

26不让我进门，我就在门口一直等！—BlockingQueue和ArrayBlockingQueue

更新时间：2019-11-26 10:24:24



“读书而不思考，等于吃饭而不消化。

——波尔克”

前面两节我们对 `ConcurrentHashMap` 的源代码进行了分析，是不是意犹未尽？那么做好准备，本小节，我们将对阻塞队列的借口 `BlockingQueue` 以及它的一个实现 `ArrayBlockingQueue` 做源码分析。说实话，源码分析十分的枯燥和艰难，不过坚持下来，收获将会很大。

1、BlockingQueue 介绍

`BlockingQueue` 顾名思义----阻塞队列。队列大家应该都很清楚了，阻塞队列是指当队列满时，入队操作需要等待；当队列空时，出队操作需要等待。而非阻塞的方式则会直接返回`false`。

`BlockingQueue` 是一个接口。定义了出队和入队的方法。

我们首先看入队的方法：

1、boolean add(E e)

该方法向队列添加元素`e`，如果有空间添加，那么添加成功返回 `true`。如果没有空间，那么直接抛出 `IllegalStateException`。

2、boolean offer(E e)

该方法向队列添加元素 `e`，如果有空间添加，那么添加成功返回 `true`。如果没有空间，则会返回 `false`。和 `add` 方法很像，只不过不是抛异常。

3、boolean offer(E e, long timeout, TimeUnit unit) throws InterruptedException

该方法向队列添加元素 `e`，如果有空间添加，那么添加成功返回 `true`。如果没有空间，则会等待一定时长，如果仍旧没有空间则返回 `false`。

4、void put(E e) throws InterruptedException

该方法向队列添加元素 `e`，如果成功立即返回，如果没有空间，则一直等待空间。等待是可被直接中断。

这四种入队的方法中前两种为非阻塞方式，后两种为阻塞方式。

接下来我们再来分析出队的方法：

1、E take() throws InterruptedException

返回并且从队列中移除 `head` 元素。如果队列为空，则会阻塞等待直到元素入队。

2、E poll(long timeout, TimeUnit unit)

返回并且从队列中移除 `head` 元素。如果队列为空，则会阻塞一定的时间，等待有元素入队。如果等待时间内没有元素入队，那么返回 `null`。

以上两种出队方法均为阻塞方法。

还有几个其它的方法，也做下介绍：

1、boolean contains(Object o)

如果队列中至少含有一个元素 `equals` 对象 `o`，那么返回 `true`。

2、int drainTo(Collection<? super E> c)和int drainTo(Collection<? super E> c, int maxElements)

第一个方法是一次性“榨干”队列，把所有元素 `remove` 掉，放入集合 `c` 中。第二个方法的区别是每次 `remove` 掉 `maxElements` 个元素，放入集合 `c`。

3、int remainingCapacity()

返回列表的剩余容量。

我们通过以下表格对出入队方法进行总结：

入队：

方法名	是否阻塞	抛出异常
<code>add(E e)</code>	否	<code>IllegalStateException</code>
<code>offer(E e)</code>	否	无。返回 <code>false</code>
<code>offer(E e, long timeout, TimeUnit unit)</code>	阻塞指定时长	<code>InterruptedException</code>
<code>put(E e)</code>	是	<code>InterruptedException</code>

出队：

方法名	是否阻塞	抛出异常
take()	是	InterruptedException
poll(long timeout, TimeUnit unit)	阻塞指定时长，超时返回null	InterruptedException

2、ArrayBlockingQueue源码分析

ArrayBlockingQueue 是 BlockingQueue 的一种实现。底层通过数组实现。ArrayBlockingQueue 支持公平和非公平的方式来入队和出队。公平是指当出现多个线程阻塞时，等待时间长的会先获得锁，非公平则不一定。

我们先看 ArrayBlockingQueue 的构造函数，有如下三个

1. public ArrayBlockingQueue(int capacity);
2. public ArrayBlockingQueue(int capacity, boolean fair);
3. public ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c) 。

capacity 为队列的大小，一旦初始化了无法改变。fair 指锁的类型，c 用来初始化队列时设置初始元素。

ArrayBlockingQueue 通过两个 Condition 信号量来控制出队和入队的阻塞，分别为 notEmpty 和 notFull。

接下来，我们先分析入队方法 put 的源代码。

2.1 put 方法源码分析

```
public void put(E e) throws InterruptedException {
    //检查e是否为null，如果为null，抛出NullPointerException
    checkNotNull(e);
    //声明显式锁
    final ReentrantLock lock = this.lock;
    //以lockInterruptibly方式上锁，如果其他线程打断等待的线程，那么等待的线程会立刻终止等待，抛出InterruptedException
    lock.lockInterruptibly();
    try {
        //如果队列已经满了，那么等待
        while (count == items.length)
            notFull.await();
        //直到消费者消费了一个元素后，通过notFull.signal()来通知等待的线程添加元素
        enqueue(e);
    } finally {
        //释放锁
        lock.unlock();
    }
}
```

