第 5 章

线程池与锁

本章将延续第 4 章的内容，对线程池中的锁机制做进一步的深入探索。

5.1 重入锁 ReentrantLock

ReentrantLock 翻译过来就是重入锁，读作 Re-entrant-Lock。

ReentrantLock 是一种线程互斥锁，它的基本行为和语义与使用 synchronized 的隐式监视器锁基本一致，但是更具有扩展性。所谓的互斥锁与重入锁概念，是指 ReentrantLock 对象在线程之间锁是互斥的，而同一个线程则可以反复调用 ReentrantLock 对象的 lock()方法，进行多次加锁。

ReentrantLock 的推荐调用模式如下：

```
class X {
    private final ReentrantLock lock = new ReentrantLock();
    public void m() {
        lock.lock();              //加锁
        try {
            // ... 方法体
        } finally {
            lock.unlock();        //解锁
         }
    }
}
```

这段代码需要注意以下两点：

（1）首先要创建一个 ReentrantLock 的成员对象实例，这是为了在该类的多个业务方法中调用同一个 ReentrantLock 对象的 lock()方法。

（2）在每个业务方法的头部进行 lock()锁定操作，在方法执行完成后，在 finally 中释放锁。由于 ReentrantLock 的锁必须手动释放（这与 synchronized 隐式锁不同），因此在 finally 中调用 unlock()方法非常关键。一旦忘记锁的释放，其他线程想获得该锁时会出现长时间的

等待。

5.1.1 重入锁

ReentrantLock 对象可以被同一线程反复递归调用，最大递归次数为 2 147 483 647。试图超出这个限制次数，将会导致从锁定方法中抛出 Error 异常。

ReentrantLock 重入调用的典型示例如下：

（1）定义类 LockX，在方法 m1()的方法体前加锁，在方法体尾部解锁。

```
public class LockX {
    private final ReentrantLock lock = new ReentrantLock();
    public void m1() {
        lock.lock();              //加锁
        try {
            // ...方法体
        } finally {
            lock.unlock();        //解锁
        }
    }
}
```

（2）在类 LockX 中定义方法 m2()，方法体前后也分别使用 lock()与 unlock()进行加锁与解锁。

```
public void m2() {
     lock.lock();                 //加锁
     try {
           // ... 方法体
     } finally {
          lock.unlock();          //解锁
     }
}
```

（3）在方法 m1()中调用方法 m2()，并跟踪当前线程对锁的持有数量。

```
public void m1() {
    System.out.println("m1:" + lock.getHoldCount());
    lock.lock();                  //加锁
    try {
        // ...方法体
        System.out.println("m1:" + lock.getHoldCount());
        m2();                     //在 m1()中调用 m2()
    } finally {
        lock.unlock();            //解锁
```

```
            System.out.println("m1:" + lock.getHoldCount());
        }
    }
    public void m2() {
        lock.lock();                //加锁
        try {
            // ... 方法体
            System.out.println("m2:" + lock.getHoldCount());
        } finally {
            lock.unlock();          //解锁
            System.out.println("m2:" + lock.getHoldCount());
        }
    }
```

（4）在测试类中调用方法 m1()。

```
public static void main(String[] args) {
    LockX x = new LockX();
    x.m1();
}
```

程序运行结果如下，可以清楚地跟踪到锁持有者的数量。

```
m1:0
m1:1
m2:2
m2:1
m1:0
```

5.1.2　互斥锁

继续使用 5.1.1 节中的类 LockX，在日志中增加 Thread.currentThread().getId()，跟踪执行 m1()和 m2()方法的线程 ID。操作步骤如下：

（1）使用线程池，并发调用 LockX 对象的 m1()方法。

```
public static void main(String[] args) {
    LockX x = new LockX();
    ExecutorService pool = Executors.newFixedThreadPool(5);
    for(int i=0;i<5;i++) {
        pool.execute(new Runnable() {
            public void run() {
                x.m1();
            }
        });
```

```
        }
        pool.shutdown();
    }
```

程序运行结果如下，5 个线程首先在 m1()方法的 lock.lock()处阻塞，争抢同一把锁。由于 ReentrantLock 为互斥锁，因此只能有一个线程进入 m1()的方法体。只有抢到锁的线程执行完毕，锁持有者数量为 0，第二个线程才有机会再次抢到锁。

```
m1:0,8
m1:0,9
m1:0,10
m1:0,11
m1:0,12
m1:1,8
m2:2,8
m2:1,8
m1:0,8
m1:1,9
m2:2,9
m2:1,9
m1:0,9
m1:1,10
m2:2,10
m2:1,10
m1:0,10
m1:1,11
m2:2,11
m2:1,11
m1:0,11
m1:1,12
m2:2,12
m2:1,12
m1:0,12
```

（2）把 m2()方法中的代码 lock.unlock()注释掉，程序运行结果如下，线程 8 首先抢到锁，然后进入 m1()和 m2()方法体，由于 m2()没有及时释放锁，因此所有的其他线程始终处于阻塞等待状态。

```
m1:0,11
m1:0,12
m1:0,8
m1:0,9
m1:0,10
```

```
m1:1,8
m2:2,8
m2:2,8
m1:1,8
```

重入锁还可以有效防止线程死锁。其简单机制是为每个锁关联一个请求计数器和一个占有它的线程对象，当计数器为 0 时，表示暂时没有线程占有它。当一个线程请求获得锁后，这个计数器会累加一次，当这个线程再次请求这个锁时，计数器再次累加一次。占有线程退出同步计数器递减一次，直到计数器为 0 时，这个锁才会被释放。

使用 ReentrantLock 进行加锁与解锁行为是显式的，我们加了多少次锁就要释放多少次锁。释放锁的行为一定要在 finally 中进行，这样可以保证锁一定会被释放。

5.1.3　ReentrantLock 与 synchronized

ReentrantLock 与 synchronized 在基本行为与语义上有很多相似性，下面对比分析一下两者的异同：

（1）它们都是互斥锁，都具有独占性和排他性。

（2）ReentrantLock 是重入锁，synchronized 也允许重入。

（3）ReentrantLock 是显式锁，需要显式地调用 lock()与 unlock()方法。synchronized 是隐式锁，加锁与解锁都依赖隐式的监视器。

（4）synchronized 是 Java 语言底层内置的锁机制，依赖于 JVM 直接解析字节码。而 ReentrantLock 是 JDK 层面的，可以查看其源代码和实现机制。

（5）ReentrantLock 应用更加灵活，具有很好的扩展性。调用 newCondition()方法，可以创建 Condition 实例，这是用于多线程交互的条件监视器。另外为了防止由于忘记调用 unlock()导致死锁，ReentrantLock 还提供了定时解锁功能。

synchronized 隐式锁重入与互斥特性测试如下：

（1）在类 LockY 中定义方法 m1()和 m2()。在 m1()与 m2()中，用 synchronized 关键字锁同一个成员变量 object。在 m1()方法中调用 m2()方法。

```
public class LockY {
   private Object object = new Object();
   public void m1() {
       synchronized(object) {
            //... 方法体
            System.out.println("m1:"+Thread.currentThread().getId()+"run...");
            m2();
       }
       System.out.println("m1:"+Thread.currentThread().getId()+"退出...");
   }
   public void m2() {
```

```
        synchronized(object) {
            //... 方法体
            System.out.println("m2:"+ Thread.currentThread().getId()+"run...");
        }
        System.out.println("m2:" + Thread.currentThread().getId() + " 退出...");
    }
}
```

（2）在主函数中，创建 LockY 对象并调用 m1()。

```
public static void main(String[] args) {
    LockY y = new LockY();
    y.m1();
}
```

程序运行结果如下，可以明确看出 synchronized 隐式锁的重入特性：

```
m1:1 run...
m2:1 run...
m2:1 退出...
m1:1 退出...
```

（3）并发调用测试。

```
public static void main(String[] args) {
    LockY y = new LockY();
    ExecutorService pool = Executors.newFixedThreadPool(3);
    for(int i=0;i<3;i++) {
        pool.execute(new Runnable() {
            public void run() {
                y.m1();
            }
        });
    }
    pool.shutdown();
}
```

程序运行结果如下，可以明确看到线程互斥现象：

```
m1:8 run...
m2:8 run...
m2:8 退出...
m1:8 退出...
m1:10 run...
m2:10 run...
```

```
m2:10 退出...
m1:10 退出...
m1:9 run...
m2:9 run...
m2:9 退出...
m1:9 退出...
```

5.1.4　尝试加锁并限时等待

调用 tryLock()方法，可以尝试获取 ReentrantLock 的锁。如果当前线程已经保存该锁，则保持计数增加 1，该方法返回 true。如果锁由另一个线程持有，则该方法将立即返回 false。

tryLock(long timeout, TimeUnit unit)方法允许限时等待获取锁：如果在给定的等待时间内当前线程未被中断，且锁没有被其他线程持有，则获取该锁。

```
public class ReentrantLock {
    public boolean tryLock() {
       return sync.nonfairTryAcquire(1);
    }
    public boolean tryLock(long timeout,TimeUnit unit)
                    throws InterruptedException{
       return sync.tryAcquireNanos(1, unit.toNanos(timeout));
    }
}
```

