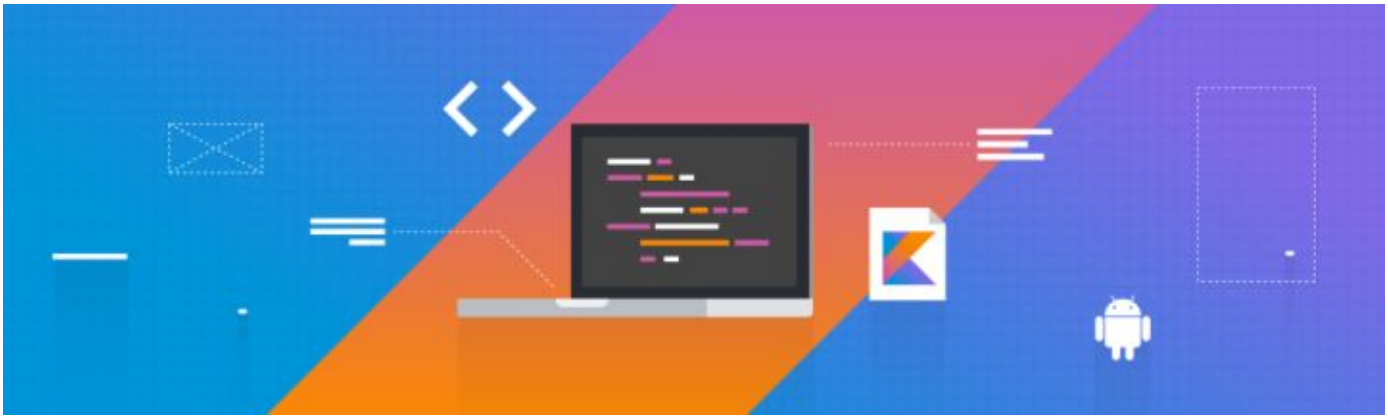
在 Android 开发中使用协程 | 上手指南

谷歌开发者
已认证的官方帐号

已关注



本文是介绍 Android 协程系列中的第二部分，这篇文章主要会介绍如何使用协程来处理任务，并且能在任务开始执行后保持对它的追踪。

保持对协程的追踪

本系列文章的第一篇，我们探讨了协程适合用来解决哪些问题。这里再简单回顾一下，协程适合解决以下两个常见的编程问题:

1. **处理耗时任务 (Long running tasks)**，这种任务常常会阻塞住主线程；
2. **保证主线程安全 (Main-safety)**，即确保安全地从主线程调用任何 suspend 函数。

协程通过在常规函数之上增加 suspend 和 resume 两个操作来解决上述问题。当某个特定的线程上的所有协程被 suspend 后，该线程便可腾出资源去处理其他任务。

协程自身并不能够追踪正在处理的任务，但是有成百上千个协程并对它们同时执行挂起操作并没有太大问题。协程是轻量级的，但处理的任务却不一定是轻量的，比如读取文件或者发送网络请求。

使用代码来手动追踪上千个协程是非常困难的，您可以尝试对所有协程进行跟踪，手动确保它们都完成了或者都被取消了，那么代码会臃肿且易出错。如果代码不是很完美，就会失去对协程的追踪，也就是所谓 "work leak" 的情况。

任务泄漏 (work leak) 是指某个协程丢失无法追踪，它类似于内存泄漏，但比它更加糟糕，这样丢失的协程可以恢复自己，从而占用内存、CPU、磁盘资源，甚至会发起一个网络请求，而这也意味着它所占用的这些资源都无法得到重用。

泄漏协程会浪费内存、CPU、磁盘资源，甚至发送一个无用的网络请求。

为了能够避免协程泄漏，Kotlin 引入了结构化并发(structured concurrency) 机制，它是一系列编程语言特性和实践指南的结合，遵循它能帮助您追踪到所有运行于协程中的任务。

在 Android 平台上，我们可以使用结构化并发来做到以下三件事:

1. **取消任务** —— 当某项任务不再需要时取消它；
2. **追踪任务** —— 当任务正在执行时，追踪它；
3. **发出错误信号** —— 当协程失败时，发出错误信号表明有错误发生。

接下来我们对以上几点一一进行探讨，看看结构化并发是如何帮助能够追踪所有协程，而不会导致泄漏出现的。

借助 scope 来取消任务

在 Kotlin 中，定义协程必须指定其 CoroutineScope 。CoroutineScope 可以对协程进行追踪，即使协程被挂起也是如此。同第一篇文章中讲到的调度程序 (Dispatcher) 不同，CoroutineScope 并不运行协程，它只是确保您不会失去对协程的追踪。

