点必须要说清楚，不说的童鞋可能会以为第一次无效，其实我们想想，对于声明周期来说，第一次无效是不符合设计思路和场景需求的。

Lifecycle 的最后我得数据说一下这个接口 LifecycleRegistryOwner ， AppCompatActivity / V4 包下的 Fragment 都是通过实现这个接口来兼容 AAC 架构的，其实这个接口很简单，里买就一个方法，需要我们返回 Lifecycle 的核心功能类 LifecycleRegistry 即可。看代码，自己实现 Lifecycle：



```
import android.arch.lifecycle.LifecycleRegistry;
import android.arch.lifecycle.LifecycleRegistryOwner;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity implements LifecycleRegistryOwner {

    private LifecycleRegistry lifecycleRegistry = new LifecycleRegistry(this);

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        proxy.zj.com.lifecycledemo.MyTv mTv = findViewById(R.id.dfttv);
        mTv.setLifecycle(getLifecycle());
        getLifecycle().addObserver(mTv);
        mTv.setLifeCycleEnable(true);

    }

    @Override
    public LifecycleRegistry getLifecycle() {
        return lifecycleRegistry;
    }
}
```

其实我们用不到 LifecycleRegistryOwner 这个借口，但是为啥要说呢，系统只是在 Activity 和 Fragment 层面兼容了 AAC，我们要是需要自己的自定义 view 对外提供声明周期管理的话记得自己实现了。按着上面的代码来就行，很简单，一看就会，不会来找我。

摘一段官方文档的翻，出自：[译] Architecture Components 之 Handling Lifecycles (<https://juejin.im/post/5937e1c8570c35005b7b262a>)

Lifecycles 的最佳实践

- 保持 UI 控制器（Activity 和 Fragment）尽可能的精简。它们不应该试图去获取它们所需的数据；相反，要用 ViewModel (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FViewModel.html>) 来获取，并且观察 LiveData (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FLiveData.html>) 将数据变化反映到视图中。
- 尝试编写数据驱动（data-driven）的 UI，即 UI 控制器的责任是在数据改变时更新视图或者将用户的操作通知给 ViewModel (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FViewModel.html>)。
- 将数据逻辑放到 ViewModel (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FViewModel.html>) 类中。ViewModel (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FViewModel.html>)



target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FViewModel.html) 应该作为 UI 控制器和应用程序其它部分的连接服务。注意：不是由 ViewModel (https://link.juejin.im?

target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FViewModel.html) 负责获取数据（例如：从网络获取）。相反，ViewModel (https://link.juejin.im?

target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FViewModel.html) 调用相应的组件获取数据，然后将数据获取结果提供给 UI 控制器。

- 使用 Data Binding (https://link.juejin.im?

target=https%3A%2F%2Fdeveloper.android.com%2Ftopic%2Flibraries%2Fdata-binding%2Findex.html) 来保持视图和 UI 控制器之间的接口干净。这样可以让视图更具声明性，并且尽可能减少在 Activity 和 Fragment 中编写更新代码。如果你喜欢在 Java 中执行该操作，请使用像 Butter Knife (https://link.juejin.im?

target=http%3A%2F%2Fjakewharton.github.io%2Fbutterknife%2F) 这样的库来避免使用样板代码并进行更好的抽象化。

- 如果 UI 很复杂，可以考虑创建一个 Presenter 类来处理 UI 的修改。虽然通常这样做不是必要的，但可能会让 UI 更容易测试。

- 不要在 ViewModel (https://link.juejin.im?

target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FViewModel.html) 中引用 View (https://link.juejin.im?

target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Fview%2FView.html) 或者 Activity (https://link.juejin.im?

target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Fapp%2FActivity.html) 的 context。因为如果 ViewModel (https://link.juejin.im?

target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FViewModel.html) 存活的比 Activity 时间长（在配置更改的情况下），Activity 将会被泄漏并且无法被正确的回收。

liveData

liveData 简单来说就是一个可以根据观察者自身生命周期，在观察者需要结束时自动解绑的 Observable，并且结合了 DataBinding 的特点，liveData 自身数据改变时可以通知所有的观察者对象。哈哈，所以说完上面 Lifecycle 才能来看 liveData，这个生命周期管理自然是依托 Lifecycle 了，他俩本身就是一个体系下的东东啊。

liveData 是 abstract 的一个类，有3个关键方法：



- **onActive**
liveData 注册的观察者数量从 0 到 1时会执行，相当于初始化方法
- **onInactive**
liveData 注册的观察者数量回到 0 时会执行
- **setValue**
数据改变通知所有观察者

其他不多说，先来看看一个最简单的 liveData 例子：

- application 中对外提供一个 liveData 对象
- MainActivity 获取这个 liveData 然后注册一个观察者对象
- MainActivity 可以启动一个新的页面

MyApplication

```
public class MyApplication extends Application {  
  
    private static MyApplication INSTANCE;  
    public MyLiveData liveData;  
  
    public static MyApplication getInstance() {  
        return INSTANCE;  
    }  
  
    @Override  
    public void onCreate() {  
        super.onCreate();  
        INSTANCE = this;  
        liveData = new MyLiveData();  
    }  
}
```

