Kotlin 协程并发问题：Mutex 我用错了吗?

AndroidPub 2023-06-12 08:18 发表于北京

收录于合集
#协程 6 #面试 15 #多线程 1

作者：equationl
juejin.cn/post/7219522025198895159

前言

最近在接手的某项目中，主管给我发来了一个遗留以久的 BUG，让我看看排查一下，把它修复了。项目的问题大概是在某项业务中，需要向数据库插入数据，而且需要保证同种类型的数据只被插入一次，但是现在却出现了数据被重复插入的情况。

我点开代码一看，上一个跑路的老哥写的非常谨慎啊，判断重复的逻辑嵌套了一层又一层，先在本地数据库查询一次没有重复后又请求服务器查询一次，最后在插入前再查询本地数据库一次。总共写了三层判重逻辑。但是为什么还是重复了呢？

再细看，哦，原来是用了协程异步查询啊，怪不得。可是，不对啊，你不是用 Mutex 上锁了吗？怎么还会重复？Mutex 你在干什么？你锁了什么？你看看你都守护了什么啊。

此时的 Mutex 就像我一般，什么都守护不住。

但是，真的怪 Mutex 吗？这篇文章我们就来浅析一下使用 Mutex 实现协程的并发可能导致失效的问题，为我们老实本份的 Mutex 洗清冤屈。

前置知识：关于协程和并发

众所周知，对于多线程程序，可能会出现同步问题，例如，下面这个经典的例子：

```
fun main() {
    var count = 0

    runBlocking {
```

```
        repeat(1000) {
            launch(Dispatchers.IO) {
                count++
            }
        }
    }

    println(count)
}
```

你们说，以上代码会输出什么？

我不知道，我也没法知道，没错，确实是这样的。因为在上面的代码中，我们循环 1000 次，每次都启动一个新的协程，然后在协程中对 count 进行自增操作。

问题就在于，我们没法保证对 count 的操作是同步的，因为我们不知道这些协程何时会被执行，也无法保证这些协程在执行时 count 的值没有被其他协程修改过。

这就导致，count 值最终会是不确定的。

另一个众所周知，kotlin 中的协程其实可以简单理解成对线程的封装，所以实际上不同的协程可能运行在同一个线程也可能运行在不同的线程。

我们给上面的代码加一个打印所在线程：

```
fun main() {
    var count = 0

    runBlocking {
        repeat(1000) {
            launch(Dispatchers.IO) {
                println("Running on ${Thread.currentThread().name}")
                count++
            }
        }
    }

    println(count)
}
```

截取其中一部分输出：

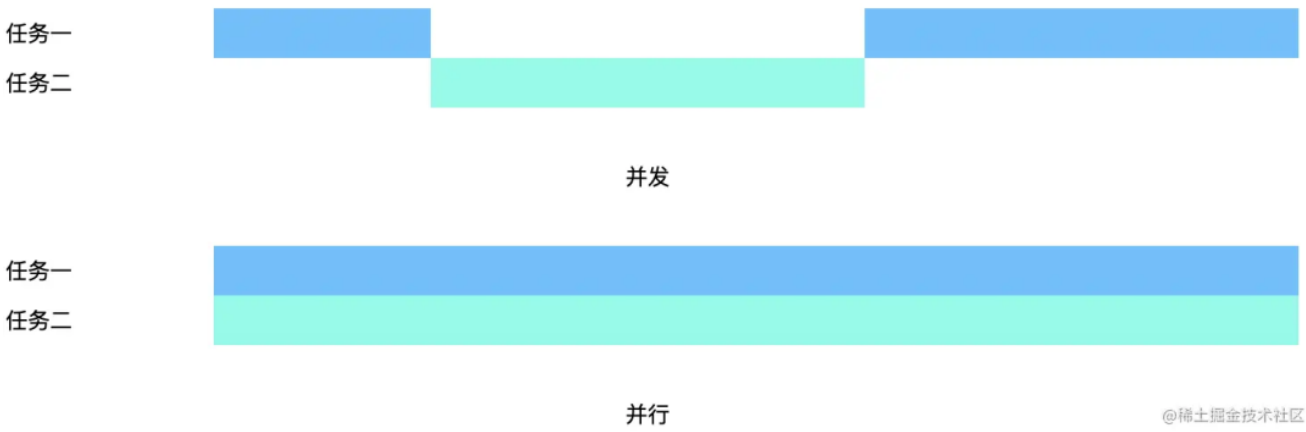
```
Running on DefaultDispatcher-worker-1
Running on DefaultDispatcher-worker-4
Running on DefaultDispatcher-worker-3
Running on DefaultDispatcher-worker-2
Running on DefaultDispatcher-worker-5
Running on DefaultDispatcher-worker-5
Running on DefaultDispatcher-worker-2
Running on DefaultDispatcher-worker-6
Running on DefaultDispatcher-worker-2
Running on DefaultDispatcher-worker-2
Running on DefaultDispatcher-worker-7
Running on DefaultDispatcher-worker-7
Running on DefaultDispatcher-worker-7
Running on DefaultDispatcher-worker-7

……
```