为了确保所有的协程都会被追踪，Kotlin 不允许在没有使用 CoroutineScope 的情况下启动新的协程。CoroutineScope 可被看作是一个具有超能力的 ExecutorService 的轻量级版本。它能启动新的协程，同时这个协程还具备我们在第一部分所说的 suspend 和 resume 的优势。

CoroutineScope 会跟踪所有协程，同样它还可以取消由它所启动的所有协程。这在 Android 开发中非常有用，比如它能够在用户离开界面时停止执行协程。

CoroutineScope 会跟踪所有协程，并且可以取消由它所启动的所有协程。

启动新的协程

需要特别注意的是，您不能随便就在某个地方调用 suspend 函数，suspend 和 resume 机制要求您从常规函数中切换到协程。

有两种方式能够启动协程，它们分别适用于不同的场景:

1. launch 构建器适合执行 "一劳永逸" 的工作，意思就是说它可以启动新协程而不将结果返回给调用方；
2. async 构建器可启动新协程并允许您使用一个名为 await 的挂起函数返回 result。

通常，您应使用 launch 从常规函数中启动新协程。因为常规函数无法调用 await (记住，它无法直接调用 suspend 函数)，所以将 async 作为协程的主要启动方法没有多大意义。稍后我们会讨论应该如何使用 async。

您应该改为使用 coroutine scope 调用 launch 方法来启动协程。

```
scope.launch {
    // 这段代码在作用域里启动了一个新协程
   // 它可以调用挂起函数
```

```
    fetchDocs()
}
```

您可以将 launch 看作是将代码从常规函数送往协程世界的桥梁。在 launch 函数体内，您可以调用 suspend 函数并能够像我们上一篇介绍的那样保证主线程安全。

Launch 是将代码从常规函数送往协程世界的桥梁。

注意: launch 和 async 之间的很大差异是它们对异常的处理方式不同。async 期望最终是通过调用 await 来获取结果 (或者异常)，所以默认情况下它不会抛出异常。这意味着如果使用 async 启动新的协程，它会静默地将异常丢弃。

由于 launch 和 async 仅能够在 CouroutineScope 中使用，所以任何您所创建的协程都会被该 scope 追踪。Kotlin 禁止您创建不能够被追踪的协程，从而避免协程泄漏。

在 ViewModel 中启动协程

既然 CoroutineScope 会追踪由它启动的所有协程，而 launch 会创建一个新的协程，那么您应该在什么地方调用 launch 并将其放在 scope 中呢? 又该在什么时候取消在 scope 中启动的所有协程呢?

在 Android 平台上，您可以将 CoroutineScope 实现与用户界面相关联。这样可让您避免泄漏内存或者对不再与用户相关的 Activities 或 Fragments 执行额外的工作。当用户通过导航离开某界面时，与该界面相关的 CoroutineScope 可以取消掉所有不需要的任务。

结构化并发能够保证当某个作用域被取消后，它内部所创建的所有协程也都被取消。

当将协程同 Android 架构组件 (Android Architecture Components) 集成起来时，您往往会需要在 ViewModel 中启动协程。因为大部分的任务都是在这里开始进行处理的，所以在这个地方启动是一个很合理的做法，您也不用担心旋转屏幕方向会终止您所创建的协程。

从生命周期感知型组件 (AndroidX Lifecycle) 的 2.1.0 版本开始 (发布于 2019 年 9 月)，我们通过添加扩展属性 ViewModel.viewModelScope 在 ViewModel 中加入了协程的支持。

推荐您阅读 Android 开发者文档 "将 Kotlin 协程与架构组件一起使用" 了解更多。

看看如下示例:

```
class MyViewModel(): ViewModel() {
    fun userNeedsDocs() {
       // 在 ViewModel 中启动新的协程
        viewModelScope.launch {
            fetchDocs()
        }
```

```
    }
}
```

当 viewModelScope 被清除 (当 onCleared() 回调被调用时) 之后，它将自动取消它所启动的所有协程。这是一个标准做法，如果一个用户在尚未获取到数据时就关闭了应用，这时让请求继续完成就纯粹是在浪费电量。