LiveData

```
public class MyLiveData extends LiveData<String> {  
  
    @Override  
    protected void setValue(String value) {  
        super.setValue(value);  
        Log.d("BBB", "setValue..." + value);  
    }  
  
    @Override  
    protected void onActive() {  
        super.onActive();  
        Log.d("BBB", "onActive...");  
    }  
  
    @Override  
    protected void onInactive() {  
        super.onInactive();  
        Log.d("BBB", "onInactive...");  
    }  
}
```

MainActivity



```

public class MainActivity extends AppCompatActivity {

    private MyLifecycleTextView tx_lifecycle;
    private Button btn_01;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);
        tx_lifecycle = findViewById(R.id.tx_lifecycle);
        getLifecycle().addObserver(tx_lifecycle);

        getLifecycle().addObserver(new LifecycleObserver() {

            @OnLifecycleEvent(Lifecycle.Event.ON_START)
            public void start() {
                Log.d("AAA", "new_onstart...");
            }
        });

        btn_01 = findViewById(R.id.btn_one);
        btn_01.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                //                data.setValue("AAAAAAAAA");
                startActivity(new Intent(MainActivity.this, SecondActivity.class));
            }
        });

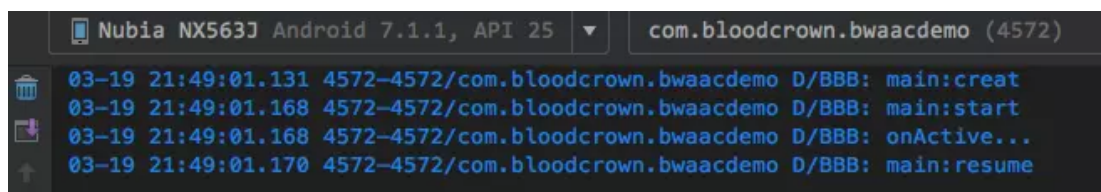
        MyApplication.getInstance().liveData.observe(this, new Observer<String>() {
            @Override
            public void onChanged(@Nullable String s) {
                Log.d("BBB", "数据改变..." + s);
            }
        });
    }
}

```

liveData 是可以使用泛型的，Google 推出 liveData 的初衷，就是用来包裹数据，包装成一个 Observable 的，所以一定要支持泛型才能使用的方便啊。

这里提醒一句，有人推荐使用 MutableLiveData，我的确是看到游 demo 使用 MutableLiveData 了。

这个例子，我们重点来看一下 onActive / onInactive 这个2个方法和观察者生命周期的互动，这点很重要，弄懂这点，这2个方法我们才能得心应手，要不会出问题的。我会打印 mainactivity 的生命周期函数 和 livadata 的相关方法



mainactivity 启动



启动一个新的页面

```
Nubia NX563J Android 7.1.1, API 25 com.bloodcrown.bwaacdemo (4572)
03-19 21:50:04.122 4572-4572/com.bloodcrown.bwaacdemo D/BBB: main:restart
03-19 21:50:04.123 4572-4572/com.bloodcrown.bwaacdemo D/BBB: main:start
03-19 21:50:04.124 4572-4572/com.bloodcrown.bwaacdemo D/BBB: onActive...
03-19 21:50:04.124 4572-4572/com.bloodcrown.bwaacdemo D/BBB: main:resume
```

返回 mainactivity 页面

根据上文所说，liveData 注册的观察者数量从 0 到 1 时会执行 onActive，liveData 注册的观察者数量回到 0 时会执行 onInactive，我们看看这个经典的例子，一个页面启动另一个页面再回来，这包含一个完整的生命周期了。

那我们要是再第二个页面 onCreate 函数里面也注册一个观察者呢，大家猜猜啊，挺有意思的。

```
03-19 22:15:07.719 5576-5576/com.bloodcrown.bwaacdemo D/BBB: main:pause
03-19 22:15:07.731 5576-5576/com.bloodcrown.bwaacdemo D/BBB: second:creat
03-19 22:15:07.738 5576-5576/com.bloodcrown.bwaacdemo D/BBB: second:start
03-19 22:15:07.740 5576-5576/com.bloodcrown.bwaacdemo D/BBB: second:resume
03-19 22:15:08.144 5576-5576/com.bloodcrown.bwaacdemo D/BBB: main:stop
```

启动一个新的页面

```
Nubia NX563J Android 7.1.1, API 25 com.bloodcrown.bwaacdemo (5576)
03-19 22:15:29.107 5576-5576/com.bloodcrown.bwaacdemo D/BBB: second:pause
03-19 22:15:29.114 5576-5576/com.bloodcrown.bwaacdemo D/BBB: main:restart
03-19 22:15:29.115 5576-5576/com.bloodcrown.bwaacdemo D/BBB: main:start
03-19 22:15:29.116 5576-5576/com.bloodcrown.bwaacdemo D/BBB: main:resume
03-19 22:15:29.429 5576-5576/com.bloodcrown.bwaacdemo D/BBB: second:stop
```

返回 mainactivity 页面

观察以上，可以得出一下结论：

- LiveData 观察者计数以 onPause 为分界点，观察者走 onPause 生命周期函数，LiveData 观察者计数会 -1，观察者走 onStart 生命周期函数，LiveData 观察者计数会 +1
- LiveData 的 onActive / onInactive 方法是根据注册的观察者数量是否为0触发的，不是每一个观察者的生命周期变动都会触发 onActive / onInactive 这2个方法，只有再观察者数量为0时才行，这点一定要注意