可以看到，不同的协程可能运行在不同的线程上，也可能同一个线程会被用来运行不同的协程。由于这个特性，所以协程也会存在多线程的并发问题。

那么，什么是并发呢？

简单理解，就是在同一个时间段内执行多个任务，此时为了实现这个目的，不同的任务可能会被拆分开来穿插着执行。

与之对应的，还有一个并行的概念，简单说就是多个任务在同一个时间点一起执行：



总之，不管是并行还是并发，都会涉及到对资源的“争夺”问题，因为在同一时间可能会有多个线程需要对同一个资源进行操作。此时就会出现上面举例的情况，由于多个线程都在对 count 进行操作，所以导致最终 count 的值会小于 1000，这也很好理解，比如此时 count 是 1，被线程 1 读取到之后，线程 1 开始对它进行 +1 操作，但是在线程1还没写完的时

候，来了个线程2，也读了一下 count 发现它是1，也对它进行 +1 操作。此时，不管线程1和2谁先写完，最终 count 也只会是 2，显然，按照我们的需求，应该是想让它是 3 才对。

那解决这个也简单啊，我们就不要让有这么多线程不就行了，只要只有一个线程不就行了？

确实，我们指定所有协程只在一个线程上执行：

```
fun main() {
    // 创建一个单线程上下文，并作为启动调度器
    val dispatcher = newSingleThreadContext("singleThread")

    var count = 0

    runBlocking {
        repeat(1000) {
            // 这里也可以直接不指定调度器，这样就会使用默认的线程执行这个协程，换言之，都
            launch(dispatcher) {
                println("Running on ${Thread.currentThread().name}")
                count++
            }
        }
    }

    println(count)
}
```

截取最后的输出结果如下：

```
……
Running on singleThread
Running on singleThread
Running on singleThread
Running on singleThread
Running on singleThread
Running on singleThread
Running on singleThread
Running on singleThread
Running on singleThread
Running on singleThread
Running on singleThread
1000
```

```
Process finished with exit code 0
```

可以看到，输出的 count 结果终于是正确了，那么，为什么还会有我这篇文章的问题呢？

哈哈，其实你被我绕进去了。

我们用协程（线程）的目的是什么？不就是为了能够执行耗时任务或者可以让多个任务同时执行，减少执行时间吗？即然你都用单线程了，那么有什么意义？

毕竟这里我们举例的代码只对 count 这一个变量进行操作，确实没有开多线程的必要，但是实际工作中肯定不止这么一个操作啊，难道我们要因为某个变量被其他线程占用了就不继续往下走了？就这么呆呆的阻塞住原地等待？显然不现实，醒醒吧，世界不是只有 count ，还有很多数据等待我们处理。所以我们用多线程的目的就是为了能够在某个变量（资源）不可用的时候可以去处理其他未被占用的资源，从而缩短总的执行时间。

