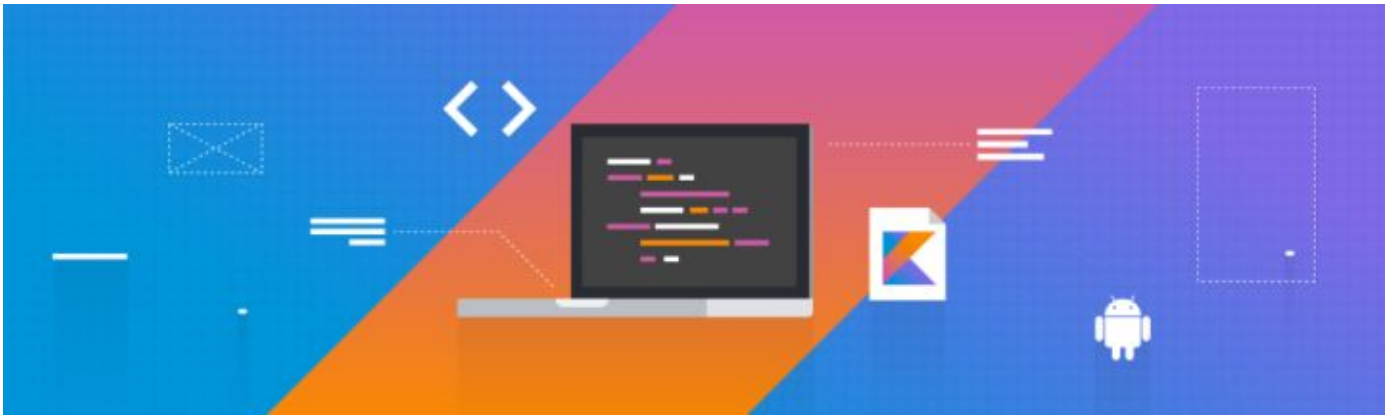
在 Android 开发中使用协程 | 代码实战

谷歌开发者
已认证的官方帐号

已关注

6 人赞同了该文章



本文是介绍 Android 协程系列中的第三部分，这篇文章通过发送一次性请求来介绍如何使用协程处理在实际编码过程中遇到的问题。在阅读本文之前，建议您先阅读本系列的前两篇文章，关于在 Android 开发中使用协程的背景介绍和上手指南。

使用协程解决实际编码问题

前两篇文章主要是介绍了如何使用协程来简化代码，在 Android 上保证主线程安全，避免任务泄漏。以此为背景，我们认为使用协程是在处理后台任务和简化 Android 回调代码的绝佳方案。

目前为止，我们主要集中在介绍协程是什么，以及如何管理它们，本文我们将介绍如何使用协程来完成一些实际任务。协程同函数一样，是在编程语言特性中的一个常用特性，您可以使用它来实现任何可以通过函数和对象能实现的功能。但是，在实际编程中，始终存在两种类型的任务非常适合使用协程来解决:

1. **一次性请求 (one shot requests)** 是那种调用一下就请求一下，请求获取到结果后就结束执行；
2. **流式请求 (streaming request)** 在发出请求后，还一直监听它的变化并返回给调用方，在拿到第一个结果之后它们也不会结束。

协程对于处理这些任务是一个绝佳的解决方案。在这篇文章中，我们将会深入介绍一次性请求，并探索如何在 Android 中使用协程实现它们。

一次性请求

一次性请求会调用一次就请求一次，获取到结果后就结束执行。这个模式同调用常规函数很像 —— 调用一次，执行，然后返回。正因为同函数调用相似，所以相对于流式请求它更容易理解。

一次性请求会调用一次就请求一次，获取到结果后就结束执行。

举例来说，您可以把它类比为浏览器加载页面。当您点击了这篇文章的链接后，浏览器向服务器发送了网络请求，然后进行页面加载。一旦页面数据传输到浏览器后，浏览器就有了所有需要的数据，然后停止同后端服务的对话。如果服务器后来又修改了这篇文章的内容，新的更改是不会显示在浏览器中的，除非您主动刷新了浏览器页面。

