JUC并发编程笔记

Object.wait()方法

- 释放锁，重新获得锁后会接着往下执行，例如：

```
public class WaitLockMain {

    public static void main(String[] args) throws InterruptedException {
        TomClass tomClass = new TomClass();
        new Thread(() -> {
            try {
                tomClass.decrement();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "A").start();
        TimeUnit.SECONDS.sleep(2);
        // 唤醒上面释放锁的线程
        tomClass.notifySelf();
        // 保持主线程存活，让上面的线程执行完毕
        TimeUnit.SECONDS.sleep(1);
    }

    static class TomClass {

        private int num = 0;

        private int count = 1;

        public synchronized void decrement() throws InterruptedException {
            System.out.println("进入方法");
            if (num != 1) {
                System.out.println("执行wait前，当前轮次：" + count);
                this.wait();
                System.out.println("执行wait后，当前轮次：" + count);
            }
            num--;
            System.out.printf("接着执行，当前轮次：%d，num值为：%d\n", count, num);
        }

        public synchronized void notifySelf() {
            this.notifyAll();
        }
    }
}
```

结果是这样的：

正确处理方案是改为如下：

```
public class WaitLockMain {
```

```
    public static void main(String[] args) throws InterruptedException {
        TomClass tomClass = new TomClass();
        new Thread(() -> {
            try {
                tomClass.decrement();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "A").start();
        TimeUnit.SECONDS.sleep(2);
        // 唤醒上面释放锁的线程
        tomClass.notifySelf();
        // 保持主线程存活，让上面的线程执行完毕
        TimeUnit.SECONDS.sleep(1);
    }

    static class TomClass {

        private int num = 0;

        private int count = 1;

        public synchronized void decrement() throws InterruptedException {
            System.out.println("进入方法");
            while (num != 1) {
                System.out.println("执行wait前，当前轮次：" + count);
                this.wait();
                System.out.println("执行wait后，当前轮次：" + count);
                count++;
            }
            num--;
            System.out.printf("接着执行，当前轮次：%d，num值为：%d\n", count, num);
        }

        public synchronized void notifySelf() {
            this.notifyAll();
        }
    }
}
```

改为while之后，即使获取了锁往下执行，执行完后仍然需要再次判断while条件。输出如下：

count为1时A线程第一次wait等待
主线程调用notifySelf()唤醒A线程
继续往下执行count++
再次回到while判断，进入第二次循环，再次wait释放锁暂停运行
如果没有手动调用notifySelf()方法，程序将永远不再继续往下执行。

Lock和Condition的使用

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @author TomShiDi
 * @description
 * @date 2021/9/19 17:25
 **/
public class LockConditionMain {
    public static void main(String[] args) {
        AtomicOperation atomicOperation = new AtomicOperation();
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                atomicOperation.increment();
            }
        }, "add1").start();
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                atomicOperation.decrement();
            }
        }, "del1").start();
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                atomicOperation.increment();
            }
        }, "add2").start();
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                atomicOperation.decrement();
            }
        }, "del2").start();
    }

    private static class AtomicOperation {
        private int num = 0;

        private Lock lock = new ReentrantLock();
        private Condition condition = lock.newCondition();

        public void increment() {
            lock.lock();
            try {
                while (num != 0) {
                    System.out.printf("%s：num!=0，被阻塞\n",
Thread.currentThread().getName());
                    condition.await();
                }
                num++;
                System.out.printf("%s：当前num值为：%d\n",
Thread.currentThread().getName(), num);
                // 唤醒其他线程
                condition.signalAll();
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }finally {
                lock.unlock();
            }
        }

        public void decrement() {
            lock.lock();
            try {
                while (num != 1) {
                    System.out.printf("%s：num!=1，被阻塞\n",
Thread.currentThread().getName());
                    condition.await();
                }
                num--;
                System.out.printf("%s：当前num值为：%d\n",
Thread.currentThread().getName(), num);
                // 唤醒其他线程
                condition.signalAll();
            } catch (Exception e) {
                e.printStackTrace();
            }finally {
                lock.unlock();
            }
        }
    }
}
```

jdk中的List（线程安全和不安全）

线程不安全List、Set

ArrayList：没有做并发加锁，线程不安全，并发修改时会报异常。也因为没有加锁操作，在不考虑并发的情况下性能最好。
HashSet：线程不安全，并发修改时会报异常。因为没有加锁，非并发条件下性能最好。

线程安全List

Vector：jdk1.0中提供的线程安全List，对方法加上了synchronized，性能差，废弃类。
Collections.synchronizedList()：该方法可以构建一个线程安全的List，是对原List做了一层synchronized封装，性能差，不建议使用。
CopyOnWriteArrayList：线程安全List，get读操作未加锁，读取时性能没有损耗。写操作时，先从原List复制一份，再对复制出的List增删，最后替换掉原List，替换的过程加了Reentrant锁。推荐使用的线程安全List。
CopyOnWriteArraySet：与上同理。

jdk中的Map（线程安全和不安全）

线程不安全 HashMap

HashMap：没有锁机制，并发修改会报异常，非并发厂家国内下性能最好。

线程安全 HashTable、ConcurrentHashMap

HashTable：线程安全，jdk1.0的遗老，方法上加synchronized锁，导致性能很差，废弃类。
ConcurrentHashMap：线程安全，用了分段锁机制，性能损耗小，并发下建议使用。

非公平锁（nofair）和公平锁（fair）

非公平锁（nofair）

1. 10个线程等待获取锁。线程序号表示线程等待锁的先后顺序，即线程1最先来争抢，线程10最后才来争抢锁。
2. 线程2获取到了锁，执行代码，其他9个线程等待，线程2执行完释放锁。
3. 线程5获取锁，线程5执行代码，其他线程等待，执行完释放锁。
4. 线程3获取，线程3执行代码，其他线程等待，线程3执行完释放锁。

   ……
   从这个例子可以看出，线程1明明是最先去争抢锁的，却一直没有抢到锁，极端情况下线程1会永远无法得到执行。这就是非公平锁。

公平锁（fair）

1. 10个线程依次排队获取锁，依次是线程1、线程2、线程3……线程10。
2. 线程1获取锁，执行完释放锁。
3. 线程2获取锁，执行完释放锁。
4. 线程3获取锁，执行完释放锁。

   ……
   先到的先获取锁，可以避免线程永远无法获取锁的情况。这就是公平锁。

可重入锁

同一线程中，一把钥匙可以开多个相同的锁，直接代码演示吧

