26.jectpack

26.1. Navigation

https://mp.weixin.qq.com/s/1URoDU0zgoYlSQM8zYqx9w

what

https://juejin.cn/post/6844904131577004039#heading-0

https://www.jianshu.com/p/5c1763b0c9eb

https://zhuanlan.zhihu.com/p/69562454

对于单个Activity嵌套多个Fragment的UI架构方式， 对Fragment的管理一直是一个比较麻烦的事情。

需要通过FragmentManager和FragmentTransaction来管理Fragment之间的切换 对应用程序的App bar的管理， Fragment间的切换动画 Fragment间的参数传递 总之，使用起来不是特别友好。

Navigation是用来管理Fragment的切换，并且可以通过可视化的方式，看见App的交互流程

why

https://www.jianshu.com/p/66b93df4b7a6 https://zhuanlan.zhihu.com/p/69562454

可以可视化的编辑各个组件之间的跳转关系

优雅的支持fragment之间的转场动画

通过第三方的插件支持fragment之间安全的参数传递

通过NavigationUI类，对菜单，底部导航，抽屉菜单导航进行方便统一的管理

支持通过deeplink直接定位到fragment

how

https://juejin.cn/post/6844904068897308679#heading-0

1 创建导航图

在 res 目录内创建一个 navigation 资源目录

根标签是navigation，它需要有一个属性startDestination，表示默认第一个显示的界面

每个fragment标签代表一个fragment类

每个action标签就相当于上图中的每条线，代表了执行切换的时候的目的地，切换动画等信息。

2 添加NavHostFragment

标签为fragment，android:name就是NavHost的实现类，这里是NavHostFragment app:navGraph 属性就是我们前面在res文件夹下创建的文件

Activity的布局中

1. 开启导航

通过Navigation#findNavController方法找到NavController，调用它的navigate方法开始导航。

view.findViewById(R.id.button).setOnClickListener(v -> { Bundle bundle = new Bundle(); bundle.putString("title","我是前面传过来的"); Navigation.findNavController(v).navigate(R.id.action_firstFragment_to_secondFragment,bundle); });

---

https://mp.weixin.qq.com/s/1URoDU0zgoYlSQM8zYqx9w

注意点:

1.页面跳转和参数传递

页面间的跳转是通过action来实现

1、Bundle方式

第一种方式是通过Bundle的方式。NavController 的navigate方法提供了传入参数是Bundle的方法

2.安全参数(SafeArg) build.gradle添加n:navigation-safe-args引用 添加 编译 Bundle bundle = new DetailFragmentArgs.Builder().setProductName("苹果").setPrice(10.5f).build().toBundle(); NavController contorller = Navigation.findNavController(view); contorller.navigate(R.id.action_homeFragment_to_detailFragment, bundle);

接收 Bundle bundle = getArguments(); if(bundle != null){ mProductName = DetailFragmentArgs.fromBundle(bundle).getProductName(); mPrice = DetailFragmentArgs.fromBundle(bundle).getPrice(); }

2 动画 action中 enterAnim: 配置进场时目标页面动画 exitAnim: 配置进场时原页面动画 popEnterAnim: 配置回退时目标页面动画 popExitAnim: 配置回退时原页面动画 配置完后

3.导航堆栈管理

Navigation 有自己的任务栈，每次调用navigate()函数，都是一个入栈操作，出栈操作有以下几种方式，下面详细介绍几种出栈方式和使用场景。 1、系统返回键

首先需要在xml中配置app:defaultNavHost="true"，才能让导航容器拦截系统返回键，点击系统返回键，是默认的出栈操作，回退到上一个导航页面。如果当栈中只剩一个页面的时候，系统返回键将由当前Activity处理。

2.popBackStack()或者navigateUp() 如果页面上有返回按钮，那么我们可以调用popBackStack()或者navigateUp()返回到上一个页面。