为了提高安全性，CoroutineScope 会进行自行传播。也就是说，如果某个协程启动了另一个新的协程，它们都会在同一个 scope 中终止运行。这意味着，即使当某个您所依赖的代码库从您创建的 viewModelScope 中启动某个协程，您也有方法将其取消。

注意: 协程被挂起时，系统会以抛出 CancellationException 的方式协作取消协程。捕获顶级异常 (如Throwable) 的异常处理程序将捕获此异常。如果您做异常处理时消费了这个异常，或从未进行 suspend 操作，那么协程将会徘徊于半取消 (semi-canceled) 状态下。

所以，当您需要将一个协程同 ViewModel 的生命周期保持一致时，使用 viewModelScope 来从常规函数切换到协程中。然后，viewModelScope 会自动为您取消协程，因此在这里哪怕是写了死循环也是完全不会产生泄漏。如下示例:

```
fun runForever() {
    // 在 ViewModel 中启动新的协程
    viewModelScope.launch {
        // 当 ViewModel 被清除后，下列代码也会被取消
        while(true) {
            delay(1_000)
           // 每过 1 秒做点什么
        }
    }
}
```

通过使用 viewModelScope，可以确保所有的任务，包含死循环在内，都可以在不需要的时候被取消掉。

任务追踪

使用协程来处理任务对于很多代码来说真的很方便。启动协程，进行网络请求，将结果写入数据库，一切都很自然流畅。

但有时候，可能会遇到稍微复杂点的问题，例如您需要在一个协程中同时处理两个网络请求，这种情况下需要启动更多协程。

想要创建多个协程，可以在 suspend function 中使用名为coroutineScope或supervisorScope这样的构造器来启动多个协程。但是这个 API 说实话，有点令人困惑。coroutineScope 构造器和 CoroutineScope 这两个的区别只是一个字符之差，但它们却是完全不同的东西。

另外，如果随意启动新协程，可能会导致潜在的任务泄漏 (work leak)。调用方可能感知不到启用了新的协程，也就意味着无法对其进行追踪。

为了解决这个问题，结构化并发发挥了作用，它保证了当 suspend 函数返回时，就意味着它所处理的任务也都已完成。

结构化并发保证了当 suspend 函数返回时，它所处理任务也都已完成。

示例使用 coroutineScope 来获取两个文档内容:

```
suspend fun fetchTwoDocs() {
    coroutineScope {
        launch { fetchDoc(1) }
        async { fetchDoc(2) }
    }
}
```

在这个示例中，同时从网络中获取两个文档数据，第一个是通过 launch 这样 "一劳永逸" 的方式启动协程，这意味着它不会返回任何结果给调用方。

第二个是通过 async 的方式获取文档，所以是会有返回值返回的。不过上面示例有一点奇怪，因为通常来讲两个文档的获取都应该使用 async，但这里我仅仅是想举例来说明可以根据需要来选择使用 launch 还是 async，或者是对两者进行混用。