尽管这样的方式缺少了流式请求那样的实时推送特性，但是它还是非常有用的。在 Android 的应用中您可以用这种方式解决很多问题，比如对数据的查询、存储或更新，它还很适用于处理列表排序问题。

问题: 展示一个有序列表

我们通过一个展示有序列表的例子来探索一下如何构建一次性请求。为了让例子更具体一些，我们来构建一个用于商店员工使用的库存应用，使用它能够根据上次进货的时间来查找相应商品，并能够以升序和降序的方式排列。因为这个仓库中存储的商品很多，所以对它们进行排序要花费将近 1 秒钟，因此我们需要使用协程来避免阻塞主线程。

在应用中，所有的数据都会存储到 Room 数据库中。由于不涉及到网络请求，因此我们不需要进行网络请求，从而专注于一次性请求这样的编程模式。由于无需进行网络请求，这个例子会很简单，尽管如此它仍然展示了该使用怎样的模式来实现一次性请求。

为了使用协程来实现此需求，您需要在协程中引入 ViewModel、Repository 和 Dao。让我们逐个进行介绍，看看如何把它们同协程整合在一起。

```
class ProductsViewModel(
    val productsRepository: ProductsRepository): ViewModel() {
   private val _sortedProducts = MutableLiveData<List<ProductListing>>()
   val sortedProducts: LiveData<List<ProductListing>> = _sortedProducts

   /**
    * 当用户点击相应排序按钮后，UI 进行调用
    */
   fun onSortAscending() = sortPricesBy(ascending = true)
   fun onSortDescending() = sortPricesBy(ascending = false)

   private fun sortPricesBy(ascending: Boolean) {
       viewModelScope.launch {
      // suspend 和 resume 使得这个数据库请求是主线程安全的，
      // 所以 ViewModel 不需要关心线程安全问题
           _sortedProducts.value =
                   productsRepository.loadSortedProducts(ascending)
       }
   }
}
```

ProductsViewModel 负责从 UI 层接受事件，然后向 repository 请求更新的数据。它使用 LiveData 来存储当前排序的列表数据，以供 UI 进行展示。当出现某个新事件时，sortProductsBy 会启动一个新的协程对列表进行排序，当排序完成后更新LiveData。在这种架构下，通常都是使用ViewModel启动协程，因为这样做的话可以在 onCleared 中取消所启动的协程。当用户离开此界面后，这些任务就没必要继续进行了。

**如果您之前没有用过 LiveData，您可以看看这篇由@CeruleanOtter写的文章，它介绍了 LiveData 是如何为 UI 保存数据的 —— ViewModels: A Simple Example。*

- @CeruleanOtter

https://twitter.com/CeruleanOtter
twitter.com

- ViewModels: A Simple Example

https://medium.com/androiddevelopers/viewmodels-a-simple-example-...
medium.com

这是在 Android 上使用协程的通用模式。由于 Android framework 不会主动调用挂起函数，所以您需要配合使用协程来响应 UI 事件。最简单的方法就是来一个事件就启动一个新的协程，最适合处理这种情况的地方就是 ViewModel 了。

在 ViewModel 中启动协程是很通用的模式。

ViewModel 实际上使用了 ProductsRepository 来获取数据，示例代码如下:

```
class ProductsRepository(val productsDao: ProductsDao) {

    /**
         这是一个普通的挂起函数，也就是说调用方必须在一个协程中。
         repository 并不负责启动或者停止协程，
         因为它并不负责对协程生命周期的掌控。
         这可能会在 Dispatchers.Main 中调用，同样它也是主线程安全的，
         因为 Room 会为我们保证主线程安全。
     */
    suspend fun loadSortedProducts(ascending: Boolean): List<ProductListing> {
        return if (ascending) {
            productsDao.loadProductsByDateStockedAscending()
        } else {
            productsDao.loadProductsByDateStockedDescending()
        }
```

```
    }
}
```