另外我们可以获取 LiveData 里面的数据

```
liveData.getValue();
```

观察者在注册到 LiveData 后，不会触发执行一次 setValue 方法，这点搞清楚基本就 OK 了



Lifecycle 和 LiveData 综合使用的例子

看这个 UserData 他可以在 view 生命周期变动时调 setValue 方法，通知所有观察者数据有变动

```
public class UserData extends LiveData implements LifecycleObserver {

    private static final String TAG = "UserData";

    public UserData() {

    }

    @Override
    protected void onActive() {
        super.onActive();
        Log.e(TAG, "onActive");
    }

    @Override
    protected void onInactive() {
        super.onInactive();
        Log.e(TAG, "onInactive");
    }

    @Override
    protected void setValue(Object value) {
        super.setValue(value);
        Log.e(TAG, "setValue");
    }
}
```

这种情况比较少使用，你想这个 LiveData 要和具体一个 view 的生命周期绑定，根据生命周期变动处理数据，发送新数据给所有注册者。在组件化的思路里，比较适合一个模块对外提供公共数据，然后这个模块有变动或是注销，再通知其他有数据关联的模块数据变动。

LiveData (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FLiveData.html>) 有以下优点：

- 没有内存泄漏：因为 Observer (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fjava%2Futil%2FObserver.html>) 被绑定到它们自己的 Lifecycle (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FLifecycle.html>) 对象上，所以，当它们的 Lifecycle (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FLifecycle.html>) 被销毁时，它们能自动的被清理。



- **不会因为 activity 停止而崩溃：**如果 Observer (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fjava%2Futil%2FObserver.html>) 的 Lifecycle (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FLifecycle.html>) 处于闲置状态（例如：activity 在后台时），它们不会收到变更事件。
- **始终保持数据最新：**如果 Lifecycle (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FLifecycle.html>) 重新启动（例如：activity 从后台返回到启动状态）将会收到最新的位置数据（除非还没有）。
- **正确处理配置更改：**如果 activity 或 fragment 由于配置更改（如：设备旋转）重新创建，将会立即收到最新的有效位置 (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Flocation%2FLocation.html>)数据。
- **资源共享：**可以只保留一个 MyLocationListener 实例，只连接系统服务一次，并且能够正确的支持应用程序中的所有观察者。
- **不再手动管理生命周期**你可能已经注意到，fragment 只是在需要的时候观察数据，不用担心被停止或者在停止之后启动观察。由于 fragment 在观察数据时提供了其 Lifecycle (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FLifecycle.html>)，所以 LiveData (<https://link.juejin.im?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Farch%2Flifecycle%2FLiveData.html>) 会自动管理这一切。

看一个经典的 LiveData 应用例子

一个定位的例子，这里写的简单，manage，factory，都没写，但是这个例子展示了：如何用 LiveData 包装数据，结合单例作为 app 的全局数据使用，这个单例可以写在自定义的 LiveData 里，也可以写在 manage 里有管理类来管理要更好一点




```

public class LocationLiveData extends LiveData<Location> {

    private static LocationLiveData sInstance;

    private LocationManager locationManager;

    @MainThread
    public static LocationLiveData get(Context context) {
        if (sInstance == null) {
            sInstance = new LocationLiveData(context.getApplicationContext());
        }
        return sInstance;
    }

    private SimpleLocationListener listener = new SimpleLocationListener() {
        @Override
        public void onLocationChanged(Location location) {
            setValue(location);
        }
    };

    private LocationLiveData(Context context) {
        locationManager = (LocationManager) context.getSystemService(Context.LOCATION_SERVICE);

        @Override
        protected void onActive() {
            locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, listener);
        }

        @Override
        protected void onInactive() {
            locationManager.removeUpdates(listener);
        }
    }
}

```

Transformations

LiveData 就是一个 Observable，那么官方提供了一个 Transformations 类，包含 map 和 switchMap 转换操作，和 Rxjava 的 map、flatMap 一样，

- Transformations.map()

```

LiveData<User> userLiveData = ...;
LiveData<String> userName = Transformations.map(userLiveData, user -> {
    user.name + " " + user.lastName
});

```

- Transformations.switchMap()

```

private LiveData<User> getUser(String id) {
    // ... 这里可以通过远程获取数据
}

LiveData<String> userId = ...;
LiveData<User> user = Transformations.switchMap(userId, id -> getUser(id));

