链滴

Java 并发编程（三）可重入锁 ReentrantLock

作者：wlgzs-sjl

原文链接：https://ld246.com/article/1574473082167

来源网站：链滴

ReentrantLock的使用

ReentrantLock可以完全替代synchronized,提供了一种更灵活的锁.

ReenTrantLock必须手动释放锁,为防止发生异常,必须将同步代码用try包裹起来,在finally代码块中释锁.

```
public class T {

    ReentrantLock lock = new ReentrantLock();

    // 使用ReentrantLock的写法
    private void m1() {
        // 尝试获得锁
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName());
        } finally {
            //
            lock.unlock();
        }
    }

    // 使用synchronized的写法
    private synchronized void m2() {
        System.out.println(Thread.currentThread().getName());
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
    }


    public static void main(String[] args) {
        T t = new T();
        new Thread(t::m1, "t1").start();
        new Thread(t::m2, "t2").start();
    }
}
```

ReentrantLock获取锁的方法

尝试锁tryLock()

使用tryLock()方法可以尝试获得锁,返回一个boolean值,指示是否获得锁.

可以给tryLock方法传入阻塞时长,当超出阻塞时长时,线程退出阻塞状态转而执行其他操作.

```
public class T{

    ReentrantLock lock = new ReentrantLock();

    void m1() {
        lock.lock(); // 相当于 synchronized
        try {
            for (int i = 0; i < 5; i++) {
                TimeUnit.SECONDS.sleep(1);
                System.out.println(i);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock(); // 使用完毕后，必须手动释放锁
            // 不同于synchronized，抛出异常后，不会自动释放锁，需要我们在finally中释放此锁
        }
    }

    void m2() {
        // 尝试获取锁，返回true拿到了
        if (lock.tryLock()) {
            // lock.tryLock(5, TimeUnit.SECONDS) // 等5s内还没拿到就返回false
            System.out.println("m2...");
            lock.unlock();
        } else {
            System.out.println(" m2 没拿到锁");
        }

    }

    public static void main(String[] args) {
        T r1 = new T();
        new Thread(r1::m1, "t1").start(); // m1 已经执行，被t1占有锁this
```

```
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        new Thread(r1::m2, "t2").start(); // 锁已经被其他线程占用，m1执行完毕后，不会执行
    }

}
```

程序运行结果如下：

```
0
 m2 没拿到锁
1
2
3
4
```

可中断锁lockInterruptibly

使用lockInterruptibly以一种可被中断的方式获取锁.获取不到锁时线程进入阻塞状态,但这种阻塞状态以被中断.调用被阻塞线程的interrupt()方法可以中断该线程的阻塞状态,并抛出InterruptedException常.

interrupt()方法只能中断线程的阻塞状态.若某线程已经得到锁或根本没去尝试获得锁,则该线程当前没处于阻塞状态,因此不能被interrupt()方法中断.

```
public class T{

    public static void main(String[] args) {
        ReentrantLock lock = new ReentrantLock();
        Thread t1 = new Thread(() -> {
            lock.lock();
            try {
                System.out.println("t1 start");
                TimeUnit.SECONDS.sleep(Integer.MAX_VALUE);// 线程一直占用锁
                System.out.println("t1 end");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }

        }, "t1");
        t1.start();

    // 线程2抢不到lock锁,若不被中断则一直被阻塞
        Thread t2 = new Thread(() -> {
            try {
                lock.lockInterruptibly(); // t2 尝试获取锁
                System.out.println("t2 start");
                TimeUnit.SECONDS.sleep(3);
                System.out.println("t1 end");
```

```
            } catch (InterruptedException e) {
                System.out.println("t2 等待中被打断");
            } finally {
                lock.unlock(); // 没有锁定进行unlock就会抛出 IllegalMonitorStateException
            }
        }, "t2");
        t2.start();

        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 打断线程2的等待
        t2.interrupt();

    }

}
```

程序运行结果如下：

```
t1 start
t2 等待中被打断
Exception in thread "t2" java.lang.IllegalMonitorStateException
    at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:151)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchroni
er.java:1261)
    at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:457)
    at c_020.ReentrantLock4.lambda$main$1(ReentrantLock4.java:41)
    at java.lang.Thread.run(Thread.java:748)
```

并不是所有处于阻塞状态的线程都可以被interrupt()方法中断,要看该线程处于具体的哪种阻塞状态.阻状态包括普通阻塞,等待队列,锁池队列.

- 普通阻塞: 调用sleep()方法的线程处于普通阻塞,调用其interrupt()方法可以中断其阻塞状态并抛出InerruptedException异常
- 等待队列: 调用锁的wait()方法将持有当前锁的线程转入等待队列,这种阻塞状态只能由锁对象的notif方法唤醒,而不能被线程的interrupt()方法中断.
- 锁池队列: 尝试获取锁但没能成功抢到锁的线程会进入锁池队列:
    - 争抢synchronized锁的线程的阻塞状态不能被中断.
    - 使用ReentrantLock的lock()方法争抢锁的线程的阻塞状态不能被中断.
    - 使用ReentrantLock的tryLock()和lockInterruptibly()方法争抢锁的线程的阻塞状态不能被中断.

