知识点 | ViewModel 四种集成方式

谷歌开发者
已认证的官方帐号

已关注

高爷等 21 人赞同了该文章

ViewModel 甫一发布，便成为了 Jetpack 中的核心组件之一。我们在 2019 年做的一份开发者问卷显示，超过 40% 的 Android 开发者已经在自己的应用中使用了 ViewModel。ViewModel 可以将数据层与 UI 分离，而这种架构不仅可以简化 UI 的生命周期的控制，也能让代码获得更好的可测试性。如果想了解更多，可以参考ViewModel: 简单介绍视频和官方文档。

由于 ViewModel 是许多功能实现的基础，我们在过去的几年里做了许多工作来改进 ViewModel 的易用性，也让它能够更加简便地与其他组件库相结合。下面的文章中，我将介绍 ViewModel 的四种集成方式:

- **ViewModel 中的 Saved State** —— 后台进程重启时，ViewModel 的数据恢复；
- **在 NavGraph 中使用 ViewModel** —— ViewModel 与导航 (Navigation) 组件库的集成；
- **ViewModel 配合数据绑定** (data-binding) —— 通过使用 ViewModel 和 LiveData 简化数据绑定；
- **viewModelScope** —— Kotlin 协程与 ViewModel 的集成。

ViewModel 的 Saved State —— 后台进程重启时，ViewModel 的数据恢复

- 于 lifecycle-viewmodel-savedstate 的 1.0.0-alpha01 版本时加入
- 支持 Java 和 Kotlin

onSaveInstanceState 带来的挑战

ViewModel 一发布，执行 onSaveInstanceState 的相关的逻辑时要如何操作 ViewModel，便成为了一个令人困惑的问题。Activity 和 Fragment 通常会在下面三种情况下被销毁:

1. **从当前界面永久离开**: 用户导航至其他界面或直接关闭 Activity (通过点击返回按钮或执行的操作调用了 finish() 方法)。对应 Activity 实例被永久关闭；
2. **Activity 配置 (configuration) 被改变**: 例如，旋转屏幕等操作，会使 Activity 需要立即重建；
3. **应用在后台时，其进程被系统杀死**: 这种情况发生在设备剩余运行内存不足，系统又亟须释放一些内存的时候。当进程在后台被杀死后，用户又返回该应用时，Activity 也需要被重建。

在后两种情况中，我们通常都希望重建 Activity。ViewModel 会帮您处理第二种情况，因为在这种情况下 ViewModel 没有被销毁；而在第三种情况下， ViewModel 被销毁了。所以一旦出现了第三种情况，便需要在 Activity 的 onSaveInstanceState 相关回调中保存和恢复 ViewModel 中的数据。我在ViewModels: 持久化、onSaveInstanceState()、恢复 UI 状态与加载器一文中更加详细地描述了这两种情况的区别。

Saved State 模块

现在，ViewModel Saved State 模块将会帮您在应用进程被杀死时恢复 ViewModel 的数据。在免除了与 Activity 繁琐的数据交换后，ViewModel 也真正意义上的做到了管理和持有所有自己的数据。

ViewModel 的这一新功能是通过 SavedStateHandle 实现的。SavedStateHandle 和 Bundle 一样，以键值对形式存储数据，它包含在 ViewModel 中，并且可以在应用处于后台时进程被杀死的情况下幸存下来。诸如用户 id 等需要在 onSaveInstanceState 时得到保存下来的数据，现在都可以存在 SavedStateHandle 中。

设置 Save State 模块

现在让我们看看如何使用 SaveState 组件。注意接下来的代码会和Lifecycles Codelab 第六步中的一段代码十分相似。那段是 Java 代码，而接下来的是 Kotlin 代码:

第一步: 添加依赖

SaveStateHandle 目前在一个独立的模块中，您需要在依赖中添加:

```
def lifecycle_version = "2.2.0"
implementation "androidx.lifecycle:lifecycle-viewmodel-savedstate:$
```

