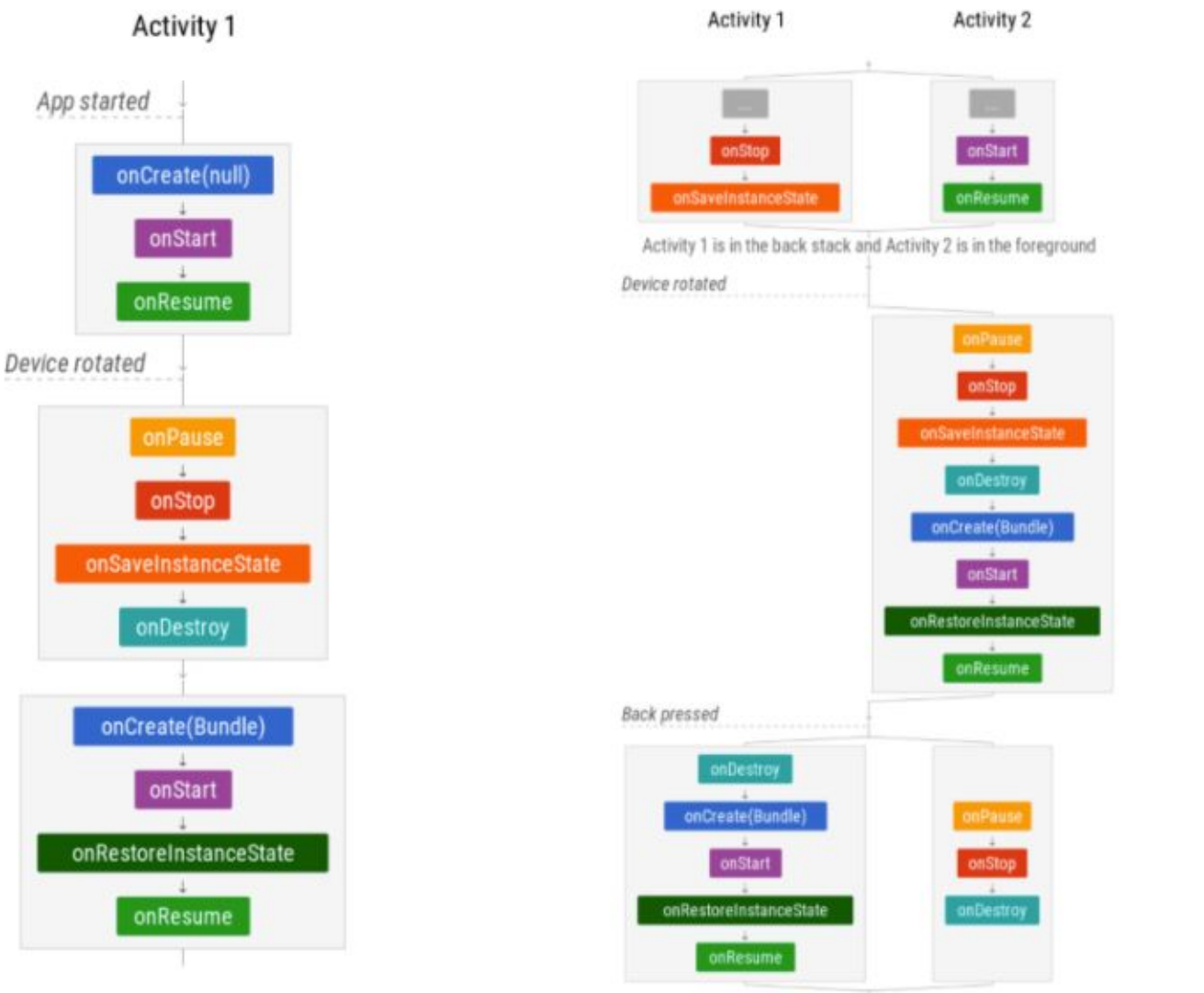
理解协程、LiveData 和 Flow

谷歌开发者
已认证的官方帐号

已关注

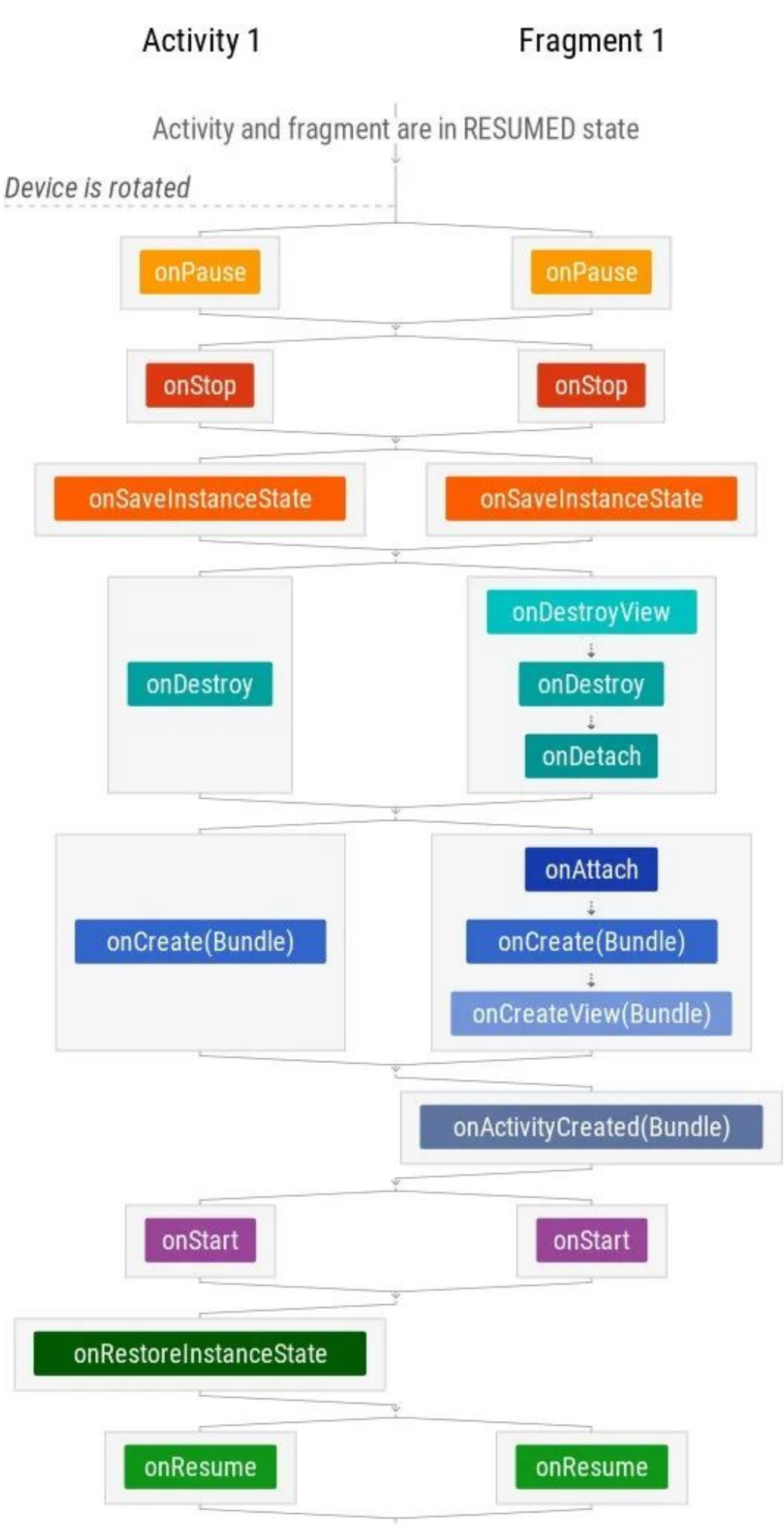
35 人赞同了该文章

从 API 1 开始，处理 Activity 的生命周期 (lifecycle) 就是个老大难的问题，基本上开发者们都看过这两张生命周期流程图:



△ Activity 生命周期流程图

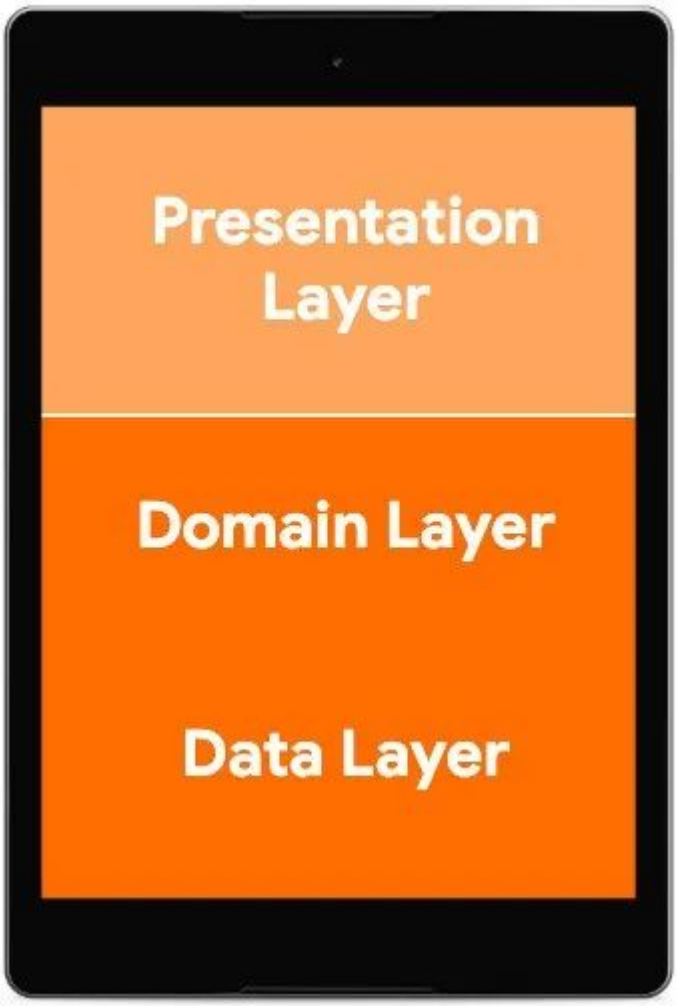
随着 Fragment 的加入，这个问题也变得更加复杂:



△ Fragment 生命周期流程图

而开发者们面对这个挑战，给出了非常稳健的解决方案: 分层架构。

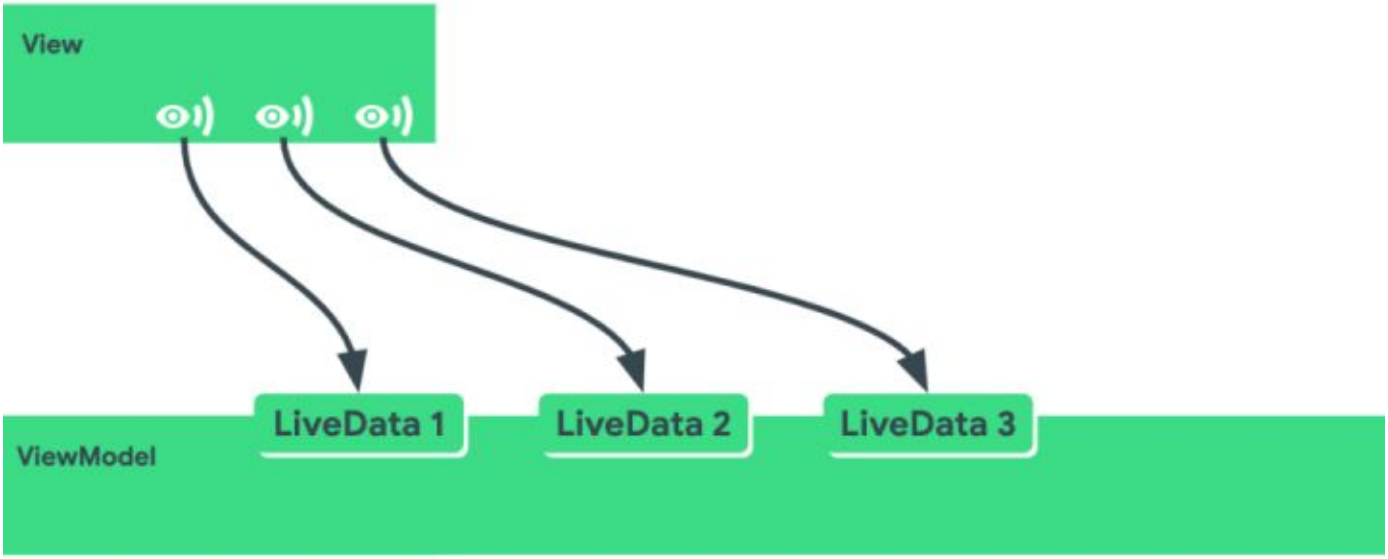
分层架构



△ 表现层 (Presentation Layer)、域层 (Domain Layer) 和数据层 (Data Layer)

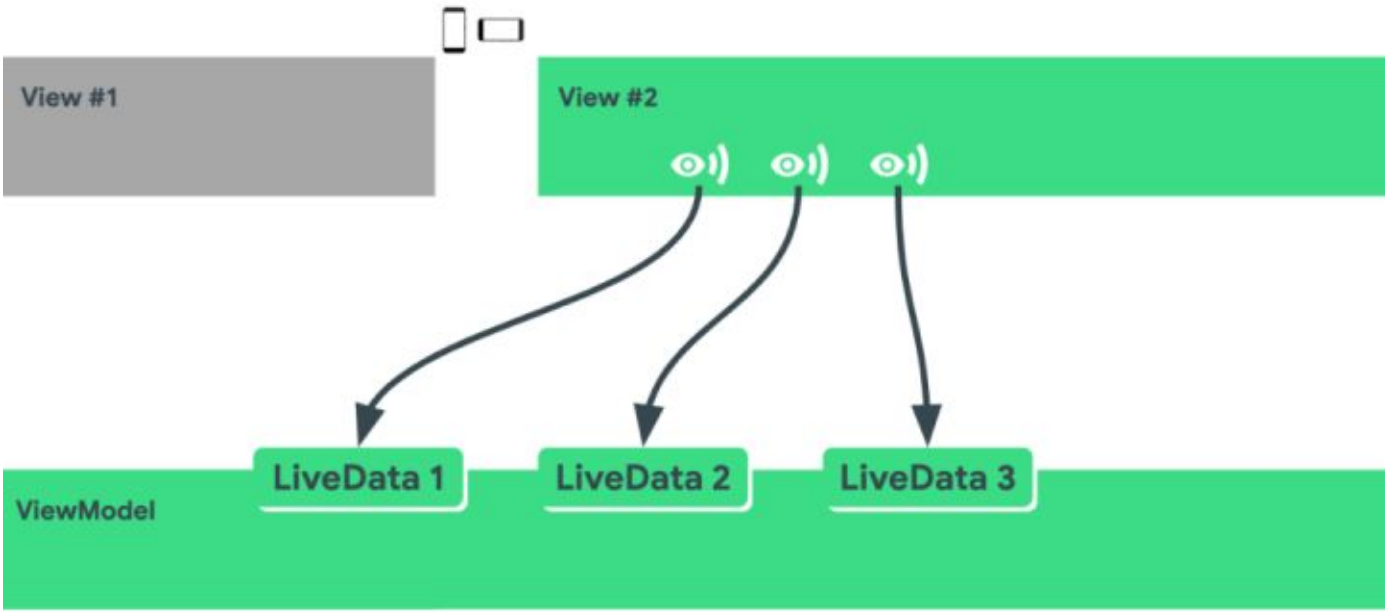
如上图所示，通过将应用分为三层，现在只有最上面的 Presentation 层 (以前叫 UI 层) 才知道生命周期的细节，而应用的其他部分则可以安全地忽略掉它。

而在 Presentation 层内部也有进一步的解决方案: 让一个对象可以在 Activity 和 Fragment 被销毁、重新创建时依然留存，这个对象就是架构组件的 ViewModel 类。下面让我们详细看看 ViewModel 工作的细节。



如上图，当一个视图 (View) 被创建，它有对应的 ViewModel 的引用地址 (注意 ViewModel 并没有 View 的引用地址)。ViewModel 会暴露出若干个 LiveData，视图会通过数据绑定或者手动订阅的方式来观察这些 LiveData。

当设备配置改变时 (比如屏幕发生旋转)，之前的 View 被销毁，新的 View 被创建:



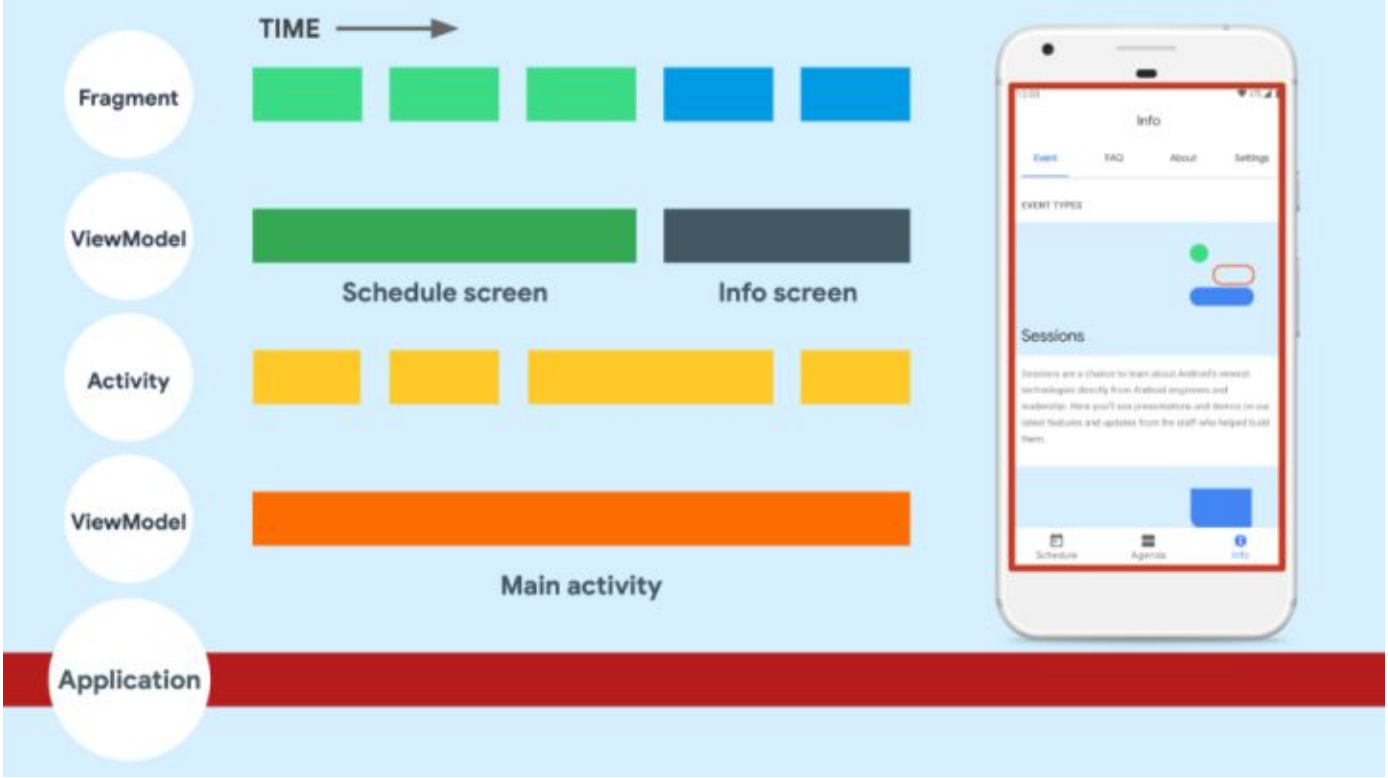
这时新的 View 会重新订阅 ViewModel 里的 LiveData，而 ViewModel 对这个变化的过程完全不知情。

归根到底，开发者在执行一个操作时，需要认真选择好这个操作的作用域 (scope)。这取决于这个操作具体是做什么，以及它的内容是否需要贯穿整个屏幕内容的生命周期。比如通过网络获取一些数据，或者是在绘图界面中计算一段曲线的控制锚点，可能所适用的作用域不同。如何取消该操作的时间太晚，可能会浪费很多额外的资源；而如果取消的太早，又会出现频繁重启操作的情况。

在实际应用中，以我们的 Android Dev Summit 应用为例，里面涉及到的作用域非常多。比如，我们这里有一个活动计划页面，里面包含多个 Fragment 实例，而与之对应的 ViewModel 的作用域就是计划页面。与之相类似的，日程和信息页面相关的 Fragment 以及 ViewModel 也是一样的作用域。

此外我们还有很多 Activity，而和它们相关的 ViewModel 的作用域就是这些 Activity。

您也可以自定义作用域。比如针对导航组件，您可以将作用域限制在登录流程或者结账流程中。我们甚至还有针对整个 Application 的作用域。



有如此多的操作会同时进行，我们需要有一个更好的方法来管理它们的取消操作。也就是 Kotlin 的协程 (Coroutine)。

协程的优势

协程的优点主要来自三个方面:

1. **很容易离开主线程**。我们试过很多方法来让操作远离主线程，AsyncTask、Loaders、ExecutorServices……甚至有开发者用到了 RxJava。但协程可以让开发者只需要一行代码就完成这个工作，而且没有累人的回调处理。
2. **样板代码最少**。协程完全活用了 Kotlin 语言的能力，包括 suspend 方法。编写协程的过程就和编写普通的代码块差不多，编译器则会帮助开发者完成异步化处理。
3. **结构并发性**。这个可以理解为针对操作的垃圾搜集器，当一个操作不再需要被执行时，协程会自动取消它。

如何启动和取消协程

在 Jetpack 组件里，我们为各个组件提供了对应的 scope，比如 ViewModel 就有与之对应的 viewModelScope，如果您想在这个作用域里启动协程，使用如下代码即可:

```
class MainActivityViewModel : ViewModel {

    init {
        viewModelScope.launch {
            // Start

        }
```

```
    }
}
```

如果您在使用 AppCompatActivity 或 Fragment，则可以使用 lifecycleScope，当 lifeCycle 被销毁时，操作也会被取消。代码如下:

```
class MyActivity : AppCompatActivity() {
    override fun onCreate(state: Bundle?) {
        super.onCreate(savedInstanceState)

        lifecycleScope.launch {
            // Run
        }

    }
}
```

有些时候，您可能还需要在生命周期的某个状态 (启动时/恢复时等) 执行一些操作，这时您可以使用 launchWhenStarted、launchWhenResumed、launchWhenCreated 这些方法:

```
class MyActivity : Activity {
    override fun onCreate(state: Bundle?) {
        super.onCreate(savedInstanceState)

        lifecycleScope.launch {
            // Run
        }

        lifecycleScope.launchWhenResumed {
            // Run
        }
    }
}
```

注意，如果您在 launchWhenStarted 中设置了一个操作，当 Activity 被停止时，这个操作也会被暂停，直到 Activity 被恢复 (Resume)。

最后一种作用域的情况是贯穿整个应用。如果这个操作非常重要，您需要确保它一定被执行，这时请考虑使用 WorkManager。比如您编写了一个发推的应用，希望撰写的推文被发送到服务器上，那这个操作就需要使用 WorkManager 来确保执行。而如果您的操作只是清理一下本地存储，那可以考虑使用 Application Scope，因为这个操作的重要性不是很高，完全可以等到下次应用启动时再做。

WorkManager 不是本文介绍的重点，感兴趣的朋友请参考《WorkManager 进阶课堂 | AndroidDevSummit 中文字幕视频》。

接下来我们看看如何在 viewModelScope 里使用 LiveData。以前我们想在协程里做一些操作，并将结果反馈到 ViewModel 需要这么操作:

```
class MyViewModel : ViewModel {
    private val _result = MutableLiveData<String>()
    val result: LiveData<String> = _result

    init {
        viewModelScope.launch {
            val computationResult = doComputation()
            _result.value = computationResult
          }
      }
}
```

看看我们做了什么:

1. 准备一个 ViewModel 私有的 MutableLiveData (MLD)
2. 暴露一个不可变的 LiveData
3. 启动协程，然后将其操作结果赋给 MLD

这个做法并不理想。在 LifeCycle 2.2.0 之后，同样的操作可以用更精简的方法来完成，也就是 LiveData 协程构造方法 (coroutine builder):

```
class MyViewModel {
    val result = liveData {
        emit(doComputation())
    }
}
```

这个 liveData 协程构造方法提供了一个协程代码块，这个块就是 LiveData 的作用域，当 LiveData 被观察的时候，里面的操作就会被执行，当 LiveData 不再被使用时，里面的操作就会取消。而且该协程构造方法产生的是一个不可变的 LiveData，可以直接暴露给对应的视图使用。而 emit() 方法则用来更新 LiveData 的数据。

让我们来看另一个常见用例，比如当用户在 UI 中选中一些元素，然后将这些选中的内容显示出来。一个常见的做法是，把被选中的项目的 ID 保存在一个 MutableLiveData 里，然后运行 switchMap。现在在 switchMap 里，您也可以使用协程构造方法:

```
private val itemId = MutableLiveData<String>()
val result = itemId.switchMap {
    liveData { emit(fetchItem(it)) }
}
```

LiveData 协程构造方法还可以接收一个 Dispatcher 作为参数，这样您就可以将这个协程移至另一个线程。

```
liveData(Dispatchers.IO) {
}
```

最后，您还可以使用 emitSource() 方法从另一个 LiveData 获取更新的结果:

```
liveData(Dispatchers.IO) {
    emit(LOADING_STRING)
    emitSource(dataSource.fetchWeather())
}
```

接下来我们来看如何取消协程。绝大部分情况下，协程的取消操作是自动的，毕竟我们在对应的作用域里启动一个协程时，也同时明确了它会在何时被取消。但我们有必要讲一讲如何在协程内部来手动取消协程。

这里补充一个大前提: **所有 kotlin.coroutines 的 suspend 方法都是可取消的**。比如这种:

```
suspend fun printPrimes() {
    while(true) {
        // Compute
  delay(1000)
    }
}
```

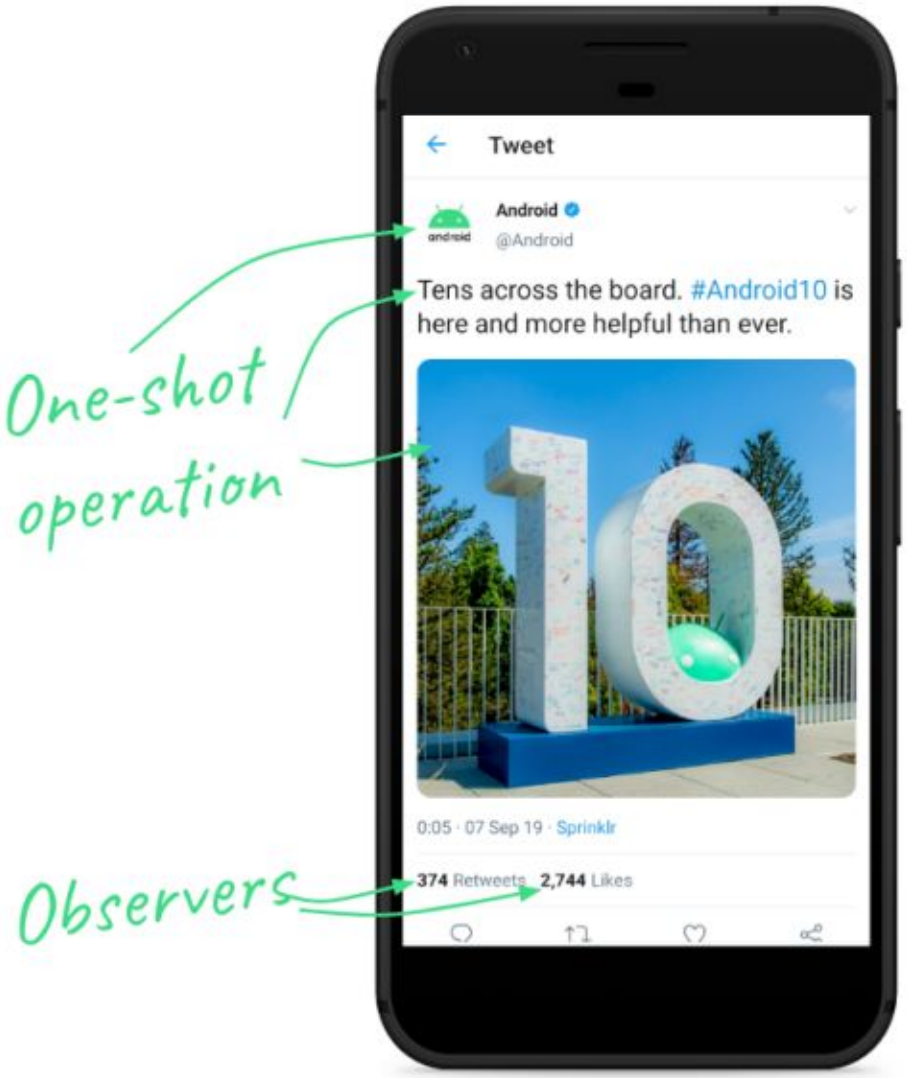
在上面这个无限循环里，每一个 delay 都会检查协程是否处于有效状态，一旦发现协程被取消，循环的操作也会被取消。

那问题来了，如果您在 suspend 方法里调用的是一个不可取消的方法呢？这时您需要使用 isActivate 来进行检查并手动决定是否继续执行操作:

```
suspend fun printPrimes() {
    while(isActive) {
        // Compute
    }
}
```

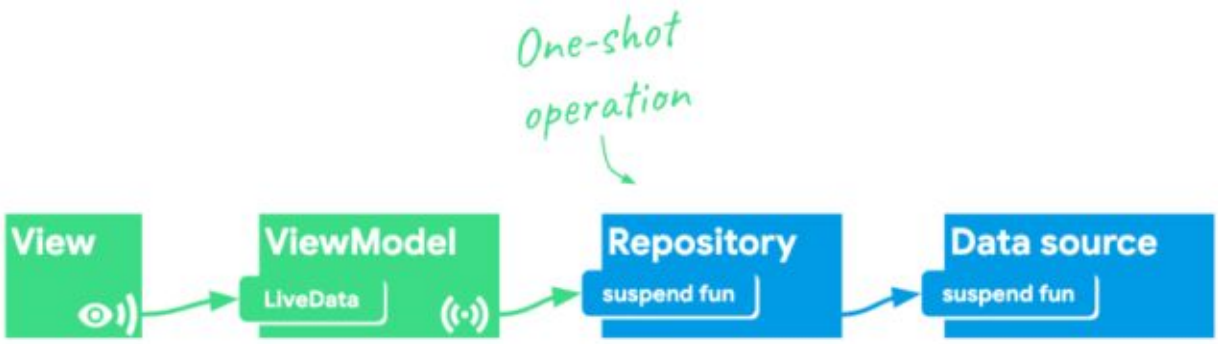
LiveData 操作实践

在进入具体的操作实践环节之前，我们需要区分一下两种操作: 单次 (One-Shot) 操作和监听 (observers) 操作。比如 Twitter 的应用:



单次操作，比如获取用户头像和推文，只需要执行一次即可。监听操作，比如界面下方的转发数和点赞数，就会持续更新数据。

让我们先看看单次操作时的内容架构:




```
class MyViewModel {
    val result = liveData {
        emit(repository.fetchData())
    }
}
```

如前所述，我们使用 LiveData 连接 View 和 ViewModel，而在 ViewModel 这里我们则使用刚刚提到的 liveData 协程构造方法来打通 LiveData 和协程，再往右就是调用 suspend 方法了。

如果我们想监听多个值的话，该如何操作呢?

第一种选择是在 ViewModel 之外也使用 LiveData:

△ Reopsitory 监听 Data Source 暴露出来的 LiveData，同时自己也暴露出 LiveData 供 ViewModel 使用

但是这种实现方式无法体现并发性，比如每次用户登出时，就需要手动取消所有的订阅。LiveData 本身的设计并不适合这种情况，这时我们就需要使用第二种选择: 使用 Flow。

ViewModel 模式

当 ViewModel 监听 LiveData，而且没有对数据进行任何转换操作时，可以直接将 dataSource 中的 LiveData 赋值给 ViewModel 暴露出来的 LiveData:

```
val currentWeather: LiveData<String> =
    dataSource.fetchWeather()
```

如果使用 Flow 的话就需要用到 liveData 协程构造方法。我们从 Flow 中使用 collect 方法获取每一个结果，然后 emit 出来给 liveData 协程构造方法使用:

```
val currentWeatherFlow: LiveData<String> = liveData {
    dataSource.fetchWeatherFlow().collect {
        emit(it)
    }
}
```

不过 Flow 给我们准备了更简单的写法:

```
val currentWeatherFlow: LiveData<String> =
    dataSource.fetchWeatherFlow().asLiveData()
```

接下来一个场景是，我们先发送一个一次性的结果，然后再持续发送多个数值:

```
val currentWeather: LiveData<String> = liveData {
    emit(LOADING_STRING)
    emitSource(dataSource.fetchWeather())
}
```

在 Flow 中我们可以沿用上面的思路，使用 emit 和 emitSource:

```
val currentWeatherFlow: LiveData<String> = liveData {
    emit(LOADING_STRING)
    emitSource(
        dataSource.fetchWeatherFlow().asLiveData()
    )
}
```

但同样的，这种情况 Flow 也有更直观的写法:

```
val currentWeatherFlow: LiveData<String> =
    dataSource.fetchWeatherFlow()
        .onStart { emit(LOADING_STRING) }
        .asLiveData()
```

接下来我们看看需要为接收到的数据做转换时的情况。

使用 LiveData 时，如果用 map 方法做转换，操作会进入主线程，这显然不是我们想要的结果。这时我们可以使用 switchMap，从而可以通过 liveData 协程构造方法获得一个 LiveData，而且 switchMap 的方法会在每次数据源 LiveData 更新时调用。而在方法体内部我们可以使用 heavyTransformation 函数进行数据转换，并发送其结果给 liveData 协程构造方法:

```
val currentWeatherLiveData: LiveData<String> =
    dataSource.fetchWeather().switchMap {
        liveData { emit(heavyTransformation(it)) }
    }
```

使用 Flow 的话会简单许多，直接从 dataSource 获得数据，然后调用 map 方法 (这里用的是 Flow 的 map 方法，而不是 LiveData 的)，然后转化为 LiveData 即可:

```
val currentWeatherFlow: LiveData<String> =
    dataSource.fetchWeatherFlow()
        .map { heavyTransformation(it) }
        .asLiveData()
```

Repository 模式

Repository 一般用来进行复杂的数据转换和处理，而 LiveData 没有针对这种情况进行设计。现在通过 Flow 就可以完成各种复杂的操作:

```
val currentWeatherFlow: Flow<String> =
    dataSource.fetchWeatherFlow()
        .map { ... }
        .filter { ... }
        .dropWhile { ... }
        .combine { ... }
        .flowOn(Dispatchers.IO)
        .onCompletion { ... }
...
```

数据源模式

而在涉及到数据源时，情况变得有些复杂，因为这时您可能是在和其他代码库或者远程数据源进行交互，但是您又无法控制这些数据源。这里我们分两种情况介绍:

1. 单次操作

如果使用 Retrofit 从远程数据源获取数值，直接将方法标记为 suspend 方法即可*:

```
suspend fun doOneShot(param: String) : String =
    retrofitClient.doSomething(param)
```

** Retrofit 从 2.6.0 开始支持 suspend 方法，Room 从 2.1.0 开始支持 suspend 方法。*

如果您的数据源尚未支持协程，比如是一个 Java 代码库，而且使用的是回调机制。这时您可以使用 suspendCancellableCoroutine 协程构造方法，这个方法是协程和回调之间的适配器，会在内部提供一个 continuation 供开发者使用:

```
suspend fun doOneShot(param: String) : Result<String> =
    suspendCancellableCoroutine { continuation ->
        api.addOnCompleteListener { result ->
            continuation.resume(result)
        }.addOnFailureListener { error ->
            continuation.resumeWithException(error)
        }
  }
```

如上所示，在回调方法取得结果后会调用 continuation.resume()，如果报错的话调用的则是 continuation.resumeWithException()。

注意，如果这个协程已经被取消，则 resume 调用也会被忽略。开发者可以在协程被取消时主动取消 API 请求。

2. 监听操作

如果数据源会持续发送数值的话，使用 flow 协程构造方法会很好地满足需求，比如下面这个方法就会每隔 2 秒发送一个新的天气值:

```
override fun fetchWeatherFlow(): Flow<String> = flow {
    var counter = 0
    while(true) {
        counter++
        delay(2000)
        emit(weatherConditions[counter % weatherConditions.size])
    }
}
```

如果开发者使用的是不支持 Flow 而是使用回调的代码库，则可以使用 callbackFlow。比如下面这段代码，api 支持三个回调分支 onNextValue、onApiError 和 onCompleted，我们可以得到结果的分支里使用 offer 方法将值传给 Flow，在发生错误的分支里 close 这个调用并传回一个错误原因 (cause)，而在顺利调用完成后直接 close 调用:

```
fun flowFrom(api: CallbackBasedApi): Flow<T> = callbackFlow {
    val callback = object : Callback {
        override fun onNextValue(value: T) {
            offer(value)
        }
        override fun onApiError(cause: Throwable) {
            close(cause)
        }
        override fun onCompleted() = close()
    }
    api.register(callback)
    awaitClose { api.unregister(callback) }
}
```

注意在这段代码的最后，如果 API 不会再有更新，则使用 awaitClose 彻底关闭这条数据通道。

相信看到这里，您对如何在实际应用中使用协程、LiveData 和 Flow 已经有了比较系统的认识。您可以重温 Android Dev Summit 上 Jose Alcérreca 和 Yigit Boyar 的演讲来巩固理解:

如果您对协程、LiveData 和 Flow 有任何疑问和想法，欢迎在评论区和我们分享。

点击这里**进一步了解 LiveData**

编辑于 05-15

协程　软件生命周期　Android 开发