ProductsRepository 提供了一个合理的同商品数据进行交互的接口，此应用中，所有内容都存储在本地 Room 数据库中，它为 @Dao 提供了针对不同排序具有不同功能的两个接口。

repository 是 Android 架构组件中的一个可选部分，如果您在应用中已经集成了它或者其他的相似功能的模块，那么它应该更偏向于使用挂起函数。因为 repository 并没有生命周期，它仅仅是一个对象，所以它不能处理资源的清理工作，所以默认情况下，repository 中启动的所有协程都有可能出现泄漏。

使用挂起函数除了避免泄漏之外，在不同的上下文中也可以重复使用 repository，任何知道如何创建协程的都可以调用 loadSortedProducts，例如 WorkManager 所调度管理的后台任务就可以直接调用它。

repository 应该使用挂起函数来保证主线程安全。

注意: 当用户离开界面后，有些在后台中处理数据保存的操作可能还要继续工作，这种情况下脱离了应用生命周期来运行是没有意义的，所以大部分情况下 viewModelScope 都是一个好的选择。

再来看看 ProductsDao，示例代码如下：

```
@Dao
interface ProductsDao {

    // 因为这个方法被标记为了 suspend，Room 将会在保证主线程安全的前提下
     // 使用自己的调度器来运行这个查询
    @Query("select * from ProductListing ORDER BY dateStocked ASC")
    suspend fun loadProductsByDateStockedAscending(): List<ProductListing>
    // 因为这个方法被标记为了 suspend，Room 将会在保证主线程安全的前提下
     // 使用自己的调度器来运行这个查询
    @Query("select * from ProductListing ORDER BY dateStocked DESC")
    suspend fun loadProductsByDateStockedDescending(): List<ProductListing>
}
```

ProductsDao 是一个 Room @Dao，它对外提供了两个挂起函数，因为这些函数都增加了 suspend 修饰，所以 Room 会保证它们是主线程安全的，这也意味着您可以直接在 Dispatchers.Main 中调用它们。

**如果您没有在 Room 中使用过协程，您可以先看看这篇由 @FMuntenescu写的文章: Room Coroutines*

- @FMuntenescu

twitter.com

- Room Coroutines

https://medium.com/androiddevelopers/room-coroutines-422b786dc4c5

medium.com

不过要注意的是，调用它的协程将会在主线程上执行。所以，如果您要对执行结果做一些比较耗时的操作，比如对列表内容进行转换，您要确保这个操作不会阻塞主线程。

注意: Room 使用了自己的调度器在后台线程上进行查询操作。您不应该再使用 withContext(Dispatchers.IO) 来调用 Room 的 suspend 查询，这只会让您的代码变复杂，也会拖慢查询速度。

Room 的挂起函数是主线程安全的，并运行于自定义的调度器中。

一次性请求模式

这是在 Android 架构组件中使用协程进行一次性请求的完整模式，我们将协程添加到了 ViewModel、Repository 和 Room 中，每一层都有着不同的责任分工。

1. ViewModel 在主线程上启动了协程，一旦有结果后就结束执行；
2. Repository 提供了保证主线程安全的挂起函数；
3. 数据库和网络层提供了保证主线程安全的挂起函数。

ViewModel负责启动协程，并保证用户离开了相应界面时它们就会被取消。它本身并不会做一些耗时的操作，而是依赖别的层级来做。一旦有了结果，就使用 LiveData 将数据发送到 UI 层。因为 ViewModel 并不做一些耗时操作，所以它是在主线程启动协程的，以便能够更快地响应用户事件。

Repository提供了挂起函数用来访问数据，它通常不会启动一些生命周期比较长的协程，因为它们一旦启动了便无法取消。无论何时 Repository 想要做一些耗时操作，比如对列表内容进行转换，都应该使用 withContext 来提供主线程安全的接口。

数据层 (网络或数据库)总是会提供挂起函数，使用 Kotlin 协程的时候要保证这些挂起函数是主线程安全的，Room 和 Retrofit 都遵循了这一点。

在一次性请求中，数据层只提供挂起函数，调用方如果想要获取最新的值，只能再次进行调用，这就像浏览器中的刷新按钮一样。

花点时间让您了解一次性请求的模式是值得，它在 Android 协程中是比较通用的模式，您会一直用到它。

第一个 bug 出现了

在经过测试后，您部署到了生产环境，运行了几周都感觉良好，直到您收到了一个很奇怪的 bug 报告:

标题: — 排序错误!

