0724分享

BlockingQueue

why，how，what

我们知道队列是先进先出的。当放入一个元素的时候，会放在队列的末尾，取出元素的时候，会从队头取。那么，当队列为空或者队列满时又想放入或取得元素怎么办，这时候就可以使用阻塞队列。如果不使用阻塞队列就需要自己写逻辑自己 notify 自己wait，但这样会有很多问题，那么当我们有这样的需求时，直接使用阻塞队列就好了。

为什么是说**阻塞(Blocking)**，当获取队列元素但是队列为空时，会阻塞等待队列中有元素再返回；也支持添加元素时，如果队列已满，那么等到队列可以放入新元素时再放入。

how，what看后面的内容可以知道。

Doug Lea大佬写的Java doc机翻后编辑

BlockingQueue 是一个先进先出的队列（Queue），当获取队列元素但是队列为空时，会阻塞等待队列中有元素再返回；也支持添加元素时，如果队列已满，那么等到队列可以放入新元素时再放入。

BlockingQueue方法有四种形式，处理不能立即满足但可能在未来某个时候满足的操作的方式各不相同：一个抛出异常，第二个返回一个特殊值（ null或false ，取决于操作），第三个无限期地阻塞当前线程，直到操作成功，第四个阻塞仅给定的最大时间限制，然后放弃。 这些方法总结在下表中：

	抛出异常	**特殊值**	**阻塞**	**超时**
插入	add(E e)	offer(E e)	put(E e)	offer(E e, long timeout, TimeUnit unit)
移除	remove()	poll()	take()	poll(long timeout, TimeUnit unit)
检查	element()	peek()		

> 重点 **put(e)** 和 **take()** 带阻塞这两个方法。

- **BlockingQueue**不接受**null**元素。 实现在尝试add 、 put或offer null抛出NullPointerException 。 null值用作标记值以指示poll操作失败。允许null就不能判断是取值失败还是值为null。
- **BlockingQueue**可能有容量限制。 在任何给定的时间，它可能有一个remainingCapacity容量，超过该容量就不能put不阻塞的情况下put其他元素。 不限制容量也有最大容量**Integer.MAX_VALUE**。
- **BlockingQueue**实现主要用于**生产者-消费者队列**，但另外还支持**Collection**接口。 因此，例如，可以使用remove(x)从队列中remove(x)任意元素。 然而，这样的操作通常执行得不是很有效，并且仅用于偶尔使用，例如当排队的消息被取消时。
- **BlockingQueue**实现是**线程安全**的。 所有排队方法都使用内部锁或其他形式的并发控制以原子方式实现其效果。 然而批量的操作addAll ， containsAll ， retainAll和removeAll不一定原子除非在实现中另有规定执行。 因此，例如，在仅添加c某些元素后， addAll(c)可能会失败（抛出异常）。
- **BlockingQueue**本质上不支持任何类型的“关闭”或“关闭”操作以指示不再添加项目。 此类功能的需求和使用往往取决于实现。 例如，一种常见的策略是让生产者插入特殊的对象，当消费者获取到特殊的对象执行特定操作。

Condition

阻塞队列实现阻塞就是依靠ReentrantLock和Condition的使用

Condition本身是个接口。Condition的java doc里面有这么一句话和示例代码

> Condition实例本质上绑定到锁。 要获取特定Lock实例的Condition实例，请使用其newCondition()方法

```
class BoundedBuffer<E> {
     final Lock lock = new ReentrantLock();
     final Condition notFull  = lock.newCondition();
     final Condition notEmpty = lock.newCondition();

     final Object[] items = new Object[100];
     int putptr, takeptr, count;

     public void put(E x) throws InterruptedException {
       lock.lock();
       try {
         while (count == items.length)
           notFull.await();
         items[putptr] = x;
         if (++putptr == items.length) putptr = 0;
         ++count;
         notEmpty.signal();
       } finally {
         lock.unlock();
       }
     }

     public E take() throws InterruptedException {
       lock.lock();
       try {
         while (count == 0)
           notEmpty.await();
         E x = (E) items[takeptr];
         if (++takeptr == items.length) takeptr = 0;
         --count;
         notFull.signal();
         return x;
       } finally {
         lock.unlock();
       }
     }
   }
```

