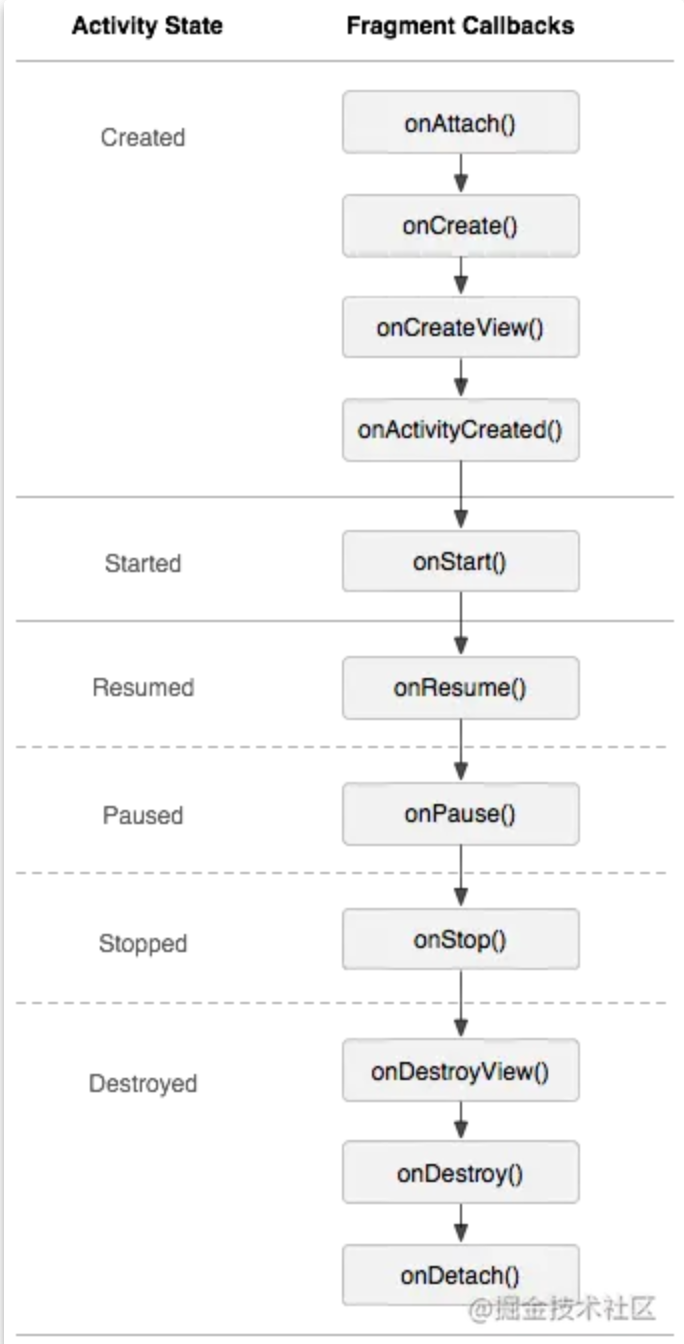
# 生命周期

## 1、初始

不管 Fragment 的直接载体是什么，最终都必须托管在 Activity 中，Fragment 包含有很多个生命周期回调方法，其生命周期会直接受宿主 Activity 生命周期的影响，但也还来源于其它方面。例如，Fragment 的直接载体可以是 Activity 或者是其它更上层的 Fragment，Fragment 会自动随着载体生命周期的变化而变化；如果直接载体是 ViewPager2 的话，切换 ViewPager2 的时候 Fragment 就会单独发生变化。

关于 Fragment 的生命周期有一张比较经典的图片，直接标明了 Activity 和 Fragment 在各个生命周期状态的映射关系：

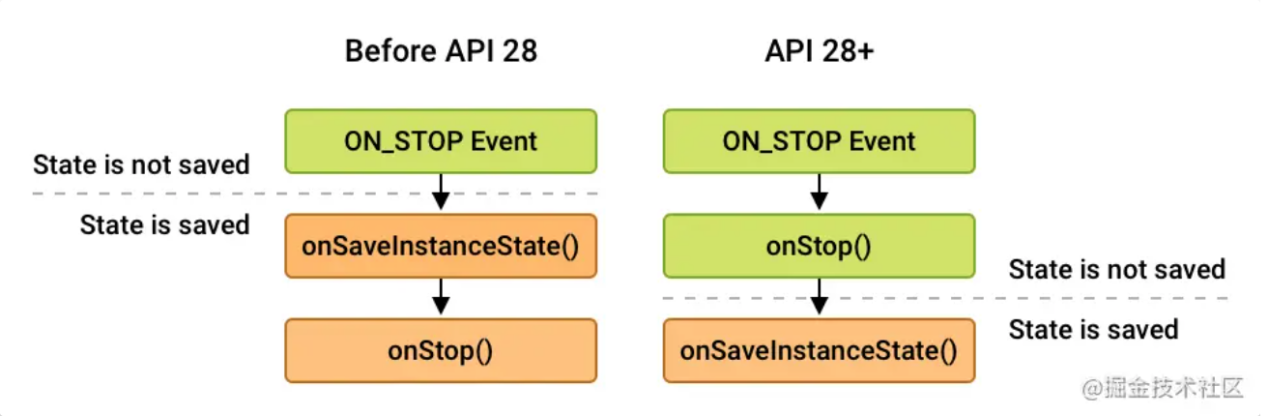


Fragment 的大部分生命周期方法都和 Activity 相映射，但两者的生命周期方法有着明确的先后顺序。以一个通过 FragmentContainerView 添加到 Activity 中的 Fragment 为例，从启动 Activity 到按返回键退出页面的整个过程中，生命周期的变化是：

* Activity 的 onCreate **方法里** 调用 Fragment 的 onAttach(Context)、onAttach(Activity)、onCreate
* Activity 的 onStart **方法里** 调用 Fragment 的 onCreateView、onViewCreated、onActivityCreated、onViewStateRestored、onStart
* Activity 的 onResume **方法后** 调用 Fragment 的 onResume
* Activity 的 onPause **方法里** 调用 Fragment 的 onPause
* Activity 的 onStop **方法里** 调用 Fragment 的 onStop
* Activity 的 onDestroy **方法里** 调用 Fragment 的 onDestroyView、onDestroy、onDetach

可以看到，整个生命周期以 onResume 方法作为分割线，该方法被回调意味着视图已经处于前台活跃状态了，Activity 作为 Fragment 的载体，就需要先保证其自身的 onResume 方法已经回调结束了才能去回调 Fragment 的 onResume 方法，因此两者不存在嵌套调用关系。而对于其它方法，当被回调时就意味着 Activity 处于非活跃状态或者是即将被销毁，此时就需要先回调完成 Fragment 的方法再结束自身，因此就存在嵌套调用关系。

此外，如果 Activity 启动了另外一个 Activity，此时其 onSaveInstanceState 方法就会被调用，此方法一样会嵌套调用 Fragment 的相应方法，但该方法和 onStop 方法的先后顺序在不同系统版本上有着差异性。在 API 28 之前的所有版本，onSaveInstanceState()会在 onStop()方法之前被调用，API 28+ 则相反



当 Activity 从后台返回前台时，如果该 Activity 属于销毁重建的情况的话，Fragment 会重新走一遍生命周期方法，此时其 onCreate、onCreateView、onActivityCreated、onViewStateRestored 方法的参数值 Bundle 就不为 null，当中就保留了我们在 onSaveInstanceState方法中插入的值，我们可以依靠该 Bundle 来帮助页面重建。

此外，目前 onAttach(Activity) 和 onActivityCreated(Bundle) 这两个方法已经被标记为废弃了，这两个方法的初衷是为了让 Fragment 的逻辑能够和载体 Activity 之间建立联系，并得到来自于 Activity 的 onCreate 事件通知，这就导致了开发者有可能会写出和 Activity 强关联的逻辑，官方现在不鼓励这种耦合。View 相关的逻辑应该放在 onViewCreated 方法中处理，其它初始化代码应该在 onCreate 方法中处理，Fragment 的初始化逻辑不应该去依赖于这两个废弃的方法，Fragment 应该更加独立才对

如果实在需要得到 Activity 的 onCreate 事件通知，可以通过在 onAttach(Context)方法中添加 LifecycleObserver 来实现

override fun onAttach(context: Context) {

super.onAttach(context)

requireActivity().lifecycle.addObserver(object : DefaultLifecycleObserver {

override fun onCreate(owner: LifecycleOwner) {

owner.lifecycle.removeObserver(this)

//TODO

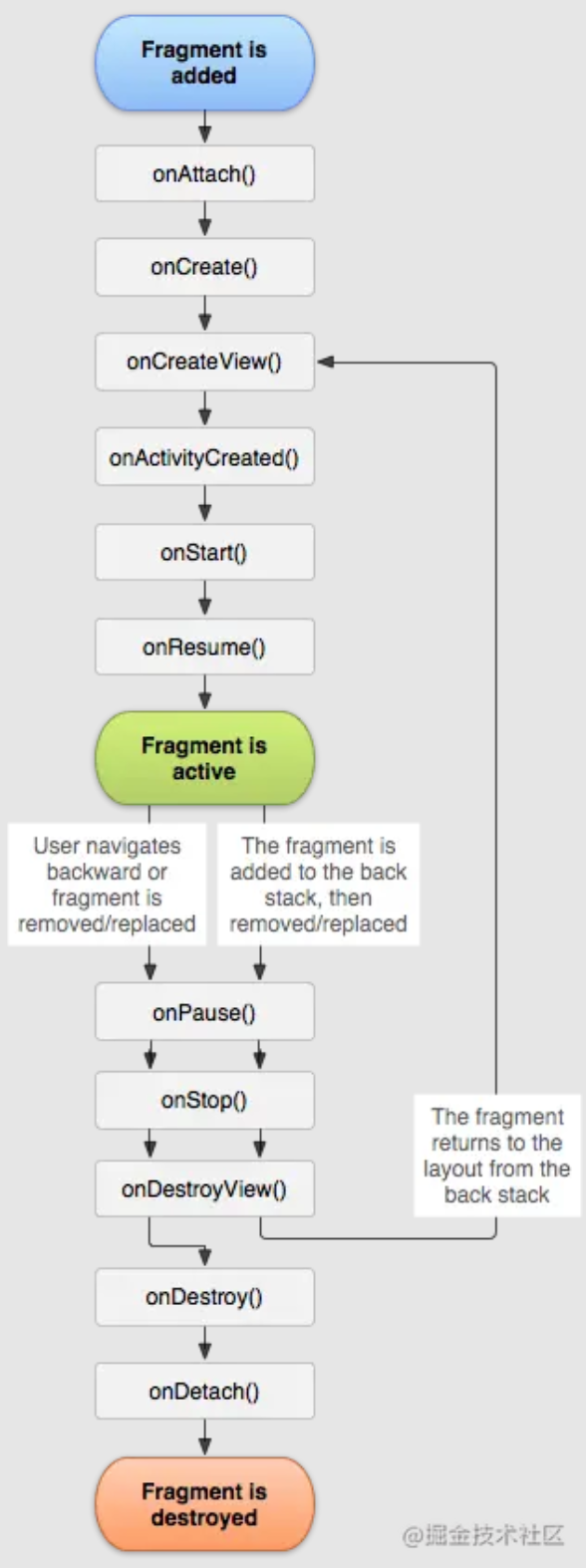
}

})