错误报告: 当我非常快速地点击排序按钮时，排序的结果偶尔是错的，这还不是每次都能复现的 。

您研究了一下，不禁问自己哪里出错了？这个逻辑很简单:

1. 开始执行用户请求的排序操作；
2. 在 Room 调度器中开始进行排序；
3. 展示排序结果。

您觉得这个 bug 不存在准备关闭它，因为解决方案很简单，"**不要那么快地点击按钮**"，但是您还是很担心，觉得还是哪个地方出了问题。于是在代码中加入一些日志，并跑了一堆测试用例后，您终于知道问题出在什么地方了!

看起来应用内展示的排序结果并不是真正的 "排序结果"，而是上一次完成排序的结果。当用户快速点击按钮时，就会同时触发多个排序操作，这些操作可能以任意顺序结束。

当启动一个新的协程来响应 UI 事件时，要去考虑一下用户若在上一个任务未完成之前又开始了新的任务，会有什么样的后果。

这其实是一个并发导致的问题，它和是否使用了协程其实没有什么关系。如果您使用回调、Rx 或者是 ExecutorService，还是可能会遇到这样的 bug。

有非常多方案能够解决这个问题，既可以在 ViewModel 中解决，又可以在 Repository 中解决。我们来看看怎么才能让一次性请求按照我们所期望的顺序返回结果。

最佳解决方案: 禁用按钮

核心问题出在我们做了两次排序，要修复的话我们可以只让它排序一次。最简单的解决方法就是禁用按钮，不让它发出新的事件就可以了。

这看起来很简单，而且确实是个好办法。实现起来的代码也很简单，还容易测试，只要它能在 UI 中体现出来这个按钮的状态，就完全可以解决问题。

要禁用按钮，只需要告诉 UI 在 sortPricesBy 中是否有正在处理的排序请求，示例代码如下:

```
// 方案 0: 当有任何排序正在执行时，禁用排序按钮

class ProductsViewModel(
    val productsRepository: ProductsRepository): ViewModel() {
```

```
    private val _sortedProducts = MutableLiveData<List<ProductListing>>()
    val sortedProducts: LiveData<List<ProductListing>> = _sortedProducts

    private val _sortButtonsEnabled = MutableLiveData<Boolean>()
    val sortButtonsEnabled: LiveData<Boolean> = _sortButtonsEnabled

    init {
        _sortButtonsEnabled.value = true
    }

    /**
        当用户点击排序按钮时，调用
     */
    fun onSortAscending() = sortPricesBy(ascending = true)
    fun onSortDescending() = sortPricesBy(ascending = false)

    private fun sortPricesBy(ascending: Boolean) {
        viewModelScope.launch {
           // 只要有排序在进行，禁用按钮
            _sortButtonsEnabled.value = false
            try {
                _sortedProducts.value =
                        productsRepository.loadSortedProducts(ascending)
            } finally {
               // 排序结束后，启用按钮
                _sortButtonsEnabled.value = true
            }
        }
    }
}
```

> *使用 sortPricesBy 中的 _sortButtonsEnabled 在排序时禁用按钮*

好了，这看起来还行，只需要在调用 repository 时在 sortPricesBy 内部禁用按钮就好了。

大部分情况下，这都是最佳解决方案，但是如果我们想在保持按钮可用的前提下解决 bug 呢？这样的话有一点困难，在本文剩余的部分看看该怎么做。

注意: 这段代码展示了从主线程启动的巨大优势，点击之后按钮立刻变得不可点了。但如果您换用了其他的调度程序，当出现某个手速很快的用户在运行速度较慢的手机上操作时，还是可能出现发送多次点击事件的情况。

并发模式

下面几个章节我们探讨一些比较高级的话题，如果您才刚刚接触协程，可以不去理解这一部分，使用禁用按钮这一方案就是解决大部分类似问题的最佳方案。

在剩余部分我们将探索在不禁用按钮的前提下，确保一次性请求能够正常运行。我们可以通过控制何时让协程运行 (或者不运行) 来避免刚刚出现的并发问题。

有三个基本的模式可以让我们确保在同一时间只会有一次请求进行:

1. 在启动更多协程之前**取消之前的任务**；
2. 让**下一个任务排队**等待前一个任务执行完成；
3. 如果有一个任务正在执行，**返回该任务**，而不是启动一个新的任务。

当介绍完这三个方案后，您可能会发现它们的实现都挺复杂的。为了专注于设计模式而不是实现细节，我创建了一个 gist 来提供这三个模式的实现作为可重用抽象 。

方案 1: 取消之前的任务

在排序这种情况下，获取新的事件后就意味着可以取消上一个排序任务了。毕竟用户通过这样的行为已经表明了他们不想要上次的排序结果了，继续进行上一次排序操作没什么意义了。

要取消上一个请求，我们首先要以某种方式追踪它。在gist中的 cancelPreviousThenRun 函数就做到了这个。

来看看如何使用它修复这个 bug:

```
// 方案 1: 取消之前的任务

// 对于排序和过滤的情况，新请求进来，取消上一个，这样的方案是很适合的。

class ProductsRepository(val productsDao: ProductsDao,
    val productsApi: ProductsService) {
   var controlledRunner = ControlledRunner<List<ProductListing>>()

   suspend fun loadSortedProducts(ascending: Boolean): List<ProductListing> {
      // 在开启新的排序之前，先取消上一个排序任务
       return controlledRunner.cancelPreviousThenRun {
           if (ascending) {
               productsDao.loadProductsByDateStockedAscending()
           } else {
               productsDao.loadProductsByDateStockedDescending()
           }
       }
   }
}
```

> 使用 cancelPreviousThenRun 来确保同一时间只有一个排序任务在进行

看一下 gist 中 cancelPreviousThenRun 中的代码实现，您可以学习到如何追踪正在工作的任务。

```
// see the complete implementation at
// 在 https://gist.github.com/objcode/7ab4e7b1df8acd88696cb0ccecad16f7
// 中查看完整实现
suspend fun cancelPreviousThenRun(block: suspend () -> T): T {
    // 如果这是一个 activeTask，取消它，因为它的结果已经不需要了
    activeTask?.cancelAndJoin()

    // ...
```

简而言之，它会通过成员变量 activeTask 来保持对当前排序的追踪。无论何时开始一个新的排序，都立即对当前 activeTask 中的所有任务执行cancelAndJoin 操作。这样会在开启一次新的排序之前就会把正在进行中的排序任务给取消掉。

使用类似于 ControlledRunner<T> 这样的抽象实现来对逻辑进行封装是比较好的方法，比直接混杂并发与应用逻辑要好很多。

选择使用抽象来封装代码逻辑，避免混杂并发和应用逻辑代码。

注意: 这个模式不适合在全局单例中使用，因为不相关的调用方是不应该相互取消。

方案 2: 让下一个任务排队等待

这里有一个对并发问题总是有效的解决方案。

让任务去排队等待依次执行，这样同一时间就只会有一个任务会被处理。就像在商场里进行排队，请求将会按照它们排队的顺序来依次处理。

对于这种特定的排序问题，其实选择方案 1 比使用本方案要更好一些，但还是值得介绍一下这个方法，因为它总是能够有效的解决并发问题。

```
// 方案 2: 使用互斥锁
// 注意: 这个方法对于排序或者是过滤来说并不是一个很好的解决方案，
// 但是它对于解决网络请求引起的并发问题非常适合。

class ProductsRepository(val productsDao: ProductsDao,
    val productsApi: ProductsService) {
    val singleRunner = SingleRunner()

    suspend fun loadSortedProducts(ascending: Boolean): List<ProductListing> {
        // 开始新的任务之前，等待之前的排序任务完成
        return singleRunner.afterPrevious {
            if (ascending) {
                productsDao.loadProductsByDateStockedAscending()
            } else {
                productsDao.loadProductsByDateStockedDescending()
            }
```

```
        }
    }
}
```

