二

# 23 Lock 有哪几个常用方法？分别有什么用？

本课时我们主要讲解 Lock 有哪几种常用的方法，以及它们分别都是干什么用的。

## 简介

Lock 接口是 Java 5 引入的，最常见的实现类是 ReentrantLock，可以起到“锁”的作用。

Lock 和 synchronized 是两种最常见的锁，锁是一种工具，用于控制对共享资源的访问，而 Lock 和 synchronized 都可以达到线程安全的目的，但是在使用上和功能上又有较大的不同。所以 Lock 并不是用来代替 synchronized 的，而是当使用 synchronized 不合适或不足以满足要求的时候，Lock 可以用来提供更高级功能的。

通常情况下，Lock 只允许一个线程来访问这个共享资源。不过有的时候，一些特殊的实现也可允许并发访问，比如 ReadWriteLock 里面的 ReadLock。

## 方法纵览

我们首先看下 Lock 接口的各个方法，如代码所示。

```
public interface Lock {

    void lock();

    void lockInterruptibly() throws InterruptedException;

    boolean tryLock();

    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

    void unlock();

    Condition newCondition();

}
```

我们可以看到与 Lock 接口加解锁相关的主要有 5 个方法，我们接下来重点分析这 5 种方法

的作用和用法，这 5 种方法分别是 lock()、tryLock()、tryLock(long time, TimeUnit unit) 和 lockInterruptibly()、unlock()。

## lock() 方法

在 Lock 接口中声明了 4 种方法来获取锁（lock()、tryLock()、tryLock(long time, TimeUnit unit)和lockInterruptibly()），那么这 4 种方法具体有什么区别呢？

首先，lock() 是最基础的获取锁的方法。在线程获取锁时如果锁已被其他线程获取，则进行等待，是最初级的获取锁的方法。

对于 Lock 接口而言，获取锁和释放锁都是显式的，不像 synchronized 那样是隐式的，所以 Lock 不会像 synchronized 一样在异常时自动释放锁（synchronized 即使不写对应的代码也可以释放），lock 的加锁和释放锁都必须以代码的形式写出来，所以使用 lock() 时必须由我们自己主动去释放锁，因此最佳实践是执行 lock() 后，首先在 try{} 中操作同步资源，如果有必要就用 catch{} 块捕获异常，然后在 finally{} 中释放锁，以保证发生异常时锁一定被释放，示例代码如下所示。

```
Lock lock = ...;

lock.lock();

try{

    //获取到了被本锁保护的资源，处理任务

    //捕获异常

}finally{

    lock.unlock();   //释放锁

}
```

在这段代码中我们创建了一个 Lock，并且用 Lock 方法加锁，然后立刻在 try 代码块中进行相关业务逻辑的处理，如果有需要还可以进行 catch 来捕获异常，但是最重要的是 finally，大家一定不要忘记在 finally 中添加 unlock() 方法，以便保障锁的绝对释放。

如果我们不遵守在 finally 里释放锁的规范，就会让 Lock 变得非常危险，因为你不知道未来什么时候由于异常的发生，导致跳过了 unlock() 语句，使得这个锁永远不能被释放了，其他线程也无法再获得这个锁，这就是 Lock 相比于 synchronized 的一个劣势，使用 synchronized 时不需要担心这个问题。

与此同时，lock() 方法不能被中断，这会带来很大的隐患：一旦陷入死锁，lock() 就会陷入

永久等待，所以一般我们用 tryLock() 等其他更高级的方法来代替 lock()，下面我们就看一看 tryLock() 方法。

## tryLock()

tryLock() 用来尝试获取锁，如果当前锁没有被其他线程占用，则获取成功，返回 true，否则返回 false，代表获取锁失败。相比于 lock()，这样的方法显然功能更强大，我们可以根据是否能获取到锁来决定后续程序的行为。

因为该方法会立即返回，即便在拿不到锁时也不会一直等待，所以通常情况下，我们用 if 语句判断 tryLock() 的返回结果，根据是否获取到锁来执行不同的业务逻辑，典型使用方法如下。

```
Lock lock = ...;
if(lock.tryLock()) {
     try{
         //处理任务
     }finally{
         lock.unlock();   //释放锁
     }
}else {
    //如果不能获取锁，则做其他事情
}
```

