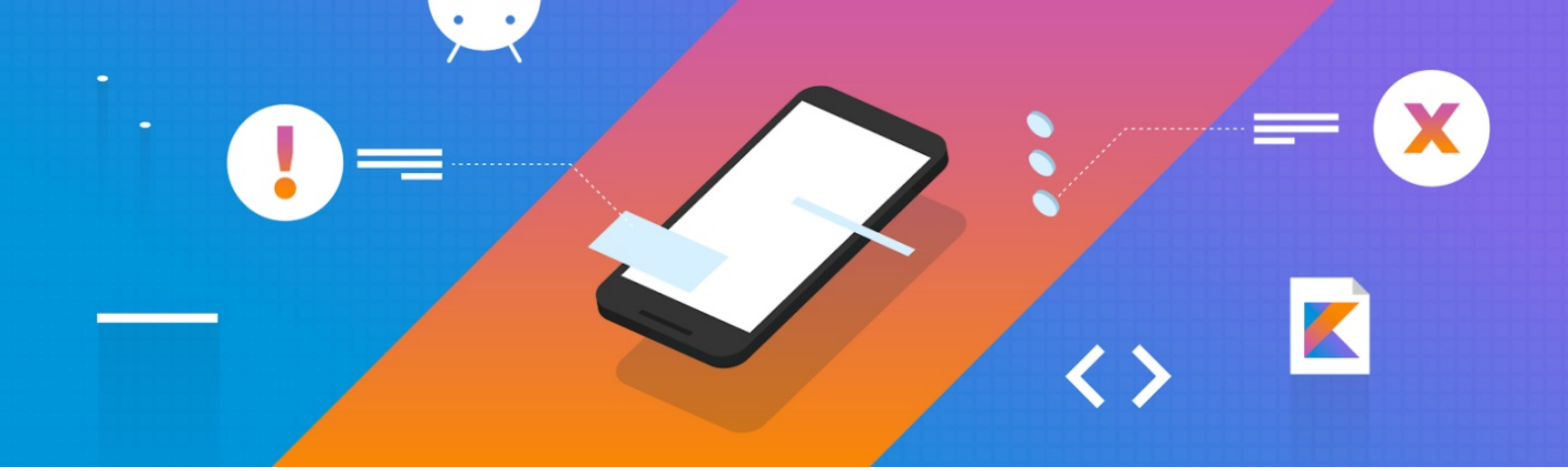

# 协程中的取消和异常 | 驻留任务详解

谷歌开发者
已认证的官方帐号

已关注

2 人赞同了该文章

在本系列第二篇文章 **协程中的取消和异常 | 取消操作详解** 中，我们学到，当一个任务不再被需要时，正确地退出十分的重要。在 Android 中，您可以使用 Jetpack 提供的两个 CoroutineScopes: **viewModelScope** 和 **lifecycleScope**，它们可以在 Activity、Fragment、Lifecycle 完成时退出正在运行的任务。如果您正在创建自己的 CoroutineScope，记得将它绑定到某个任务中，并在需要的时候取消它。

然而，在有些情况下，您会希望即使用户离开了当前界面，操作依然能够执行完成。因此，您就不会希望任务被取消，例如，向数据库写入数据或者向您的服务器发送特定类型的请求。

下面我们就来介绍实现此类情况的模式。

## 协程还是 WorkManager?

协程会在您的应用进程活动期间执行。如果您需要执行一个能够在应用进程之外活跃的操作 (比如向远程服务器发送日志)，在 Android 平台上建议使用 **WorkManager**。WorkManager 是一个扩展库，用于那些预期会在将来的某个时间点执行的重要操作。

请针对那些在当前进程中有效的操作使用协程，同时保证可以在用户关闭应用时取消操作 (例如，进行一个您希望缓存的网络请求)。那么，实现这类操作的最佳实践是什么呢?

## 协程的最佳实践

由于本文所介绍的模式是在协程的其它最佳实践的基础之上实现的，我们可以借此机会回顾一下:

### 1. 将调度器注入到类中

不要在创建协程或调用 withContext 时硬编码调度器。

✅ **好处**: 便于测试。您可以在进行单元测试或仪器测试时轻松替换掉它们。

### 2. 应当在 ViewModel 或 Presenter 层创建协程

如果是仅与 UI 相关的操作，则可以在 UI 层执行。如果您认为这条最佳实践在您的工程中不可行，则很有可能是您没有遵循第一条最佳实践 (测试没有注入调度器的 ViewModel 会变得更加困难；这种情况下，暴露出挂起函数会使测试变得可行)。

✅ **好处**: UI 层应该尽量简洁，并且不直接触发任何业务逻辑。作为代替，应当将响应能力转移到 ViewModel 或 Presenter 层实现。在 Android 中，测试 UI 层需要执行插桩测试，而执行插桩测试需要运行一个模拟器。

### 3. ViewModel 或 Presenter 以下的层级，应当暴露挂起函数与 Flow

如果您需要创建协程，请使用 **coroutineScope** 或 **supervisorScope**。而如果您想要将协程限定在其他作用域，请继续阅读，接下来本文将对此进行讨论。

✅ **好处**: 调用者 (通常是 ViewModel 层) 可以控制这些层级中任务的执行和生命周期，也可以在需要时取消这些任务。

## 协程中那些不应当被取消的操作

假设我们的应用中有一个 ViewModel 和一个 Repository，它们的相关逻辑如下:

```
class MyViewModel(private val repo: Repository) : ViewModel() {
  fun callRepo() {
    viewModelScope.launch {
      repo.doWork()
    }
  }
}

class Repository(private val ioDispatcher: CoroutineDispatcher) {
  suspend fun doWork() {
    withContext(ioDispatcher) {
      doSomeOtherWork()
     veryImportantOperation() // 它不应当被取消
    }
  }
}
```

我们不希望用 viewModelScope 来控制 veryImportantOperation()，因为 viewModelScope 随时都可能被取消。我们想要此操作的运行时长超过 viewModelScope，这个目的要如何达成呢?

**我们需要在 Application 类中创建自己的作用域，并在由它启动的协程中调用这些操作**。这个作用域应当被注入到那些需要它的类中。

与稍后将在本文中看到的其他解决方案 (如 GlobalScope) 相比，创建自己的 CoroutineScope 的好处是您可以根据自己的想法对其进行配置。无论您是需要 **CoroutineExceptionHandler**，还是想使用自己的线程池作为调度器，这些常见的配置都可以放在自己的 CoroutineScope 的 CoroutineContext 中。

您可以称其为 applicationScope。applicationScope 必须包含一个 SupervisorJob()，这样协程中的故障便不会在层级间传播 (见本系列第三篇文章: **协程中的取消和异常 | 异常处理详解**):

```
class MyApplication : Application() {
  // 不需要取消这个作用域，因为它会随着进程结束而结束
   val applicationScope = CoroutineScope(SupervisorJob() + otherConfig)
}
```

由于我们希望它在应用进程存活期间始终保持活动状态，所以我们不需要取消 applicationScope，进而也不需要保持 SupervisorJob 的引用。当协程所需的生存期比调用处作用域的生存期更长时，我们可以使用 applicationScope 来运行协程。

**从 application CoroutineScope 创建的协程中调用那些不应当被取消的操作**

**每当您创建一个新的 Repository 实例时，请传入上面创建的 applicationScope**。对于测试，可以参考后文的 Testing 部分。

### 应该使用哪种协程构造器?

您需要基于 veryImportantOperation 的行为来使用 launch 或 async 启动新的协程:

- 如果需要返回结果，请使用 async 并调用 await 来等待其完成；
- 如果不是，请使用 launch 并调用 join 来等待其完成。请注意，如 **本系列第三部分所述**，您必须在 launch 块内部手动处理异常。

下面是使用 launch 启动协程的方式:

```
class Repository(
  private val externalScope: CoroutineScope,
  private val ioDispatcher: CoroutineDispatcher
) {
  suspend fun doWork() {
    withContext(ioDispatcher) {
      doSomeOtherWork()
      externalScope.launch {
        //如果这里会抛出异常，那么要将其包裹进 try/catch 中;
```

```
        //或者依赖 externalScope 的 CoroutineScope 中的
        //CoroutineExceptionHandler
        veryImportantOperation()
      }.join()
    }
  }
}
```

或使用 async:

```
class Repository(
  private val externalScope: CoroutineScope,
  private val ioDispatcher: CoroutineDispatcher
) {
  suspend fun doWork(): Any { // 在结果中使用特定类型
    withContext(ioDispatcher) {
      doSomeOtherWork()
      return externalScope.async {
        // 异常会在调用 await 时暴露，它们会在调用了 doWork 的协程中传播。
        // 注意，如果正在调用的上下文被取消，那么异常将会被忽略。
        veryImportantOperation()
    }.await()
    }
  }
}
```

在任何情况下，都无需改动上面的 ViewModel 的代码。就算 ViewModelScope 被销毁，使用 externalScope 的任务也会持续运行。就像其他挂起函数一样，只有在 veryImportantOperation() 完成之后，doWork() 才会返回。

**有没有更简单的解决方案呢?**

另一种可以在一些用例中使用的方案 (可能是任何人都会首先想到的方案)，便是将 veryImportantOperation 像下面这样用 withContext 封装进 externalScope 的上下文中:

```
class Repository(
  private val externalScope: CoroutineScope,
  private val ioDispatcher: CoroutineDispatcher
) {
  suspend fun doWork() {
    withContext(ioDispatcher) {
      doSomeOtherWork()
      withContext(externalScope.coroutineContext) {
        veryImportantOperation()
      }
```

```
    }
  }
}
```

但是，此方法有下面几个注意事项，使用的时候需要注意:

- 如果调用 doWork() 的协程在 veryImportantOperation 开始执行时被退出，它将继续执行直到下一个退出节点，而不是在 veryImportantOperation 结束后退出；
- CoroutineExceptionHandler 不会如您预期般工作，这是因为在 withContext 中使用上下文时，异常会被重新抛出。

## 测试

由于我们可能需要同时注入调度器和 CoroutineScop，那么这些场景里分别需要注入什么呢?

| | 单元测试 | 插桩测试 |
|---|---|---|
| 调度器 | TestCoroutineDispatcher<br>考虑使用<br>MainCoroutineRule | AsyncTask.THREAD_POOL_EXECUTOR.<br>asCoroutineDispatcher()<br>可以让 Espresso 自动接管工作 |
| 作用域 | TestCoroutineScope | TestCoroutineScope |

测试时要注入什么

说明文档:

- TestCoroutineDispatcher
- MainCoroutineRule
- TestCoroutineScope
- AsyncTask.THREAD_POOL_EXECUTOR.asCoroutineDispatcher()

## 替代方案

其实还有一些其他的方式可以让我们使用协程来实现这一行为。不过，这些解决方案不是在任何条件下都能有条理地实现。下面就让我们看看一些替代方案，以及为何适用或者不适用，何时使用或者不使用它们。

## ❌ GlobalScope

下面是几个不应该使用 GlobalScope 的理由:

- **诱导我们写出硬编码值** 。直接使用 GlobalScope 可能会让我们倾向于写出硬编码的调度器，这是一种很差的实践方式。
- **导致测试非常困难** 。由于您的代码会在一个不受控制的作用域中执行，您将无法对从中启动的任务进行管理。
- 就如同我们对 applicationScope 所做的那样，**您无法为所有协程都提供一个通用的、内建于作用域中的 CoroutineContext**。相反，您必须传递一个通用的 CoroutineContext 给 GlobalScope 启动的所有协程。

**建议: 不要直接使用它。**

## ❌ Android 中的 ProcessLifecycleOwner 作用域

在 Android 中的 androidx.lifecycle:lifecycle-process 库中，有一个 applicationScope，您可以使用 ProcessLifecycleOwner.get().lifecycleScope 来调用它。

在使用它时，您需要注入一个 LifecycleOwner 来代替我们之前注入的 CoroutineScope。在生产环境中，您需要传入 ProcessLifecycleOwner.get()；而在单元测试中，您可以用 LifecycleRegistry 来创建一个虚拟的 LifecycleOwner。 注意，这个作用域的默认 CoroutineContext 是 Dispatchers.Main.immediate，所以它可能不太适合去执行后台任务。就像使用 GlobalScope 时那样，您也需要传递一个通用的 CoroutineContext 到所有通过 GlobalScope 启动的协程中。

由于上述原因，此替代方案相比起直接在 Application 类中创建一个 CoroutineScope 要麻烦许多。而且，我个人不喜欢在 ViewModel 或 Presenter 层之下与 Android lifecycle 建立关系，我希望这些层级是平台无关的。

**建议: 不要直接使用它。**

### ⚠ 特别说明**

如果您将您的 applicationScope 中的 CoroutineContext 等于 GlobalScope 或 ProcessLifecycleOwner.get().lifecycleScope，您就可以像下面这样直接使用它:

```
class MyApplication : Application() {
  val applicationScope = GlobalScope
}
```

您仍然可以获得上文所述的所有**优点**，并且将来可以根据需要轻松进行更改。

## ❌✅ 使用 NonCancellable

正如您在本系列第二篇文章 **协程中的取消和异常 | 取消操作详解** 中看到的，您可以使用 withContext(NonCancellable) 在被取消的协程中调用挂起函数。我们建议您使用它来进行可挂起的代码清理，但是，您不应该滥用它。

这样做的风险很高，因为您将会无法控制协程的执行。确实，它可以使代码更简洁，可读性更强，但与此同时，它也可能在将来引起一些无法预测的问题。

使用示例如下:

```
class Repository(
  private val ioDispatcher: CoroutineDispatcher
) {
  suspend fun doWork() {
    withContext(ioDispatcher) {
      doSomeOtherWork()
    withContext(NonCancellable){
        veryImportantOperation()
      }
    }
  }
}
```

尽管这个方案很有诱惑力，但是您可能无法总是知道 someImportantOperation() 背后有什么逻辑。它可能是一个扩展库；也可能是一个接口背后的实现。它可能会导致各种各样的问题:

- 您将无法在测试中结束这些操作；
- 使用延迟的无限循环将永远无法被取消；
- 从其中收集 Flow 会导致 Flow 也变得无法从外部取消；
- ……

而这些问题会导致出现细微且非常难以调试的错误。

**建议: 仅用它来挂起清理操作相关的代码。**

每当您需要执行一些超出当前作用域范围的工作时，我们都建议您在您自己的 Application 类中创建一个自定义作用域，并在此作用域中执行协程。同时要注意，在执行这类任务时，避免使用 GlobalScope、ProcessLifecycleOwner 作用域或 NonCancellable。

发布于 10-15

Android　Kotlin　协程