Kotlin 协程和 Android SQLite API 中的线程模型

谷歌开发者
已认证的官方帐号

已关注

9 人赞同了该文章

从 Room 2.1 版本之后，开发者们可以通过定义 suspend DAO 函数来使用 Kotlin 协程了。协程在处理异步操作时表现得异常优秀，它可以让您用顺序自然的代码处理诸如操作数据库一类的耗时操作，而不再需要专门在线程之间来回切换任务、处理结果或错误了。Room 支持协程后，可以在数据库操作中使用由并发作用域、生命周期、嵌套所带来的一些便利。

在我们为 Room 添加协程的支持期间，我们遇到并解决了在协程模型和 Android SQL API 中没想到的一些问题。在本篇文章中，我们会向您阐述我们遇到的这些问题，以及我们的解决方案。

意料外的问题

先看一下下面这段代码，看似是安全的，但实际上这段代码存在着问题:

```
/**
 * 将指定的 [amount] 的金额，从 [accountA] 转移到 [accountB]
 */
suspend fun transferMoney(accountA: String, accountB: String,
    amount: Int) {
   // 使用了 IO dispatcher，所以该 DB 的操作在 IO 线程上进行
    withContext(Dispatchers.IO) {
        database.beginTransaction() //在 IO-Thread-1 线程上开始执行事务
        try {
    // 协程可以在与调度器（这里就是 Dispatchers.IO）相关联的任何线程上绑定并继续执行。
    // 同时，由于事务也是在 IO-Thread-1 中开始的，因此我们可能恰好可以成功执行查询。
            moneyDao.decrease(accountA, amount) //挂起函数

    // 如果协程又继续在 IO-Thread-2 上执行，那么下列操作数据库的代码可能会引起死锁，
    // 因为它需要等到 IO-Thread-1 的线程执行结束后才可以继续。
            moneyDao.increase(accountB, amount) //挂起函数

            database.setTransactionSuccessful() //永远不会执行这一行
        } finally {
            database.endTransaction() //永远不会执行这一行
        }
    }
}
```

Android 的 SQLite 事务受制于单个线程

上述代码中的问题在于 Android 的 SQLite 事务是受制于单个线程的。当一个正在进行的事务中的某个查询在当前线程中被执行时，它会被视为是该事务的一部分并允许继续执行。但当这个查询在另外一个线程中被执行时，那它就不再属于这个事务的一部分了，这样的话就会导致这个查询被阻塞，直到事务在另外一个线程执行完成。这也是 beginTransaction 和 endTransaction 这两个 API 能够保证原子性的一个前提。当数据库的事务操作都是在一个线程上完成的，这样的 API 不会有任何问题，但是使用协程之后问题就来了，因为协程是不绑定在任何特定的线程上的。也就是说，问题的根源就是在协程挂起之后会继续执行所绑定的那个线程，而这样是不能保证和挂起之前所绑定的线程是同一个线程。

在协程中使用数据库事务操作可能会引起死锁

简单实现

为了解决 Android SQLite 的这个限制，我们需要一个类似于 runInTransaction 这样可以接受挂起代码块的 API，这个 API 实现起来就像写一个单线程的调度器一样:

```
suspend fun <T> RoomDatabase.runInTransaction(
    block: suspend () -> T
): T = withContext(newSingleThreadContext("DB")) {
    beginTransaction()
    try {
        val result = block.invoke(this)
        setTransactionSuccessful()
```

```
            return@runBlocking result
        } finally {
            endTransaction()
        }
    }
```

以上实现仅仅是个开始，但是当在挂起代码块中使用另一个调度器的话就会出问题了:

```
// 一个很简单的退税函数
suspend fun sendTaxRefund(federalAccount: String,
    taypayerList: List<Taxpayer>) {
    database.runInTransaction {
        val refundJobs = taypayerList.map { taxpayer ->
            coroutineScope {
             // 并行去计算退税金额
                async(Dispatchers.IO) {
                    val amount = irsTool.calculateRefund(taxpayer)
                    moneyDao.decrease(federalAccount, amount)
                    moneyDao.increase(taxpayer.account, amount)
                }
            }
        }
        // 等待所有计算任务结束
        refundJobs.joinAll()
    }
}
```

因为接收的参数是一个挂起代码块，所以这部分代码就有可能使用一个不同的调度器来启动子协程，这样就会导致执行数据库操作的是另外的一个线程。因此，一个比较好的实现是应该允许使用类似于 async、launch 或 withContext 这样的标准协程构造器。而在实际应用中，只有数据库操作才需要被调度到单事务线程。

介绍 withTransaction

为了解决上面的问题，我们构建了 **withTransaction API**，它模仿了 withContext API，但是提供了专为安全执行 Room 事务而构建的协程上下文，您可以按照如下方式编写代码:

```
fun transferMoney(
    accountA: String,
    accountB: String,
    amount: Int
) = GlobalScope.launch(Dispatchers.Main) {
    roomDatabase.withTransaction {
        moneyDao.decrease(accountA, amount)
        moneyDao.increase(accountB, amount)
```

```
    }
    Toast.makeText(context, "Transfer Completed.", Toast.LENGTH_SHORT).show()
}
```