但是，如果其他的代码执行到一定程度，绕不开必须要使用被占用的资源怎么办？

不管正在占用的线程是否解除占用直接硬去拿这个资源继续处理？显然不现实，因为这样就会造成我们前言中所述的情况发生。

所以如果我们遇到需要使用被占用的资源时，应当暂停当前线程，直至占用被解除。

在 java 中通常有三种方式解决这个问题：

1. `synchronized`
2. `AtomicInteger`
3. `ReentrantLock`

但是在 kotlin 的协程中使用它们不太合适，因为协程是非阻塞式的，当我们需要协程“暂停”的时候（如 delay(1000)），协程通常是被挂起，挂起的协程并不会阻塞它所在的线程，此时这个线程就可以腾出身去执行其他的任务。

而在 java 中需要线程暂停时（如 `Thread.sleep(1000)` ），通常就是直接阻塞这个线程，此时这个线程就会被限制，直到阻塞结束。

在 kotlin 中，提供了一个轻量级的同步锁：`Mutex`

什么是 **Mutex**

Mutex 是在 kotlin 协程中用于替代 java 线程中 synchronized 或 ReentrantLock 的类，用于为不应该被多个协程同时执行的代码上锁，例如为前面例子中的 count 自增代码上锁，这样可以保证它在同一时间点只会被一个协程执行，从而避免了由于多线程导致的数据修改问题。

Mutex 有两个核心方法：`lock()` 和 `unlock()` ，分别用于上锁和解锁：

```
fun main() {
    var count = 0
    val mutex = Mutex()

    runBlocking {
        repeat(1000) {
            launch(Dispatchers.IO) {
                println("Running on ${Thread.currentThread().name}")
                mutex.lock()
                count++
                mutex.unlock()
            }
        }
    }

    println(count)
}
```

上述代码输出截取如下：

```
……
Running on DefaultDispatcher-worker-47
Running on DefaultDispatcher-worker-20
Running on DefaultDispatcher-worker-38
Running on DefaultDispatcher-worker-15
Running on DefaultDispatcher-worker-14
Running on DefaultDispatcher-worker-19
Running on DefaultDispatcher-worker-48
1000

Process finished with exit code 0
```

可以看到，虽然协程运行在不同的线程，但是依然能够正确的对 count 进行修改操作。

这是因为我们在修改 count 值时调用了 `mutex.lock()` 此时保证了之后的代码块仅允许被当前协程执行，直至调用 `mutex.unlock()` 解除了锁定，其他协程才能继续执行这个代码块。

Mutex 的 lock 和 unlock 原理可以简单的理解成，当调用 lock 时，如果这个锁没有被其他协程持有则持有该锁，并执行后面的代码；如果这个锁已经被其他协程持有，则当前协程进入挂起状态，直至锁被释放，并拿到了锁。当被挂起时，它所在的线程并不会被阻塞，而是可以去执行其他的任务。

在实际使用中，我们一般不会直接使用 `lock()` 和 `unlock()` ，因为如果在上锁后执行的代码中出现异常的话，将会造成持有的锁永远不会被释放，此时就会造成死锁，其他的协程将永远等待不到这个锁被释放，从而永远被挂起：

```
fun main() {
    var count = 0
    val mutex = Mutex()

    runBlocking {
        repeat(1000) {
            launch(Dispatchers.IO) {
                try {
                    mutex.lock()
                    println("Running on ${Thread.currentThread().name}")
                    count++
                    count / 0
                    mutex.unlock()
                } catch (tr: Throwable) {
                    println(tr)
                }
            }
        }
    }

    println(count)
}
```

上述代码输出：

```
Running on DefaultDispatcher-worker-1
java.lang.ArithmeticException: / by zero
```