注意，本文发布时 lifecycle 组件的最新稳定版为 2.2.0，如果您希望持续关注相关组件库的进展，可以查看lifecycle 版本发布文档。

第二步: 修改调用 ViewModelProvider 的方式

接下来，您需要创建一个持有 SaveStateHandle 的 ViewModel。在 Activity 或 Fragment 的 onCreate 方法中，将 ViewModelProvider 的调用修改为:

```
//下面的 Kotlin 扩展需要依赖以下或更新新版本的 ktx 库:
//androidx.fragment:fragment-ktx:1.0.0（最新版本 1.2.4） 或
//androidx.activity:activity-ktx:1.0.0 （最新版本 1.1.0）
val viewModel by viewModels { SavedStateViewModelFactory(application, this) }
// 或者不使用 ktx
val viewModel = ViewModelProvider(this,
        SavedStateViewModelFactory(application, this))
            .get(MyViewModel::class.java)
```

创建 ViewModel 的类是 ViewModel 工厂 (ViewModel factory)，而创建包含 SaveStateHandle 的 View Model 的工厂类是 SavedStateViewModelFactory。通过此工厂创建的 ViewModel 将持有一个基于传入 Activity 或 Fragment 的 SaveStateHandle。

第三步: 使用 SaveStateHandle

当前面的步骤准备完成时，您就可以在 ViewModel 中使用 SavedStateHandle 了。下面是一个保存用户 ID 的示例：

```
class MyViewModel(state :SavedStateHandle) :ViewModel() {

    // 将Key声明为常量
    companion object {
        private val USER_KEY = "userId"
    }

    private val savedStateHandle = state

    fun saveCurrentUser(userId: String) {
        // 存储 userId 对应的数据
        savedStateHandle.set(USER_KEY, userId)
    }

    fun getCurrentUser(): String {
        // 从 saveStateHandle 中取出当前 userId
        return savedStateHandle.get(USER_KEY)?: ""
    }
}
```

1. **构造方法**: SavedStateHandle 作为构造方法参数传入 MyViewModel；
2. **保存**: saveNewUser 方法展示了使用键值对的形式保存 USER_KEY 和 userId 到 SaveStateHandle 的例子。每当数据更新时，要保存新的数据到 SavedStateHandle；
3. **获取**: 如代码中所示，调用 savedStateHandle.get(USER_KEY) 方法获取被保存的 userId。

现在，无论是第二还是第三种情况下，SavedStateHandle 都可以帮您恢复界面数据了。

如果您想要在 ViewModel 中使用 LiveData，可以调用SavedStateHandle.getLiveData()，示例如下:

```
// getLiveData 方法会取得一个与 key 相关联的 MutableLiveData
// 当与 key 相对应的 value 改变时 MutableLiveData 也会更新。
private val _userId : MutableLiveData<String> =
    savedStateHandle.getLiveData(USER_KEY)

// 只暴露一个不可变 LiveData 的情况
val userId : LiveData<String> = _userId
```

