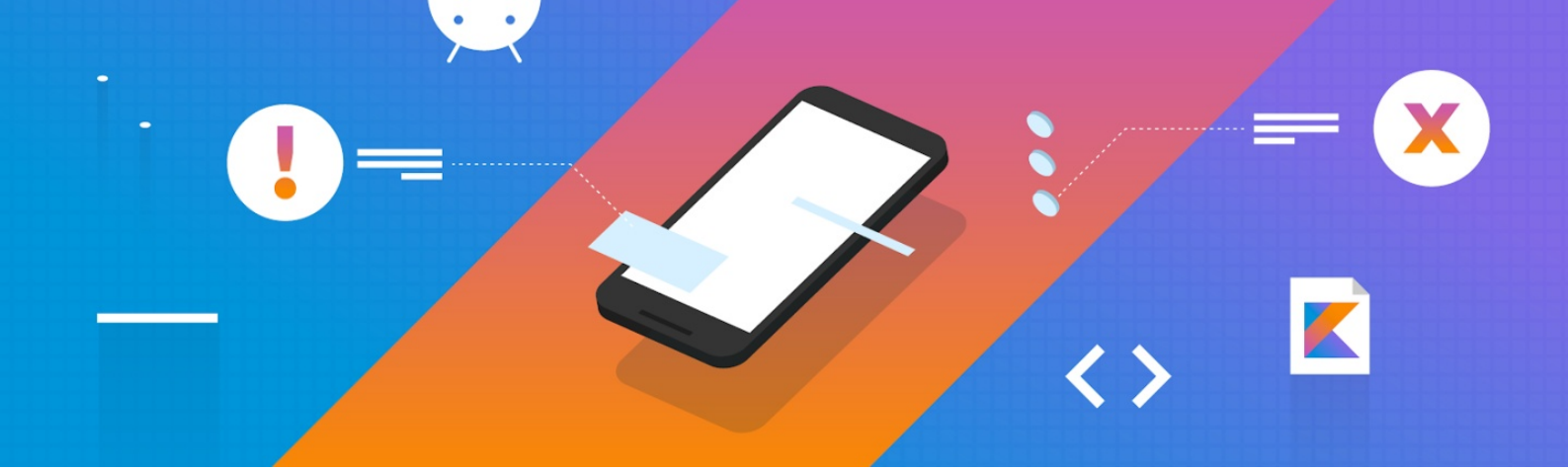

协程中的取消和异常 | 取消操作详解

谷歌开发者
已认证的官方帐号

已关注

高爷等 7 人赞同了该文章

在日常的开发中，我们都知道应该避免不必要的任务处理来节省设备的内存空间和电量的使用——这一原则在协程中同样适用。您需要控制好协程的生命周期，在不需要使用的时候将它取消，这也是结构化并发所倡导的，继续阅读本文来了解有关协程取消的来龙去脉。

⚠ 为了能够更好地理解本文所讲的内容，建议您首先阅读本系列中的第一篇文章: **协程中的取消和异常 | 核心概念介绍**。

调用 cancel 方法

当启动多个协程时，无论是追踪协程状态，还是单独取消各个协程，都是件让人头疼的事情。不过，我们可以通过直接取消协程启动所涉及的整个作用域 (scope) 来解决这个问题，因为这样可以取消所有已创建的子协程。

```
// 假设我们已经定义了一个作用域

val job1 = scope.launch { … }
val job2 = scope.launch { … }

scope.cancel()
```

取消作用域会取消它的子协程

有时候，您也许仅仅需要取消其中某一个协程，比如用户输入了某个事件，作为回应要取消某个进行中的任务。如下代码所示，调用 job1.cancel 会确保只会取消跟 job1 相关的特定协程，而不会影响其余兄弟协程继续工作。

```
// 假设我们已经定义了一个作用域
```

```
val job1 = scope.launch { … }
val job2 = scope.launch { … }

// 第一个协程将会被取消，而另一个则不受任何影响
job1.cancel()
```

被取消的子协程并不会影响其余兄弟协程

协程通过抛出一个特殊的异常 CancellationException 来处理取消操作。在调用 .cancel 时您可以传入一个 CancellationException 实例来提供更多关于本次取消的详细信息，该方法的签名如下:

```
fun cancel(cause: CancellationException? = null)
```

如果您不构建新的 CancellationException 实例将其作为参数传入的话，会创建一个默认的 CancellationException (请查看 **完整代码**)。

```
public override fun cancel(cause: CancellationException?) {
    cancelInternal(cause ?: defaultCancellationException())
}
```

一旦抛出了 CancellationException 异常，您便可以使用这一机制来处理协程的取消。有关如何执行此操作的更多信息，请参考下面的处理取消的副作用一节。

