二

55 Condition、object.wait() 和 notify() 的关系?

本课时我们主要介绍 Condition、Object 的 wait() 和 notify() 的关系。

下面先讲一下 Condition 这个接口，来看看它的作用、如何使用，以及需要注意的点有哪些。

Condition接口

作用

我们假设线程 1 需要等待某些条件满足后，才能继续运行，这个条件会根据业务场景不同，有不同的可能性，比如等待某个时间点到达或者等待某些任务处理完毕。在这种情况下，我们就可以执行 Condition 的 await 方法，一旦执行了该方法，这个线程就会进入 WAITING 状态。

通常会有另外一个线程，我们把它称作线程 2，它去达成对应的条件，直到这个条件达成之后，那么，线程 2 调用 Condition 的 signal 方法 [或 signalAll 方法]，代表“**这个条件已经达成了，之前等待这个条件的线程现在可以苏醒了**”。这个时候，JVM 就会找到等待该 Condition 的线程，并予以唤醒，根据调用的是 signal 方法或 signalAll 方法，会唤醒 1 个或所有的线程。于是，线程 1 在此时就会被唤醒，然后它的线程状态又会回到 Runnable 可执行状态。

代码案例

我们用一个代码来说明这个问题，如下所示：

```
public class ConditionDemo {

    private ReentrantLock lock = new ReentrantLock();

    private Condition condition = lock.newCondition();

    void method1() throws InterruptedException {

        lock.lock();
```

```
        try{
            System.out.println(Thread.currentThread().getName()+":条件不满足，开始aw
            condition.await();
            System.out.println(Thread.currentThread().getName()+":条件满足了，开始执行
        }finally {
            lock.unlock();
        }
    }
    void method2() throws InterruptedException {
        lock.lock();
        try{
            System.out.println(Thread.currentThread().getName()+":需要5秒钟的准备时间
            Thread.sleep(5000);
            System.out.println(Thread.currentThread().getName()+":准备工作完成，唤醒其
            condition.signal();
        }finally {
            lock.unlock();
        }
    }
    public static void main(String[] args) throws InterruptedException {
        ConditionDemo conditionDemo = new ConditionDemo();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    conditionDemo.method2();
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
            }
        }).start();
        conditionDemo.method1();
    }
}
```

在这个代码中，有以下三个方法。

- **method1**，它代表主线程将要执行的内容，首先获取到锁，打印出“条件不满足，开始 await”，然后调用 condition.await() 方法，直到条件满足之后，则代表这个语句可以继续向下执行了，于是打印出“条件满足了，开始执行后续的任务”，最后会在 finally 中解锁。
- **method2**，它同样也需要先获得锁，然后打印出“需要 5 秒钟的准备时间”，接着用 sleep 来模拟准备时间；在时间到了之后，则打印出“准备工作完成”，最后调用 condition.signal() 方法，把之前已经等待的线程唤醒。
- **main 方法**，它的主要作用是执行上面这两个方法，它先去实例化我们这个类，然后再用子线程去调用这个类的 method2 方法，接着用主线程去调用 method1 方法。

最终这个代码程序运行结果如下所示：

```
main:条件不满足，开始 await
Thread-0:需要 5 秒钟的准备时间
Thread-0:准备工作完成，唤醒其他的线程
main:条件满足了，开始执行后续的任务
```

同时也可以看到，打印这行语句它所运行的线程，第一行语句和第四行语句打印的是在 main 线程中，也就是在主线程中去打印的，而第二、第三行是在子线程中打印的。这个代码就模拟了我们前面所描述的场景。

注意点

下面我们来看一下，在使用 Condition 的时候有哪些注意点。

- 线程 2 解锁后，线程 1 才能获得锁并继续执行

线程 2 对应刚才代码中的子线程，而线程 1 对应主线程。这里需要额外注意，并不是说子线程调用了 signal 之后，主线程就可以立刻被唤醒去执行下面的代码了，而是说在调用了 signal 之后，还需要等待子线程完全退出这个锁，即执行 unlock 之后，这个主线程才有可能去获取到这把锁，并且当获取锁成功之后才能继续执行后面的任务。刚被唤醒的时候主线程还没有拿到锁，是没有办法继续往下执行的。

- signalAll() 和 signal() 区别

signalAll() 会唤醒所有正在等待的线程，而 signal() 只会唤醒一个线程。

用 Condition 和 wait/notify 实现简易版阻塞队列

在第 05 讲，讲过如何用 Condition 和 wait/notify 来实现生产者/消费者模式，其中的精髓就在于用 Condition 和 wait/notify 来实现简易版阻塞队列，我们来分别回顾一下这两段代码。