```

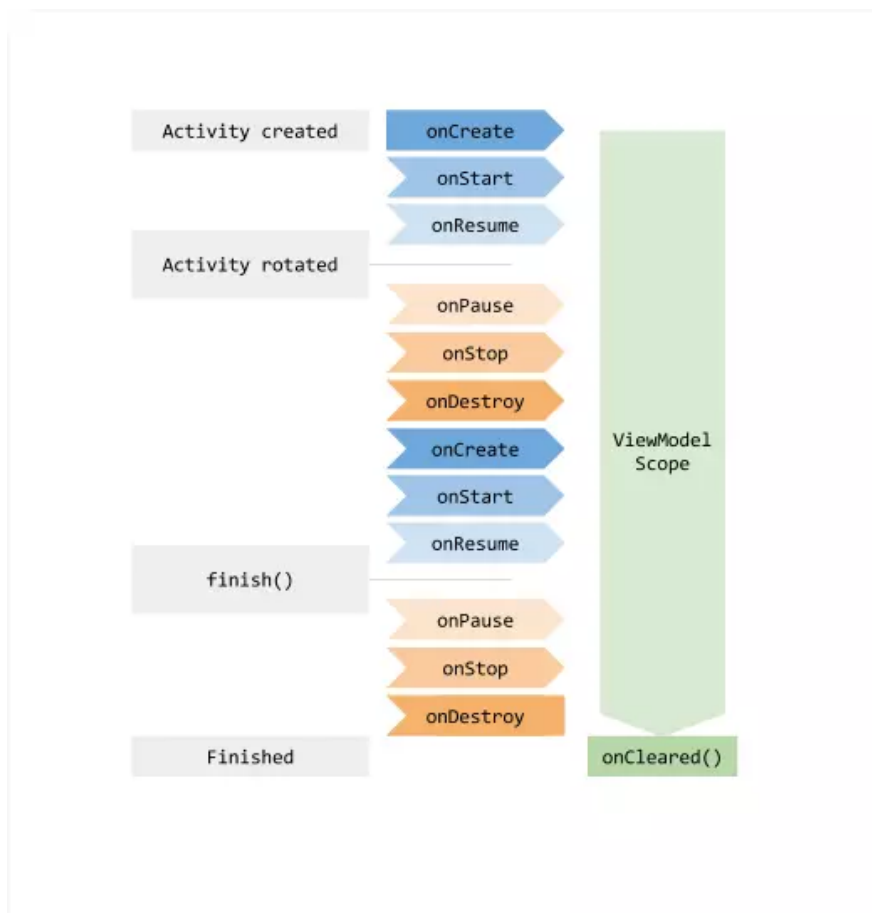


map() 方法接受一个数据，返回一个新的数据，这个看着不难，switchMap() 方法接受一个数据，然后依赖另一个 LiveData 加工，然后返回数据。要是看着不太懂的话，看这里：[译] Architecture Components 之 LiveData
(<https://juejin.im/post/5937e402a0bb9f005808d00e>)

ViewModel

这个最好理解，ViewModel 就是 MVP 中，P 的角色，当然 ViewModel 还有他的独特之处。ViewModel 的生命周期虽然总体上还是跟着 view 的，但是 ViewModel 存在的时间比 view 要长一些。我们看看这几个场景：

- 屏幕旋转，系统主动杀死造成的 Activity 或者 Fragment 被销毁或重新创建，所以保存于其中的数据有可能会丢失
- 在 Activity 或者 Fragment 中会经常发起一些需要一定时间才会返回结果的异步请求调用



ViewModel 生命周期图

ViewModel 生命周期图表明了 ViewModel 的生命周期是有些区别，单总体趋同与 view 的生命周期的，提供了一种新的 view 数据保存模式，比如屏幕旋转时 activity 重新创建 ViewModel 还是原来那个对象。

ViewModel 是个基类，需要我们继承他，没有特殊方法需要去处理，集成完后，直接创建对象就可以使用了。



```
MyViewModule myViewModule2 = ViewModelProviders.of(this).get(MyViewModule.class);
```

还有一个 `AndroidViewModel`，期中可以获取 `application`，只不过这个 `application` 需要我们自己传入，另外 `ViewModel` 的 `of` 方法还可以接受一个 `ViewModelProvider.NewInstanceFactory` 的参数，可以支持自定义构造方法，想传几个参数都没问题

```
public class MyViewModule extends AndroidViewModel {

    public String name;

    public MyViewModule(@NonNull Application application, String name) {
        super(application);
        this.name = name;
    }

    public void show() {
        Toast.makeText(getApplication(), "测试...", Toast.LENGTH_SHORT).show();
    }

    public static class Factroy extends ViewModelProvider.NewInstanceFactory {

        public Application application;
        public String name;

        public Factroy(Application application, String name) {
            this.application = application;
            this.name = name;
        }

        @NonNull
        @Override
        public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {
            return (T) new MyViewModule(application, name);
        }
    }
}

创建 MyViewModule 对象

MyViewModule.Factroy factroy = new MyViewModule.Factroy(getApplication(), "AAA");
MyViewModule myViewModule = ViewModelProviders.of(this, factroy).get(MyViewModule.c
```

使用 `ViewModule` 可以在同一个 `actitivity` 的多个 `Fragment` 之间共享数据



```
public class SharedViewModel extends ViewModel {
    private final MutableLiveData<Item> selected = new MutableLiveData<Item>();
    public void select(Item item) {
        selected.setValue(item);
    }
    public LiveData<Item> getSelected() {
        return selected;
    }
}

public class MasterFragment extends Fragment {
    private SharedViewModel model;
    public void onActivityCreated() {
        model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);
        itemSelector.setOnClickListener(item -> {
            model.select(item);
        });
    }
}

public class DetailFragment extends LifecycleFragment {
    public void onActivityCreated() {
        SharedViewModel model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);
        model.getSelected().observe(this, { item ->
            // 更新 UI
        });
    }
}
```

- Activity 不需要知道该通信的任何事情
- Fragment 之间不受相互影响。除了 ViewModel 之外，Fragment 不需要了解彼此，就算一个 Fragment 被销毁了，另一个也可以正常工作。而且每个 Fragment 都有自己独立的生命周期，不受其他 Fragment 的影响。

写到这里其实我们可以看到一个问题，ViewModel 内部不应该持有外部 view 的引用，ViewModel 的声明周期比 Fragment / Activity 都长，ViewModel 要是持有了外部 view 的引用就会造成内存泄露，需要注意！