可以知道使用condition必须先持有相应的锁，condition 是依赖于 ReentrantLock 的，不管是调用 await 进入等待还是 signal 唤醒，**都必须获取到锁才能进行操作。**

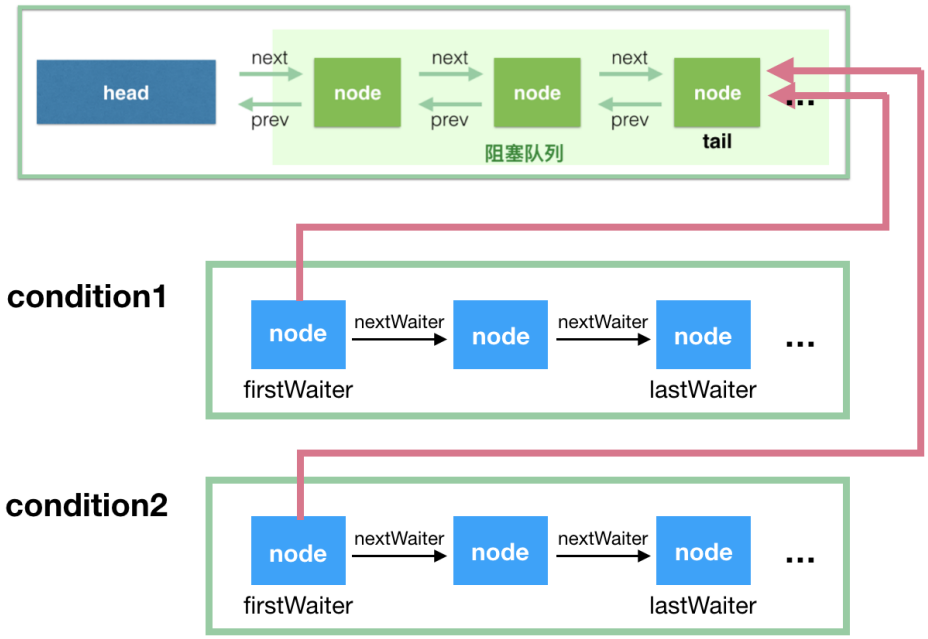
每个 ReentrantLock 实例可以通过调用多次 newCondition 产生多个 ConditionObject 的实例：

```
final ConditionObject newCondition() {
    // 实例化一个 ConditionObject
    return new ConditionObject();
}
```

ConditionObject是AbstractQueuedSynchronizer中的内部类，实现了Condition接口

```
public class ConditionObject implements Condition, java.io.Serializable {
        private static final long serialVersionUID = 1173984872572414699L;
        // 条件队列的第一个节点
          // 不要管这里的关键字 transient，是不参与序列化的意思
        private transient Node firstWaiter;
        // 条件队列的最后一个节点
        private transient Node lastWaiter;
        ......
```

和AbstractQueuedSynchronizer中队列的关系



...

主要方法

```
 public final void await() throws InterruptedException {
            //先判断中断状态
            if (Thread.interrupted())
                throw new InterruptedException();
            //把当前线程以node添加到当前condition的条件队列
            Node node = addConditionWaiter();
            //完全释放锁，返回之前的savedState
            int savedState = fullyRelease(node);
            int interruptMode = 0;
            // 1. isOnSyncQueue(node) 返回 true，即当前 node 已经转移到阻塞队列了
            // 2. checkInterruptWhileWaiting(node) != 0 会到 break，然后退出循环，代
表的是线程中断
            while (!isOnSyncQueue(node)) {
                //挂起当前线程，当别的线程调用了signal()，并且是当前线程被唤醒的时候才从
park()方法返回
                LockSupport.park(this);
                // 1. 如果在 signal 之前已经中断，返回 THROW_IE
```

```
                // 2. 如果是 signal 之后中断，返回 REINTERRUPT
                // 3. 没有发生中断，返回 0
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                    break;
            }
            //唤醒后再次获取锁，失败后进入阻塞队列  state 为释放锁之前的savedState
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null) // signal之前中断的处理
                unlinkCancelledWaiters();
            if (interruptMode != 0)
                reportInterruptAfterWait(interruptMode); // 处理中断状态
        }

public final void signal() {
            // 调用 signal 方法的线程必须持有当前的独占锁
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            Node first = firstWaiter;
            if (first != null)
                doSignal(first);
        }
```

