```
public boolean offer(E e) {
    //(1)e为null则抛出空指针异常
    checkNotNull(e);

   //(2)构造Node节点，在构造函数内部调用unsafe.putObject
    final Node<E> newNode = new Node<E>(e);


    //(3)从尾节点进行插入
    for (Node<E> t = tail, p = t;;) {

        Node<E> q = p.next;

        //(4)如果q==null说明p是尾节点，则执行插入
        if (q == null) {

            //(5)使用CAS设置p节点的next节点
            if (p.casNext(null, newNode)) {
                //(6)CAS成功，则说明新增节点已经被放入链表，然后设置当前尾节点(包含head，第

                    //1，3，5...个节点为尾节点)
                if (p != t)
                    casTail(t, newNode);  // Failure is OK.
                return true;
            }
        }
        else if (p == q)//(7)
            //多线程操作时，由于poll操作移除元素后可能会把head变为自引用，也就是head的next变

             //成了head，所以这里需要
            //重新找新的head
            p = (t != (t = tail)) ? t : head;
        else
            //(8) 寻找尾节点
            p = (p != t && t != (t = tail)) ? t : q;
    }
}
```

下面结合图来讲解该方法的执行流程。

（1）首先看当一个线程调用 offer（item）时的情况。首先代码（1）对传参进行空检查，如果为 null 则抛出 NPE 异常，否则执行代码（2）并使用 item 作为构造函数参数创建一个新的节点，然后代码（3）从队列尾部节点开始循环，打算从队列尾部添加元素，当执行到代码（4）时队列状态如图 7-2 所示。

仅供非商业用途或交流学习使用

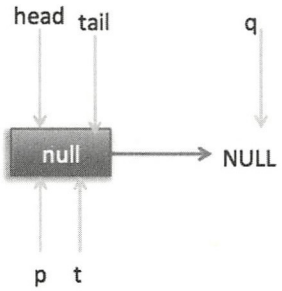


图 7-2

这时候节点 p、t、head、tail 同时指向了 item 为 null 的哨兵节点，由于哨兵节点的 next 节点为 null，所以这里 q 也指向 null。代码（4）发现 q==null 则执行代码（5），通过 CAS 原子操作判断 p 节点的 next 节点是否为 null，如果为 null 则使用节点 newNode 替换 p 的 next 节点，然后执行代码（6），这里由于 p==t 所以没有设置尾部节点，然后退出 offer 方法，这时候队列的状态如图 7-3 所示。

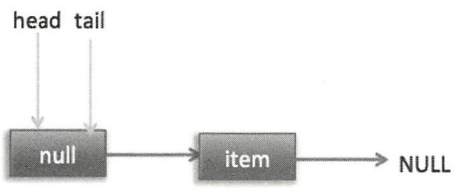


图 7-3

（2）上面是一个线程调用 offer 方法的情况，如果多个线程同时调用，就会存在多个线程同时执行到代码（5）的情况。假设线程 A 调用 offer（item1），线程 B 调用 offer(item2)，同时执行到代码（5）p.casNext(null, newNode)。由于 CAS 的比较设置操作是原子性的，所以这里假设线程 A 先执行了比较设置操作，发现当前 p 的 next 节点确实是 null，则会原子性地更新 next 节点为 item1，这时候线程 B 也会判断 p 的 next 节点是否为 null，结果发现不是 null（因为线程 A 已经设置了 p 的 next 节点为 item1），则会跳到代码（3），然后执行到代码（4），这时候的队列分布如图 7-4 所示。