```
        // 第一层锁
        synchronized (lockObject){
            // 第二层锁
            synchronized (lockObject) {
                System.out.println("同一线程中，获取了第一层锁之后，第二层不需要再次获取
锁");
            }
        }
```

死锁的四个条件

1. 资源互斥：同一时刻只能被一个线程持有。
2. 占有资源并等待：线程持有部分资源且等待获取其他的资源。
3. 非抢占：资源只有在线程执行完毕后才会释放，无法被抢占。
4. 循环等待：线程一直等待需要的那部分资源。

Callable可以返回结果的线程接口

Callable接口和Runnable接口类似，都是用于声明一个线程任务。区别在于Callable可以拿到任务的返回值。
Callable不能直接给Thread创建线程，需要做为FutureTask构造方法的入参，创建出的FutureTask，再用FutureTask构建Thread线程。

```
public static void main(String[] args) throws InterruptedException,
ExecutionException {
        FutureTask<Integer> futureTask = new FutureTask<>(() -> {
            TimeUnit.SECONDS.sleep(5);
            return count.incrementAndGet();
        });
        new Thread(futureTask, "A").start();
        while (!futureTask.isDone()) {
            System.out.printf("%s：任务还未完成\n",
Thread.currentThread().getName());
            TimeUnit.MICROSECONDS.sleep(200);
        }
        System.out.printf("%s：任务执行完毕，结果是：%d\n",
                Thread.currentThread().getName(),
                futureTask.get());
    }
```

CountDownLatch

一个jdk提供的并发计数器。创建实例时初始化一个数字count，使用countDown()方法进行count减一操作。await()方法中判断，如果count大于0则使得线程等待，如果count等于0则继续执行后续代码逻辑。

```
public static void main(String[] args) throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(10);

        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                System.out.printf("【%s】乘客上车了\n",
Thread.currentThread().getName());
                countDownLatch.countDown();
            }, "乘客" + i).start();
        }

        countDownLatch.await();
        System.out.println("所有乘客都已上车，发车");
    }
```

CyclicBarrier

与CountDownLatch类似的功能，有点类似 共轭 这个词。设定一个count，每次执行await()使得count减一。当count大于0时，线程等待；当count等于0时，唤醒所有等待的线程。

```
public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(count,
```

```
                () -> System.out.printf("【%s】：乘客均已上车\n",
Thread.currentThread().getName()));
        for (int i = 0; i < count; i++) {
            new Thread(() -> {
                System.out.printf("【%s】上车\n",
Thread.currentThread().getName());
                try {
                    cyclicBarrier.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
                System.out.printf("【%s】线程已被唤醒\n",
Thread.currentThread().getName());
            }, "乘客" + i).start();
        }
    }
```

Semaphore

可以理解为上面两个的升级版。在Semaphore中可以设置资源的最大数量，并且资源可以循环利用。acquire()方法可以获取一个或者多个资源，如果资源不足线程将会被阻塞。与之对应的release()方法可以释放一个或多个资源。

```
public static void main(String[] args) {
        // 初始化三个车位
        Semaphore semaphore = new Semaphore(3);

        for (int i = 0; i < 6; i++) {
            new Thread(() -> {
                try {
                    // 占用车位
                    semaphore.acquire();
                    System.out.printf("【%s】：进入了车位\n",
Thread.currentThread().getName());
                    TimeUnit.SECONDS.sleep((long) (Math.random() * 10));
                    System.out.printf("【%s】：离开了车位\n",
Thread.currentThread().getName());
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    // 释放车位
                    semaphore.release();
                }
            }, "车辆" + i).start();
        }
    }
```

读写锁

读锁：读取数据不会对原数据产生影响，即多次读取的数据都是相同的。因此，对于读取操作可以不用加锁，或者是加共享锁。
写锁：写操作会改动原数据，会导致多次读取的数据不一致，或者是只修改了部分数据，即脏数据。因此，对于写操作，使用互斥锁，保证单次写操作的原子性。

读锁和不加锁：读锁是共享锁，既然多次读写不存在并发数据问题，那为什么还要加读锁呢？首先，加了读锁，其他加读锁的线程仍然可以获取锁，这是共享锁的特性。但是其他加写锁的线程这时是无法获取到锁的，即保证了读锁区间的代码在这一次操作中不会受到其他线程写操作的影响。

锁降级：同一个线程中，在获取了写锁的情况下，仍然可以获取读锁。1.获取写锁；2.获取读锁；3.释放写锁。以上3执行完后锁将降级为读锁，之所以说是锁的降级，是因为读锁是共享锁，其他线程现在就能再获取读锁了。

锁不能升级：在同一个线程中，首先获取了读锁，再获取写锁时，程序将会一直等待获取写锁，直到读锁释放。