BlockingQueue的实现

ArrayBlockingQueue

由数组支持的有界阻塞队列。 该队列对元素 FIFO（先进先出）进行排序。 队列的头部是在队列中停留时间最长的那个元素。 队列的尾部是在队列中停留时间最短的那个元素。 新元素插入队列尾部，队列检索操作获取队列头部元素。

这是一个经典的“有界缓冲区”，其中一个**固定大小的数组**保存由生产者插入并由消费者提取的元素。 **容量一旦创建，就无法更改。** 尝试put元素放入已满队列将导致操作阻塞； 尝试从空队列中take元素也会类似地阻塞。

此类支持用于排序等待生产者和消费者线程的可选公平策略。 默认情况下，不保证此顺序。 但是，在公平性设置为true构造的队列以 FIFO 顺序授予线程访问权限。 公平通常会降低吞吐量，但会减少可变性并避免饥饿。

此类及其迭代器实现了Collection和Iterator接口的所有可选方法。

1. 可重入锁ReentrantLock，构造方法指定是否公平
2. 不可扩容，需指定容量
3. 一个锁，两个条件

```
public class ArrayBlockingQueue<E> extends AbstractQueue<E> implements
    BlockingQueue<E>, java.io.Serializable {
    //...
    final Object[] items;
    // 队头指针
    int takeIndex;
    // 队尾指针
    int putIndex;
    int count;
    // 核心为1个锁外加两个条件
    final ReentrantLock lock;
    private final Condition notEmpty;
```

```
    private final Condition notFull;
    //...
        public void put(E e) throws InterruptedException {
        Objects.requireNonNull(e);
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length)
                notFull.await();
            enqueue(e);
        } finally {
            lock.unlock();
        }
    }

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

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

        private E dequeue() {
        final Object[] items = this.items;
        @SuppressWarnings("unchecked")
        E e = (E) items[takeIndex];
        items[takeIndex] = null;
        if (++takeIndex == items.length) takeIndex = 0;
        count--;
        if (itrs != null)
            itrs.elementDequeued();
        notFull.signal();
        return e;
    }
}
```

LinkedBlockingQueue

基于链接节点的可选有界阻塞队列。 该队列对元素 FIFO（先进先出）进行排序。 队列的头部是在队列中停留时间最长的那个元素。 队列的尾部是在队列中停留时间最短的那个元素。 新元素插入队列尾部，队列检索操作获取队列头部元素。 链接队列通常比基于数组的队列具有更高的吞吐量，但在大多数并发应用程序中的可预测性能较差。

可选的容量绑定构造函数参数是一种防止队列过度扩展的方法。 如果未指定，容量等于 **Integer.MAX_VALUE** 。 链接节点在每次插入时动态创建，除非这会使队列超出容量。