ViewModel 和外部 view 的交互看过上面的大家应该都明白了，就是 LiveData 了，用这个 LiveData 代替 view 的交互接口同 ViewModel 通信，ViewModel 返回 LiveData，然后 view 获取这个 LiveData 再去注册 update 方法，这样避免 ViewModel 持有外部 view 造成的内存泄露

ViewModel vs SavedInstanceState

ViewModels 提供了一种在配置更改时保存数据的简便方式，但是如果应用进程被操作系统杀死，那么数据则没有机会被恢复。

通过 SavedInstanceState 保存的数据，存在于操作系统进程的内存中。当用户离开应用数个小时之后，应用的进程很有可能被操作系统杀死，通过 SavedInstanceState 保存的数据，则可以在 Activity 或者 Fragment 重新创建的时候，在其中的 onCreate() 方法中通过 Bundle 恢复数据。

Room



谷歌推出的一个Sqlite ORM库，不过使用起来还不错，使用注解，极大简化数据库的操作，有点类似Retrofit的风格

不过 Room 还是没有脱离 SQL 语言，在注解中还是要写 SQL 语句的，一般移动端的数据库没有太高的要求，这样我们其实可以使用 Room 这个数据库，Room 存的是对象，取的也是对象，是对象类型数据库了。要是 app 项目的数据库要求高的话，在来使用 Room 也是个不错的选择

Room 的详细学习看这个：浅谈Android Architecture Components

(<http://blog.csdn.net/guijiaoba/article/details/73692397#livedata>) | Android 架构组件

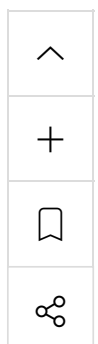
Room 介绍及使用 ([https://mp.weixin.qq.com/s?__biz=MzlxNzU1Nzk3OQ==&mid=2247486728&idx=1&sn=e9211eb04079ff6b666d54836b12c913&chksm=97f6b3bca0813aaa5f3bd192d6262af881f99c716c70ad478e3fd16cacc40ecbdeb5a5ec1435&mpshare=1&scene=1&srcid=03223t4Al3bv45HpglSuqszH&key=b58032bc20cd466d4cd383c86b852adeb36d6a01e394fa7fa6734d7abf586e6d9a88899b3332e0f6318122e1d330587399cf56a09f96ffe2829cb1461c749f09fb5cf50f2379e06a5267a8d29047f115&ascene=0&uin=NTI5MjcyMDk1&devicetype=iMac+MacBookPro13%2C1+OSX+OSX+10.12+build\(16A2323a\)&version=12020010&nettype=WIFI&lang=zh_CN&fontScale=100&pass_ticket=VL%2Frq0mEoQ%2F6jgN8D8SB0ArE1i%2F5UrPishCI4oKGN2lt%2B%2BHipsFnlt4EPG8Hse](https://mp.weixin.qq.com/s?__biz=MzlxNzU1Nzk3OQ==&mid=2247486728&idx=1&sn=e9211eb04079ff6b666d54836b12c913&chksm=97f6b3bca0813aaa5f3bd192d6262af881f99c716c70ad478e3fd16cacc40ecbdeb5a5ec1435&mpshare=1&scene=1&srcid=03223t4Al3bv45HpglSuqszH&key=b58032bc20cd466d4cd383c86b852adeb36d6a01e394fa7fa6734d7abf586e6d9a88899b3332e0f6318122e1d330587399cf56a09f96ffe2829cb1461c749f09fb5cf50f2379e06a5267a8d29047f115&ascene=0&uin=NTI5MjcyMDk1&devicetype=iMac+MacBookPro13%2C1+OSX+OSX+10.12+build(16A2323a)&version=12020010&nettype=WIFI&lang=zh_CN&fontScale=100&pass_ticket=VL%2Frq0mEoQ%2F6jgN8D8SB0ArE1i%2F5UrPishCI4oKGN2lt%2B%2BHipsFnlt4EPG8Hse))

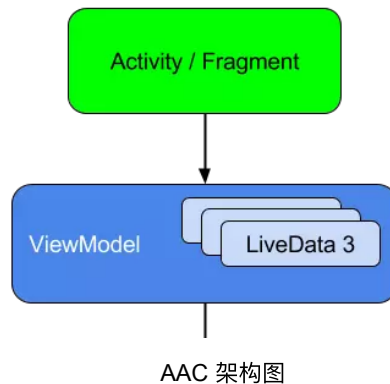
- [译] Architecture Components 之 Room Persistence Library
(<https://juejin.im/post/593df980ac502e006c049607>)

AAC 框架带给我们的改变

上面我们已经基本看过了 AAC 框架新的 API，经过 AAC 改造，我们不用担心内存泄露的场景了，页面的生命周期函数处理可以写在页面之外，更灵活了，整个 app 的架构可以像 Rxjava 一样，改成基于数据和事件的流式架构了，可以让我们更简单容易分离，组合代码结构。

说带具体的还是 LiveData 对数据层的改造对我们影响最大，我们再来看看上面官方给出的 AAC 架构图





Google 官方的页面，数据流程如下：

- Respositroy 判断环境，管理缓存，返回数据
- ViewModel 管理数据的 LiveData
- 页面获取 LiveData 注册 update 界面更新方法
- 最后 ViewModel 触发 Respositroy 请求数据，通过 LiveData 更新数据

