二

37 阻塞和非阻塞队列的并发安全原理是什么?

本课时我们主要研究阻塞和非阻塞队列的并发安全原理。

之前我们探究了常见的阻塞队列的特点，在本课时，我们以 ArrayBlockingQueue 为例，首先分析 BlockingQueue 即阻塞队列的线程安全原理，然后再看看它的兄弟——非阻塞队列的并发安全原理。通过本课时的学习，我们就可以了解到关于并发队列的底层原理了。

ArrayBlockingQueue 源码分析

我们首先看一下 ArrayBlockingQueue 的源码，ArrayBlockingQueue 有以下几个重要的属性：

```
// 用于存放元素的数组
final Object[] items;
// 下一次读取操作的位置
int takeIndex;
// 下一次写入操作的位置
int putIndex;
// 队列中的元素数量
int count;
```

第一个就是最核心的、用于存储元素的 Object 类型的数组；然后它还会有两个位置变量，分别是 takeIndex 和 putIndex，这两个变量就是用来标明下一次读取和写入位置的；另外还有一个 count 用来计数，它所记录的就是队列中的元素个数。

另外，我们再来看下面这三个变量：

```
// 以下3个是控制并发用的工具
final ReentrantLock lock;
```

```
private final Condition notEmpty;

private final Condition notFull;
```

这三个变量也非常关键，第一个就是一个 ReentrantLock，而下面两个 Condition 分别是由 ReentrantLock 产生出来的，这三个变量就是我们实现线程安全最核心的工具。

ArrayBlockingQueue 实现并发同步的原理就是利用 ReentrantLock 和它的两个 Condition，读操作和写操作都需要先获取到 ReentrantLock 独占锁才能进行下一步操作。进行读操作时如果队列为空，线程就会进入到读线程专属的 notEmpty 的 Condition 的队列中去排队，等待写线程写入新的元素；同理，如果队列已满，这个时候写操作的线程会进入到写线程专属的 notFull 队列中去排队，等待读线程将队列元素移除并腾出空间。

下面，我们来分析一下最重要的 put 方法：

```
public void put(E e) throws InterruptedException {

    checkNotNull(e);

    final ReentrantLock lock = this.lock;

    lock.lockInterruptibly();

    try {

        while (count == items.length)

        notFull.await();

        enqueue(e);

    } finally {

        lock.unlock();

    }

}
```

在 put 方法中，首先用 checkNotNull 方法去检查插入的元素是不是 null。如果不是 null，我们会用 ReentrantLock 上锁，并且上锁方法是 lock.lockInterruptibly()。这个方法我们在第 23 课时的时候讲过，在获取锁的同时是可以响应中断的，这也正是我们的阻塞队列在调用 put 方法时，在尝试获取锁但还没拿到锁的期间可以响应中断的底层原因。

紧接着，是一个非常经典的 try finally 代码块，finally 中会去解锁，try 中会有一个 while 循环，它会检查当前队列是不是已经满了，也就是 count 是否等于数组的长度。如果等于就

代表已经满了，于是我们便会进行等待，直到有空余的时候，我们才会执行下一步操作，调用 enqueue 方法让元素进入队列，最后用 unlock 方法解锁。

你看到这段代码不知道是否眼熟，在第 5 课时我们讲过，用 Condition 实现生产者/消费者模式的时候，写过一个 put 方法，代码如下：

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

可以看出，这两个方法几乎是一模一样的，所以当时在第 5 课时的时候我们就说过，我们自己用 Condition 实现生产者/消费者模式，实际上其本质就是自己实现了简易版的 BlockingQueue。你可以对比一下这两个 put 方法的实现，这样对 Condition 的理解就会更加深刻。

和 ArrayBlockingQueue 类似，其他各种阻塞队列如 LinkedBlockingQueue、PriorityBlockingQueue、DelayQueue、DelayedWorkQueue 等一系列 BlockingQueue 的内部也是利用了 ReentrantLock 来保证线程安全，只不过细节有差异，比如 LinkedBlockingQueue 的内部有两把锁，分别锁住队列的头和尾，比共用同一把锁的效率更高，不过总体思想都是类似的。

非阻塞队列ConcurrentLinkedQueue

看完阻塞队列之后，我们就来看看非阻塞队列 ConcurrentLinkedQueue。顾名思义，ConcurrentLinkedQueue 是使用链表作为其数据结构的，我们来看一下关键方法 offer 的源码：

```
public boolean offer(E e) {
    checkNotNull(e);
    final Node<E> newNode = new Node<E>(e);
    for (Node<E> t = tail, p = t;;) {
        Node<E> q = p.next;
        if (q == null) {
            // p is last node
            if (p.casNext(null, newNode)) {
                // Successful CAS is the linearization point
                // for e to become an element of this queue,
                // and for newNode to become "live".
                if (p != t) // hop two nodes at a time
                    casTail(t, newNode);  // Failure is OK.
                return true;
            }
            // Lost CAS race to another thread; re-read next
        }
        else if (p == q)
            // We have fallen off list.  If tail is unchanged, it
            // will also be off-list, in which case we need to
            // jump to head, from which all live nodes are always
            // reachable.  Else the new tail is a better bet.
            p = (t != (t = tail)) ? t : head;
        else
            // Check for tail updates after two hops.
            p = (p != t && t != (t = tail)) ? t : q;
    }
}
```

在这里我们不去一行一行分析具体的内容，而是把目光放到整体的代码结构上，在检查完空判断之后，可以看到它整个是一个大的 for 循环，而且是一个非常明显的死循环。在这个循环中有一个非常亮眼的 p.casNext 方法，这个方法正是利用了 CAS 来操作的，而且这个死循环去配合 CAS 也就是典型的乐观锁的思想。我们就来看一下 p.casNext 方法的具体实现，其方法代码如下：

```
boolean casNext(Node<E> cmp, Node<E> val) {
    return UNSAFE.compareAndSwapObject(this, nextOffset, cmp, val);
}
```

可以看出这里运用了 UNSAFE.compareAndSwapObject 方法来完成 CAS 操作，而 compareAndSwapObject 是一个 native 方法，最终会利用 CPU 的 CAS 指令保证其不可中断。

可以看出，非阻塞队列 ConcurrentLinkedQueue 使用 CAS 非阻塞算法 + 不停重试，来实现线程安全，适合用在不需要阻塞功能，且并发不是特别剧烈的场景。

总结

最后我们来做一下总结。本课时我们分析了阻塞队列和非阻塞队列的并发安全原理，其中阻塞队列最主要是利用了 ReentrantLock 以及它的 Condition 来实现，而非阻塞队列则是利用 CAS 方法实现线程安全。

上一页 下一页