此类及其迭代器实现了Collection和Iterator接口的所有可选方法。

1. 可重入锁ReentrantLock
2. 不可扩容，未指定容量默认Integer.MAX_VALUE
3. 两个锁，两个条件

```
public class LinkedBlockingQueue<E> extends AbstractQueue<E> implements
    BlockingQueue<E>, java.io.Serializable {
    // ...
    private final int capacity;
    // 原子变量
    private final AtomicInteger count = new AtomicInteger(0);
    // 单向链表的头部
    private transient Node<E> head;
    // 单向链表的尾部
    private transient Node<E> last;
    // 两把锁，两个条件
    private final ReentrantLock takeLock = new ReentrantLock();
    private final Condition notEmpty = takeLock.newCondition();
    private final ReentrantLock putLock = new ReentrantLock();
    private final Condition notFUll = putLock.newCondition();
    // ...

    public void put(E e) throws InterruptedException {
        if (e == null) throw new NullPointerException();
        final int c;
        final Node<E> node = new Node<E>(e);
        final ReentrantLock putLock = this.putLock;
        final AtomicInteger count = this.count;
        putLock.lockInterruptibly();
        try {
            while (count.get() == capacity) {
                notFull.await();
            }
            enqueue(node);
            c = count.getAndIncrement();
            if (c + 1 < capacity)
                notFull.signal();
        } finally {
            putLock.unlock();
        }
        if (c == 0)
            signalNotEmpty();
    }

    public E take() throws InterruptedException {
        final E x;
        final int c;
        final AtomicInteger count = this.count;
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lockInterruptibly();
        try {
            while (count.get() == 0) {
                notEmpty.await();
```

```
            }
            x = dequeue();
            c = count.getAndDecrement();
            if (c > 1)
                notEmpty.signal();
        } finally {
            takeLock.unlock();
        }
        if (c == capacity)
            signalNotFull();
        return x;
    }
}
```

PriorityBlockingQueue

一个**无界阻塞队列**，它使用与类PriorityQueue相同的排序规则并提供阻塞检索操作。 虽然这个队列在逻辑上是无界的，但由于资源耗尽（导致OutOfMemoryError ），尝试添加可能会失败。 此类不允许null元素。 依赖于自然排序的优先级队列也不允许插入不可比较的对象（这样做会导致ClassCastException ）。

此类及其迭代器实现了Collection和Iterator接口的所有可选方法。 方法iterator()提供的 Iterator不能保证以任何特定顺序遍历 PriorityBlockingQueue 的元素。 如果您需要有序遍历，请考虑使用Arrays.sort(pq.toArray()) 。 此外， drainTo方法可用于按优先级顺序删除部分或所有元素，并将它们放置在另一个集合中。

对此类的操作不保证具有相同优先级的元素的顺序。 如果您需要强制排序，您可以定义自定义类或比较器，这些类或比较器使用辅助键来打破主要优先级值之间的联系。 例如，这是一个将先进先出打破平局应用于可比元素的类。 要使用它，您需要插入一个new FIFOEntry(anEntry)而不是普通的条目对象。

1. 可重入锁ReentrantLock
2. 有序，没有自定义比较操作，采用元素默认比较方式，最小二叉堆
3. 可扩容，默认值11
4. 没有notFull条件，超出容量时，进行扩容操作
5. 按优先级从小到大出队列

```
public class PriorityBlockingQueue<E> extends AbstractQueue<E> implements
    BlockingQueue<E>, java.io.Serializable {
    //...
    // 用数组实现的二插小根堆
    private transient Object[] queue;
    private transient int size;
    private transient Comparator<? super E> comparator;
    // 1个锁+一个条件，没有非满条件
    private final ReentrantLock lock;
    private final Condition notEmpty;
    //...

     public boolean offer(E e) {
        if (e == null)
            throw new NullPointerException();
        final ReentrantLock lock = this.lock;
        lock.lock();
        int n, cap;
```

```
        Object[] es;
        while ((n = size) >= (cap = (es = queue).length))
            tryGrow(es, cap);
        try {
            final Comparator<? super E> cmp;
            if ((cmp = comparator) == null)
                siftUpComparable(n, e, es);
            else
                siftUpUsingComparator(n, e, es, cmp);
            size = n + 1;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
        return true;
    }

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

DelayQueue

Delayed元素的**无界阻塞队列**，其中**一个元素只能在其延迟到期时被占用。** 队列的头部是过去延迟过期最远的那个Delayed元素。 如果没有延迟到期，则没有 head 并且poll将返回null 。 当元素的getDelay(TimeUnit.NANOSECONDS)方法返回小于或等于零的值时，就会发生过期。 尽管无法使用take或poll删除未过期的元素，但它们仍被视为普通元素。 例如， size方法返回过期和未过期元素的计数。 此队列不允许空元素。

此类及其迭代器实现了Collection和Iterator接口的所有可选方法。 方法iterator()提供的 Iterator不能保证以任何特定顺序遍历 DelayQueue 的元素。

1. 可重入锁ReentrantLock
2. 有序，按延迟时间，最小的在堆顶，也是最小二叉堆
3. 堆顶元素延迟实现小于等于0才会出队列，除了队列为空时也会阻塞
4. take时，如果有其他线程正在等待则无限期等待，否则为leader时阻塞有限时间，即堆顶元素延迟时间
5. put时，只有新元素在堆顶时，才会中途唤醒leader线程