很明显，这是一个最简单的数据，显示例子，展示的一个页面拥有一个 LiveData 对象来更新数据，这是传统的逻辑思路。但是大家想想，LiveData 是数据流式的，我们完全可以让 Respositroy 返回一个静态的全局的 LiveData，需要的页面去注册 update 方法，页面关闭的时候可以自动接触绑定，这才是 LiveData 的初衷啊，当然使用嘛，不要死板，怎么灵活，怎么简单，便利，怎么来。

上面涉及到我们对数据的应用范围了和类型了，LiveData 的出现让我们对数据的包装使用产生了一些思考

数据的作用范围我分为3种：

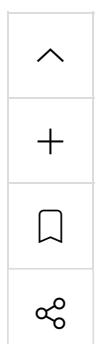
- 单个页面范围内
- 单个功能模块范围内 (module)
- 整个 app 范围内

数据的类型我分为2种：

- 原始数据类型，不做处理
- Reponse<Object> ,把数据和加载状态综合一起
- Reponse + Object , 把数据和加载状态分开，这样可以视情况选择

结合不同的数据的应用范围，对数据的处理：

- 单个页面范围
对于单个页面范围的 data，Respositroy 返回非静态的 LiveData <Reponse<Object>> 带加载状态的 data
- 单个功能模块范围内 (module)
Respositroy 需要单独成 module，Respositroy 返回 LiveData <Reponse>，



LiveData < Object> , 有可能这个 module 的数据, 别的 module 也是需要的, 但是对于加载状态来说只是本功能 module 有意义, 其他 module 只关心 data 的。

- 整个 app 范围内

同上面单个功能模块范围内 (module) 的处理, 视情况可以不提供加载状态的 Response

数据的经典应用场景举例:

- 单个页面范围内

这是最常见的, 我们请求的大数据接口都是, 比如商品详情页, 不同 ID 的商品有不同数据, 那么别的页面显然是无法使用这个数据的

- 单个功能模块范围内 (module)

这个比较少见, 是随着 app 的组件化出现的, 经典的例子有定位, 定位的数据只要对外提供静态的 LiveData <Adress> 就行, 这个定位数据随着定位模块的卸载而删除, 要是设计成整个 app 范围内页无妨, 号保存即可。

- 整个 app 范围内

这个场景更少, 但是是难设计的, 最考验代码功底的。比如全局缓存, 这个每个 app 都需要, 还有个人中心数据, 这个设计好了非常厉害了, 对于编码水平提供有很大助力啊

总结一下就是用 LiveData 改造了一下 repository 数据层, UI 层通过 ViewModule 获取这个 LiveData, 建立通道联系刷新数据, 这样把整个 app 基于数据流改造成响应式架构, 适应组件化, 平台化高度封装, 业务模块分离的需求。

本文 demo 思路

首先本文 demo 如下:

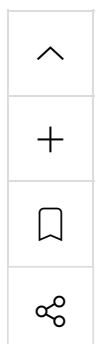
- BW_AAC_Demo (https://github.com/zb25810045/BW_AAC_Demo)

思路是用 LiveData 改造了一下 repository 数据层, UI 层通过 ViewModule 获取这个 LiveData, 建立通道联系刷新数据。简单用 RxJava 延迟 2 秒发送一条数据, 模拟一下网络请求。

比较好的代码研究资料:

- Architecture Components 之 Guide to App Architecture
(<https://www.jianshu.com/p/c6e452537b2f>)
- 使用Android Architecture Component开发应用 (附demo)
(<https://www.jianshu.com/p/8333768481c3>)

我的 demo 思路如下:





device-2018-03-30-205908.png

先来看看我的页面，主界面中游2个红色背景的 textview 分别用来显示2个级别的数据：单次请求的网络数据和全局数据，全局数据在另外一个页面中加载，然后再返回看看 LiveData 的同步效果是否好用。

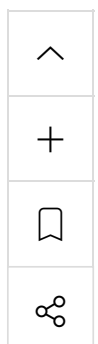
数据层都封装再 Respositroy 中，Respositroy 持有一个 netDataSource 用于获取网络数据，然后 Respositroy 处理数据然后返回。其中数据都是用 response 来包裹，response 中有 code 用来发送数据加载中的各种状态

Snip20180330_5.png

画了一个简单的数据流动图，可以看到我这里的 Respositroy 数据层封装的还是比较简单的，Respositroy 内部只维护了一个 BookNetDataSource 远程数据源。

来简单看下代码：

BookNetDataSource -> 远程数据源



```
public class BookNetDataSource {  
  
    private Book mPrivateBook = new Book("私有数据在此");  
    private Book mPublicBook = new Book("共有数据在此");  
  
    public Observable<Book> getPrivateBook() {  
        return getData(new Book("私有数据如下: private"));  
    }  
  
    public Observable<Book> getPublicBook() {  
        return getData(new Book("全局数据如下: public"));  
    }  
  
    private Observable<Book> getData(final Book book) {  
        return Observable.timer(2, TimeUnit.SECONDS)  
            .map(new Function<Long, Book>() {  
                @Override  
                public Book apply(Long aLong) throws Exception {  
                    return book;  
                }  
            }).subscribeOn(Schedulers.io());  
    }  
}
```

BookNetDataSource 返回了2个数据，private 数据表示一次性数据，public 表示全局缓存数据，用 rxjava 延迟2秒模拟一下网络状态