}

## 2、回退栈 & 事务

Fragment 的生命周期不仅仅只有以上那种线性流程那么简单，不然 Fragment 也不会总是被人吐槽状态过于复杂多变了。Fragment 生命周期的开始节点是 onAttach 方法，结束节点是 onDetach 方法，这个过程中 onCreateView 到 onDestroyView 之间可以被执行 N 多次，就像下图所示



onCreateView 到 onDestroyView 方法之间之所以可能会执行 N 多次，就在于 FragmentTransaction 可以先后执行 N 多次 \*\*移除（remove）\*\*或者 **替换（replace）操作，即先后加载不同 Fragment。新加载的 Fragment 就会取代之前的 Fragment 切换到前台，旧的 Fragment 的视图 View 就会被销毁。如果加载新 Fragment 的操作有添加到回退栈**中，那么当用户点击返回键时旧的 Fragment 就会重新呈现到前台，此时就会重新走一遍 onCreateView 到 onDestroyView 方法之间的流程了

这里就涉及到了关于 **Fragment 回退栈** 的概念了。假设 FragmentA 通过以下代码 replace 为了 FragmentB，那么当用户按返回键时，FragmentB 就会被销毁，而 FragmentA 就会重新执行 onCreateView 到 onDestroyView 之间的所有方法，重新回到前台

supportFragmentManager.commit {

setReorderingAllowed(true)

addToBackStack(null)

replace(

R.id.fragmentContainerView, FragmentB()

)

}

**我们可以通过点击返回键退出 FragmentB，因此直观感受上似乎就是 FragmentB 被添加到了回退栈中，但实际上被添加的并不是 Fragment，而是那些包含了 addToBackStack(String) 方法的一整个事务**

这句话看着很抽象，我举一个例子来帮助读者理解

我们先 add 一个 Fragment 1，然后再 replace 为 Fragment 2，replace 事务中就调用了 addToBackStack(null) 方法，那么当点击返回键时，Fragment 2 就会被销毁，Fragment 1 重新加载到前台页面。大致代码以及运行效果：

supportFragmentManager.commit {

setReorderingAllowed(true)

addToBackStack(null)

add(R.id.fragmentContainerView, Fragment1())

}

supportFragmentManager.commit {

setReorderingAllowed(true)

addToBackStack(null)

replace(R.id.fragmentContainerView, Fragment2())

}

从执行 add 操作到点击返回键，两个 Fragment 的生命周期方法的调用顺序就大致如下所示，省去了部分方法

//先执行 add 操作

Fragment-1: onStart-start

Fragment-1: onResume-start

//再执行 replace 操作

Fragment-1: onPause-start

Fragment-1: onStop-start

Fragment-2: onAttach-context-start

Fragment-2: onCreate-start

Fragment-2: onResume-start

Fragment-1: onDestroyView-start

//点击返回键

Fragment-2: onPause-start

Fragment-2: onStop-start

Fragment-1: onCreateView-start

Fragment-1: onActivityCreated-start

Fragment-1: onViewStateRestored-start

Fragment-1: onStart-start

Fragment-1: onResume-start

Fragment-2: onDestroyView-start

Fragment-2: onDestroy-start

Fragment-2: onDetach-start

这整个过程的调用关系可以总结为：

* 当执行 replace 操作时。Fragment 1 会先执行到 onStop，意味着 Fragment 1 已经被切换到后台了。然后就开始执行 Fragment 2 的生命周期直到 onResume ，意味着 Fragment 2 已经被切换到前台了。之后就会接着执行 Fragment 1 的 onDestroyView ，此时就意味着 Fragment 1 不仅被切换到了后台，其 View 也被销毁了
* 当点击返回键时。Fragment 2 会先执行到 onStop，意味着 Fragment 2 已经被切换到后台了。Fragment 1 则会再次执行从 onCreateView 到 onResume 之间的所有方法，意味着重新回到了前台。之后就会接着执行 Fragment 2 的生命周期直到 onDetach ，此时 Fragment 2 就被完全销毁了，无法再次回到该实例页面

点击返回键时，由于 replace 操作调用了addToBackStack(null) 方法，意味着该事务加入到了回退栈中，因此此时响应了返回键事件的其实就是该事务，所以 replace 操作就会被撤销，FragmentManager 负责将视图恢复到 replace 之前的状态，因此 Fragment 2 整个实例被完全销毁了，Fragment 1 得以重新回到前台

所以说，**FragmentTransaction 的回退栈中保留的是事务而非具体的 Fragment 实例，能响应返回事件的是我们向其中提交的事务，具体的响应结果就是将该事务撤销，恢复到之前的状态**

我们先 add 一个 Fragment 1，然后再 replace 为 Fragment 2，replace 事务中**不调用** addToBackStack(null) 方法，那么此时就需要点击两次返回键才能退出 Fragment 2 了，且此时 Activity 也会随着退出。大致代码以及运行效果：

supportFragmentManager.commit {

setReorderingAllowed(true)

addToBackStack(null)

add(R.id.fragmentContainerView, Fragment1())

}

supportFragmentManager.commit {

setReorderingAllowed(true)

//addToBackStack(null)

replace(R.id.fragmentContainerView, Fragment2())

}

从执行 add 操作到点击两次返回键，两个 Fragment 的生命周期方法的调用顺序就大致如下所示，省去了部分方法

//先执行 add 操作

Fragment-1: onStart-start

Fragment-1: onResume-start

//再执行 replace 操作

Fragment-1: onPause-start

Fragment-1: onStop-start

Fragment-2: onAttach-context-start

Fragment-2: onCreate-start

Fragment-2: onResume-start

Fragment-1: onDestroyView-start

//点击返回键

Fragment-1: onDestroy-start

Fragment-1: onDetach-start

//再次点击返回键

Fragment-2: onPause-start

Fragment-2: onStop-start

Fragment-2: onDestroyView-start

Fragment-2: onDestroy-start

Fragment-2: onDetach-start

 当第一次点击返回键时。由于 replace 事务并没有被添加到回退栈中，而 add 操作有，所以此时响应了返回事件的是 add 操作，点击返回键就相当于把 add 操作给撤销掉了，因此 Fragment 1 就会执行到 onDetach 方法，Fragment 2 不受影响

 当第二次点击返回键时，由于此时 FragmentTransaction 的回退栈为空，所以此时响应了返回事件的其实是 Activity，所以 Activity 会退出，连带着 Fragment 2 也一起被销毁了

## 3、FragmentLifecycle

在最早的时候，**生命周期**这个词对于 Activity 和 Fragment 来说指的就是其特定的回调方法是否已经被执行了，例如 onCreate、onStart、onDestroy 等方法。现如今也指这两者的 Lifecycle.State 的当前值是什么

Activity 和 Fragment 都实现了 LifecycleOwner 接口，都包含了一个 Lifecycle 对象用于标记其生命周期状态，因此我们能够在这两者中以和生命周期绑定的方式对 LiveData 进行监听，就像以下代码一样，当中 textLiveData 关联的 this 即 LifecycleOwner 对象，从而保证了只有当 Fragment 处于前台活跃状态时才会收到数据回调

class PageFragment : Fragment() {

private val pageViewModel by lazy {

ViewModelProvider(this@PageFragment).get(PageViewModel::class.java).apply {

textLiveData.observe(this@PageFragment, {

})

}

}

}

Lifecycle.State 一共包含五种值，FragmentLifecycle 会在这五个值中不断流转，例如当切换为 DESTROYED 状态时，也即意味 onDestory()、onDetach() 等方法被调用了，至此 Fragment 的本次生命周期也就结束了

public enum State {

DESTROYED,

INITIALIZED,

CREATED,

STARTED,

RESUMED;

}

## 4、FragmentViewLifecycle

Fragment 相对于 Activity 来说比较特殊，因为其关联的 View 对象可以在单次 FragmentLifecycle 过程中先后多次加载和销毁，因此实际上 **FragmentView 的生命周期和 Fragment 并不同步**

Fragment 内部也声明了九种状态值用于标记其自身的生命周期状态，当中就包含一个 VIEW\_CREATED，即表示 FragmentView 已经被创建了，从这也可以看出 Fragment 内部维护的生命周期密度要比 FragmentLifecycle 小得多

static final int INITIALIZING = -1; // Not yet attached.

static final int ATTACHED = 0; // Attached to the host.

static final int CREATED = 1; // Created.

static final int VIEW\_CREATED = 2; // View Created.

static final int AWAITING\_EXIT\_EFFECTS = 3; // Downward state, awaiting exit effects

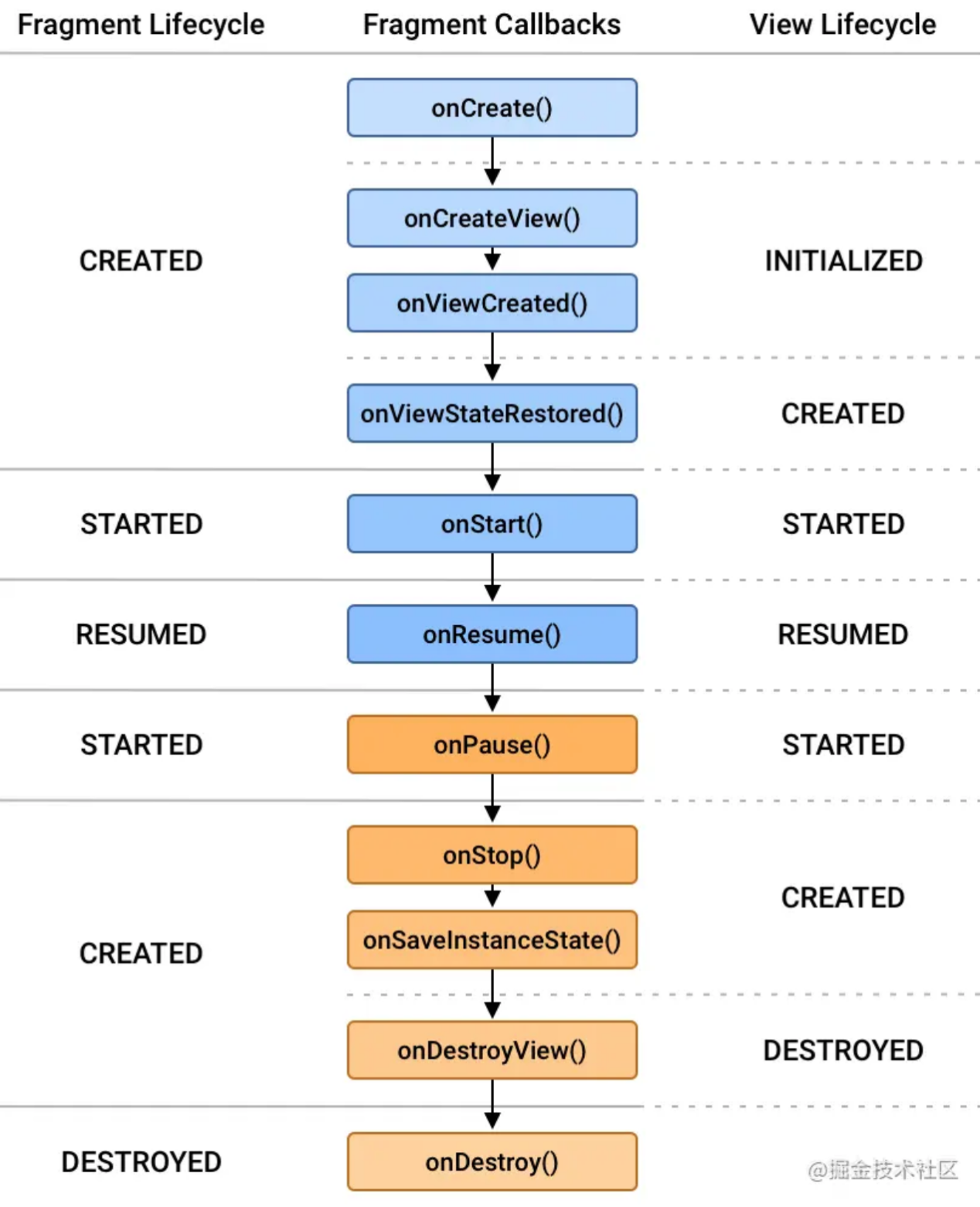
static final int ACTIVITY\_CREATED = 4; // Fully created, not started.