在底层实现中，子协程会通过抛出异常的方式将取消的情况通知到它的父级。父协程通过传入的取消原因来决定是否来处理该异常。如果子协程因为 CancellationException 而被取消，对于它的父级来说是不需要进行其余额外操作的。

不能在已取消的作用域中再次启动新的协程

如果您使用的是 androidx KTX 库的话，在大部分情况下都不需要创建自己的作用域，所以也就不需要负责取消它们。如果您是在 ViewModel 的作用域中进行操作，请使用 **viewModelScope**，或者如果在生命周期相关的作用域中启动协程，那就应该使用 **lifecycleScope**。viewModelScope 和 lifecycleScope 都是 CoroutineScope 对象，它们都会在适当的时间点被取消。例如，当 ViewModel 被清除时，在其作用域内启动的协程也会被一起取消。

为什么协程处理的任务没有停止?

如果我们仅是调用了 cancel 方法，并不意味着协程所处理的任务也会停止。如果您使用协程处理了一些相对较为繁重的工作，比如读取多个文件，那么您的代码不会自动就停止此任务的进行。

让我们举一个更简单的例子看看会发生什么。假设我们需要使用协程来每秒打印两次 "Hello"。我们先让协程运行一秒，然后将其取消。其中一个版本实现如下所示:

```
import kotlinx.coroutines.*

fun main(args: Array<String>) = runBlocking<Unit> {
   val startTime = System.currentTimeMillis()
    val job = launch (Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (i < 5) {
            // print a message twice a second
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("Hello ${i++}")
                nextPrintTime += 500L
            }
        }
    }
    delay(1000L)
    println("Cancel!")
    job.cancel()
    println("Done!")
}
```

Target platform: JVM Running on kotlin v. 1.3.7

我们一步一步来看发生了什么。当调用 launch 方法时，我们创建了一个活跃 (active) 状态的协程。紧接着我们让协程运行了 1,000 毫秒，打印出来的结果如下:

```
Hello 0
Hello 1
Hello 2
```

当 job.cancel 方法被调用后，我们的协程转变为取消中 (cancelling) 的状态。但是紧接着我们发现 Hello 3 和 Hello 4 打印到了命令行中。当协程处理的任务结束后，协程又转变为了已取消 (cancelled) 状态。

协程所处理的任务不会仅仅在调用 cancel 方法时就停止，相反，我们需要修改代码来定期检查协程是否处于活跃状态。

让您的协程可以被取消

您需要确保所有使用协程处理任务的代码实现都是协作式的，也就是说它们都配合协程取消做了处理，因此您可以在任务处理期间定期检查协程是否已被取消，或者在处理耗时任务之前就检查当前协程是否已取消。例如，如果您从磁盘中获取了多个文件，在开始读取文件内容之前，先检查协程是否被取消了。类似这样的处理方式，您可以避免处理不必要的 CPU 密集型任务。

```
val job = launch {
    for(file in files) {
        // TODO 检查协程是否被取消
        readFile(file)
    }
}
```

所有 kotlinx.coroutines 中的挂起函数 (withContext, delay 等) 都是可取消的。如果您使用它们中的任一个函数，都不需要检查协程是否已取消，然后停止任务执行，或是抛出 CancellationException 异常。但是，如果没有使用这些函数，为了让您的代码能够配合协程取消，可以使用以下两种方法:

- 检查 job.isActive 或者使用 ensureActive()
- 使用 yield() 来让其他任务进行

检查 job 的活跃状态

先看一下第一种方法，在我们的 while(i<5) 循环中添加对于协程状态的检查:

```
// 因为处于 launch 的代码块中，可以访问到 job.isActive 属性
while (i < 5 && isActive)
```

这样意味着我们的任务只会在协程处于活跃的状态下执行。同样，这也意味着在 while 循环之外，我们若还想处理别的行为，比如在 job 被取消后打日志出来，那就可以检查 !isActive 然后再继续进行相应的处理。

Coroutine 的代码库中还提供了另一个很有用的方法 —— ensureActive()，它的实现如下:

```
fun Job.ensureActive(): Unit {
    if (!isActive) {
         throw getCancellationException()
    }
}
```

如果 job 处于非活跃状态，这个方法会立即抛出异常，我们可以在 while 循环开始就使用这个方法。

```
while (i < 5) {
    ensureActive()
    …
}
```

通过使用 ensureActive 方法，您可以避免使用 if 语句来检查 isActive 状态，这样可以减少样板代码的使用量，但是相应地也失去了处理类似于日志打印这种行为的灵活性。

使用 yield() 函数运行其他任务

如果要处理的任务属于 1) CPU 密集型，2) 可能会耗尽线程池资源，3) 需要在不向线程池中添加更多线程的前提下允许线程处理其他任务，那么请使用 **yield()**。如果 job 已经完成，由 yield 所处理

的首要任务将会是检查任务的完成状态，完成的话则直接通过抛出 CancellationException 来退出协程。yield 可以作为定期检查所调用的第一个函数，例如上面提到的 ensureActive() 方法。