BookRespositroy 数据层



```

public class BookRespositroy {

    public static MutableLiveData<Response<Book>> PUBLIC_LIVEDATA;

    private MutableLiveData<Response<Book>> mBookLiveData;
    private BookNetDataSource mBookNetDataSource;

    static {
        PUBLIC_LIVEDATA = new MutableLiveData<>();
    }

    public static MutableLiveData<Response<Book>> getPublicLiveData() {
        return PUBLIC_LIVEDATA;
    }

    public static void refreshPublicData() {
        PUBLIC_LIVEDATA.setValue(new Response<Book>(Response.CODE_LOADING, null));
        new BookNetDataSource().getPublicBook()
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<Book>() {
                @Override
                public void accept(Book book) throws Exception {
                    PUBLIC_LIVEDATA.setValue(new Response<Book>(Response.CODE_SU
                }
            });
    }

    public BookRespositroy() {
        mBookNetDataSource = new BookNetDataSource();
        mBookLiveData = new MutableLiveData<>();
    }

    public MutableLiveData<Response<Book>> getPrivateLiveData() {
        return mBookLiveData;
    }

    public void refreshPrivateData() {
        mBookLiveData.setValue(new Response<Book>(Response.CODE_LOADING, null));
        mBookNetDataSource.getPrivateBook()
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<Book>() {
                @Override
                public void accept(Book book) throws Exception {
                    mBookLiveData.setValue(new Response<Book>(Response.CODE_SUCC
                }
            });
    }

}

```

PUBLIC_LIVEDATA 这个是全球缓存数据，这个由 Respositroy 数据层维护是比较恰当的，mBookLiveData 这个是一次性数据，这个一次性对应的是 BookRespositroy 这个对象的生命周期，MainActivity 通过 viewModule 获取到这2个 LiveData 管道然后注册自己的 UI 刷新方法

省略无关代码



```

public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;
    private MyViewModule myViewModule;
    private ProgressDialog dialog;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // 初始化 DataBinding
        binding = DataBindingUtil.setContentView(this, R.layout.activity_main);

        // 使用自定义构造器方式创建 ViewModel 对象
        MyViewModule.Factroy factroy = new MyViewModule.Factroy(getApplication(), "A");
        myViewModule = ViewModelProviders.of(this, factroy).get(MyViewModule.class);

        // 从 ViewModel 中获取页面私有数据源的管道, 建立联系, Response 中包含请求响应状态码
        myViewModule.getPrivateBookLiveData().observe(this, new Observer<Response<Book>>() {
            @Override
            public void onChanged(@Nullable Response<Book> response) {
                if (response == null) {
                    return;
                }
                int code = response.code;
                if (code == Response.CODE_LOADING) {
                    dialog.show();
                    return;
                }
                if (code == Response.CODE_SUCCESS) {
                    dialog.dismiss();
                    binding.setViewData(response.data.getName());
                    return;
                }
            }
        });

        // 从 ViewModel 中获取全局公共数据源的管道, 建立联系
        BookRespositroy.getPublicLiveData().observe(this, new Observer<Response<Book>>() {
            @Override
            public void onChanged(@Nullable Response<Book> response) {
                if (response == null) {
                    return;
                }
                int code = response.code;
                if (code == Response.CODE_LOADING) {
                    dialog.show();
                    return;
                }
                if (code == Response.CODE_SUCCESS) {
                    dialog.dismiss();
                    binding.setAppData(response.data.getName());
                    return;
                }
            }
        });
    }
}

```

最后 MainActivity 通过 viewModule 通知 respositroy 获取数据, 然后刷新 livedata , livedata 调用 setVale 方法就能通知所有注册其上的对象更新数据了。



```
public void refreshPrivateData() {
    mBookLiveData.setValue(new Response<Book>(Response.CODE_LOADING, null));
    mBookNetDataSource.getPrivateBook()
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Consumer<Book>() {
            @Override
            public void accept(Book book) throws Exception {
                mBookLiveData.setValue(new Response<Book>(Response.CODE_SUCCESS, book));
            }
        });
}
```

注意这里面，loading 的状态是由 respositroy 发送的而不是 viewmodule ，加载状态通过 response 中的 code 码来承载。

核心点：

- 数据的流动基于观察者模式来注册，数据源蹭蹭包装抛出 LiveData 这个包裹数据源的 observable 用于注册，UI 层获取 LiveData 注册 UI 刷新方法，那么整个数据通道就正式建立了，这时通知数据层加载数据就可以了
- 数据层中对数据源进行封装，对 remote，file，sql，cache 需要分别封装，具体的数据源返回的不是具体的数据，而是 rxjava 中的 observable ，因为整个数据流是基于响应式的，要是这里返回个具体数据那么思路就不对了，或者用 callback 传递给数据源都是对响应式变成的不理解。

最后

最后想说一下 respositroy 数据层的封装一定要量力而行，我这里 respositroy 层支持持有 net 远程数据源了，这个一般即使我们的需求了，没必要过度去封装，反而不美。但是要是数据源比较多的缓存需求，那么就需要再 respositroy 中封装一个 DataSource 来管理数据源了，比如图片加载库，glide、fresco，他们都是对数据缓存有要读要起的，你看代码的代码思路专门有一个数据源管理类对外提供数据。

另外若是 respositroy 数据层中业务处理很复杂的话，我们可以把具体的业务逻辑处理分离成一个个的 helper 类，google 源码中就有很多的 helper 类来辅助管理具体的功能