enqueue 中实现元素的入队操作，代码如下：

```
private void enqueue(E x) {
    final Object[] items = this.items;
    items[putIndex] = x;
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
    notEmpty.signal();
}
```

putIndex 记录了当前可以写入的数组下标。

count 是 queue 中保存的元素总数。

这段代码中，第 4、5 行不太好理解。当 `++putIndex` 和数组长度一样时，说明到了数组的最后一个元素。然后 `putIndex` 被置为0。这是因为这里循环使用数组，由于数组长度就是队列的长度，并且出一个才能进一个，所以进到数组最后一位时，下一个等待出队的位置肯定是 `>=0`。此时循环使用数组，下一个 `put` 的位置转到数组第一位。如果消费得比较快，消费的位置 `>0`，那么没问题，再次 `put` 时候就会写入 `index=0` 的位置。如果恰巧出队位置就是 0。那么下次 `put` 的时候，由于 `count == items.length`，而会进入等待。直到位置 0 的元素被消费掉，那么写入 0 位置，也不会有任何问题。

这么做的好处是循环使用数组，而不需要每次消费都向前移动数组中元素的位置。

2.2 offer 方法和 add 方法源码分析

下面我们再看看 `offer` 的源代码：

```
public boolean offer(E e) {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        if (count == items.length)
            return false;
        else {
            enqueue(e);
            return true;
        }
    } finally {
        lock.unlock();
    }
}
```

区别就是使用 `lock`，在等待锁的期间不能被打断。另外如果 `count == items.length`，队列满了直接返回 `false`，而不是阻塞等待。

这里一并讲解 `add` 方法：

```
public boolean add(E e) {
    if (offer(e))
        return true;
    else
        throw new IllegalStateException("Queue full");
}
```

其实 `add` 方法中调用的 `offer` 方法，只不过 `add` 在 `offer` 返回 `false` 时，抛出异常。

2.3 有等待超时参数的 offer 方法源码分析

```

public boolean offer(E e, long timeout, TimeUnit unit)
    throws InterruptedException {

    checkNotNull(e);
    long nanos = unit.toNanos(timeout);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length) {
            if (nanos <= 0)
                return false;
            nanos = notFull.awaitNanos(nanos);
        }
        enqueue(e);
        return true;
    } finally {
        lock.unlock();
    }
}

```

其实这个方法和 `put` 更像一点，`put` 阻塞直到有空位出现。而此方法阻塞时会有超时时间设置，`notFull.awaitNanos(nanos)` 返回剩余的超时时长，当 `nanos <= 0`，也就是说没有时长了，如果还没有等到空位，那么也会放回 `false`。

以上入队的方法已经讲完了，接下来我们看一下出队的方法，出队方法的区别其实和入队类似，所以这里我不再详细讲解每一个，我们看一个典型就可以了。

2.4 take 方法源代码分析

`take` 方法和 `put` 方法相对应，代码如下：

```

public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}

```

可以看到，和 `put` 一样都是使用的 `lockInterruptibly` 方法上锁，可以被中断。当 `count` 为0时，说明队列已经空了，无法取出元素，那么通过调用 `notEmpty.await()` 阻塞。直到某个入队方法添加元素后调用了 `notEmpty.signal()`，通知该线程可以继续出队操作了。

`poll()` 方法不会阻塞，如果没有元素，则直接返回 `null`。

`poll(long timeout, TimeUnit unit)`，则会阻塞一定时长，如果还是没有元素，才会返回 `null`。

3、总结

关于 `ArrayBlockingQueue` 的源代码就讲解到这里，可以看出源代码并不复杂，而且不同方法的实现也很类似，只有细微的差别。如果大家感兴趣，相信你轻松就能读懂源代码。另外 `BlockingQueue` 的实现还有很多，还有个比较常用的是 `LinkedBlockingQueue`，通过链表存储来实现，如果感兴趣也可以自己去看源代码。

阻塞队列用来实现生产者和消费者很好用。另外我们还应该知道在 **Executor** 中使用了 **BlockingQueue**。大家如果已经忘了，可以回过头再去看看 **Executor** 源码分析那一节的内容。

}