```
public class DelayQueue<E extends Delayed> extends AbstractQueue<E> implements
    BlockingQueue<E> {
    // ...
    // 一把锁和一个非空条件
    private final transient ReentrantLock lock = new ReentrantLock();
```

```
    private final Condition available = lock.newCondition();
    // 优先级队列
    private final PriorityQueue<E> q = new PriorityQueue<E>();
    // ...

    public boolean offer(E e) {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            q.offer(e);
            if (q.peek() == e) {
                leader = null;
                available.signal();
            }
            return true;
        } finally {
            lock.unlock();
        }
    }

    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            for (;;) {
                E first = q.peek();
                if (first == null)
                    available.await();
                else {
                    long delay = first.getDelay(NANOSECONDS);
                    if (delay <= 0L)
                        return q.poll();
                    first = null; // don't retain ref while waiting
                    if (leader != null)
                        available.await();
                    else {
                        Thread thisThread = Thread.currentThread();
                        leader = thisThread;
                        try {
                            available.awaitNanos(delay);
                        } finally {
                            if (leader == thisThread)
                                leader = null;
                        }
                    }
                }
            }
        } finally {
            if (leader == null && q.peek() != null)
                available.signal();
            lock.unlock();
        }
    }
}
```

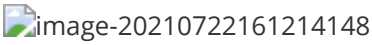
SynchronousQueue

一个阻塞队列，其中每个插入操作都必须等待另一个线程执行相应的移除操作，反之亦然。 同步队列没有任何内部容量，甚至没有容量。 您无法peek同步队列，因为元素仅在您尝试删除它时才存在； 你不能插入一个元素（使用任何方法），除非另一个线程试图删除它； 你不能迭代，因为没有什么可以迭代的。 队列的头部是第一个排队的插入线程试图添加到队列中的元素； 如果没有这样的排队线程，则没有元素可用于删除， poll()将返回null 。 对于其他Collection方法（例如contains ），SynchronousQueue充当空集合。 此队列不允许null元素。

同步队列类似于 CSP 和 Ada 中使用的集合通道。 它们非常适合切换设计，在这种设计中，在一个线程中运行的对象必须与在另一个线程中运行的对象同步，以便将某些信息、事件或任务交给它。

此类支持用于排序等待生产者和消费者线程的可选公平策略。 默认情况下，不保证此顺序。 但是，在公平性设置为true构造的队列以 FIFO 顺序授予线程访问权限。

此类及其迭代器实现了Collection和Iterator接口的所有可选方法。



1. 公平模式使用TransferQueue实现，非公平模式使用TransferStack实现。默认TransferStack。
2. 先调put(...)，线程会阻塞；直到另外一个线程调用了take()，两个线程才同时解锁，反之亦然。

```
public class SynchronousQueue<E> extends AbstractQueue<E> implements
    BlockingQueue<E>, java.io.Serializable {
    // ...
    static final class TransferQueue<E> extends Transferer<E> {
        static final class QNode {
            volatile QNode next;
            volatile Object item;
            volatile Thread waiter;
            final boolean isData;
            //...
        }
        transient volatile QNode head;
        transient volatile QNode tail;
        // ...
    }