代码测试步骤如下：

（1）新建测试类，定义成员变量 ReentrantLock 对象。在方法 m1()中，调用 tryLock(5, TimeUnit.SECONDS)获取锁，超时则退出。

```
public class TryLockTest {
    private ReentrantLock lock = new ReentrantLock();
   public void m1(){
      try {
         if (lock.tryLock(5, TimeUnit.SECONDS)) {
            System.out.println(Thread.currentThread().getId()+"获取
                                了锁...");
            TimeUnit.SECONDS.sleep(3);
         } else {
            System.out.println(Thread.currentThread().getId()+"等待
                               超时...");
         }
      }catch(Exception e){
```

```
            e.printStackTrace();
        }finally {
            if(lock.isHeldByCurrentThread() && lock.isLocked()) {
                lock.unlock();
            }
        }
    }
}
```

（2）并发调用 m1()方法，由于锁只能由一个线程获得，故其他线程需要等待。

```
public static void main(String[] args) {
    final TryLockTest test = new TryLockTest();
    ExecutorService pool = Executors.newFixedThreadPool(5);
    for(int i = 0; i < 5; i++) {
        pool.execute(new Runnable() {
            public void run() {
                test.m1();
            }
        });
    }
    pool.shutdown();
}
```

程序运行结果如下，超过等待时间的线程，会退出等待：

```
8 获取了锁...
9 获取了锁...
10 等待超时...
11 等待超时...
12 等待超时...
```

注意：调用 lock.unlock()方法时，需要进行判断；否则未获得锁的线程，调用 unlock()方法，会抛出 java.lang.IllegalMonitorStateException 异常。

```
if(lock.isHeldByCurrentThread() && lock.isLocked()) {
    lock.unlock();
}
```

尝试加锁并限时等待在某些场景下非常有用，它可以有效避免由于处理不当长时间占用资源，从而发生死锁的情况。如果某些线程由于异常等原因遗漏了 unlock()调用，也不会导致其他线程的无限期等待。

5.2　重入锁与 Condition

synchronized 关键字与 Object 对象的 wait()、notify()方法配合，可以实现多线程间的等待、唤醒操作，参见 1.3.4 节的代码示例。

ReentrantLock 锁与 Condition 条件对象配合，也可以实现多线程的等待/通知操作。

```
public interface Condition {
    void await() throws InterruptedException;
    boolean await(long time,TimeUnit unit)
           throws InterruptedException;
    void signal();
    void signalAll();
}
```

Condition 对象的 await()方法，与 Object 对象的 wait()方法类似。而 Condition 对象的 signal()方法，与 Object 类中的 notify()方法类似。

把 1.3.4 节的代码进行修改，用 ReentrantLock 与 Condition 实现线程交互，操作步骤如下：

（1）主函数中创建 ReentrantLock 与 Condition 对象。

```
public static void main(String[] args) {
   final ReentrantLock lock = new ReentrantLock();
   final Condition condition  = lock.newCondition();
   ...
}
```

（2）定义线程对象 t1，当 i=5 时，阻塞当前线程。调用 condition.await()阻塞当前线程前，必须要调用 lock.lock()获取锁。

```
Thread t1 =  new Thread(new Runnable() {
        public void run() {
             for(int i=0;i<10;i++) {
                 System.out.println(Thread.currentThread().getName()
                                    + ",i=" +i);
                 if(i==5) {
                     lock.lock();
                     System.out.println(Thread.currentThread().getName()
                                        + "加锁");
                     try {
                         System.out.println(Thread.currentThread()
                                            .getName()  + "开始等待...");
```

```
                    condition.await();
                }catch(Exception e) {
                    e.printStackTrace();
                }finally {
                    lock.unlock();
                    System.out.println(Thread.currentThread()
                                        .getName() + "解锁");
                }
            }
    } } });
```

（3）定义线程对象 t2，把线程 t1 从 WAITING 状态唤醒。调用 condition.signal()唤醒 t1 线程前，必须要调用 lock.lock()获取锁。

```
Thread t2 =  new Thread(new Runnable() {
        public void run() {
            System.out.println(Thread.currentThread().getName()
                                + " running...");
                lock.lock();
                System.out.println(Thread.currentThread().getName()
                                    + "加锁");
                try {
                   System.out.println(Thread.currentThread().getName()
                                       + "，发送 notify 通知...");
                   condition.signal();
                } catch (Exception e) {
                    e.printStackTrace();
                }finally {
                    lock.unlock();
                    System.out.println(Thread.currentThread().getName()
                                        + "解锁");
                }
    }});
```

（4）分别启动线程 t1 和 t2。

```
public static void main(String[] args) {
    final ReentrantLock lock = new ReentrantLock();
    final Condition condition  = lock.newCondition();
    Thread t1 =  new Thread(...);
    Thread t2 =  new Thread(...);
    //启动第一个线程
    t1.start();
    double d = 0;
```