若是 app 设计有自己的特殊业务 code 的话，我们可以把处理放在 respositroy 数据层中，在 app 初始化时配置一个静态的处理类出来，然后我们在 respositroy 数据层中获取这个处理类来优先处理特务业务 code，为啥不放在 respositroy 的基类中，是因为很多 api 我们不需要处理特殊 code，为了代码灵活一下牺牲下代码封装性。

ps：最后的最后，说一下写的不好，我的认识也浅显，欢迎大家踊跃喷我，希望通过大家的评论提高进步。

小礼物走一走，来简书关注我



赞赏支持

 android_进阶 (/nb/12952113)

[举报文章](#) © 著作权归作者所有



前行的乌龟 (/u/fb093dd92ed8) ♂

写了 248770 字, 被 191 人关注, 获得了 431 个喜欢
(/u/fb093dd92ed8)

✓ 已关注

你要明白在泥泞中生存并不可怕, 可怕的是你不知不觉变成了一只老鼠, 而且还在找理由, 继续这么生存...

喜欢 | 15



更多分享

被以下专题收入, 发现更多相似内容

+ 收入我的专题



Android开发 (/c/2c3211291f88?utm_source=desktop&utm_medium=notes-included-collection)



框架【库】 (/c/628a00a75683?utm_source=desktop&utm_medium=notes-included-collection)

Android - 收藏集 (/p/dad51f6c9c4d?utm_campaign=maleskine&utm_c...

用两张图告诉你, 为什么你的 App 会卡顿? - Android - 掘金 Cover 有什么料? 从这篇文章中你能获得这些料: 知道setContentView()之后发生了什么? ... Android 获取 View 宽高的常用正确方式, 避免为零 - 掘金...



passiontim (/u/e946d18f163c?

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommenc

掘金 Android 文章精选合集 (/p/5ad013eb5364?utm_campaign=maleski...

用两张图告诉你, 为什么你的 App 会卡顿? - Android - 掘金Cover 有什么料? 从这篇文章中你能获得这些料: 知道setContentView()之后发生了什么? ... Android 获取 View 宽高的常用正确方式, 避免为零 - 掘金...



掘金官方 (/u/5fc9b6410f4f?

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommenc

awesome-android (/p/325d0794fcd0?utm_campaign=maleskine&utm_...



作者: snowdream 微信: sn0wdr1am 原文地址: <https://github.com/snowdream/awesome-android>
 awesome-android Introduction android libs from github System re...



雪梦科技 (/u/748f0f7e6432?)

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommend

awesome-android (/p/1a792b14af3b?utm_campaign=maleskine&utm_...

afinalAfinal是一个android的ioc, orm框架 <https://github.com/yangfuhai/afinal> xUtils**android orm, bitmap, http, view inject... <https://github.com...>



passiontim (/u/e946d18f163c?)

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommend

必读的 Android 文章 (/p/c8488cc14ffe?utm_campaign=maleskine&utm...

写给 Android 开发者的混淆使用手册 - Android - 掘金 本文转自: 点击打开链接 毫无疑问, 混淆是打包过程中最重要的流程之一, 在没有特殊原因的情况下, 所有 app 都应该开启混淆。首先, 这里说的混淆其实是...



掘金官方 (/u/5fc9b6410f4f?)

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommend

(/p/980366e0452c?)

utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommend

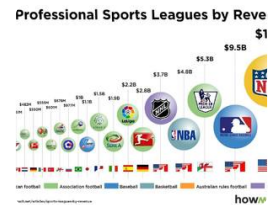
NBA是如何通过推特提升影响力的? (/p/980366e0452...

2015-2016赛季, NBA实现了一个明显的突破: 篮球推特战略 (Basketball Twitter)。今年2月份, NBA成为所有联赛、球队、球员账户中, 第一个社媒...



禹唐 (/u/9f15af43b678?)

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommend



话别离 诉衷肠 (/p/3585288f2a0a?utm_campaign=maleskine&utm_conte...

当我拍下你的背影 我知道 再一次的别离已经开始 都说分别是为了更好的相聚 可是只有经历过的人才知道其中的煎熬 已经数不清这是第几次分别 也不想去回忆 每一次相聚 我会期许很久 直到你按门铃开始 内心的欢...



琐城无忧 (/u/0f69d7d777ec?)

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommend

16-08-06 6 w32 65.3% (/p/1796e5979dc8?utm_campaign=maleskine&ut...

读书 (50%) □ 《毛泽东农村调查文集》 (0-100% 0)(0%) ✓ 《精要主义》 (83-100% 100%)(100%) □ 《把时间当朋友》 (0-50% 25%)(50%) 学习 (66.7%) □ 重拾 solife 公众号 ✓ 尝试写作 《写作 - 如何坚持每天写一千字 - ...



二石兄 (/u/ee459a0ddc6e?)

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommend

@荷边书房@注解四书《论语》【194】认清自己, 不忘初心 (/p/b3d3c068f...

【子曰: “巍巍乎, 舜禹之有天下也, 而不与焉。”】巍巍, 是高大之貌; 不与, 犹言不相关, 不以位为荣。孔子说: “伟大啊, 舜和禹贵为天子, 但好像荣华富贵跟他们没有关系似的, 他们并不以为荣。” 这就涉及...





荷边书房 (/u/762b351b1293?

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommend

