在 View 上使用挂起函数

谷歌开发者
已认证的官方帐号

已关注

2 人赞同了该文章

Kotlin 协程 让我们可以用同步代码来建立异步问题的模型。这是非常好的特性，但是目前大部分用例都专注于 I/O 任务或是并发操作。其实协程不仅在处理跨线程的问题有优势，还可以用来处理同一线程中的异步问题。

我认为有一个地方可以真正从中受益，那就是在 Android 视图系统中使用协程。

Android 视图 回调

Android 视图系统中尤其热衷于使用回调: 目前在 Android Framework 中，view 和 widgets 类中的回调有 80+ 个，在 Jetpack 中回调的数目更是超过了 200 个 (这里也包含了没有界面的依赖库)。

最常见的用法有以下几项:

- AnimatorListener 获取动画结束相关的事件
- RecyclerView.OnScrollListener 获取滑动状态变更事件
- View.OnLayoutChangeListener 获取 View 布局改变的事件

然后还有一些通过接受 Runnable 来执行异步操作的API，比如 View.post()、View.postDelayed() 等等。

正是因为 Android 上的 UI 编程从根本上就是异步的，所以造成了如此之多的回调。从测量、布局、绘制，到调度插入，整个过程都是异步的。通常情况下，一个类 (通常是 View) 调用系统方法，一段时间之后系统来调度执行，然后通过回调触发监听。

KTX 扩展方法

上述提及的 API，在 Jetpack 中都增加了扩展方法来提高开发效率。其中 **View.doOnPreDraw()**方法是我最喜欢的一个，该方法对等待下一次绘制被执行进行了极大的精简。其实还有很多我常用的方法，比如 **View.doOnLayout()**、**Animator.doOnEnd()**。

但是这些扩展方法也是仅止步于此，他们只是将旧风格的回调 API 改成了 Kotlin 中比较友好的基于 lambda 风格的 API。虽然用起来很优雅，但我们只是在用另一种方式处理回调，这还是没有解决复杂的 UI 的回调嵌套问题。既然我们在讨论异步操作，那在这种情况下，我们可以使用协程优化这些问题么？

使用协程解决问题

这里假定您已经对协程有一定的理解，如果接下来的内容对您来说会有些陌生，可以通过我们今年早期的系列文章进行回顾: **在 Android 开发中使用协程 | 背景介绍**。

挂起函数 (Suspending functions) 是协程的基础组成部分，它允许我们以非阻塞的方式编写代码。这种特性非常适用于我们处理 Android UI，因为我们不想阻塞主线程，阻塞主线程会带来性能上的问题，比如: **jank**。

suspendCancellableCoroutine

在 Kotlin 协程库中，有很多协程的构造器方法，这些构造器方法内部可以使用挂起函数来封装回调的 API。最主要的 API 是 **suspendCoroutine()** 和 **suspendCancellableCoroutine()**，后者是可以被取消的。

我们推荐始终使用 **suspendCancellableCoroutine()**，因为这个方法可以从两个维度处理协程的取消操作:

#1: 可以在异步操作完成之前取消协程。如果某个 view 从它所在的层级中被移除，那么根据协程所处的作用域 (scope)，它有可能会被取消。举个例子: Fragment 返回出栈，通过处理取消事件，我们可以取消异步操作，并清除相关引用的资源。

#2: 在协程被挂起的时候，异步 UI 操作被取消或者抛出异常。并不是所有的操作都有已取消或出错的状态，但是这些操作有。就像后面 Animator 的示例中那样，我们必须把这些状态传递到协程中，让调用者可以处理错误的状态。

等待 View 被布局完成

让我们看一个例子，它封装了一个等待 View 传递下一次布局事件的任务 (比如说，我们改变了一个 TextView 中的内容，需要等待布局事件完成后才能获取该控件的新尺寸):

```
suspend fun View.awaitNextLayout() =
    suspendCancellableCoroutine<Unit> { cont ->
```

```
        // 这里的 lambda 表达式会被立即调用，允许我们创建一个监听器
        val listener = object : View.OnLayoutChangeListener {
            override fun onLayoutChange(...) {
                // 视图的下一次布局任务被调用
                // 先移除监听，防止协程泄漏
                view.removeOnLayoutChangeListener(this)
                // 最终，唤醒协程，恢复执行
                cont.resume(Unit)
            }
        }
        // 如果协程被取消，移除该监听
        cont.invokeOnCancellation { removeOnLayoutChangeListener(listener) }
        // 最终，将监听添加到 view 上
        addOnLayoutChangeListener(listener)

        // 这样协程就被挂起了，除非监听器中的 cont.resume() 方法被调用

    }
```

此方法仅支持协程中一个维度的取消 (# 1 操作)，因为布局操作没有错误状态供我们监听。

接下来我们就可以这样使用了:

```
viewLifecycleOwner.lifecycleScope.launch {
    // 将该视图设置为不可见，再设置一些文字
    titleView.isInvisible = true
    titleView.text = "Hi everyone!"

    // 等待下一次布局事件的任务，然后才可以获取该视图的高度
    titleView.awaitNextLayout()

    // 布局任务被执行
    // 现在，我们可以将视图设置为可见，并其向上平移，然后执行向下的动画
    titleView.isVisible = true
    titleView.translationY = -titleView.height.toFloat()
    titleView.animate().translationY(0f)
}
```

我们为 View 的布局创建了一个 await 函数。用同样的方法可以替代很多常见的回调，比如 **doOnPreDraw()**，它是在 View 得到绘制时调用的方法；再比如 postOnAnimation()，在动画的下一帧开始时调用的方法，等等。

作用域

不知道您有没有发现这样一个问题，在上面的例子中，我们使用了 lifecycleScope 来启动协程，为什么要这样做呢?

为了避免发生内存泄漏，在我们操作 UI 的时候，选择合适的作用域来运行协程是极其重要的。幸运的是，我们的 View 有一些范围合适的 **Lifecycle**。我们可以使用扩展属性 **lifecycleScope** 来获得一个绑定生命周期的 **CoroutineScope**。

LifecycleScope 被包含在 AndroidX 的 lifecycle-runtime-ktx 依赖库中，可以在 这里**找到更多信息**

我们最常用的生命周期的持有者 (lifecycle owner) 就是 Fragment 中的 **viewLifecycleOwner**，只要加载了 Fragment 的视图，它就会处于活跃状态。一旦 Fragment 的视图被移除，与之关联的 **lifecycleScope** 就会自动被取消。又由于我们已经为挂起函数中添加了对取消操作的支持，所以 lifecycleScope 被取消时，所有与之关联的协程都会被清除。

等待 Animator 执行完成

我们再来看一个例子来加深理解，这次是等待 **Animator** 执行结束:

```
suspend fun Animator.awaitEnd() = suspendCancellableCoroutine<Unit> { cont ->

    // 增加一个处理协程取消的监听器，如果协程被取消，
    // 同时执行动画监听器的 onAnimationCancel() 方法，取消动画
    cont.invokeOnCancellation { cancel() }

    addListener(object : AnimatorListenerAdapter() {
        private var endedSuccessfully = true

        override fun onAnimationCancel(animation: Animator) {
            // 动画已经被取消，修改是否成功结束的标志
            endedSuccessfully = false
        }

        override fun onAnimationEnd(animation: Animator) {

            // 为了在协程恢复后的不发生泄漏，需要确保移除监听
            animation.removeListener(this)
            if (cont.isActive) {

                // 如果协程仍处于活跃状态
                if (endedSuccessfully) {
                    // 并且动画正常结束，恢复协程
                    cont.resume(Unit)
                } else {
                    // 否则动画被取消，同时取消协程
                    cont.cancel()
```

```
                }
            }
        }
    })
}
```

这个方法支持两个维度的取消，我们可以分别取消动画或者协程:

#1: 在 Animator 运行的时候，协程被取消 。我们可以通过 **invokeOnCancellation** 回调方法来监听协程何时被取消，这能让我们同时取消动画。

#2: 在协程被挂起的时候，Animator 被取消 。我们通过 **onAnimationCancel()** 回调来监听动画被取消的事件，通过调用协程的 cancel() 方法来取消挂起的协程。

这就是使用挂起函数等待方法执行来封装回调的基本使用了。

组合使用

到这里，您可能有这样的疑问，"看起来不错，但是我能从中收获什么呢？" 单独使用其中某个方法，并不会产生多大的作用，但是如果把它们组合起来，便能发挥巨大的威力。

下面是一个使用 Animator.awaitEnd() 来依次运行 3 个动画的示例:

```
viewLifecycleOwner.lifecycleScope.launch {
    ObjectAnimator.ofFloat(imageView, View.ALPHA, 0f, 1f).run {
        start()
        awaitEnd()
    }

    ObjectAnimator.ofFloat(imageView, View.TRANSLATION_Y, 0f, 100f).run {
        start()
        awaitEnd()
    }

    ObjectAnimator.ofFloat(imageView, View.TRANSLATION_X, -100f, 0f).run {
        start()
        awaitEnd()
    }
}
```

这是一个很常见的使用案例，您可以把这些动画放进 **AnimatorSet** 中来实现同样的效果。

但是这里使用的方法适用于不同类型的异步操作: 我们使用一个 **ValueAnimator**，一个 **RecyclerView** 的平滑滚动，以及一个 **Animator** 来举例:

```
viewLifecycleOwner.lifecycleScope.launch {
    // #1: ValueAnimator
    imageView.animate().run {
        alpha(0f)
        start()
        awaitEnd()
    }

    // #2: RecyclerView smooth scroll
    recyclerView.run {
        smoothScrollToPosition(10)
        // 该方法和其他方法类似，等待当前的滑动完成，我们不需要刻意关注实现
        // 代码可以在文末的引用中找到
        awaitScrollEnd()
    }

    // #3: ObjectAnimator
    ObjectAnimator.ofFloat(textView, View.TRANSLATION_X, -100f, 0f).run {
        start()
        awaitEnd()
    }
}
```

试着用 AnimatorSet 实现一下吧！ 如果不用协程，那就意味着我们要监听每一个操作，在回调中执行下一个操作，这回调层级想想都可怕。

通过把不同的异步操作转换为协程的挂起函数，我们获得了简洁明了地编排它们的能力。

我们还可以更进一步...

如果我们希望 **ValueAnimator 和平滑滚动同时开始，然后在两者都完成之后启动 **ObjectAnimator**，该怎么做呢？**那么在使用了协程之后，我们可以使用 **async()** 来并发地执行我们的代码：

```
viewLifecycleOwner.lifecycleScope.launch {
    val anim1 = async {
        imageView.animate().run {
            alpha(0f)
            start()
            awaitEnd()
        }
    }

    val scroll = async {
        recyclerView.run {
            smoothScrollToPosition(10)
```

```
            awaitScrollEnd()
        }
    }

    // 等待以上两个操作全部完成
    anim1.await()
    scroll.await()

    // 此时，anim1 和滑动都完成了，我们开始执行 ObjectAnimator
    ObjectAnimator.ofFloat(textView, View.TRANSLATION_X, -100f, 0f).run {
        start()
        awaitEnd()
    }
}
```

但是如果您还想让滚动延迟执行怎么办呢？ (类似 Animator.startDelay 方法) 那么使用协程也有很好的实现，我们可以用 **delay()** 方法:

```
viewLifecycleOwner.lifecycleScope.launch {
    val anim1 = async {
        // ...
    }

    val scroll = async {
        // 我们希望在 anim1 完成后，延迟 200ms 执行滚动
        delay(200)

        recyclerView.run {
            smoothScrollToPosition(10)
            awaitScrollEnd()
        }
    }

    // …
}
```

如果我们想重复动画，那么我们可以使用 **repeat()** 方法，或者使用 for 循环实现。下面是一个 view 淡入淡出 3 次的例子:

```
viewLifecycleOwner.lifecycleScope.launch {
    repeat(3) {
        ObjectAnimator.ofFloat(textView, View.ALPHA, 0f, 1f, 0f).run {
            start()
            awaitEnd()
        }
```

```
    }
}
```

您甚至可以通过重复计数来实现更精妙的功能。假设您希望淡入淡出在每次重复中逐渐变慢:

```
viewLifecycleOwner.lifecycleScope.launch {
    repeat(3) { repetition ->
        ObjectAnimator.ofFloat(textView, View.ALPHA, 0f, 1f, 0f).run {
            // 第一次执行持续 150ms，第二次：300ms，第三次：450ms
            duration = (repetition + 1) * 150L
            start()
            awaitEnd()
        }
    }
}
```

在我看来，这就是在 Android 视图系统中使用协程能真正发挥作用的地方。我们就算不去组合不同类型的回调，也能创建复杂的异步变换，或是将不同类型的动画组合起来。

通过使用与我们应用中数据层相同的协程开发原语，还能使 UI 编程更便捷。对于刚接触代码的人来说， await 方法要比看似会断开的回调更具可读性。

最后

希望通过本文，您可以进一步思考协程还可以在哪些其他的 API 中发挥作用。

接下来的文章中，我们将探讨如何使用协程来组织一个复杂的变换动画，其中也包括了一些常见 View 的实现，感兴趣的读者请继续关注我们的更新。

发布于 9 小时前

协程　Android 开发　Kotlin