无论何时进行一次新的排序， 都使用一个 SingleRunner 实例来确保同时只会有一个排序任务在进行。

它使用了Mutex，可以把它理解为一张单程票 (或是锁)，协程在必须要获取锁才能进入代码块。如果一个协程在运行时，另一个协程尝试进入该代码块就必须挂起自己，直到所有的持有 Mutex 的协程完成任务，并释放 Mutex 后才能进入。

Mutex 保证同时只会有一个协程运行，并且会按照启动的顺序依次结束。

方案 3: 复用前一个任务

第三种可以考虑的方案是复用前一个任务，也就是说新的请求可以重复使用之前存在的任务，比如前一个任务已经完成了一半进来了一个新的请求，那么这个请求直接重用这个已经完成了一半的任务，就省事很多。

但其实这种方法对于排序来说并没有多大意义，但是如果是一个网络数据请求的话，就很适用了。

对于我们的库存应用来说，用户需要一种方式来从服务器获取最新的商品库存数据。我们提供了一个刷新按钮这样的简单操作来让用户点击一次就可以发起一次新的网络请求。

当请求正在进行时，禁用按钮就可以简单地解决问题。但是如果我们不想这样，或者说不能这样，我们就可以选择这种方法复用已经存在的请求。

查看下面的来自 gist 的使用了joinPreviousOrRun的示例代码:

```
class ProductsRepository(val productsDao: ProductsDao,
    val productsApi: ProductsService) {
   var controlledRunner = ControlledRunner<List<ProductListing>>()

   suspend fun fetchProductsFromBackend(): List<ProductListing> {
      // 如果已经有一个正在运行的请求，那么就返回它。如果没有的话，开启一个新的请求。
       return controlledRunner.joinPreviousOrRun {
           val result = productsApi.getProducts()
           productsDao.insertAll(result)
           result
       }
   }
}
```

上面的代码行为同 cancelPreviousAndRun 相反，它会直接使用之前的请求而放弃新的请求，而 cancelPreviousAndRun 则会放弃之前的请求而创建一个新的请求。如果已经存在了正在运行的请求，它会等待这个请求执行完成，并将结果直接返回。只有不存在正在运行的请求时才会创建新的请求来执行代码块。

您可以在 joinPreviousOrRun 开始时看到它是如何工作的，如果 activeTask 中存在任何正在工作的任务，就直接返回它。

```
// 在 https://gist.github.com/objcode/7ab4e7b1df8acd88696cb0ccecad16f7
//#file-concurrencyhelpers-kt-L124 中查看完整实现

suspend fun joinPreviousOrRun(block: suspend () -> T): T {
   // 如果存在 activeTask，直接返回它的结果，并不会执行代码块
    activeTask?.let {
        return it.await()
    }
    // ...
```

这个模式很适合那种通过 id 来查询商品数据的请求。您可以使用 map 来建立 id 到 Deferred 的映射关系，然后使用相同的逻辑来追踪同一个产品之前的请求数据。

直接复用之前的任务可以有效避免重复的网络请求。

下一步

在这篇文章中，我们探讨了如何使用 Kotlin 协程来实现一次性请求。我们实现了如何在 ViewModel 中启动协程，然后在 Repository 和 Room Dao 中提供公开的 suspend function，这样形成了一个完整的编程范式。

对于大部分任务来说，在 Android 上使用 Kotlin 协程按照上面这些方法就已经足够了。这些方法就像上面所说的排序一样可以应用在很多场景中，您也可以使用这些方法来解决查询、保存、更新网络数据等问题。

然后我们探讨了一下可能出现 bug 的地方，并给出了解决方案。最简单 (往往也是最好的) 的方案就是从 UI 上直接更改，排序运行时直接禁用按钮。

最后，我们探讨了一些高级并发模式，并介绍了如何在 Kotlin 协程中实现它们。虽然这些代码有点复杂，但是为一些高级协程方面的话题做了很好的介绍。

在下一篇文章中，我们将会研究一下流式请求，并探索如何使用 liveData 构造器，感兴趣的读者请继续关注我们的更新。

发布于 05-08