static final int STARTED = 5; // Created and started, not resumed.

static final int AWAITING\_ENTER\_EFFECTS = 6; // Upward state, awaiting enter effects

static final int RESUMED = 7; // Created started and resumed.

状态值切换到 DESTROYED，对于 FragmentLifecycle 来说意味着 onDestroy、onDetach 等方法被调用，对于 FragmentViewLifecycle 来说则意味着onDestroyView方法被调用，因此 FragmentViewLifecycle 的跨度范围要比 FragmentLifecycle 小一些。而且 FragmentLifecycle 切换到 DESTROYED 后状态值是不可逆的，无法再次更改，而 FragmentViewLifecycle 切换到 DESTROYED 后是有机会再次更改的，因为 View 对象可以先后多次加载和销毁，每次加载就意味着生命周期的重新开始



Fragment 提供了一个getViewLifecycleOwner()方法由于提供 FragmentViewLifecycle，从中可以看出该方法只能在 onCreateView() 到 onDestroyView() 之间被调用，即只能在 FragmentView 创建了且销毁之前使用，否则将直接抛出异常

@Nullable

FragmentViewLifecycleOwner mViewLifecycleOwner;

@MainThread

@NonNull

public LifecycleOwner getViewLifecycleOwner() {

if (mViewLifecycleOwner == null) {

throw new IllegalStateException("Can't access the Fragment View's LifecycleOwner when "

+ "getView() is null i.e., before onCreateView() or after onDestroyView()");

}

return mViewLifecycleOwner;

}

FragmentViewLifecycle 非常有用，我们在日常开发中可以根据实际情况使用 FragmentViewLifecycle 来替代 FragmentLifecycle，因为 FragmentLifecycle 存在着一些并不明显的使用误区，很容易就造成 bug

举个例子。假设我们的 Fragment 需要监听 ViewModel 中某个 LiveData 值的变化，并根据监听到的值来设置界面，此时就要考虑在 Fragment 中的哪里来订阅 LiveData 了。看了以上内容后，我们已经知道 onCreateView 方法到 onDestoryView之间是可能会先后执行多次的，那么监听操作就不应该放在这里面了，否则就会造成重复订阅。此时我们想到的可能就是这两种方式了：**声明全局变量 ViewModel 时顺便监听** 或者是 **在 onCreate 方法中进行监听**

private val pageViewModel by lazy {

ViewModelProvider(this).get(PageViewModel::class.java).apply {

textLiveData.observe(this@PageFragment, Observer {

//TODO

})

}

}

private val pageViewModel by lazy {

ViewModelProvider(this).get(PageViewModel::class.java)

}

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

pageViewModel.textLiveData.observe(this@PageFragment, Observer {

//TODO

})

}

这两种方式都可以避免重复订阅的问题，但此时还存在另一个藏得很深的问题：**假如 FragmentView 真的销毁重建了，重建后的 FragmentView 也收不到 textLiveData 已有的数据！！！**

会出现该问题的本质原因还是因为 FragmentView 的生命周期和 Fragment 并不同步。如果 Fragment 已经接收过 textLiveData 的回调了，那么当 FragmentView 销毁重建后，由于 textLiveData 的值没有发生变化，和 textLiveData 绑定的 LifecycleOwner 也还一直存在着，那么重建后的 FragmentView 自然就不会收到 textLiveData 的回调了，从而导致无法根据回调来重建页面

为了解决该问题，就需要使用到 FragmentViewLifecycle 了。由于 FragmentViewLifecycle 的生命周期在 onDestoryView的时候就结束了，此时也会自动移除 Observer，因此我们可以直接在 onViewCreated 方法中使用 viewLifecycleOwner 来监听 textLiveData，从而保证每次重建后的 FragmentView 都能收到回调

private val pageViewModel by lazy {

ViewModelProvider(this).get(PageViewModel::class.java)

}

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

super.onViewCreated(view, savedInstanceState)

pageViewModel.textLiveData.observe(viewLifecycleOwner, Observer {

//TODO

})

}

在大部分情况下，我们在 Fragment 中执行的操作都是和 FragmentView 强关联的，属于视图操纵行为，此时就可以使用 FragmentViewLifecycle 来替代 FragmentLifecycle，从而保证事件一定只有在 FragmentView 存在且活跃的情况下才会被回调，且保证了每次 FragmentView 被销毁重建的时候都能够得到最新数据。而对于那些依赖于 Fragment 完整生命周期的事件，就还是只能继续使用 FragmentLifecycle 了

# setMaxLifecycle

在以前，Fragment 往往会和 ViewPager 配套使用，由于 ViewPager 存在页面**预加载**的机制，相邻 Fragment 会在对用户不可见的时候就执行onResume方法，从而触发一些不必要的业务逻辑，导致多余的流量消耗和性能消耗

为了解决以上问题，实现 Fragment **懒加载**的效果，我们需要在onViewCreated、onResume、onHiddenChanged、setUserVisibleHint等多个方法中添加标记位，通过组合标记位来准确得知 Fragment 当前是否真的对用户可见，且只在可见的情况下才去触发关联的业务逻辑。这种实现方式长久以来一直很流行，也完全可用，但也真的不够优雅，我们需要组合多个条件才能实现目的，且 Fragment 在不可见的时候就去回调onResume方法本身也是一种很违反常识的行为

现如今我们也有了更好的选择，setUserVisibleHint方法已经被废弃了，从注释可以看到官方现在推荐使用setMaxLifecycle 方法来更为精准地控制 Fragment 的生命周期

**SetMaxLifecycle**方法从名字就可以看出来是用于为 Fragment 设置一个最大的生命周期状态，实际上也的确是，state 参数值我们只能选择 CREATED、STARTED、RESUMED 三者之一

@NonNull

public FragmentTransaction setMaxLifecycle(@NonNull Fragment fragment,

@NonNull Lifecycle.State state) {

addOp(new Op(OP\_SET\_MAX\_LIFECYCLE, fragment, state));

return this;

}

在正常情况下，我们在 Activity 中 add 一个 Fragment 后其生命周期流程是会直接执行到 **onResume** 方法的

supportFragmentManager.commit {

val fragment = FragmentLifecycleFragment.newInstance(

fragmentTag = (tagIndex++).toString(),

bgColor = Color.parseColor("#0091EA")

)

add(

R.id.fragmentContainerView, fragment

)

setReorderingAllowed(true)

addToBackStack(null)

}

Fragment-1: onAttach

Fragment-1: onCreate

Fragment-1: onCreateView

Fragment-1: onViewCreated

Fragment-1: onActivityCreated

Fragment-1: onViewStateRestored

Fragment-1: onStart

Fragment-1: onResume

而如果设置了setMaxLifecycle(fragment, Lifecycle.State.CREATED)的话，就会发现 Fragment 只会执行到 onCreate 方法，且 Fragment 也不会显示，毕竟 FragmentView 没有被创建，自然也不会被挂载到 Activity 中

supportFragmentManager.commit {

val fragment = FragmentLifecycleFragment.newInstance(

fragmentTag = (tagIndex++).toString(),

bgColor = Color.parseColor("#0091EA")

)

add(

R.id.fragmentContainerView, fragment

)

setMaxLifecycle(fragment, Lifecycle.State.CREATED)

setReorderingAllowed(true)

addToBackStack(null)

}

Fragment-1: onAttach

Fragment-1: onCreate

而如果 Fragment 已经回调了 **onResume** 方法，此时再来为其设置 **setMaxLifecycle(fragment, Lifecycle.State.STARTED)** 的话，就会发现 Fragment 会回调 **onPause** 方法