3.popUpTo 和 popUpToInclusive

我们看下面这个例子。假设有A,B,C 3个页面，跳转顺序是 A to B，B to C，C to A。依次执行几次跳转后，栈中的顺序是A>B>C>A>B>C>A。此时如果用户按返回键，会发现反复出现重复的页面，此时用户的预期应该是在A页面点击返回，应该退出应用。此时就需要在C到A的action中设置popUpTo="@id/a". 这样在C跳转A的过程中会把B,C出栈。但是还会保留上一个A的实例，加上新创建的这个A的实例，就会出现2个A的实例. 此时就需要设置popUpToInclusive=true. 这个配置会把上一个页面的实例也弹出栈，只保留新建的实例。下面再分析一下设置成false的场景。还是上面3个页面，跳转顺序A to B，B to C. 此时在B跳C的action中设置 popUpTo="@id/a", popUpToInclusive=false. 跳到C后，此时栈中的顺序是AC。B被出栈了。如果设置popUpToInclusive=true. 此时栈中的保留的就是C。AB都被出栈了。

4 DeepLink

Navigation组件提供了对深层链接（DeepLink）的支持。通过该特性，我们可以利用PendingIntent或者一个真实的URL链接，直接跳转到应用程序的某个destination

1、PendingIntent rivate PendingIntent getPendingIntent() { Bundle bundle = new Bundle(); bundle.putString("productName", "香蕉"); bundle.putFloat("price",6.66f); return Navigation .findNavController(this,R.id.fragment) .createDeepLink() .setGraph(R.navigation.nav_graph) .setDestination(R.id.detailFragment) .setArguments(bundle) .createPendingIntent(); } 2、URL连接

源码理解

https://mp.weixin.qq.com/s/1URoDU0zgoYlSQM8zYqx9w

NavHostFragment

这是一个特殊的布局文件，Navigation Graph中的页面通过该Fragment展示

NavHostFragment

1.onInflate解析在xml配置的两个参数defaultNavHost， 和navGraph

2、onCreate 创建NavController

3、onCreateView 创建一个FrameLayout

4、onViewCreated 在这个函数中，把NavController设置给了父布局的view的中的ViewTag中 Navigation.findNavController(View)中 递归遍历view的父布局，查找是否有view含有id为R.id.nav_controller_view_tag的tag, tag有值就找到了NavController。如果tag没有值.说明当前父容器没有NavController

NavController

导航的主要工作都在NavController中，涉及xml解析，导航堆栈管理，导航跳转等方面

1.NavHostFragment把导航文件的资源id传给了NavController

2.NavController把导航xml文件传递给了NavInflater解析导航xml文件

3.生成NavGraph保存着xml中配置的导航目标NavDestination

NavController的navigate函

把所有Navigator的实例保存在了NavigatorProvider

navigator.navigate

Fragment实例是通过instantiateFragment创建的，这个函数中是通过反射的方式创建的Fragment实例，Fragment还是通过FragmentManager进行管理，是用replace方法替换新的Fragment, 这就是说每次导航产生的Fragment都是一个新的实例，不会保存之前Fragment的状态

NavGraph

里面包含了一组NavDestination，每个NavDestination就是一个一个的页面，也就是导航目的地

NavigatorProvider

内部有个HashMap，用来存放Navigator，Navigator它是个抽象类，有三个比较重要的子类FragmentNavigator，ActivityNavigator，DialogFragmentNavigator

使用 https://juejin.cn/post/6844904131577004039

1.注解处理器的目标是，扫描出所有带FragmentDestination或者ActivityDestination的类，拿到注解中的参数和类的全类名，封装成对象放到map中，使用fastjson将map生成json字符串，保存在src/main/assets目录下面

https://blog.csdn.net/weixin_42575043/article/details/108709467

2.可以自定义FragmentNavigator解决Fragment重复创建的问题

26.2. DataBinding

https://blog.csdn.net/LucasXu01/article/details/103807451

https://juejin.cn/post/6844903494831308814#heading-6

https://www.jianshu.com/p/c56a987347ff

原理

1.APT预编译方式生成ActivityMainBinding和ActivityMainBindingImpl

2.处理布局的时候生成了两个xml文件

activity_main-layout.xml（DataBinding需要的布局控件信息）

activity_main.xml（Android OS 渲染的布局文件）

Model是如何刷新View

1.DataBindingUtil.setContentView方法将xml中的各个View赋值给ViewDataBinding，完成findviewbyid的任务

2.当VM层调用notifyPropertyChanged方法时，最终在ViewDataBindingImpl的executeBindings方法中处理逻辑

View是如何刷新Model

ViewDataBindingImpl的executeBindings方法中在设置了双向绑定的控件上，为其添加对应的监听器，监听其变动，如：EditText上设置TextWatcher，具体的设置逻辑放置到了TextViewBindingAdapter.setTextWatcher里