如需了解更多，请移步至Lifecycles Codelab 第六步和官方文档。

ViewModel 与 Jetpack 导航: 在 NavGraph 中使用 ViewModel

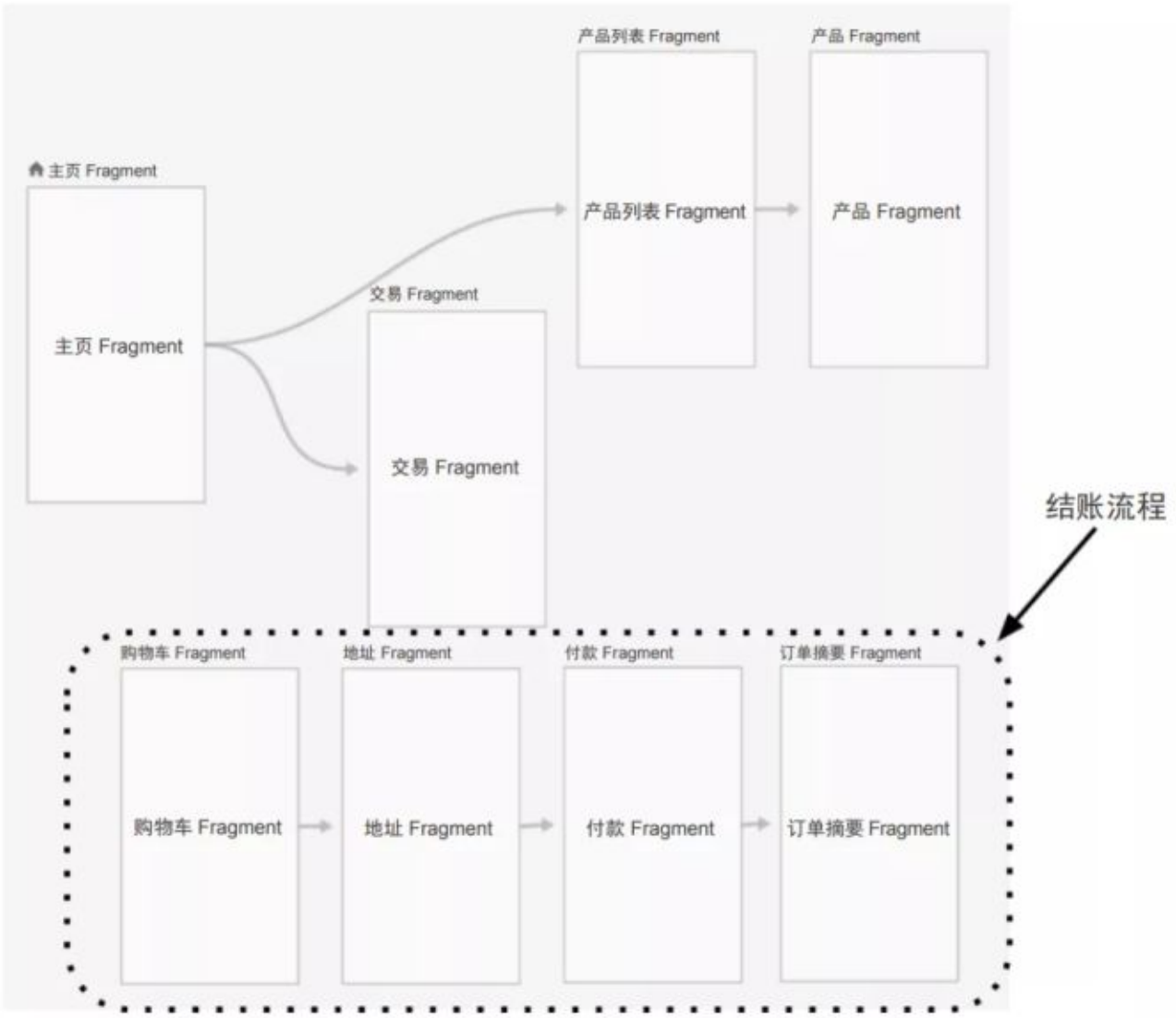
- 于 navigation 的 2.1.0-rc01 版本时加入
- 支持 Java 与 Kotlin

共享 ViewModel 数据所带来的挑战

Jetpack 导航组件 (Navigation) 十分适用于那些只有少量或一个 Activity，但是 Activity 中会包含多个 Fragment 的应用。Ian Lake 在他的演讲: 单 Activity 架构: 为什么、什么情况下以及如何使用 中介绍了一些我们选择单一 Activity 架构的原因，而与本文相关的一点，是这种架构允许在多个界面 (destination) 间共享 ActivityViewModel。您可以用 Activity 创建一个 ViewModel 实例，然后从这个 Activity 中的任一个 Fragment 中获得 ViewModel 的引用:

```
// 在Fragment的 onCreate 或 onActivityCreated 方法中执行
// 这个Kotlin扩展需要依赖最KTX库：androidx.fragment:fragment-ktx:1.1.0
val sharedViewModel: ActivityViewModel by activityViewModels()
```

假设我们有这样一个单 Activity 应用，它包含了八个 Fragment，其中四个 Fragment 是购买支付流程:



△ 包含一些购买支付流程的导航图 (Navigation Graph)