并且程序将会一直执行下去，无法终止。

其实要解决这个问题也很简单，我们只需要加上 finally ，使这段代码无论是否执行成功都要释放掉锁即可：

```
fun main() {
    var count = 0
    val mutex = Mutex()

    runBlocking {
        repeat(1000) {
            launch(Dispatchers.IO) {
                try {
                    mutex.lock()
                    println("Running on ${Thread.currentThread().name}")
                    count++
                    count / 0
                    mutex.unlock()
                } catch (tr: Throwable) {
                    println(tr)
                } finally {
                    mutex.unlock()
                }
            }
        }
    }

    println(count)
}
```

上述代码输出结果截取如下：

```
……

Running on DefaultDispatcher-worker-45
java.lang.ArithmeticException: / by zero
Running on DefaultDispatcher-worker-63
java.lang.ArithmeticException: / by zero
Running on DefaultDispatcher-worker-63
java.lang.ArithmeticException: / by zero
Running on DefaultDispatcher-worker-63
java.lang.ArithmeticException: / by zero
1000
```

```
Process finished with exit code 0
```

可以看到，虽然每个协程都报错了，但是程序是能执行完毕的，不会被完全挂起不动。其实这里我们可以直接使用 Mutex 的扩展函数 `withLock` ：

```
fun main() {
    var count = 0
    val mutex = Mutex()

    runBlocking {
        repeat(1000) {
            launch(Dispatchers.IO) {
                mutex.withLock {
                    try {
                        println("Running on ${Thread.currentThread().name}"
                        count++
                        count / 0
                    } catch (tr: Throwable) {
                        println(tr)
                    }
                }
            }
        }
    }

    println(count)
}
```

上述代码输出内容截取如下：

```
……
Running on DefaultDispatcher-worker-31
java.lang.ArithmeticException: / by zero
Running on DefaultDispatcher-worker-31
java.lang.ArithmeticException: / by zero
Running on DefaultDispatcher-worker-51
java.lang.ArithmeticException: / by zero
Running on DefaultDispatcher-worker-51
java.lang.ArithmeticException: / by zero
Running on DefaultDispatcher-worker-51
```

```
java.lang.ArithmeticException: / by zero
1000
```

可以看到，使用 `withLock` 后就不需要我们自己处理上锁和解锁了，只需要把要保证只被同时执行一次的代码放进它的参数中的高阶函数里就行。

这里看一下 withLock 的源码：

```
public suspend inline fun <T> Mutex.withLock(owner: Any? = null, action: ()
    // ……

    lock(owner)
    try {
        return action()
    } finally {
        unlock(owner)
    }
}
```

其实也非常的简单，就是在执行我们传入的 action 函数前调用 `lock()` 执行完毕后在 finally 中调用 `unlock()`。

说了这么多，可能读者想问了，你在这讲了半天，是不是偏题了啊？你的标题呢？怎么还不说?别急别急，这不就来了吗

为什么我都 mutex.withLock 了却没用呢？

回到我们的标题和前言中的场景，为什么项目中明明使用了 `mutex.Unlock` 将查重代码上锁了，还是会出现重复插入的情况？

我知道你很急，但是你别急，容我再给你看个例子：

```
fun main() {
    var count = 0
    val mutex = Mutex()

    runBlocking {
```

```
        mutex.withLock {
            repeat(10000) {
                launch(Dispatchers.IO) {
                    count++
                }
            }
        }
    }

    println(count)
}
```

你猜这段代码能输出 10000 吗？再看一段代码：

```
fun main() {
    var count = 0
    val mutex = Mutex()

    runBlocking {
        mutex.withLock {
            repeat(100) {
                launch(Dispatchers.IO) {
                    repeat(100) {
                        launch(Dispatchers.IO) {
                            count++
                        }
                    }
                }
            }
        }
    }

    println(count)
}
```

这段呢？你们猜能输出 10000 吗？

其实只要我们稍微想一想就知道，这个显然不可能输出 10000 啊。虽然我们在最顶层加了 `mutex.lockWith` 。但是，我们却在其中新开了很多新的协程，这就意味着，事实上这个锁约等于没有加。