我们创建 lock() 方法之后使用 tryLock() 方法并用 if 语句判断它的结果，如果 if 语句返回 true，就使用 try finally 完成相关业务逻辑的处理，如果 if 语句返回 false 就会进入 else 语句，代表它暂时不能获取到锁，可以先去做一些其他事情，比如等待几秒钟后重试，或者跳过这个任务，有了这个强大的 tryLock() 方法我们便可以解决死锁问题，代码如下所示。

```
    public void tryLock(Lock lock1, Lock lock2) throws InterruptedException {
        while (true) {
            if (lock1.tryLock()) {
                try {
                    if (lock2.tryLock()) {
```

```
                    try {
                        System.out.println("获取到了两把锁，完成业务逻辑");
                        return;
                    } finally {
                        lock2.unlock();
                    }
                }
            } finally {
                lock1.unlock();
            }
        } else {
            Thread.sleep(new Random().nextInt(1000));
        }
    }
}
```

如果代码中我们不用 tryLock() 方法，那么便可能会产生死锁，比如有两个线程同时调用这个方法，传入的 lock1 和 lock2 恰好是相反的，那么如果第一个线程获取了 lock1 的同时，第二个线程获取了 lock2，它们接下来便会尝试获取对方持有的那把锁，但是又获取不到，于是便会陷入死锁，但是有了 tryLock() 方法之后，我们便可以避免死锁的发生，首先会检测 lock1 是否能获取到，如果能获取到再尝试获取 lock2，但如果 lock1 获取不到也没有关系，我们会在下面进行随机时间的等待，这个等待的目标是争取让其他的线程在这段时间完成它的任务，以便释放其他线程所持有的锁，以便后续供我们使用，同理如果获取到了 lock1 但没有获取到 lock2，那么也会释放掉 lock1，随即进行随机的等待，只有当它同时获取到 lock1 和 lock2 的时候，才会进入到里面执行业务逻辑，比如在这里我们会打印出“获取到了两把锁，完成业务逻辑”，然后方法便会返回。

## tryLock(long time, TimeUnit unit)

tryLock() 的重载方法是 tryLock(long time, TimeUnit unit)，这个方法和 tryLock() 很类似，区别在于 tryLock(long time, TimeUnit unit) 方法会有一个超时时间，在拿不到锁时会等待一定的时间，如果在时间期限结束后，还获取不到锁，就会返回 false；如果一开始就获取锁或者等待期间内获取到锁，则返回 true。

这个方法解决了 lock() 方法容易发生死锁的问题，使用 tryLock(long time, TimeUnit unit) 时，在等待了一段指定的超时时间后，线程会主动放弃这把锁的获取，避免永久等待；在等待的期间，也可以随时中断线程，这就避免了死锁的发生。本方法和下面介绍的 lockInterruptibly() 是非常类似的，让我们来看一下 lockInterruptibly() 方法。

## lockInterruptibly()

这个方法的作用就是去获取锁，如果这个锁当前是可以获得的，那么这个方法会立刻返回，但是如果这个锁当前是不能获得的（被其他线程持有），那么当前线程便会开始等待，除非它等到了这把锁或者是在等待的过程中被中断了，否则这个线程便会一直在这里执行这行代码。一句话总结就是，除非当前线程在获取锁期间被中断，否则便会一直尝试获取直到获取到为止。

顾名思义，lockInterruptibly() 是可以响应中断的。相比于不能响应中断的 synchronized 锁，lockInterruptibly() 可以让程序更灵活，可以在获取锁的同时，保持对中断的响应。我们可以把这个方法理解为超时时间是无穷长的 tryLock(long time, TimeUnit unit)，因为 tryLock(long time, TimeUnit unit) 和 lockInterruptibly() 都能响应中断，只不过 lockInterruptibly() 永远不会超时。

这个方法本身是会抛出 InterruptedException 的，所以使用的时候，如果不在方法签名声明抛出该异常，那么就要写两个 try 块，如下所示。

```
public void lockInterruptibly() {
    try {
        lock.lockInterruptibly();
        try {
            System.out.println("操作资源");
        } finally {
            lock.unlock();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

在这个方法中我们首先执行了 lockInterruptibly 方法，并且对它进行了 try catch 包装，然后同样假设我们能够获取到这把锁，和之前一样，就必须要使用 try finall 来保障锁的绝对释放。

## unlock()

最后要介绍的方法是 unlock() 方法，是用于解锁的，u方法比较简单，对于 ReentrantLock 而言，执行 unlock() 的时候，内部会把锁的“被持有计数器”减 1，直到减到 0 就代表当前这把锁已经完全释放了，如果减 1 后计数器不为 0，说明这把锁之前被“重入”了，那么锁并没有真正释放，仅仅是减少了持有的次数。

上一页 下一页