公平锁

在初始化ReentrantLock时给其fair参数传入true,可以指定该锁为公平锁,

synchronized 是不公平锁。

CPU默认的进程调度是不公平的,也就是说,CPU不能保证等待时间较长的线程先被执行.但公平锁可以证等待时间较长的线程先被执行。

公平锁，先获取锁的人，在锁被释放时，优先获得锁。

不公平锁，无论先后，线程调度器将会随机给某个线程锁，不用计算线程时序，效率较高。

```
public class T extends Thread {

    private static ReentrantLock lock = new ReentrantLock(true);// 指定锁为公平锁

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            lock.lock();
            try {
                System.out.println(Thread.currentThread().getName() + "获取锁");
                sleep(100);
            }catch (InterruptedException e){
                e.printStackTrace();
            }finally {
                lock.unlock(); // 公平锁 t1 unlock 后，等待时间长的一定是 t2 所以下次一定是 t2 执行
            }
        }
    }

    public static void main(String[] args) {
        T t1 = new T();
        new Thread(t1).start();
        new Thread(t1).start();

    }
}
```

程序运行结果如下：

```
Thread-1获取锁
Thread-2获取锁
Thread-1获取锁
Thread-2获取锁
Thread-1获取锁
Thread-2获取锁
```

生产者/消费者模式

经典面试题：写一个固定容量的容器，拥有put和get方法，以及getCount方法。能够支持2个生产者程以及10个消费者线程的阻塞调用。如果调用 get方法时，容器为空，get方法就需要阻塞等待；如调用 put方法时，容器满了，put方法就需要阻塞等待

1、使用synchronized的wait()/notify()实现

```
public class MyContainer1<T> {

    private final LinkedList<T> list = new LinkedList<>();
```

```
    private final int MAX = 10;
    private int count = 0;

    public synchronized void put(T t) {
        while (list.size() == MAX) { // 如果容量最大，释放锁等待    //【这里为什么使用while，而不使用if？？？】
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // 否则 put
        list.add(t);
        ++count;
        this.notifyAll(); // 通知消费者线程，可以消费了
        // 【这里为什么调用 notifyAll 而不是 notify？】
    }

    public synchronized T get() {
        T t = null;
        while (list.size() == 0) { // 如果容量为空，释放锁等待
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // 否则获取
        t = list.removeFirst();
        count--;
        this.notifyAll(); // 通知生产者线程生产
        return t;
    }

    public static void main(String[] args) {
        MyContainer1<String> c = new MyContainer1<>();
        //启动消费者线程
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                for (int j = 0; j < 5; j++) {
                    System.out.println(c.get());
                }
            }, "c" + i).start();
        }

        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //启动生产者线程
        for (int i = 0; i < 2; i++) {
```

```
            new Thread(() -> {
                for (int j = 0; j < 10; j++) {
                    c.put(Thread.currentThread().getName() + " " + j);
                }
            }, "p" + i).start();
        }
    }
}
```

为什么使用while 而不是使用 if ?

在与wait()的配合中，百分之99的程序都是与while而不是if结合使用。

上述代码中，在容器已满的情况下，put方法会wait等待，当容器中的元素被消费者消费了一部分，会唤醒所有put方法，

put方法会继续向下执行，直接执行list.add(t)，那么多个生产者线程执行list.add()就有可能出现数据致性的问题。

如果使用while则会循环判断，就避免了这些问题。

不是有锁吗？为什么会需要循环判断?

wait之后，锁就会失去，再次被唤醒时，并且得到锁之后，**是从list.add()开始执行的**，会无判断直加入到容器中。

为什么调用 notifyAll 而不是 notify ?

因为notify有可能再次叫醒一个生产者线程

2、使用Condition对象实现

用Lock和Condition实现，可以精确唤醒某些线程

```
public class MyContainer2<T> {

    private final LinkedList<T> list = new LinkedList<>();
    private final int MAX = 10;
    private int count = 0;

    private Lock lock = new ReentrantLock();// 锁对象
    // 绑定在锁上的一个条件,阻塞在该条件上的线程为生产者线程
    private Condition producer = lock.newCondition();
    // 绑定在锁上的一个条件,阻塞在该条件上的线程为消费者线程
    private Condition consumer = lock.newCondition();


    public  void put(T t) {
        lock.lock();
        try {
            while (list.size() == MAX) {
                // 当前线程应该是生产者线程,因此将当前线程阻塞在producerCondition上
                producer.await();
            }
            list.add(t);
            ++count;
```

```
            // 唤醒所有阻塞在consumerCondition的线程,这些被唤醒的线程都应该是消费者线程
            consumer.signalAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public  T get() {
        T t=null;
        lock.lock();
        try {
            while (list.size() == 0) {
                // 当前线程应该是消费者线程,因此将当前线程阻塞在consumerCondition上
                consumer.await();
            }
            t = list.removeFirst();
            count--;
            // 唤醒所有阻塞在producerCondition的线程,这些被唤醒的线程都应该是生产者线程
            producer.signalAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
        return t;
    }

    public static void main(String[] args) {
        MyContainer2<String> c = new MyContainer2<>();
        //启动消费者线程
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                for (int j = 0; j < 5; j++) {
                    System.out.println(c.get());
                }
            }, "c" + i).start();
        }

        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //启动生产者线程
        for (int i = 0; i < 2; i++) {
            new Thread(() -> {
                for (int j = 0; j < 10; j++) {
                    c.put(Thread.currentThread().getName() + " " + j);
                }
            }, "p" + i).start();
        }
```

```
    }
}
```

原文链接：Java 并发编程（三）可重入锁 ReentrantLock