当数据发生变化的时候，TextWatcher在回调onTextChanged()的最后，会通过textAttrChanged.onChange()回调到传入的mboundView2androidTextAttrChanged的onChange()。

使用

https://juejin.cn/post/6844903872520011784#heading-0

26.3. ViewModel

https://blog.csdn.net/c10wtiybq1ye3/article/details/89934891

https://www.jianshu.com/p/41c56570a266?utm_campaign=haruki&utm_content=note&utm_medium=seo_notes&utm_source=recommendation

https://www.jianshu.com/p/ebdf656b6dd4

https://blog.csdn.net/qq_15988951/article/details/105106867

how

viewmodel = ViewModelProvider(this).get(MyViewModel::class.java)

what

ViewModel 类旨在以注重生命周期的方式存储和管理界面相关的数据

why

Activity配置更改重建时(比如屏幕旋转)保留数据

问题

https://blog.csdn.net/u014093134/article/details/104082453

例如你的 APP 某个 Activity 中包含一个 列表，因为配置更改而重新创建 Activity 后（例如众所周知的屏幕旋转发生后需手动保存数据在旋转后进行恢复），新 Activity 必须重新提取列表数据，对于简单数据，Activity 可以使用 onSaveInstanceState() 方法从 onCreate() 中的捆绑包恢复数据，但这种方法仅适合可以序列化再反序列化但少量数据，不适合数量可能较大但数据，如用户列表或位图

因为ViewModel的生命周期是比Activity还要长，所以ViewModel可以持久保存UI数据。

通常在系统首次调用 Activity 对象的 onCreate() 方法时请求 ViewModel。系统可能会在 Activity 的整个生命周期内多次调用 onCreate()，如在旋转设备屏幕时。 所以当前Activity的生命周期不断变化，经历了被销毁重新创建，而ViewModel的生命周期没有发生变化，Activity因为配置更改或者被系统意外回收的时候，会自动保存数 据。在Activity重建的时候就可以继续使用销毁之前保存的数据。

源码 https://www.jianshu.com/p/ebdf656b6dd4

ComponentActivity 中

onRetainNonConfigurationInstance是在onStop() 和 onDestroy()之间被调用，它内部会保存ViewModel数据;

它会被ActivityThread中performDestroyActivity方法调用，它执行在onDestroy生命周期之前

Activity的attach时会调用getLastNonConfigurationInstance来恢复数据

ViewModel将一直留在内存中，直到限定其存在时间范围的Lifecycle(activity dstroy掉用clear) 永久消失：

UI组件(Activity与Fragment、Fragment与Fragment)间实现数据共享

当这两个 Fragment 各自获取 ViewModelProvider 时，它们会收到相同的 ViewModel 实例 ViewModelProvider通过ViewModelStore获取ViewModel，FragmentActivity自身是持有ViewModelStore

避免内存泄漏的发生。

https://www.jianshu.com/p/41c56570a266

引入了ViewModel和LiveData之后，可以实现vm和view的解耦，只是view引用vm，而vm是不持有view的引用的。在activity退出之后即是还有网络在继续也不会引发内存泄漏和空指针异常

源码解析

https://blog.csdn.net/c10wtiybq1ye3/article/details/89934891

1.Factory是ViewModelProvider的一个内部接口，它的实现类是拿来构建ViewModel实例

3.get mViewModelStore.get(key) create通过newInstance(application)去实例化

ViewModelStore：和名字一样，就是存储ViewModel的，它里面定义了一个HashMap来存储ViewModel，key值是ViewModel全路径+一个默认的前缀

26.4.vm+livedata

Viewmodel

https://juejin.cn/post/6844904079265644551#heading-0

https://www.jianshu.com/p/35d143e84d42

https://www.jianshu.com/p/109644858928

1.how

1.通过ViewModelProviders.of()方法创建ViewModel对象

2.在Activity或者Fragment中，是由Activity和Fragment来提ViewModelStore类对象， 每个Activity或者Fragment都有一个，目的是用于保存该页面的ViewModel对象

2.why

1.管理UI界面数据,数据持久化(将加载数据与数据恢复从 Activity or Fragment中解耦)