Job.join Deferred.await cancellation**

等待协程处理结果有两种方法: 来自 launch 的 job 可以调用 join 方法，由 async 返回的 Deferred (其中一种 job 类型) 可以调用 await 方法。

Job.join 会挂起协程，直到任务处理完成。与 job.cancel 一起使用时，会按照以下方式进行:

- 如果您调用 job.cancel 之后再调用 job.join，那么协程会在任务处理完成之前一直处于挂起状态；
- 在 job.join 之后调用 job.cancel 没有什么影响，因为 job 已经完成了。

如果您关心协程处理结果，那么应该使用 **Deferred**。当协程完成后，结果会由 **Deferred.await** 返回。Deferred 是 Job 的其中一种类型，它同样可以被取消。

在已取消的 deferred 上调用 await 会抛出 JobCancellationException 异常。

```
val deferred = async { … }

deferred.cancel()
val result = deferred.await() // 抛出 JobCancellationException 异常
```

为什么会拿到这个异常呢？await 的角色是负责在协程处理结果出来之前一直将协程挂起，因为如果协程被取消了那么协程就不会继续进行计算，也就不会有结果产生。因此，在协程取消后调用 await 会抛出 JobCancellationException 异常: 因为 Job 已被取消。

另一方面，如果您在 deferred.cancel 之后调用 deferred.await 不会有任何情况发生，因为协程已经处理结束。

处理协程取消的副作用

假设您要在协程取消后执行某个特定的操作，比如关闭可能正在使用的资源，或者是针对取消需要进行日志打印，又或者是执行其余的一些清理代码。我们有好几种方法可以做到这一点:

检查 !isActive

如果您定期地进行 isActive 的检查，那么一旦您跳出 while 循环，就可以进行资源的清理。之前的代码可以更新至如下版本:

```
while (i < 5 && isActive) {
    if (…) {
        println(“Hello ${i++}”)
        nextPrintTime += 500L
    }
}

// 协程所处理的任务已经完成，因此我们可以做一些清理工作
println(“Clean up!”)
```

您可以查看 **完整版本**。

所以现在，当协程不再处于活跃状态，会退出 while 循环，就可以处理一些清理工作了。

Try catch finally

因为当协程被取消后会抛出 CancellationException 异常，我们可以将挂起的任务放置于 try/catch 代码块中，然后在 finally 代码块中执行需要做的清理任务。

```
val job = launch {
   try {
      work()
   } catch (e: CancellationException){
      println(“Work cancelled!”)
    } finally {
      println(“Clean up!”)
    }
}

delay(1000L)
println(“Cancel!”)
job.cancel()
println(“Done!”)
```

但是，一旦我们需要执行的清理工作也挂起了，那上述代码就不能够继续工作了，因为一旦协程处于取消中状态，它将不能再转为挂起 (suspend) 状态。您可以查看 **完整代码**。

处于取消中状态的协程不能够挂起

当协程被取消后需要调用挂起函数，我们需要将清理任务的代码放置于 NonCancellable CoroutineContext 中。这样会挂起运行中的代码，并保持协程的取消中状态直到任务处理完成。

```
val job = launch {
   try {
      work()
```

```
    } catch (e: CancellationException){
       println("Work cancelled!")
     } finally {
       withContext(NonCancellable){
          delay(1000L) // 或一些其他的挂起函数
          println("Cleanup done!")
       }
     }
}

delay(1000L)
println("Cancel!")
job.cancel()
println("Done!")
```

您可以查看其 **工作原理**。

suspendCancellableCoroutine 和 invokeOnCancellation

如果您通过 suspendCoroutine 方法将回调转为协程，那么您更应该使用 suspendCancellableCoroutine 方法。可以使用 continuation.invokeOnCancellation 来执行取消操作:

```
suspend fun work() {
    return suspendCancellableCoroutine { continuation ->
        continuation.invokeOnCancellation {
           // 处理清理工作
        }
    // 剩余的实现代码
}
```

为了享受到结构化并发带来的好处，并确保我们并没有进行多余的操作，那么需要保证代码是可被取消的。

使用在 Jetpack: viewModelScope 或者 lifecycleScope 中定义的 CoroutineScopes，它们在 scope 完成后就会取消它们处理的任务。如果要创建自己的 CoroutineScope，请确保将其与 job 绑定并在需要时调用 cancel。

协程代码的取消需要是协作式的，因此请将代码更新为对协程的取消操作以延后的方式进行检查，并避免不必要的操作。

现在，大家了解了本系列的第一部分 **协程的一些基本概念**、第二部分协程的取消，在接下来的文章中，我们将继续深入探讨学习第三部分异常处理，感兴趣的读者请继续关注我们的更新。

发布于 10-25

Android Android 开发 Kotlin