还记得我们上面看过的 `mutex.lockWith` 的源码吗？此处相当于刚 lock 上，启动了一个新协程，直接 unlock 了，但是实际需要上锁的代码应该是新启动的协程里面的代码啊。

所以，我们在上锁时应该尽可能的缩小上锁的粒度，只给需要的代码上锁：

```
fun main() {
    var count = 0
    val mutex = Mutex()

    runBlocking {
        repeat(100) {
            launch(Dispatchers.IO) {
                repeat(100) {
                    launch(Dispatchers.IO) {
                        mutex.withLock {
                            count++
                        }
                    }
                }
            }
        }
    }

    println(count)
}
```

这里，我们需要上锁的其实就是对 count 的操作，所以我们只需要把上锁代码加给 count++ 即可，运行代码，完美输出 10000 。

有了上面的铺垫，我们再来看看我接手项目的简化代码原型：

```
fun main() {
    val mutex = Mutex()

    runBlocking {
        mutex.withLock {
         // 模拟同时调用了很多次插入函数
            insertData("1")
            insertData("1")
            insertData("1")
            insertData("1")
            insertData("1")
```

```
        }
    }
}

fun insertData(data: String) {
    CoroutineScope(Dispatchers.IO).launch {
        // 这里写一些无关数据的业务逻辑
        // xxxxxxx

        // 这里进行查重 查重结果 couldInsert
        if (couldInsert) {
            launch(Dispatchers.IO) {
                // 这里将数据插入数据库
            }
        }
    }
}
```

你们猜，此时数据库会被插入几个 1 ？答案显然是无法预知，一二三四五次都有可能。

我们来猜一猜，这哥们在写这段代码时的心路历程：

> 产品：这里的插入数据需要注意一个类型只让插入一个数据啊
>
> 开发：好嘞，这还不简单，我在插入前加个查重就行了
>
> 提测后
>
> 测试：开发兄弟，你这里有问题啊，这个数据可以被重复插入啊
>
> 开发：哦？我看看，哦，这里查询数据库用了协程异步执行，那不就是并发问题吗？我搜搜看 kotlin 的协程这么解决并发，哦，用 mutex 啊，那简单啊。
>
> 于是开发一顿操作，直接在调用查重和插入数据的最上级函数中加了个 mutex.withlock 将整个处理逻辑全部上锁。并且觉得这样就万无一失了，高枕无忧了，末了还不忘给 kotlin 点个赞，加锁居然这么方便，不像 java 还得自己写一堆处理代码。

那么，我是这么解决这个问题的呢？最好的解决方案，其实应该是能够将上锁粒度细化到具体的数据库操作的地方，但是还记得我上面说的吗，这个项目中嵌套了一层又一层的查询代码，

想要在这其中插入上锁代码显然不容易，我可不想因为往里面插一个锁直接导致整座大山倒塌。

所以我的选择是给每个 launch 了新协程的地方又加了一堆锁……这座山，因为我，变得更高了，哈哈哈哈哈。所以，其实并不是 mutex 有问题，有问题的只是使用的人罢了。

-- END --

推荐阅读

- 图解 Java 中那 18 把锁
- kotlin协程：并发 & 线程安全
- 建议收藏！Kotlin 线程同步的 N 种方法

AndroidPub

资深Android研发&面试官，定期分享原创技术文章、学习资料、实用面经等。欢迎关注...

73篇原创内容

公众号

收录于合集 #协程 6

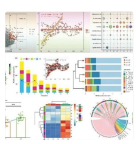
上一篇 · 有关协程 Dispatcher 的七个灵魂拷问

阅读原文

喜欢此内容的人还喜欢

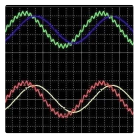
R语言绘图及数据分析合集！

科研后花园



ADC常用的十大滤波算法(C语言)

一起学嵌入式



Vue 3 中的 WebSocket 使用

一个不务正业的程序员