在深入研究 Room withTransaction API 的实现前，让我们先回顾一下已提到的一些协程的概念。**CoroutineContext** 包含了需要对协程任务进行调度的信息，它携带了当前的 **CoroutineDispatcher** 和 **Job 对象**，以及一些额外的数据，当然也可以对它进行扩展来使其包含更多信息。CoroutineContext 的一个重要特征是它们被同一协程作用域下的子协程所继承，比如 withContext 代码块的作用域。这一机制能够让子协程继续使用同一个调度器，或在父协程被取消时，它们会被一起取消。本质上，Room 提供的挂起事务 API 会创建一个专门的协程上下文来在同一个事务作用域下执行数据库操作。

withTransaction API 在上下文中创建了三个关键元素:

- 单线程调度器，用于执行数据库操作；
- 上下文元素，帮助 DAO 函数判断其是否处在事务中；
- **ThreadContextElement**，用来标记事务协程中所使用的调度线程。

事务调度器

CoroutineDispatcher 会决定协程该绑定到哪个线程中执行。比如，**Dispatchers.IO** 会使用一个共享线程池分流执行那些会发生阻塞的操作，而 **Dispatchers.Main** 会在 Android 主线程中执行协程。由 Room 创建的事务调度器能够从 **Room 的 Executor** 获取单一线程，并将事务分发给该线程，而不是分发给一个随意创建的新线程。这一点很重要，因为 executor 可以由用户来配置，并且可作为测试工具使用。在事务开始时，Room 会获得 executor 中某个线程的控制权，直到事务结束。在事务执行期间，即使调度器因子协程发生了变化，已执行的数据库操作仍会被分配到该事务线程上。

获取一个事务线程并不是一个阻塞操作，它也不应该是阻塞操作，因为如果没有可用线程的话，应该执行挂起操作，然后通知调用方，避免影响其他协程的执行。它还会将一个 runnable 插入队列，然后等待其运行，这也是线程可运行的一个标志。**suspendCancellableCoroutine** 函数为我们搭建了连接基于回调的 API 和协程之间的桥梁。在这种情况下，一旦之前入队列的 runnable 执行了，就代表着一个线程可用，我们会使用 runBlocking 启动一个事件循环来获取此线程的控制权。然后 **runBlocking** 所创建的调度器会将要执行的代码块分发给已获得的线程。另外，Job 被用来挂起和保持线程的可用性，直到事务执行完成为止。要注意的是，一旦协程被取消了或者是无法获取到线程，就要有防范措施。获取事务线程的相关代码如下:

```
/**
 *构建并返回一个 [ContinuationInterceptor] 用来将协程分发到获取到的线程中，并执行事务。
 *[controlJob] 用来通过取消任务来控制线程的释放。
 */
private suspend fun Executor.acquireTransactionThread(
    controlJob: Job
): ContinuationInterceptor = suspendCancellableCoroutine { continuation ->
```

```
    continuation.invokeOnCancellation {
        // 当我们在等待获取到可用线程时，如果失败了或者任务取消，
        // 我们是不能够停止等待这一动作的，但我们可以取消 controlJob，
        // 这样一旦获取到控制权，很快就会被释放。
        controlJob.cancel()
    }
    try {
        execute {
            // runBlocking 创建一个 event loop 来执行协程中的任务代码
            runBlocking {
               // 获取到线程后，通过返回有 runBlocking 创建的拦截器来恢复
               // suspendCancellableCoroutine，
               // 拦截器将会被用来拦截和分发代码块到获取的线程中
                continuation.resume(
                        coroutineContext[ContinuationInterceptor]!!)

               // 挂起 runBlocking 协程，直到 controlJob 完成。
               // 由于协程是空的，所以这将会阻止 runBlocking 立即结束。
                controlJob.join()
            }
        }
    } catch (ex: RejectedExecutionException) {
       // 无法获取线程，取消协程
        continuation.cancel(
            IllegalStateException(
                "Unable to acquire a thread to perform the transaction.", ex)
        )
    }
}
```