Fragment-1: onAttach

Fragment-1: onCreate

Fragment-1: onCreateView

Fragment-1: onViewCreated

Fragment-1: onActivityCreated

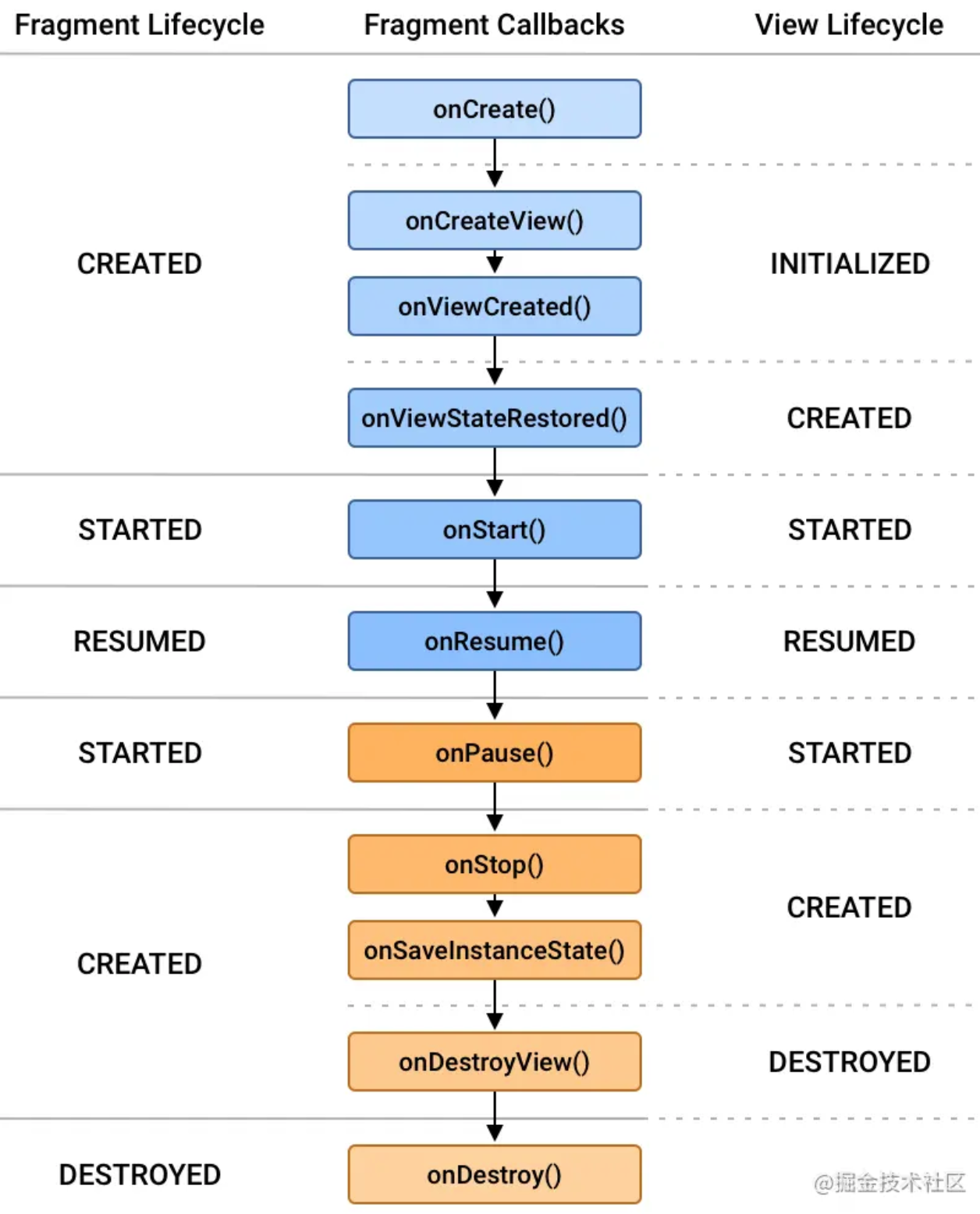
Fragment-1: onViewStateRestored

Fragment-1: onStart

Fragment-1: onResume

Fragment-1: onPause

其实，setMaxLifecycle 控制的是 FragmentLifecycle，同个 FragmentLifecycle 状态值对应的可能是不同的生命周期方法，就像以下图片所示。onResume方法对应的是 RESUMED，此时我们要求 FragmentLifecycle 的最大值是 STARTED，那么 Fragment 只能执行onPause 方法将状态值切换到 STARTED。而如果我们一开始就设置了最大值是 STARTED 的话，Fragment 就只会从 onAttach 执行到 onStart 方法，此时 FragmentLifecycle 一样是 STARTED



所以说，setMaxLifecycle 方法会根据 Fragment 的现有状态来加大或者回退当前的 FragmentLifecycle，选择合适的回调路线来进行状态切换

setMaxLifecycle 方法目前已经应用于 ViewPager2 中的 FragmentStateAdapter 了。当 Fragment 被切出时，就将其最大状态设置为 STARTED，当 Fragment 被切入时，就将其最大状态设置为 RESUMED，从而使得只有当前可见的 Fragment 才会被回调 onResume 方法，被切出的 Fragment 则会回调 onPause 方法，保证了每个 Fragment 都能处于正确的生命周期状态

void updateFragmentMaxLifecycle(boolean dataSetChanged) {

···

Fragment toResume = null;

for (int ix = 0; ix < mFragments.size(); ix++) {

long itemId = mFragments.keyAt(ix);

Fragment fragment = mFragments.valueAt(ix);

if (!fragment.isAdded()) {

continue;

}

if (itemId != mPrimaryItemId) {

//重点

transaction.setMaxLifecycle(fragment, STARTED);

} else {

toResume = fragment; // itemId map key, so only one can match the predicate

}

fragment.setMenuVisibility(itemId == mPrimaryItemId);

}

if (toResume != null) { // in case the Fragment wasn't added yet

//重点

transaction.setMaxLifecycle(toResume, RESUMED);

}

if (!transaction.isEmpty()) {

transaction.commitNow();

}

}

# FragmentFactory

在以前，Fragment 要求开发者必须为每个子类均声明一个**无参构造函数**，因为当系统发生 Configuration Change 时，系统需要恢复重建每个 Fragment，如果 Fragment 包含一个或者多个**有参构造函数**的话，系统不知道应该调用哪个构造函数，而且也无法自动生成**构造参数**，因此只能选择通过**反射无参构造函数**的方式来完成实例化，这就要求每个子类都必须拥有一个无参构造函数了

为了解决该问题，按以往的方式我们都是通过声明一个**静态工厂方法**来提供实例化的入口，并通过 Bundle 来传递请求参数。当系统在恢复重建 Fragment 时也会自动将重建前的 Bundle 注入到新的实例中，从而保证请求参数不会缺失

class FragmentFactoryFragment : BaseFragment(R.layout.fragment\_fragment\_factory) {

companion object {

private const val KEY\_BG\_COLOR = "keyBgColor"

fun newInstance(bgColor: Int): FragmentFactoryFragment {

return FragmentFactoryFragment().apply {

arguments = Bundle().apply {

putInt(KEY\_BG\_COLOR, bgColor)

}

}

}

}

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

val bgColor = arguments?.getInt(KEY\_BG\_COLOR) ?: throw IllegalArgumentException()

val flRoot = view.findViewById<View>(R.id.flRoot)

flRoot.setBackgroundColor(bgColor)

}

}

通过反射无参构造函数来实例化 Fragment 并注入 **arguments** 的过程，就对应 Fragment 中的 **instantiate** 方法

public static Fragment instantiate(@NonNull Context context, @NonNull String fname,

@Nullable Bundle args) {

try {

Class<? extends Fragment> clazz = FragmentFactory.loadFragmentClass(

context.getClassLoader(), fname);

//反射无参构造函数

Fragment f = clazz.getConstructor().newInstance();

if (args != null) {

args.setClassLoader(f.getClass().getClassLoader());

//注入 Bundle

f.setArguments(args);

}

return f;

} catch (java.lang.InstantiationException e) {

throw new InstantiationException("Unable to instantiate fragment " + fname

+ ": make sure class name exists, is public, and has an"

+ " empty constructor that is public", e);

} catch (IllegalAccessException e) {

throw new InstantiationException("Unable to instantiate fragment " + fname

+ ": make sure class name exists, is public, and has an"

+ " empty constructor that is public", e);

} catch (NoSuchMethodException e) {

throw new InstantiationException("Unable to instantiate fragment " + fname

+ ": could not find Fragment constructor", e);

} catch (InvocationTargetException e) {

throw new InstantiationException("Unable to instantiate fragment " + fname

+ ": calling Fragment constructor caused an exception", e);

}

}

为了解决**无法自由定义有参构造函数的问题**，Fragment 如今也提供了 FragmentFactory 来参与实例化 Fragment 的过程

首先，假设 Fragment 的构造函数需要一个 int 类型的构造参数

class FragmentFactoryFragment(private val bgColor: Int) :

BaseFragment(R.layout.fragment\_fragment\_factory) {

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

val flRoot = view.findViewById<View>(R.id.flRoot)

flRoot.setBackgroundColor(bgColor)

}

}

继承 FragmentFactory，在 **instantiate**方法中判断当前要实例化的是哪一个 Fragment，在此根据 bgColor 实例化 Fragment 并返回

class MyFragmentFactory(private val bgColor: Int) : FragmentFactory() {

override fun instantiate(classLoader: ClassLoader, className: String): Fragment {

val clazz = loadFragmentClass(classLoader, className)

if (clazz == FragmentFactoryFragment::class.java) {

return FragmentFactoryFragment(bgColor)

}

return super.instantiate(classLoader, className)

}

}

之后我们在代码中仅需要向 **supportFragmentManager** 声明需要注入的 Fragment Class 即可，无需显式实例化，实例化过程交由 MyFragmentFactory 来完成

class FragmentFactoryActivity : BaseActivity() {

override val bind by getBind<ActivityFragmentFactoryBinding>()

override fun onCreate(savedInstanceState: Bundle?) {

//需要在 super.onCreate 之前调用，因为 Activity 需要依靠 FragmentFactory 来完成 Fragment 的恢复重建

supportFragmentManager.fragmentFactory = MyFragmentFactory(Color.parseColor("#00BCD4"))

super.onCreate(savedInstanceState)

supportFragmentManager.commit {

add(R.id.fragmentContainerView, FragmentFactoryFragment::class.java, null)

setReorderingAllowed(true)

disallowAddToBackStack()

}

}

}

FragmentFactory 的好处有：

* 将本应该直接传递给 Fragment 的构造参数转交给了 FragmentFactory，这样系统在恢复重建时就能统一通过 instantiate 方法来重新实例化 Fragment，而无需关心 Fragment 的构造函数
* 只要 FragmentFactory 包含了所有 Fragment 均需要的构造参数，那么同个 FragmentFactory 就可以用于实例化多种不同的 Fragment，从而解决了需要为每个 Fragment 均声明静态工厂方法的问题，Fragment 也省去了向 Bundle 赋值取值的操作，减少了开发者的工作量

FragmentFactory 也存在着局限性：

* 由于需要考虑 Fragment 恢复重建的场景，因此我们在 super.onCreate 之前就需要先初始化 supportFragmentManager.fragmentFactory，这样 Activity 在恢复重建的时候才能根据已有参数来重新实例化 Fragment，这就要求我们必须在一开始的时候就确定 FragmentFactory 的构造参数，也即 Fragment 的构造参数，而这在日常开发中并非总是能够做到的，因为 Fragment 的构造参数可能是需要动态生成的

# Activity Result API

Activity 和 Fragment 中有两组历史悠久的 API 如今都已经被废弃了：

* startActivityForResult()、onActivityResult()
* requestPermissions()、onRequestPermissionsResult()

虽然这两组 API 所执行的操作并不一样，但都属于**发起请求并等待回调结果**的操作类型，官方建议使用新增的 registerForActivityResult 方法来实现这类需求，因为通过 Activity Result API 实现的代码更有复用性，减少了模板代码且易于测试

看个小例子。假设存在一个 ActivityResultApiFragment 和一个 ActivityResultApiTestActivity。Fragment 用于输入用户名，在拿到用户名后需要将值传递给 Activity，由其负责查询用户所在地，查到后再将值回传给 Fragment

按照需求，userName 和 location 都需要保存在 Intent 中，以便在两者之间实现数据交换，保存 userName 和取出 location 的过程都由 ActivityResultContract 来实现，对应 GetLocation