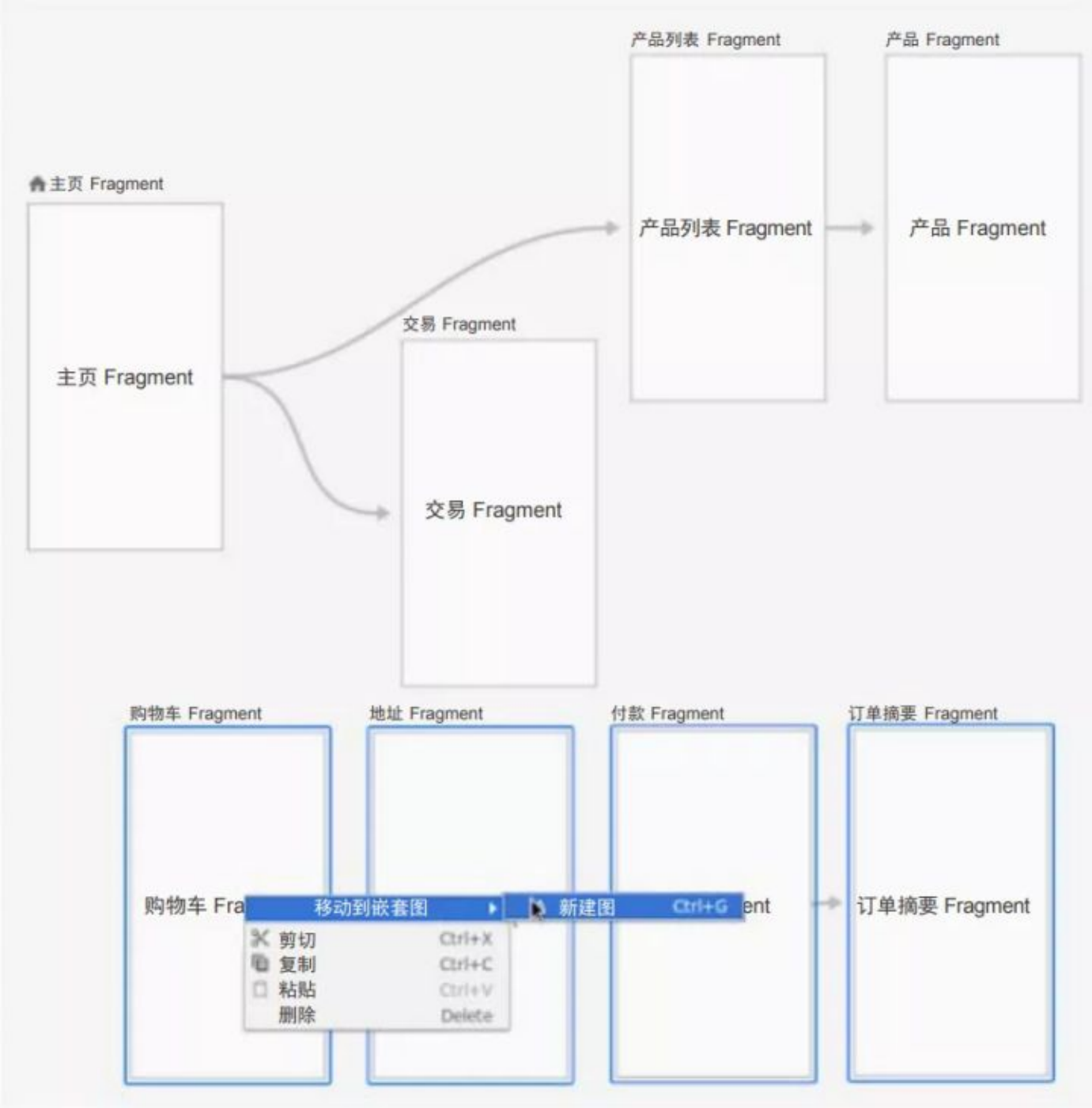
这四个页面需要共享一些诸如收货地址、是否使用了优惠券等信息。按照前面所讲的做法，需要共享的数据会放在一个 ActivityViewModel 中，但这同时也意味着所有八个页面都会共享这些数据。

支付流程外的界面并不需要关心这些数据，这么做显然并不合适。

ViewModel 与 NavGraph 集成

Navigation 2.1.0 中引入了依托一个导航图 (navigation graph) 创建 ViewModel 的功能。在使用时，您需要先把一个界面集合 (例如: 登录流程、支付流程的相关界面)，放到一个嵌套导航图 (nested navigation graph)中。此时再通过嵌套导航图创建出 ViewModel，便可以在相关界面中共享数据了。

想要创建嵌套导航图，您需要选中对应流程相关的界面，点击鼠标右键，并选择**Nested Graph → New Graph**:



△ 创建嵌套导航图的截图

注意嵌套导航图在 XML 文件中的 id，在这里是 checkout_graph:

```
<navigation app:startDestination="@id/homeFragment" ...>
    <fragment android:id="@+id/homeFragment" .../>
    <fragment android:id="@+id/productListFragment" .../>
    <fragment android:id="@+id/productFragment" .../>
    <fragment android:id="@+id/bargainFragment" .../>

    <navigation
      android:id="@+id/checkout_graph"
      app:startDestination="@id/cartFragment">

        <fragment android:id="@+id/orderSummaryFragment".../>
        <fragment android:id="@+id/addressFragment" .../>
        <fragment android:id="@+id/paymentFragment" .../>
        <fragment android:id="@+id/cartFragment" .../>

    </navigation>

</navigation>
```