    static final class TransferStack extends Transferer {
        static final int REQUEST = 0;
        static final int DATA = 1;
        static final int FULFILLING = 2;
        static final class SNode {
            volatile SNode next; // 单向链表
            volatile SNode match; // 配对的节点
            volatile Thread waiter; // 对应的阻塞线程
            Object item; int mode; // 三种模式
            //...
        }
        volatile SNode head;
    }
}
```

ConcurrentLinkedQueue

基于链接节点的无界线程安全队列。 该队列对元素 FIFO（先进先出）进行排序。 队列的头部是在队列中停留时间最长的那个元素。 队列的尾部是在队列中停留时间最短的那个元素。 新元素插入队列尾部，队列检索操作获取队列头部元素。 当许多线程将共享对公共集合的访问时， ConcurrentLinkedQueue是合适的选择。 与大多数其他并发集合实现一样，此类不允许使用null元素。

此实现采用了一种高效的非阻塞算法，该算法基于 Maged M. Michael 和 Michael L. Scott 在Simple、Fast、Practical Non-Blocking and Blocking Concurrent Queue Algorithms 中描述的算法 。

迭代器是弱一致的，返回元素反映了在迭代器创建时或之后的某个时刻的队列状态。 它们不会抛出java.util.ConcurrentModificationException ，并且可能与其他操作同时进行。 自迭代器创建以来队列中包含的元素将只返回一次。

请注意，与大多数集合不同， size方法不是恒定时间操作。 由于这些队列的异步性质，确定当前元素数量需要遍历元素，因此如果在遍历期间修改此集合，则可能会报告不准确的结果。

添加、删除或检查多个元素的批量操作（例如addAll 、 removeIf或forEach ）不能保证以原子方式执行。 例如，与addAll操作并发的forEach遍历可能只观察到一些添加的元素。

此类及其迭代器实现了Queue和Iterator接口的所有可选方法。

内存一致性影响：与其他并发集合一样，在将对象放入ConcurrentLinkedQueue之前线程中的操作发生在另一个线程中从ConcurrentLinkedQueue访问或删除该元素之后的操作之前。

```
public ConcurrentLinkedQueue() {
        head = tail = new Node<E>();
    }

public boolean offer(E e) {
        final Node<E> newNode = new Node<E>(Objects.requireNonNull(e));

        for (Node<E> t = tail, p = t; ; ) {
            Node<E> q = p.next;
            if (q == null) {
                // p 是最后一个节点
                if (NEXT.compareAndSet(p, null, newNode)) {//next节点如果为空，就把e
添加
                    // 成功的 CAS 是 e 成为该队列元素的线性化点，
                    // 以及 newNode 成为“活”的线性化点。
                    if (p != t) // 一次跳两个节点；失败是可以的
                        TAIL.weakCompareAndSet(this, t, newNode);
                    //tail节点如果内容是t  就  把tail的值赋为e  相当于移动到e(新节点)插入
的位置
                    return true;
                }
                //失去了到另一个线程的 CAST 竞赛；下次再读
            } else if (p == q)
                // 我们已经从名单上掉下来了。
                // 如果tail没有改变，它也会被off-list，
                // 在这种情况下我们需要跳转到head，
                // 所有存活的节点总是可以到达的。
                // 否则新的尾巴是更好的选择。
                p = (t != (t = tail)) ? t : head;
            else
                // 两跳后检查尾部更新。
                p = (p != t && t != (t = tail)) ? t : q;
        }
    }
```

```
public E poll() {
        restartFromHead:
        for (; ; ) {
            for (Node<E> h = head, p = h, q; ; p = q) {
                final E item;
                if ((item = p.item) != null && p.casItem(item, null)) {
                    // 成功的CAS是线性化点
                    // 用于要从此队列中删除的项目。
                    if (p != h) // 一次跳两个节点
                        updateHead(h, ((q = p.next) != null) ? q : p);
                    return item;
                } else if ((q = p.next) == null) {
                    updateHead(h, p);
                    return null;
                } else if (p == q)
                    continue restartFromHead;
            }
        }
    }
```

常见面试题

LinkedBlockingQueue相比于ArrayBlockingQueue的区别

1. 为了提高并发度，用2把锁，分别控制队头、队尾的操作。意味着在put(...)和put(...)之间、take()与take()之间是互斥的，put(...)和take()之间并不互斥。但对于count变量，双方都需要操作，所以必须是原子类型。
2. 因为各自拿了一把锁，所以当需要调用对方的condition的signal时，还必须再加上对方的锁，就是signalNotEmpty()和signalNotFull()方法。
3. 不仅put会通知 take，take 也会通知 put。当put 发现非满的时候，也会通知其他 put线程；当take发现非空的时候，也会通知其他take线程。