* GetLocation 的两个泛型声明就分别对应着**请求参数的数据类型**以及**返回参数的数据类型**
* createIntent方法用于保存请求参数 userName，并指定要启动的是 ActivityResultApiTestActivity
* parseResult 方法会拿到 ActivityResultApiTestActivity 返回的 Intent 对象，如果判断为成功状态的话则直接取出 location 并返回

class GetLocation : ActivityResultContract<String, String>() {

companion object {

private const val KEY\_REQ\_LOCATION = "keyReqLocation"

private const val KEY\_RES\_LOCATION = "keyResLocation"

fun getUserName(intent: Intent): String {

return intent.getStringExtra(KEY\_REQ\_LOCATION)

?: throw RuntimeException("userName must not be null")

}

fun putResultOk(location: String): Intent {

return Intent().apply {

putExtra(KEY\_RES\_LOCATION, location)

}

}

}

override fun createIntent(context: Context, input: String): Intent {

val intent = Intent(context, ActivityResultApiTestActivity::class.java)

intent.putExtra(KEY\_REQ\_LOCATION, input)

return intent

}

override fun parseResult(resultCode: Int, intent: Intent?): String? {

if (resultCode == Activity.RESULT\_OK) {

return intent?.getStringExtra(KEY\_RES\_LOCATION)

}

return null

}

}

ActivityResultApiTestActivity 按照我们惯常的方式获取到请求参数，返回 RESULT\_OK 状态码和结果值

class ActivityResultApiTestActivity : BaseActivity() {

override val bind by getBind<ActivityActivityResultApiTestBinding>()

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

bind.btnFinish.setOnClickListener {

val userName = GetLocation.getUserName(intent)

setResult(Activity.RESULT\_OK, GetLocation.putResultOk("$userName-广州"))

finish()

}

}

}

ActivityResultApiFragment 通过 registerForActivityResult 方法来注册 GetLocation，需要的时候再通过getLocation.launch("业志陈")来传入用户名并启动 GetLocation，最终在 onActivityResult 方法中获取到请求结果

class ActivityResultApiFragment :

Fragment(R.layout.fragment\_activity\_result\_api) {

private val getLocation: ActivityResultLauncher<String> =

registerForActivityResult(

GetLocation(),

object : ActivityResultCallback<String> {

override fun onActivityResult(result: String) {

Toast.makeText(activity, result, Toast.LENGTH\_SHORT).show()

}

})

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

val btnStartActivity = view.findViewById<Button>(R.id.btnStartActivity)

btnStartActivity.setOnClickListener {

getLocation.launch("业志陈")

}

}

}

类似地，官方也提供了一个用于请求权限的 ActivityResultContract 实现类：RequestMultiplePermissions，通过该 Contract 我们也能以很简单的方式完成以前割裂且麻烦的申请操作

class ActivityResultApiFragment :

Fragment(R.layout.fragment\_activity\_result\_api) {

private val requestPermissions: ActivityResultLauncher<Array<String>> =

registerForActivityResult(

ActivityResultContracts.RequestMultiplePermissions(),

object : ActivityResultCallback<Map<String, Boolean>> {

override fun onActivityResult(result: Map<String, Boolean>) {

for (entry in result) {

Toast.makeText(activity, entry.key + " " + entry.value, Toast.LENGTH\_SHORT)

.show()

}

}

})

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

val btnRequestPermissions = view.findViewById<Button>(R.id.btnRequestPermissions)

btnRequestPermissions.setOnClickListener {

requestPermissions.launch(

arrayOf(

Manifest.permission.ACCESS\_NETWORK\_STATE,

Manifest.permission.ACCESS\_FINE\_LOCATION

)

)

}

}

}

使用 Activity Result API 的好处有：

* 完全隔绝了请求的发起者和处理者之间的引用关系，发起者直接面向于 ActivityResultContract，无需知道请求最终是交由谁处理，这样开发者就可以随时替换 ActivityResultContract 的具体实现逻辑了，避免耦合且易于测试
* 请求的发起逻辑和处理逻辑都能够得到复用，减少了重复代码
* 统一了 startActivityForResult 和 requestPermissions的请求方式，并提供了更加简洁方便的回调方法，避免了以前申请逻辑和回调逻辑是完全割裂开的情况

# Fragment Result API

Activity Result API 适用于 Fragment 和非载体 Activity 之间进行数据通信的场景，如果是**多个 Fragment 之间**或者是 **Fragment 和载体 Activity 之间**要进行数据通信的话，我们可以选择的实现方法有很多种：

* Fragment 通过 FragmentManager 拿到另一个 Fragment 的实例对象，通过直接调用对方的方法来实现多个 Fragment 之间的数据通信
* Fragment 和 Activity 之间相互注册回调接口，以此来实现 Fragment 和载体 Activity 之间的数据通信
* Fragment 以 Activity 作为中转站将事件转交给其它 Fragment，以此来实现多个 Fragment 之间的数据通信
* Fragment 和 Activity 同时持有相同一个 Activity 级别的 ViewModel 实例，以此来实现多个 Fragment 之间、Fragment 和载体 Activity 之间的数据通信

以上四种方法都可以实现需求，但也存在着一些问题：

* 前三种方法造成了 Fragment 之间、Fragment 和 Activity 之间存在强耦合，且由于并不清楚对方处于什么生命周期状态，一不小心就可能导致 NPE
* 最后一种方法对于生命周期安全有保障，但如果只是简单的数据通信场景的话，使用 ViewModel 就有点大材小用了，ViewModel 比较适用于数据量稍大的业务场景

如今 Fragment 也有了一种新的选择：FragmentResult。使用 FragmentResult 进行数据通信不需要持有任何 Fragment 或者 Activity 的引用，仅需要使用 FragmentManager 就可以实现

看个小例子。声明一个 Activity 和两个 Fragment，分别向对方正在监听的 requestKey 下发数据，数据通过 Bundle 来进行传递，Activity 和 Fragment 只需要面向特定的 requestKey 来发送数据或者监听数据即可，无需关心数据的来源和去向

* setFragmentResult 方法表示的是向 requestKey 下发数据
* setFragmentResultListener 表示的是对 requestKey 进行监听

const val requestKeyToActivity = "requestKeyToActivity"

private const val requestKeyFirst = "requestKeyFirst"

private const val requestKeySecond = "requestKeySecond"

class FragmentResultApiAFragment : Fragment(R.layout.fragment\_fragment\_result\_api\_a) {

private val requestKeyFirst = "requestKeyFirst"

private val requestKeySecond = "requestKeySecond"

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

super.onViewCreated(view, savedInstanceState)

val btnSend = view.findViewById<Button>(R.id.btnSend)

val btnSendMsgToActivity = view.findViewById<Button>(R.id.btnSendMsgToActivity)

val tvMessage = view.findViewById<TextView>(R.id.tvMessage)

btnSend.setOnClickListener {

val bundle = Bundle()

bundle.putString("fromFragmentResultApiAFragment", Random.nextInt().toString())

//向对 requestKeyFirst 进行监听的 Fragment 传递数据

parentFragmentManager.setFragmentResult(requestKeyFirst, bundle)

}

btnSendMsgToActivity.setOnClickListener {

val bundle = Bundle()

bundle.putString("fromFragmentResultApiAFragment", Random.nextInt().toString())

//向对 requestKeyToActivity 进行监听的 Activity 传递数据

parentFragmentManager.setFragmentResult(requestKeyToActivity, bundle)

}

//对 requestKeySecond 进行监听

parentFragmentManager.setFragmentResultListener(

requestKeySecond,

this,

{ requestKey, result ->

tvMessage.text = "requestKey: $requestKey \n result: $result"

})

}

}

class FragmentResultApiBFragment : Fragment(R.layout.fragment\_fragment\_result\_api\_b) {

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

super.onViewCreated(view, savedInstanceState)

val btnSend = view.findViewById<Button>(R.id.btnSend)

val tvMessage = view.findViewById<TextView>(R.id.tvMessage)

btnSend.setOnClickListener {

val bundle = Bundle()

bundle.putString("fromFragmentResultApiBFragment", Random.nextInt().toString())

//向对 requestKeySecond 进行监听的 Fragment 传递数据

parentFragmentManager.setFragmentResult(requestKeySecond, bundle)

}

//对 requestKeyFirst 进行监听

parentFragmentManager.setFragmentResultListener(

requestKeyFirst,

this,

{ requestKey, result ->

tvMessage.text = "requestKey: $requestKey \n result: $result"

})

}

}

class FragmentResultApiActivity : BaseActivity() {

override val bind by getBind<ActivityFragmentResultApiBinding>()

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

supportFragmentManager.setFragmentResultListener(

requestKeyToActivity,

this,

{ requestKey, result ->

bind.tvMessage.text = "requestKey: $requestKey \n result: $result"

})

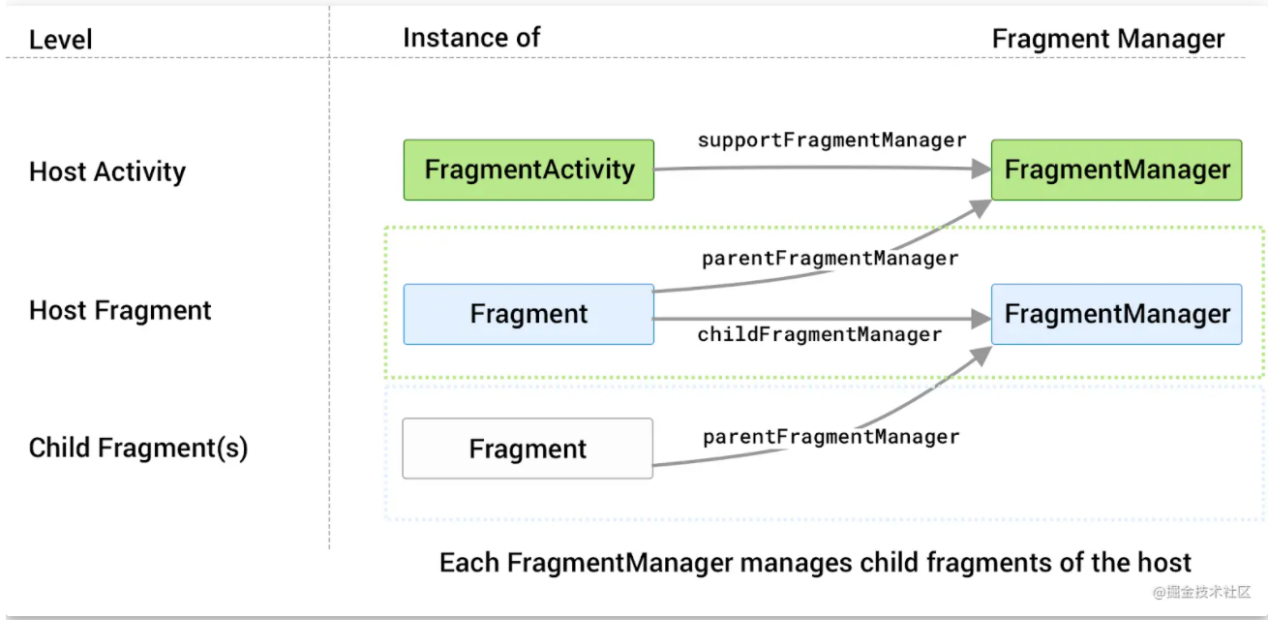
}

}

使用 Fragment Result API 的好处有：

* Fragment Result API 并不局限于 Fragment 之间，只要 Activity 也持有和 Fragment 相同的 FragmentManager 实例，也可以用以上方式在 Activity 和 Fragment 之间实现数据通信。但需要注意，一个 requestKey 所属的数据只能被一个消费者得到，后注册的 FragmentResultListener 会把先注册的给覆盖掉
* Fragment Result API 也可以用于在父 Fragment 和子 Fragment 之间传递数据，但前提是两者持有的是相同的 FragmentManager 实例。例如，如果要将数据从子 Fragment 传递给父 Fragment，父 Fragment 应通过getChildFragmentManager() 来调用setFragmentResultListener()，而不是getParentFragmentManager()
* 数据的发送者只负责下发数据，不关心也不知道有多少个接收者，也不知道数据是否有被消费。数据的接收者只负责接收数据，不关心也不知道有多少个数据源。避免了 Activity 和 Fragment 之间存在引用关系，使得每个个体之间更加独立
* 只有当 Activity 和 Fragment 至少处于 STARTED 状态，即只有处于活跃状态时才会接收到数据回调，非活跃状态下连续传值也只会保留最新值，当切换到 DESTROYED 状态时也会自动移除监听，从而保证了生命周期的安全性