coroutineScope 和 supervisorScope 可以让您安全地从 suspend 函数中启动协程。

但是请注意，这段代码不会显式地等待所创建的两个协程完成任务后才返回，当 fetchTwoDocs 返回时，协程还正在运行中。

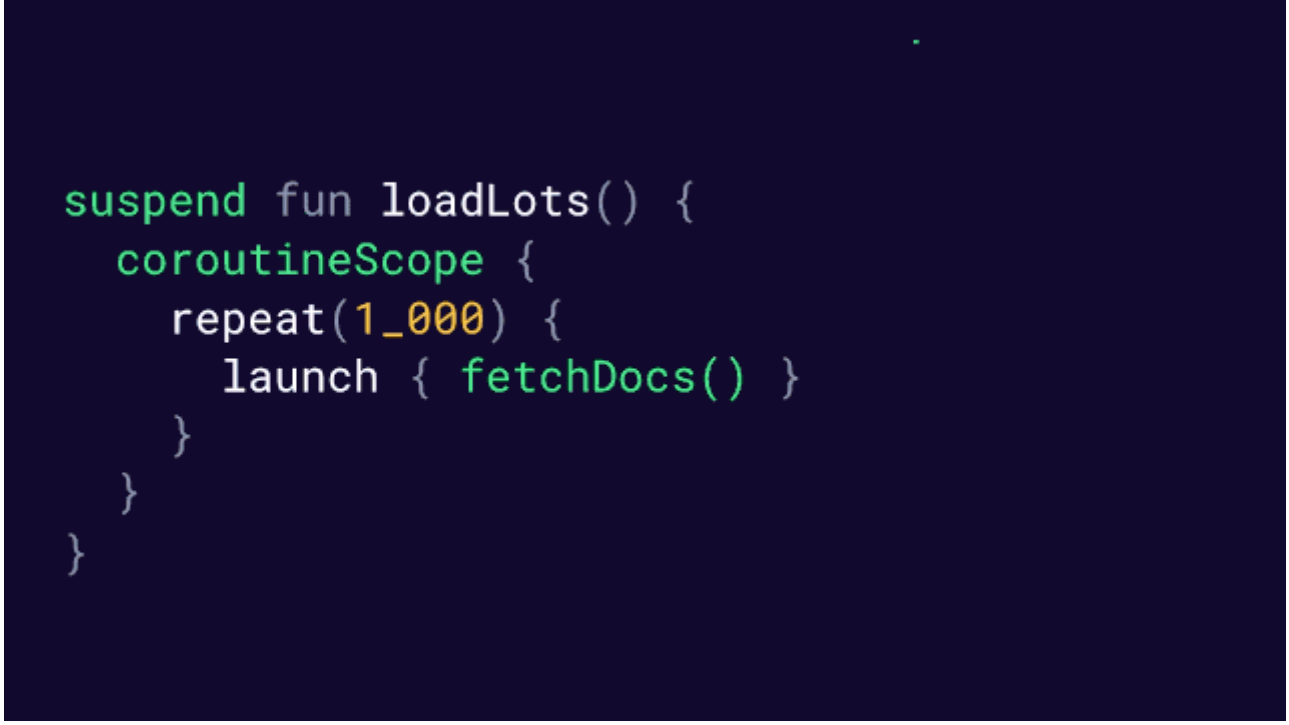
所以，为了做到结构化并发并避免泄漏的情况发生，我们想做到在诸如 fetchTwoDocs 这样的 suspend 函数返回时，它们所做的所有任务也都能结束。换个说法就是，fetchTwoDocs 返回之前，它所启动的所有协程也都能完成任务。

Kotlin 确保使用 coroutineScope 构造器不会让 fetchTwoDocs 发生泄漏，coroutinScope 会先将自身挂起，等待它内部启动的所有协程完成，然后再返回。因此，只有在 coroutineScope 构建器中启动的所有协程完成任务之后，fetchTwoDocs 函数才会返回。

处理一堆任务

既然我们已经做到了追踪一两个协程，那么来个刺激的，追踪一千个协程来试试！

先看看下面这个动画:



这个动画展示了 coroutineScope 是如何追踪一千个协程的。

这个动画向我们展示了如何同时发出一千个网络请求。当然，在真实的 Android 开发中最好别这么做，太浪费资源了。

这段代码中，我们在 coroutineScope 构造器中使用 launch 启动了一千个协程，您可以看到这一切是如何联系到一起的。由于我们使用的是 suspend 函数，因此代码一定使用了 CoroutineScope 创建了协程。我们目前对这个 CoroutineScope 一无所知，它可能是viewModelScope 或者是其他地方定义的某个 CoroutineScope，但不管怎样，coroutineScope 构造器都会使用它作为其创建新的 scope 的父级。

然后，在 coroutineScope 代码块内，launch 将会在新的 scope 中启动协程，随着协程的启动完成，scope 会对其进行追踪。最后，一旦所有在 coroutineScope 内启动的协程都完成后，loadLots 方法就可以轻松地返回了。

注意: scope 和协程之间的父子关系是使用 Job 对象进行创建的。但是您不需要深入去了解，只要知道这一点就可以了。

coroutineScope 和 supervisorScope 将会等待所有的子协程都完成。

以上的重点是，使用 coroutineScope 和 supervisorScope 可以从任何 suspend function 来安全地启动协程。即使是启动一个新的协程，也不会出现泄漏，因为在新的协程完成之前，调用方始终处于挂起状态。

更厉害的是，coroutineScope 将会创建一个子 scope，所以一旦父 scope 被取消，它会将取消的消息传递给所有新的协程。如果调用方是 viewModelScope，这一千个协程在用户离开界面后都会自动被取消掉，非常整洁高效。

在继续探讨报错 (error) 相关的问题之前，有必要花点时间来讨论一下 supervisorScope 和 coroutineScope，它们的主要区别是当出现任何一个子 scope 失败的情况，coroutineScope 将会被取消。如果一个网络请求失败了，所有其他的请求都将被立即取消，这种需求选择

coroutineScope。相反，如果您希望即使一个请求失败了其他的请求也要继续，则可以使用 supervisorScope，当一个协程失败了，supervisorScope 是不会取消剩余子协程的。