以上工作完成时，便可以使用 by navGraphViewModels 获取到对应的 ViewModel:

```
val viewModel: CheckoutViewModel by navGraphViewModels(R.id.checko
```

Java 中同样适用，代码如下:

```
public void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // 设置其他 fragment
    NavController navController = NavHostFragment.findNavController(this);

    ViewModelProvider viewModelProvider = new ViewModelProvider(this,
        navController.getViewModelStore(R.id.checkout_graph));

    CheckoutViewModel viewModel =
        viewModelProvider.get(CheckoutViewModel.class);

    // 使用 Checkout ViewModel
}
```

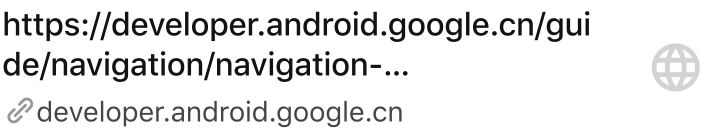
需要注意的是，嵌套导航图相对于导航图的其他部分是一个独立的整体。您无法导航至嵌套导航图中包含的某个特定界面；当您导航至一个嵌套导航图时，打开的只会是其中的开始界面 (startDestination)。这种特性使得嵌套导航图适合用于封装特定流程的界面组合，比如前面提到过的登录和支付流程。

ViewModel 与 NavGraph 的集成，是 2019 年 I/O 大会所发布的关于 Navigation 框架的新特性之一。

详细了解更多，请参阅:

- 主题演讲: Jetpack Navigation 的主题演讲

- 官方文档: 以编程方式与导航组件交互



ViewModel 与 Data Binding: 在 Data Binding 中使用 ViewModel 和 LiveData

- 于 Android Studio 的 3.1 版本时加入
- 支持 Java 与 Kotlin

移除 LiveData 相关的模板代码

ViewModel、LiveData 与 Data Binding 的集成方式并不是什么新功能，但它始终非常好用。ViewModel 通常都包含一些 LiveData，而 LiveData 意味着可以被监听。所以最常见的使用场景是在 Fragment 中给 LiveData 添加一个观察者:

```
override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)

    myViewModel.name.observe(this, { newName ->
        // 更新UI，这里是一个TextView
        nameTextView.text = newName
    })

}
```

Data Binding是一个通过观察数据变化来更新 UI 的组件库。通过 ViewModel、LiveData 和 Data Binding 的组合，您可以移除以往给 LiveData 添加观察者的做法，改为直接在 XML 中绑定 View Model 和 LiveData。

使用 Data Binding、ViewModel 和 LiveData

假设您希望在 XML 布局文件中引用 ViewModel:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
<data>
<variable name="viewmodel"
type="com.android.MyViewModel"/>
</data>
<... Rest of your layout ...>
</layout>
```

调用 binding.setLifecycleOwner(this) 方法，然后将 ViewModel 传递给 binding 对象，就可以将 LiveData 与 Data Binding 结合起来:

```
class MainActivity : AppCompatActivity() {

    // 这个ktx扩展需要依赖 androidx.activity:activity-ktx:1.0.0
    // 或更新版本
    private val myViewModel: MyViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

         //填充视图并创建 Data Binding 对象
        val binding: MainActivityBinding =
            DataBindingUtil.setContentView(this, R.layout.main_activity)

        //声明这个 Activity 为 Data Binding 的 lifecycleOwner
        binding.lifecycleOwner = this

        // 将 ViewModel 传递给 binding
        binding.viewmodel = myViewModel
    }
}
```

现在，您可以像下面这样使用 ViewModel:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable name="viewmodel"
                  type="com.android.MyViewModel"/>
    </data>
    <TextView
            android:id="@+id/name"
            android:text="@{viewmodel.name}"
            android:layout_height="wrap_content"
```

```
            android:layout_width="wrap_content"/>
</layout>
```

注意，这里的 viewmodel.name 既可以是 String 类型，也可以是 LiveData。如果它是 LiveData，那么 UI 将根据 LiveData 值的改变自动刷新。