每一个载体（Activity 或者 Fragment）都包含一个和自身同等级的 FragmentManager 用于管理子 Fragment，对应 Activity 的 supportFragmentManager 和 Fragment 的 childFragmentManager；每一个 子 Fragment 也都包含一个来自于载体的 FragmentManager，对应 Fragment 的 parentFragmentManager



# OnBackPressed

Activity 包含有一个 onBackPressed方法用于响应用户点击返回键的操作，可用来控制是否允许用户退出当前页面，但 Fragment 并不包含类似方法。如果想要让 Fragment 也可以拦截用户的返回操作的话，按照以往的方法就需要通过在 Activity 和 Fragment 之间建立回调来进行拦截，这无疑也造成了两者之间存在耦合

如今开发者有了更好的实现方式：通过 OnBackPressedDispatcher 使得任意可以访问 Activity 的代码逻辑都可以拦截 onBackPressed方法

我们可以在 Fragment 中向 Activity 添加一个 OnBackPressedCallback 回调，传递的值 true 即代表该 Fragment 会拦截用户的每一次返回操作并进行回调，我们需要根据业务逻辑在合适的时候将其置为 false，从而放开对onBackPressed的控制权。此外，addCallback 方法的第一个参数是 LifecycleOwner 类型，也即当前的 Fragment 对象，该参数确保了仅在 Fragment 的生命周期至少处于 ON\_START 状态时才进行回调，并在 ON\_DESTROY 时自动移除监听，从而保证了生命周期的安全性

class PlaceholderFragment(private val sectionNumber: Int) :

Fragment(R.layout.fragment\_placeholder) {

private val onBackPressedCallback = object : OnBackPressedCallback(true) {

override fun handleOnBackPressed() {

//TODO

}

}

override fun onAttach(context: Context) {

super.onAttach(context)

requireActivity().onBackPressedDispatcher.addCallback(this, onBackPressedCallback)

}

}

# Fragment 妙用

## 1、监听生命周期

我们在使用 Fragment 时往往都会为其指定布局文件，从而显示特定视图。但 Fragment 也可以不加载任何布局文件，此时 Fragment 就是完全无 UI 界面的，但只要该 Fragment 也挂载到了 Activity 或者父 Fragment 上，一样可以收到相应的生命周期回调，此时就可以通过**对上层完全无感知的形式**来间接获得载体的生命周期回调通知了

根据 Fragment 的这个特性实现的比较出名的组件和开源库有：Jetpack Lifecycle 和 Glide

### Jetpack Lifecycle

熟悉 Jetpack 的同学应该都知道，我们是通过 Lifecycle 来收到 Activity 生命周期状态变化的通知的，而 Lifecycle 就是通过向 Activity 注入一个无 UI 界面的 Fragment 得知 Activity 生命周期状态变化的

现在我们在日常开发中使用的 AppCompatActivity 最终会继承于 ComponentActivity，ComponentActivity 的 onCreate 方法是这样的：

@SuppressLint("RestrictedApi")

@Override

protected void onCreate(@Nullable Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

ReportFragment.injectIfNeededIn(this);

}

ReportFragment 的 **injectIfNeededIn()** 是一个静态方法，以 **android.app.Activity** 对象作为入参参数，此方法内部就会向 Activity 添加一个无 UI 界面的 ReportFragment

public static void injectIfNeededIn(Activity activity) {

if (Build.VERSION.SDK\_INT >= 29) {

// On API 29+, we can register for the correct Lifecycle callbacks directly

activity.registerActivityLifecycleCallbacks(

new LifecycleCallbacks());

}

//向 activity 添加一个不可见的 framework 中的 fragment，以此来取得 Activity 生命周期事件的回调

android.app.FragmentManager manager = activity.getFragmentManager();

if (manager.findFragmentByTag(REPORT\_FRAGMENT\_TAG) == null) {

manager.beginTransaction().add(new ReportFragment(), REPORT\_FRAGMENT\_TAG).commit();

// Hopefully, we are the first to make a transaction.

manager.executePendingTransactions();

}

}

ReportFragment 就通过自身的各个回调方法来间接获得 Activity 生命周期事件的回调通知

@Override

public void onActivityCreated(Bundle savedInstanceState) {

super.onActivityCreated(savedInstanceState);

···

dispatch(Lifecycle.Event.ON\_CREATE);

}

@Override

public void onStart() {

super.onStart();

···

dispatch(Lifecycle.Event.ON\_START);

}

@Override

public void onDestroy() {

super.onDestroy();

dispatch(Lifecycle.Event.ON\_DESTROY);

···

}

### Glide

我们加载的图片通常是要显示在 ImageView 中的，而 ImageView 是会挂载在 Activity 或者 Fragment 等容器上的，当容器处于后台或者已经被 finish 时，此时加载图片的操作就应该被取消或者停止，否则也是在浪费宝贵的系统资源和网络资源，甚至可能发生内存泄露或者 NPE。那么，显而易见的一个问题就是，Glide 是如何判断容器是否还处于活跃状态的呢？

类似于 Jetpack 组件中的 Lifecycle 的实现思路，Glide 也是通过一个无 UI 界面的 Fragment 来间接获取容器的生命周期状态的

首先，Glide 包含一个 LifecycleListener，定义了三种事件通知回调，用于通知容器的活跃状态：是处于前台、后台、还是已经退出了。具体的实现类是 ActivityFragmentLifecycle

public interface LifecycleListener {

void onStart();

void onStop();

void onDestroy();

}

ActivityFragmentLifecycle 用于 SupportRequestManagerFragment 这个 Fragment 中来使用（省略了部分代码）。可以看到，在 Fragment 的三个生命周期回调事件中，都会相应通知 ActivityFragmentLifecycle。那么，不管 ImageView 的载体是 Activity 还是 Fragment，我们都可以向其注入一个无 UI 界面的 SupportRequestManagerFragment，以此来监听载体在整个生命周期内活跃状态的变化

public class SupportRequestManagerFragment extends Fragment {

private static final String TAG = "SupportRMFragment";

private final ActivityFragmentLifecycle lifecycle;

public SupportRequestManagerFragment() {

this(new ActivityFragmentLifecycle());

}

@VisibleForTesting

@SuppressLint("ValidFragment")

public SupportRequestManagerFragment(@NonNull ActivityFragmentLifecycle lifecycle) {

this.lifecycle = lifecycle;

}

@NonNull

ActivityFragmentLifecycle getGlideLifecycle() {

return lifecycle;

}

@Override

public void onStart() {

super.onStart();

lifecycle.onStart();

}

@Override

public void onStop() {

super.onStop();

lifecycle.onStop();

}

@Override

public void onDestroy() {

super.onDestroy();

lifecycle.onDestroy();

unregisterFragmentWithRoot();

}

@Override

public String toString() {

return super.toString() + "{parent=" + getParentFragmentUsingHint() + "}";

}

}

## 2、方法代理

Fragment 还有一种特殊的用法。我们平时是通过 requestPermissions()、onRequestPermissionsResult() 等方法来申请权限的，在一个地方发起权限申请操作，在另一个特定方法里拿到申请结果，整个流程是异步且割裂开的。而 Fragment 包含了很多个和 Activity 完全一样的 API，其中就包括这两个权限申请方法，因此我们可以通过自动向载体挂载一个无 UI 界面的 Fragment，让 Fragment 来代理整个权限申请流程，使得载体可以用流式申请的方式来完成整个操作

/\*\*

\* @Author: leavesC

\* @Date: 2021/9/5 17:10

\* @Desc:

\* @Github：https://github.com/leavesC

\*/

class RequestPermissionsFragment : Fragment() {

companion object {

private const val EXTRA\_PERMISSIONS = "github.leavesC.extra.PERMISSIONS"

private const val REQUEST\_CODE = 100

fun request(

activity: FragmentActivity,

permissions: Array<String>,

onRequestResult: ((Array<String>, IntArray) -> Unit)

) {

val bundle = Bundle()

bundle.putStringArray(EXTRA\_PERMISSIONS, permissions)

val requestPermissionsFragment = RequestPermissionsFragment()

requestPermissionsFragment.arguments = bundle

requestPermissionsFragment.onRequestResult = onRequestResult

activity.supportFragmentManager.commit {

setReorderingAllowed(true)

addToBackStack(null)

add(requestPermissionsFragment, null)

}

}

}

private val permissions by lazy {

requireArguments().getStringArray(EXTRA\_PERMISSIONS) ?: throw IllegalArgumentException()

}

private var onRequestResult: ((Array<String>, IntArray) -> Unit)? =

null

override fun onAttach(context: Context) {

super.onAttach(context)

requestPermissions(permissions, REQUEST\_CODE)

}

override fun onRequestPermissionsResult(

requestCode: Int,

permissions: Array<String>,

grantResults: IntArray

) {

onRequestResult?.invoke(permissions, grantResults)

parentFragmentManager.popBackStack()

}

override fun onDestroyView() {

super.onDestroyView()

onRequestResult = null

}

}

之后在 FragmentActivity 或者 Fragment 中就可以通过以下方式来完成整个权限申请操作了，直接在回调里拿到申请结果

RequestPermissionsFragment.request(

fragmentActivity,

arrayOf(

Manifest.permission.ACCESS\_FINE\_LOCATION,

Manifest.permission.CAMERA

)