协程失败时发出报错信号

在协程中，报错信号是通过抛出异常来发出的，就像我们平常写的函数一样。来自 suspend 函数的异常将通过 resume 重新抛给调用方来处理。跟常规函数一样，您不仅可以使用 try/catch 这样的方式来处理错误，还可以构建抽象来按照您喜欢的方式进行错误处理。

但是，在某些情况下，协程还是有可能会弄丢获取到的错误的。

```
val unrelatedScope = MainScope()
// 丢失错误的例子
suspend fun lostError() {
   // 未使用结构化并发的 async
    unrelatedScope.async {
        throw InAsyncNoOneCanHearYou("except")
    }
}
```

注意: 上述代码声明了一个无关联协程作用域，它将不会按照结构化并发的方式启动新的协程。还记得我在一开始说的结构化并发是一系列编程语言特性和实践指南的集合，在 suspend 函数中引入无关联协程作用域违背了结构化并发规则。

在这段代码中错误将会丢失，因为 async 假设您最终会调用 await 并且会重新抛出异常，然而您并没有去调用 await，所以异常就永远在那等着被调用，那么这个错误就永远不会得到处理。

结构化并发保证当一个协程出错时，它的调用方或作用域会被通知到。

如果您按照结构化并发的规范去编写上述代码，错误就会被正确地抛给调用方处理。

```
suspend fun foundError() {
    coroutineScope {
        async {
            throw StructuredConcurrencyWill("throw")
        }
    }
}
```

coroutineScope 不仅会等到所有子任务都完成才会结束，当它们出错时它也会得到通知。如果一个通过 coroutineScope 创建的协程抛出了异常，coroutineScope 会将其抛给调用方。因为我们用的是coroutineScope 而不是 supervisorScope，所以当抛出异常时，它会立刻取消所有的子任务。

使用结构化并发

在这篇文章中，我介绍了结构化并发，并展示了如何让我们的代码配合 Android 中的 ViewModel 来避免出现任务泄漏。

同样，我还帮助您更深入去理解和使用 suspend 函数，通过确保它们在函数返回之前完成任务，或者是通过暴露异常来确保它们正确发出错误信号。

如果我们使用了不符合结构化并发的代码，将会很容易出现协程泄漏，即调用方不知如何追踪任务的情况。这种情况下，任务是无法取消的，同样也不能保证异常会被重新抛出来。这样会使得我们的代码很难理解，并可能会导致一些难以追踪的 bug 出现。

您可以通过引入一个新的不相关的 CoroutineScope (注意是大写的 C)，或者是使用 GlobalScope 创建的全局作用域，但是这种方式的代码不符合结构化并发要求的方式。

但是当出现需要协程比调用方的生命周期更长的情况时，就可能需要考虑非结构化并发的编码方式了，只是这种情况比较罕见。因此，使用结构化编程来追踪非结构化的协程，并进行错误处理和任务取消，将是非常不错的做法。

如果您之前一直未按照结构化并发的方法编码，一开始确实一段时间去适应。这种结构确实保证与 suspend 函数交互更安全，使用起来更简单。在编码过程中，尽可能多地使用结构化并发，这样让代码更易于维护和理解。

在本文的开始列举了结构化并发为我们解决的三个问题:

1. **取消任务** —— 当某项任务不再需要时取消它；
2. **追踪任务** —— 当任务正在执行时，追踪它；
3. **发出错误信号** —— 当协程失败时，发出错误信号表明有错误发生。

实现这种结构化并发，会为我们的代码提供一些保障:

1. **作用域取消**时，它内部所有的**协程也会被取消**；
2. **suspend 函数返回**时，意味着它的所有**任务都已完成**；
3. **协程报错**时，它所在的**作用域或调用方会收到报错通知**。

总结来说，结构化并发让我们的代码更安全，更容易理解，还避免了出现任务泄漏的情况。

下一步

本篇文章，我们探讨了如何在 Android 的 ViewModel 中启动协程，以及如何在代码中运用结构化并发，来让我们的代码更易于维护和理解。

在下一篇文章中，我们将探讨如何在实际编码过程中使用协程，感兴趣的读者请继续关注我们的更新。

发布于 04-23

协程　Android 开发　Android