用 Condition 实现简易版阻塞队列

代码如下所示：

```
public class MyBlockingQueueForCondition {

    private Queue queue;

    private int max = 16;

    private ReentrantLock lock = new ReentrantLock();

    private Condition notEmpty = lock.newCondition();

    private Condition notFull = lock.newCondition();

    public MyBlockingQueueForCondition(int size) {

        this.max = size;

        queue = new LinkedList();

    }

    public void put(Object o) throws InterruptedException {

        lock.lock();

        try {

            while (queue.size() == max) {

                notFull.await();
```

```
            }
            queue.add(o);
            notEmpty.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public Object take() throws InterruptedException {
        lock.lock();
        try {
            while (queue.size() == 0) {
                notEmpty.await();
            }
            Object item = queue.remove();
            notFull.signalAll();
            return item;
        } finally {
            lock.unlock();
        }
    }
}
```

在上面的代码中，首先定义了一个队列变量 queue，其最大容量是 16；然后定义了一个 ReentrantLock 类型的 Lock 锁，并在 Lock 锁的基础上创建了两个 Condition，一个是 notEmpty，另一个是 notFull，分别代表队列没有空和没有满的条件；最后，声明了 put 和 take 这两个核心方法。

用 wait/notify 实现简易版阻塞队列

我们再来看看如何使用 wait/notify 来实现简易版阻塞队列，代码如下：

```
class MyBlockingQueueForWaitNotify {

    private int maxSize;

    private LinkedList<Object> storage;

    public MyBlockingQueueForWaitNotify (int size) {

        this.maxSize = size;

        storage = new LinkedList<>();

    }

    public synchronized void put() throws InterruptedException {

        while (storage.size() == maxSize) {

            this.wait();

        }

        storage.add(new Object());

        this.notifyAll();

    }

    public synchronized void take() throws InterruptedException {

        while (storage.size() == 0) {

            this.wait();

        }

        System.out.println(storage.remove());

        this.notifyAll();

    }

}
```

如代码所示，最主要的部分仍是 put 与 take 方法。我们先来看 put 方法，该方法被 synchronized 保护，while 检查 List 是否已满，如果不满就往里面放入数据，并通过 notifyAll() 唤醒其他线程。同样，take 方法也被 synchronized 修饰，while 检查 List 是否为空，如果不为空则获取数据并唤醒其他线程。

在第 05 讲，有对这两段代码的详细讲解，遗忘的小伙伴可以到前面复习一下。

Condition 和 wait/notify的关系

对比上面两种实现方式的 put 方法，会发现非常类似，此时让我们把这两段代码同时列在屏幕中，然后进行对比：

左：

```
public void put(Object o) throws InterruptedException {
   lock.lock();
   try {
      while (queue.size() == max) {
         condition1.await();
      }
      queue.add(o);
      condition2.signalAll();
   } finally {
      lock.unlock();
   }
}
```

右：

```
public synchronized void put() throws InterruptedException {
   while (storage.size() == maxSize) {
      this.wait();
   }
   storage.add(new Object());
   this.notifyAll();
}
```

可以看出，左侧是 Condition 的实现，右侧是 wait/notify 的实现：

```
lock.lock() 对应进入 synchronized 方法
```

```
condition.await() 对应 object.wait()

condition.signalAll() 对应 object.notifyAll()

lock.unlock() 对应退出 synchronized 方法
```

实际上，如果说 Lock 是用来代替 synchronized 的，那么 Condition 就是用来代替相对应的 Object 的 wait/notify/notifyAll，所以在用法和性质上几乎都一样。

Condition 把 Object 的 wait/notify/notifyAll 转化为了一种相应的对象，其实现的效果基本一样，但是把更复杂的用法，变成了更直观可控的对象方法，是一种升级。

await 方法会自动释放持有的 Lock 锁，和 Object 的 wait 一样，不需要自己手动释放锁。

另外，调用 await 的时候必须持有锁，否则会抛出异常，这一点和 Object 的 wait 一样。

总结

首先介绍了 Condition 接口的作用，并给出了基本用法；然后讲解了它的几个注意点，复习了之前 Condition 和 wait/notify 实现简易版阻塞队列的代码，并且对这两种方法，不同的实现进行了对比；最后分析了它们之间的关系。

上一页 下一页