) { permissions: Array<String>,

grantResults: IntArray ->

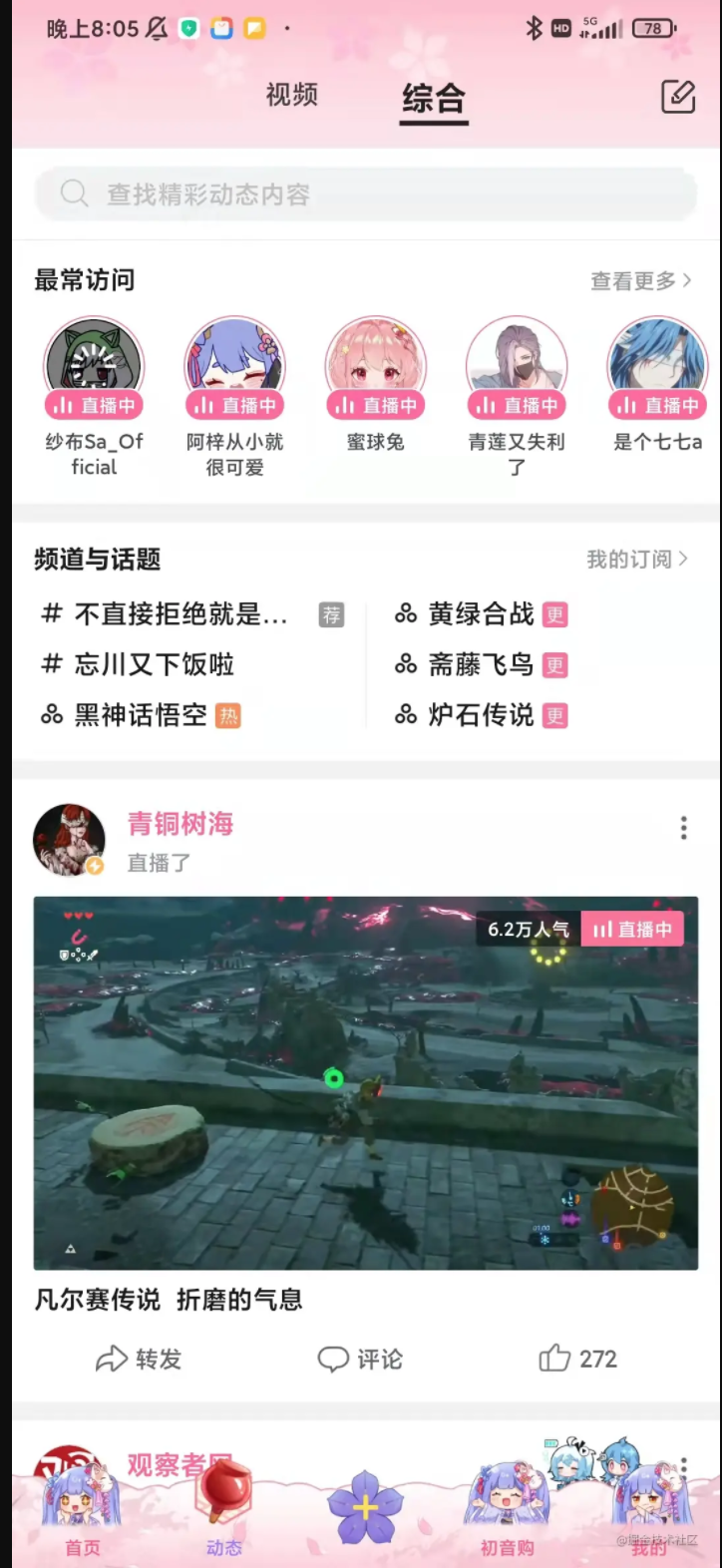
permissions.forEachIndexed { index, s ->

showToast("permission：" + s + " grantResult：" + grantResults[index])

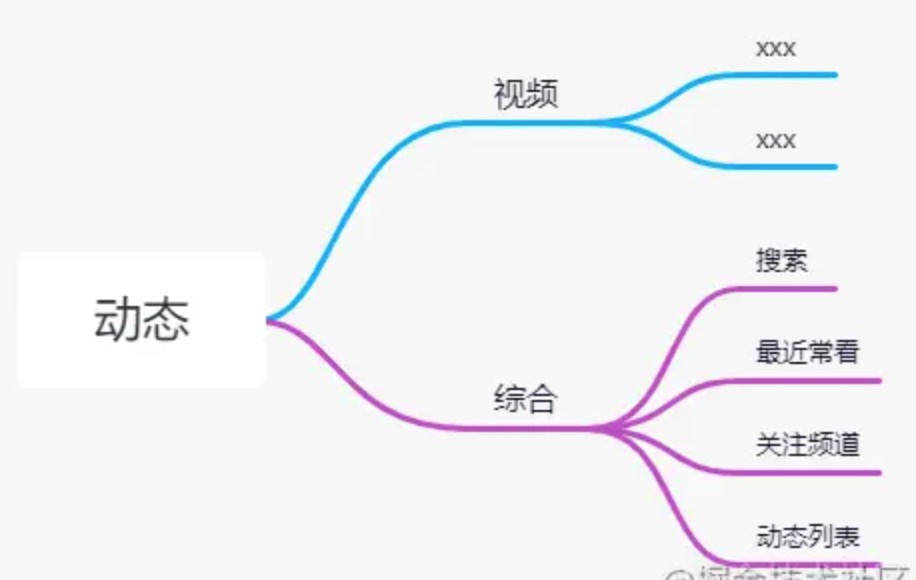
}

}

**如何优雅地使用Fragment**



我们来简单划分一下这个页面的结构：



如果我们想也不想地来直接实现，那么这个页面大概可以分成这样：

1. 顶部的两个Tab，ViewPager + Fragment。
2. 底下的滚动页面，一个RecyclerView（或者NestedScrollView + RecyclerView）全搞定。

1 这部分没什么好说的，大家应该都是这么做的（应该不会有人这里不用ViewPager + Fragment吧？），我们简单用伪代码模拟一下。

/\*\*

\* 动态

\*/

public class MomentsFragment extends Fragment {

/\*\*

\* 顶部视频和综合的Tab

\*/

private ViewPager tabPager;

@Nullable

@Override

public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {

// return inflater.inflate()...

return super.onCreateView(inflater, container, savedInstanceState);

}

@Override

public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

tabPager = view.findViewById(R.id.view\_pager);

FragmentPagerAdapter adapter = new MomentsFragmentAdaper();

// 视频 Tab

adapter.addFragment(new VideoFragment());

// 综合 Tab

adapter.addFragment(new CommonFragment());

tabPager.setAdapter(adapter);

}

}

至于 2 部分……

我们当然可以像上面说的那么做，把所有的内容往这一整页的Fragment里面一塞，剩下的1.用Adapter的Type全弄一起，2.或者写好几个View，addView，3.或者自定义一种容器来稍作解耦，然后容器提供getView方法，最后还是addView。

但设想一下这么做的后果：1.假设放在Adapter里面，我们会得到一个3000行的Adapter，页面的每一部分可能仅仅是某个方法，Adapter里面还装着网络请求和用来解析网络请求的相关代码，这样的“上帝类”我们也许能忍受，但是很可能在一个月后我们就不得不审视这所有的3000行代码，而目的仅仅是因为产品经理希望我们把“最常访问”和“频道与话题”这两个部分换个位置。

public class VideoPageAdapter extends RecyclerView.Adapter {

private static final int TYPE\_VIDEO = 0;

private static final int TYPE\_SEARCH = 1;

private static final int TYPE\_RECENT = 2;

private static final int TYPE\_CHANNEL = 3;

@Override

public int getItemViewType(int position) {

int type = TYPE\_VIDEO;

switch (position) {

case 0:

type = TYPE\_SEARCH;

break;

case 1:

type = TYPE\_RECENT;

break;

case 2:

type = TYPE\_CHANNEL;

break;

}

return type;

}

// 此处省略三千行

}

那么，关于2和3呢？稍有经验的人肯定能想到，我们应该把代表不同功能的部分分别解耦，把他们至少拆到不同的类里面去——我们可以用不同的View来代表不同的功能，但是考虑到MVP，MVVM等思想，直接用View来显然也是在给未来挖坑，那么我们就可以自定义一个容器来作为Controller或者Presenter，从而实现不同的功能……

那么，与其自己造轮子，我们何不用Fragment呢？假设我们不用Fragment，而是自定义了某个Container或者Presenter，那么我们就要在最外层的Fragment里面持有它们所有的引用，然后得在不同的生命周期里面，手动调用方法触发它们的生命周期（当然你也可以用JetPack的LifeCycle来更优雅地实现这一点）。至于这些不同的Container之间，如果我们希望他们数据互通，最坏的结果可能就是他们需要互相持有互相的引用，代码最后还是变得一团糟……

/\*\*

\* 视频Tab

\*/

public class VideoFragment extends Fragment {

private ContainerOrPresenter searchPresenter;

private ContainerOrPresenter recentPresenter;

private ContainerOrPresenter channelPresenter;

private ContainerOrPresenter videoPresenter;

@Override

public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

searchPresenter = new ContainerOrPresenter();

recentPresenter = new ContainerOrPresenter();

// 假设两个 Presenter 需要互相通信。

searchPresenter.setRecentPresenter(recentPresenter);

}

@Override

public void onResume() {

super.onResume();

// 模拟繁琐的生命周期控制

searchPresenter.onResume(this);

recentPresenter.onResume(this);

channelPresenter.onResume(this);

videoPresenter.onResume(this);

}

}

这段代码的问题就在于我们维护了完全没必要维护的生命周期，为此不得不持有了多余的引用。

所以用Fragment吧！有官方来为我们维护它的生命周期，官方还提供了Fragment + ViewModel的最佳实践让我们实现不同部分之间的数据传递，我们能够不费吹灰之力地实现MVVM架构，而未来如果我们希望把一部分功能复用或者删除，简单地操作一下Fragment就可以。这就是Fragment之于解耦，于其自带生命周期的优势所在了。

那么，回到最开始的问题，这个页面在我们加入了Fragment之后，一个更好的实现方式可能是这样……

1. 顶部的两个Tab，ViewPager + Fragment。
2. 底下的滚动页面：NestedScrollView + 4个Fragment。

大概这样：



我们无需在意滚动页面里面的Fragment装的到底是什么，我们只需要简单的在外层Fragment里面根据需要，new Fragment()，然后 add 即可，不同的部分所要实现的不同功能，我们可以全部放在不同的Fragment里面，而不同的Fragment之间我们通过ViewModel来通信即可。

这里我们还是用简单的代码来表示这种实践： 首先，最外层代表“动态”的Fragment我们对其进行了一些改动， 主要是添加了ViewModel，以及子Fragment的创建方式有了一些差异（这里我们过后提） 具体都在注释里面了：

/\*\*

\* B站的“动态”Tab

\*/

public class MomentsFragment extends Fragment {

/\*\*

\* 顶部视频和综合的Tab

\*/

private ViewPager tabPager;

private MomentsViewModel momentsViewModel;

private FragmentActivity hostActivity;

@Override

public void onAttach(@NonNull Context context) {

super.onAttach(context);

hostActivity = (FragmentActivity) context;

}

@Override

public void onCreate(@Nullable Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// 这里 of 方法传入的参数可以是Fragment也可以是Activity