事务上下文元素

有了调度器后，我们就可以创建事务中的元素来添加到上下文中，并保持着对调度器的引用。如果在事务作用域内调用了 DAO 函数，就可以把 DAO 函数重新路由到相应的线程中。我们创建的事务元素如下:

```
internal class TransactionElement(
    private val transactionThreadControlJob: Job,
    internal val transactionDispatcher: ContinuationInterceptor
) : CoroutineContext.Element {

   // Singleton key 用于检索此上下文中的 element
    companion object Key : CoroutineContext.Key<TransactionElement>

    override val key: CoroutineContext.Key<TransactionElement>
        get() = TransactionElement
```

```
        /**
         * 这个 element 用来统计事务数量（包含嵌套事务）。
         * 调用 [acquire] 来增加计数，调用 [release] 来减少计数。
         * 如果在调用 [release] 时计数达到 0，则事务被取消，事务线程会被释放
         */
        private val referenceCount = AtomicInteger(0)

        fun acquire() {
            referenceCount.incrementAndGet()
        }

        fun release() {
            val count = referenceCount.decrementAndGet()
            if (count < 0) {
                throw IllegalStateException(
                    "Transaction was never started or was already released.")
            } else if (count == 0) {
               // 取消控制事务线程的 job 会导致它被 release
                transactionThreadControlJob.cancel()
            }
        }
    }
```

TransactionElement 函数中的 acquire 和 release 是用来跟踪嵌套事务的。由于 beginTransaction 和 endTransaction 允许嵌套调用，我们也想保留这个特性，但是我们只需要在最外层事务完成时释放事务线程即可。这些功能的用法在稍后的 withTransaction 实现中会介绍。

事务线程标记

上文中提到的创建事务上下文中所需的最后一个关键元素是 **ThreadContextElement**。CoroutineContext 中的这个元素类似于 **ThreadLocal**，它能够跟踪线程中是否有正在进行的事务。这个 element 是由一个 ThreadLocal 支持，对于调度器所用的每个线程，它都会在 ThreadLocal 上设置一个值来执行协程代码块。线程一旦完成任务后，这个值会被重置。在我们的例子中，这个值是没有意义的，在 Room 中也只需要确定这个值是否存在即可。如果协程上下文可以访问平台中存在的 **ThreadLocal**，则可以从协程所绑定的任何线程向其分发 begin/ends 命令，如果做不到，那在事务完成前只能阻塞线程。但我们仍然需要追踪每个阻塞的数据库方法是在哪个事务上运行，以及哪个线程负责平台事务。

Room 的 withTransaction API 中使用的 ThreadContextElement 会标识数据库中的阻塞函数。Room 中的阻塞函数，包含 DAO 生成的那些，在它们被事务协程调用后会被特殊处理，用来保证它们不会在其他的调度器上运行。如果您的 DAO 同时具有这两种类型的功能，则可以在 withTransaction 块中将阻塞函数与挂起函数混合和匹配。通过将 ThreadContextElement 添加到协程上下文中，并从 DAO 函数中访问它，我们可以验证阻塞函数是否处于正确的作用域中。如果

不是， **我们会抛出异常而不是造成死锁** 。在之后，我们计划将阻塞函数也重新路由到事务线程中。

```
private final ThreadLocal<Integer> mSuspendingTransactionId =
    new ThreadLocal<>();
public void assertNotSuspendingTransaction() {
    if (!inTransaction() && mSuspendingTransactionId.get() != null) {
        throw new IllegalStateException(
    "Cannot access database on a different"
            + " coroutine context inherited from a suspending transaction.");
    }
}
```

这三个元素的组合构成了我们的事务上下文:

```
private suspend
fun RoomDatabase.createTransactionContext(): CoroutineContext {
    val controlJob = Job()
    val dispatcher = queryExecutor.acquireTransactionThread(controlJob)
    val transactionElement = TransactionElement(controlJob, dispatcher)
    val threadLocalElement =
        suspendingTransactionId.asContextElement(controlJob.hashCode())
    return dispatcher + transactionElement + threadLocalElement
}
```

事务 API 的实现

创建了事务上下文之后，我们终于可以提供一个安全的 API 用于在协程中执行数据库事务。接下来要做的就是将这个上下文和通常的 begin/end 事务模式结合起来:

```
suspend fun <R> RoomDatabase.withTransaction(
    block: suspend () -> R
): R {
    // 如果可以的话就使用继承的事务上下文，这样允许嵌套挂起的事务
    val transactionContext =
        coroutineContext[TransactionElement]?.transactionDispatcher
            ?: createTransactionContext()
    return withContext(transactionContext) {
        val transactionElement = coroutineContext[TransactionElement]!!
        transactionElement.acquire()
        try {
            beginTransaction()
            try {
                // 在一个新的 scope 中封装 suspend 代码块，来等待子协程
```

```
                val result = coroutineScope {
                    block.invoke(this)
                }
                setTransactionSuccessful()
                return@withContext result
            } finally {
                endTransaction()
            }
        } finally {
            transactionElement.release()
        }
    }
}
```

Android 中 SQLite 的线程限制是合理的，这在 Kotlin 还没出现时已然如此设计了。协程引入了新的编程范式，改变了传统 Java 并发编程的一些思维模式。直接取消 Android 线程对 SQLite 事务的限制是不可行的，因为我们希望提供一个向后兼容的解决方案，而上述这些方法的组合最终让我们在使用协程和 Fluent API 的解决方案中发挥了创造性。

发布于 09-17

Android Kotlin SQLite