ViewMode 与 Kotlin 协程: viewModelScope

- 于 Lifecycle 的 2.1.0 版本时加入
- 只支持 Kotlin

Android 平台上的协程

通常情况下，我们使用回调 (Callback) 处理异步调用，这种方式在逻辑比较复杂时，会导致回调层层嵌套，代码也变得难以理解。Kotlin 协程 (Coroutines)同样适用于处理异步调用，它让逻辑变得简单的同时，也确保了操作不会阻塞主线程。如果您不了解协程，这里有一系列很棒的博客《在 Android 开发中使用协程》以及 codelab: 在 Android 应用中使用 Kotlin 协程以供参考。

一段简单的协程代码:

```
// 下面是示例代码，真实情景下不要使用 GlobalScope
GlobalScope.launch {
    longRunningFunction()
    anotherLongRunningFunction()
}
```

这段示例代码只启动了一个协程，但我们在真实的使用环境下很容易创建出许多协程，这就难免会导致有些协程的状态无法被跟踪。如果这些协程中刚好有您想要停止的任务时，就会导致**任务泄漏** (work leak)。

为了防止任务泄漏，您需要将协程加入到一个CoroutineScope中。CoroutineScope 可以持续跟踪协程的执行，它可以被取消。当 CoroutineScope 被取消时，它所跟踪的所有协程都会被取消。上面的代码中，我使用了GlobalScope，正如我们不推荐随意使用全局变量一样，这种方式通常**不推荐**使用。所以，如果想要使用协程，您要么限定一个作用域 (scope)，要么获得一个作用域的访问权限。而在 ViewModel 中，我们可以使用 viewModelScope 来管理协程的作用域。

viewModelScope

当 ViewModel 被销毁时，通常都会有一些与其相关的操作也应当被停止。

例如，假设您正在准备将一个位图 (bitmap) 显示到屏幕上。这种操作就符合我们前面提到的一些特征: 既不能在执行时阻塞主线程，又要求在用户退出相关界面时停止执行。使用协程进行此类操作时，就应当使用viewModelScope。

viewModelScope 是一个 ViewModel 的 Kotlin 扩展属性。正如前面所说，它能在 ViewModel 销毁时 (onCleared()方法调用时) 退出。这样一来，只要您使用了 ViewModel，您就可以使用 viewModelScope 在 ViewModel 中启动各种协程，而不用担心任务泄漏。

示例如下:

```
class MyViewModel() : ViewModel() {

    fun initialize() {
        viewModelScope.launch {
            processBitmap()
        }
    }

    suspend fun processBitmap() = withContext(Dispatchers.Default) {
        // 在这里做耗时操作
    }

}
```

详细了解更多，请参阅:

- 文章: 更简便地在 Android 中使用协程:

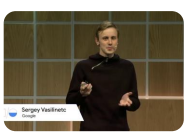
viewModelScopehttps://medium.com/androiddevelopers/easy-coroutines-in-...
medium.com

- 官方文档: 将 Kotlin 协程与架构组件一起使用

https://developer.android.google.cn/topic/libraries/architecture/coroutines
developer.android.google.cn

- 视频演讲: 理解 Android 中的 Kotlin 协程

Understand Kotlin Coroutines on Android (Google I/
v.youku.com



总结

本文中，我们讲了:

1. ViewModel 使用 SaveStateHandle 组件处理 onSaveInstanceState 相关逻辑；
2. 通过配合 View Model 和导航图来精确限定数据在 Fragment 中的共享范围；
3. 使用 DataBinding 库时，可以将 ViewModel 传递给数据绑定 (binding)，如果同时有在 ViewModel 中使用 LiveData，则可以通过 binding.setLifecycleOwner(lifecycleOwner) 让 UI 根据 LiveData 自动更新；
4. 在 ViewModel 中使用 Kotlin 协程时，使用 viewModelScope 来让协程在 ViewModel 被销毁时自动取消。

以上这些功能很多都来自社区提交的请求和反馈，如果您正在寻找 ViewModel 相关的功能，可以留意功能需求列表或者考虑提交自己的需求。

如果您想了解架构组件和 Android Jetpack 的最新进展，请关注 Android 开发者博客，并留意 AndroidX 发布文档。

如果您对这些功能仍有疑问，可以在下方留言。感谢阅读！

发布于 05-25

Android　Android 开发　VM