// 如果此 Fragment 需要和Activity或者和Activity下的其他Fragments通讯，建议传入Activity作为参数

// 值得注意的是，传入Activity创建的ViewModel会一直存续到Activity销毁，所以注意内存泄露问题

momentsViewModel = ViewModelProviders.of(hostActivity).get(MomentsViewModel.class);

momentsViewModel.newMessageLiveData.observe(this, new Observer<Integer>() {

@Override

public void onChanged(Integer integer) {

// 这里模拟网络请求回传了“动态”tab右上角的新消息数量

}

});

}

@Nullable

@Override

public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {

// return inflater.inflate(res, container, false);

return super.onCreateView(inflater, container, savedInstanceState);

}

@Override

public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

tabPager = view.findViewById(R.id.view\_pager);

// 这里传入的应该是 getChildFragmentManager，不做过多解释

MomentsFragmentAdapter adapter = new MomentsFragmentAdapter(

getChildFragmentManager(), FragmentPagerAdapter.BEHAVIOR\_RESUME\_ONLY\_CURRENT\_FRAGMENT);

List<String> childList = new ArrayList<>();

// 视频 Tab，这里相比原版有一些改动，后面解释

// 这里也能简单看出来，相比直接传入Fragment的实例，这样做其实做到了解耦

// 我们无需关注Moments里面到底有什么，只要把对应页面的路径传入即可

childList.add(Routes.ROUTE\_VIDEO);

// 综合 Tab

childList.add(Routes.ROUTE\_COMMON);

adapter.setDataList(childList);

tabPager.setAdapter(adapter);

}

@Override

public void onResume() {

super.onResume();

// 这里模拟进行了一个网络请求

momentsViewModel.requestNewMessage();

}

@Override

public void onDestroy() {

super.onDestroy();

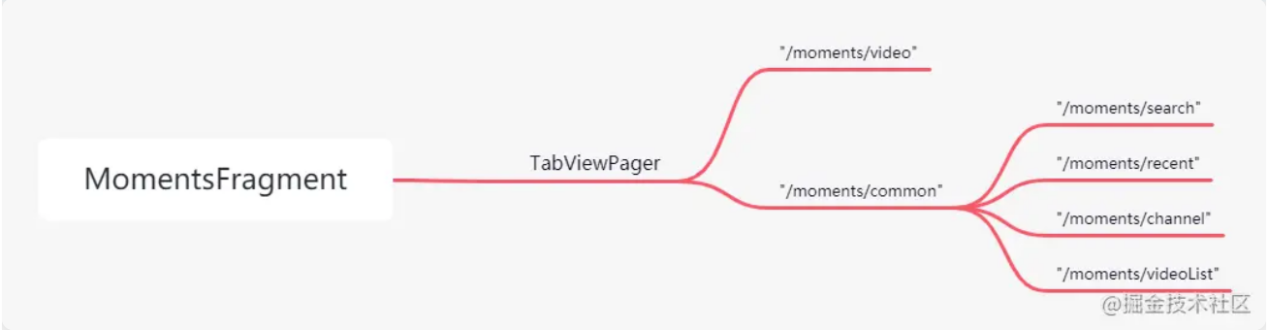
// 避免内存泄露

momentsViewModel.newMessageLiveData.removeObservers(this);

}

}

顺便，这里我们提到了ARouter对于Fragment的解耦效果。下面我们简单展示ARouter+Fragment所能达到的效果。



可以看到，这下我们连具体的Fragment都不需要了，原本的实现方式我们虽然可以无需了解我们需要的Fragment是哪个，但是我们还是需要手动来 new XXXFragment，而在使用ARouter之后，我们就只需要不同Fragment对应的路径了。

我们在通过Fragment实现具体细节之后，动态-综合这页本身作为一个容器，只需要把容器的内容填上即可，无需自己实现任何额外的逻辑，用代码标识就是：

/\*\*

\* 动态-综合

\* 容器Fragment，用来装里面具体的 搜索，最近访问，频道，动态列表 等子模块

\*/

@Route(path = Routes.ROUTE\_COMMON)

public class CommonFragment extends Fragment {

private CommonViewModel commonViewModel;

private FragmentActivity hostActivity;

@Override

public void onAttach(@NonNull Context context) {

super.onAttach(context);

hostActivity = (FragmentActivity) context;

}

@Override

public void onCreate(@Nullable Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// 这里 of 方法传入的参数可以是Fragment也可以是Activity

// 如果此 Fragment 需要和Activity或者和Activity下的其他Fragments通讯，建议传入Activity作为参数

// 值得注意的是，传入Activity创建的ViewModel会一直存续到Activity销毁，所以注意内存泄露问题

commonViewModel = ViewModelProviders.of(this).get(CommonViewModel.class);

}

@Override

public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

// 假设这个LinearLayout包裹在NestedScrollView里面，假设它叫“ll\_container”

// 借用ARouter来创建Fragment后，我们只需要四个路径，此容器Fragment的工作就全部完成啦~

FragmentManager manager = getChildFragmentManager();

FragmentTransaction transaction = manager.beginTransaction();

transaction.add(R.id.ll\_container, createFragmentByPath(Routes.ROUTE\_SEARCH), Routes.ROUTE\_SEARCH);

transaction.add(R.id.ll\_container, createFragmentByPath(Routes.ROUTE\_RECENT), Routes.ROUTE\_RECENT);

transaction.add(R.id.ll\_container, createFragmentByPath(Routes.ROUTE\_CHANNEL), Routes.ROUTE\_CHANNEL);

transaction.add(R.id.ll\_container, createFragmentByPath(Routes.ROUTE\_VIDEO\_LIST), Routes.ROUTE\_VIDEO\_LIST);

transaction.commit();

}

private Fragment createFragmentByPath(String path) {

return (Fragment) ARouter.getInstance().build(path).navigation();

}

}

这里创建ViewModel是为了实现父Fragment和其所有子Fragment之间的通信，具体的后面再说。

然后这里我们实现一下频道Fragment和视频列表Fragment，上代码：

/\*\*

\* 动态-综合页-频道

\*/

@Route(path = Routes.ROUTE\_CHANNEL)

public class ChannelFragment extends Fragment {

private CommonViewModel commonViewModel;

private RecyclerView channelRecyclerView;

private RecyclerView.Adapter channelAdapter;

private FragmentActivity hostActivity;

@Override

public void onAttach(@NonNull Context context) {

super.onAttach(context);

hostActivity = (FragmentActivity) context;

}

@Override

public void onCreate(@Nullable Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

commonViewModel = ViewModelProviders.of(this).get(CommonViewModel.class);

commonViewModel.channelData.observe(this, new Observer<List<ChannelData>>() {

@Override

public void onChanged(List<ChannelData> channelData) {

if (null != channelData) {

// channelAdapter.setDataList(channelData);

channelAdapter.notifyDataSetChanged();

}

}

});

}

@Override

public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

// channelRecyclerView = view.findViewById(R.id.rv);

// channelAdapter = new ChannelAdapter();

channelRecyclerView.setLayoutManager(new GridLayoutManager(hostActivity, 2));

channelRecyclerView.setAdapter(channelAdapter);

}

@Override

public void onResume() {

super.onResume();

commonViewModel.requestChannel();

}

@Override

public void onDestroy() {

super.onDestroy();

commonViewModel.channelData.removeObservers(this);

}

}

/\*\*

\* 动态-综合页-视频列表

\*/

@Route(path = Routes.ROUTE\_VIDEO\_LIST)

public class VideoListFragment extends Fragment {

private CommonViewModel commonViewModel;

private RecyclerView videoRecyclerView;

private RecyclerView.Adapter videoAdapter;

private FragmentActivity hostActivity;

@Override

public void onAttach(@NonNull Context context) {

super.onAttach(context);

hostActivity = (FragmentActivity) context;

}

@Override

public void onCreate(@Nullable Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

commonViewModel = ViewModelProviders.of(this).get(CommonViewModel.class);

commonViewModel.channelData.observe(this, new Observer<List<ChannelData>>() {

@Override

public void onChanged(List<ChannelData> channelData) {

if (null != channelData) {

// channelAdapter.setDataList(channelData);

videoAdapter.notifyDataSetChanged();

}

}

});

}

@Override

public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

// videoRecyclerView = view.findViewById(R.id.rv);

// videoAdapter = new VideoAdapter();

videoRecyclerView.setLayoutManager(new LinearLayoutManager(hostActivity));

videoRecyclerView.setAdapter(videoAdapter);

}

@Override

public void onResume() {

super.onResume();

commonViewModel.requestVideo();

}

@Override

public void onDestroy() {

super.onDestroy();

commonViewModel.videoData.removeObservers(this);

}

}

可以看到，我们从原本的一个大类负责所有的功能，逐渐变成了大类只做容器，功能细化拆分给不同的child fragment，从而提高灵活性和复用性。

这样做还有一个好处，那就是不同的页面现在可以交给不同的人来开发了，避免了冲突，提高了效率。

我们这里简单说说用Fragment + ARouter好处都有啥？

我个人认为，好处如下：

1. 解耦
2. 传参优化&依赖注入
3. 通过路径统一实现页面的多端统一

第一点不用说，我们原本想要创建一个Fragment，需要知道类名，现在知道路由地址path就可以了。

第二点是ARouter提供的功能，具体可以在ARouter的项目里查看，我们都知道如果给Fragment的构造函数添加参数来传参是个天坑，是个Crash的经典来源，使用ARouter可以帮我们简化许多工作。

第三点则是，当页面和路径对应起来以后，我们通过一个路径可以在安卓，ios，甚至是h5端都对应到同一个页面。理论上服务器只要下发一套这样的路径配置，我们就能把同一个页面通过不同的样式组合起来，或是轻松实现把首页的不同tab调换顺序。又比如服务器下发路径就可以在路径对应的页面上配置弹窗或者广告，悬浮窗等，十分灵活。最后还有，一个页面我们可以分别对应一个原生页和H5页，如果原生页路径找不到，这个路径（我们可以把它拼装成url)就可以直接降级打开一个web页面。

前面有写到过Fragment + ViewModel的优势和实践，最大好处有二：

1. 实现Fragment与Activity，Fragment与Fragment之间的通信。
2. 轻松实现MVVM模式。

同一 Activity 中的两个或更多 Fragment 通常需要相互通信。例如，假设有一个 Fragment 显示一个列表，另一个 Fragment 允许用户对该列表应用各种过滤器。如果 Fragment 不直接通信（这意味着，它们不再独立），这种情况可能不容易实现。此外，这两个 Fragment 还必须处理另一个 Fragment 尚未创建或不可见的情况。

这很好实现，我们只需要 ViewModelProviders.of(hostActivity).get(CommonViewModel.class);即可，只要of传入的实例是同一个，那么这个方法我们就能得到相同的ViewModel。

这两个 Fragment 可以使用其 Activity 范围共享 ViewModel 来处理这种通信。通过以这种方式共享 ViewModel，Fragment 不需要相互了解，Activity 也不需要执行任何操作来促进通信。

同样的，我们只需要 ViewModelProviders.of(getParentFragment()).get(CommonViewModel.class);即可，只要of传入的实例是同一个，那么这个方法我们就能得到相同的ViewModel。