在 Android 系统中，需要数据恢复有如下两种场景：

场景1：资源相关的配置发生改变导致 Activity 被杀死并重新创建。 场景2：资源内存不足导致低优先级的 Activity 被杀死。

使用 onSaveInstanceState 与 onRestoreInstanceState

onSaveInstanceState只适合保存少量的可以被序列化、反序列化的数据

onRetainNonConfigurationInstance 方法，用于处理配置发生改变时数据的保存。随后在重新创建的 Activity 中调用 getLastNonConfigurationInstance 获取上次保存的数据

官方最终采用了 onRetainNonConfigurationInstance 的方式来恢复 ViewModel 。

其实就是在屏幕旋转的时候，AMS通过Binder回调Activity的retainNonConfigurationInstances()方法，数据保存就是通过retainNonConfigurationInstances()方法保存在NonConfigurationInstances对象，而再一次使用取出ViewModel的数据的时候，就是从nc对象中取出ViewModelStore对象，而ViewModelStore对象保存有ViewModel集合 ，官方重写了 onRetainNonConfigurationInstance 方法，在该方法中保存了 ViewModelStor

监听 Activity 声明周期，在 onDestory 方法被调用时，判断配置是否改变。如果没有发送改变，则调用 Activity 中的 ViewModelStore 的 clear() 方法，清除所有的 ViewModel

2.Fragments 间共享数据

获取到了Activity的ViewModelStore对象，从而实现了Fragment之间共享ViewModel

为什么不同的Fragment使用相同的Activity对象来获取ViewModel，可以轻易的实现ViewModel共享？

讲道理，如果同学们仔细看了ViewModel的创建流程，这个问题自然迎刃而解。

因为不同的Fragment使用相同的Activity对象来获取ViewModel，在创建ViewModel之前都会先从Activity提供的ViewModelStore中先查询一遍是否已经存在该ViewModel对象。 所以我们只需要先在Activity中同样调用一遍ViewModel的获取代码，即可让ViewModel存在于ViewModelStore中，从而不同的Fragment可以共享一份ViewModel了。

https://juejin.cn/post/6844903919064186888#heading-2（vm总结）

livedata

https://zhuanlan.zhihu.com/p/76747541

what LiveData是一个可被观察的数据容器类

它将数据包装起来，使得数据成为“被观察者”，页面成为“观察者”。这样，当该数据发生变化时，页面能够获得通知，进而更新UI。

可以看到它接收的第一个参数是一个LifecycleOwner对象，在我们的示例中即Activity对象。第二个参数是一个Observer对象。通过最后一行代码将Observer与Activity的生命周期关联在一起。

只有在页面处于激活状态（Lifecycle.State.ON_STARTED或Lifecycle.State.ON_RESUME）时，页面才会收到来自LiveData的通知，如果页面被销毁（Lifecycle.State.ON_DESTROY）

how

在页面中，我们通过LiveData.observe()方法对LiveData包装的数据进行观察，反过来，当我们想要修改LiveData包装的数据时，可通过LiveData.postValue()/LiveData.setValue()来完成。postValue()是在非UI线程中使用，如果在UI线程中，则使用setValue()方法。

why

不会发生内存泄漏

观察者会绑定到 Lifecycle 对象，并在其关联的生命周期遭到销毁后进行自我清理。

不再需要手动处理生命周期

如果观察者的生命周期处于非活跃状态（如返回栈中的 Activity ），则它不会接收任何 LiveData 事件。

---

getLifecycle().addObserver进行观察

activity实现LifecycleOwner，reprotfrgament注册

livedata继承LifecycleObserver，detsroy销毁 其他

26.5.liferecycle

https://liuwangshu.cn/application/jetpack/3-lifecycle-theory.html

https://juejin.cn/post/6844903784166998023

Lifecycle使用

LifecycleObserver:是一个空方法接口，用于标识观察者，对这个 Lifecycle 对象进行监听

LifecycleOwner: 是一个接口，持有方法Lifecycle getLifecycle()。

LifecycleRegistry 类用于注册和反注册需要观察当前组件生命周期的 LifecycleObserver

1.实现LifecycleOwner重写getLifecycle 返回mLifecycleRegistry，mLifecycleRegistry不同生命周期markState

2.继承LifecycleObserver

3.getLifecycle.addObserver注册LifecycleObserver