仅供非商业用途或交流学习使用

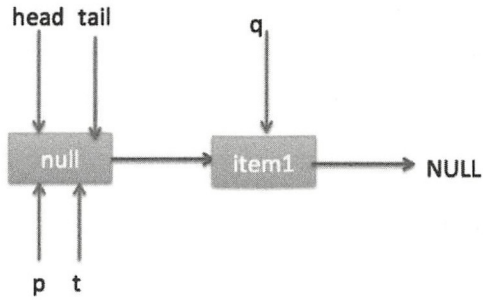


图 7-4

根据上面的状态图可知线程 B 接下来会执行代码（8），然后把 q 赋给了 p，这时候队列状态如图 7-5 所示。

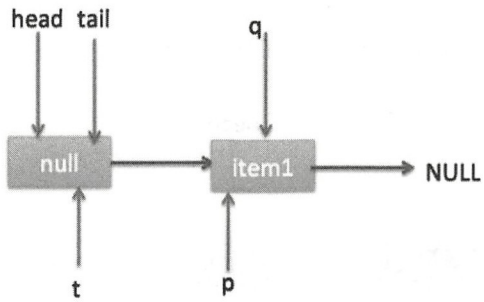


图 7-5

然后线程 B 再次跳转到代码（3）执行，当执行到代码（4）时队列状态如图 7-6 所示。

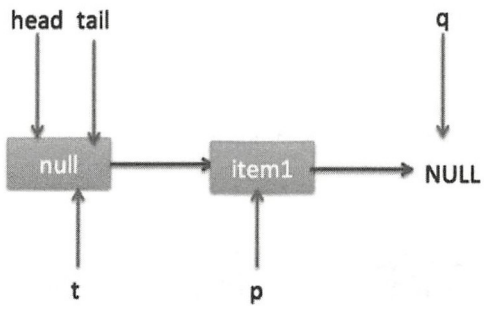


图 7-6

仅供非商业用途或交流学习使用

由于这时候 q==null，所以线程 B 会执行代码（5），通过 CAS 操作判断当前 p 的 next 节点是否是 null，不是则再次循环尝试，是则使用 item2 替换。假设 CAS 成功了，那么执行代码（6），由于 p!=t，所以设置 tail 节点为 item2，然后退出 offer 方法。这时候队列分布如图 7-7 所示。

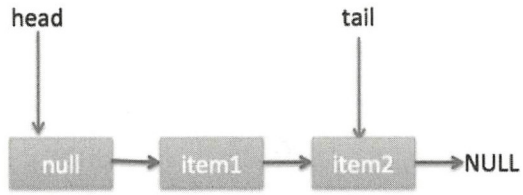


图 7-7

分析到现在，就差代码（7）还没走过，其实这一步要在执行 poll 操作后才会执行。这里先来看一下执行 poll 操作后可能会存在的一种情况，如图 7-8 所示。

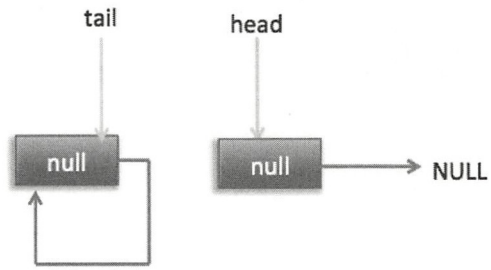


图 7-8

下面分析当队列处于这种状态时调用 offer 添加元素，执行到代码（4）时的状态图（见图 7-9）。

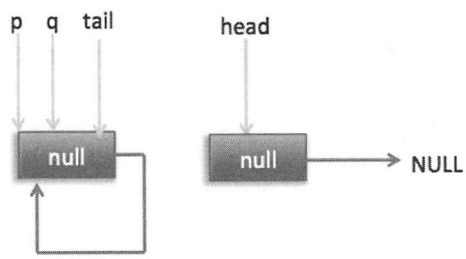


图 7-9

仅供非商业用途或交流学习使用

这里由于 q 节点不为空并且 p==q 所以执行代码（7），由于 t==tail 所以 p 被赋值为 head，然后重新循环，循环后执行到代码（4），这时候队列状态如图 7-10 所示。

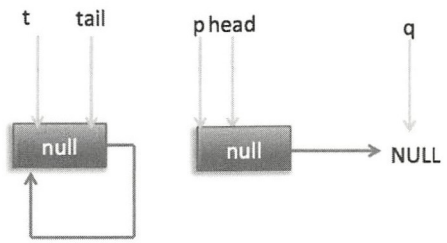


图 7-10

这时候由于 q==null，所以执行代码（5）进行 CAS 操作，如果当前没有其他线程执行 offer 操作，则 CAS 操作会成功，p 的 next 节点被设置为新增节点。然后执行代码（6），由于 p!=t 所以设置新节点为队列的尾部节点，现在队列状态如图 7-11 所示。

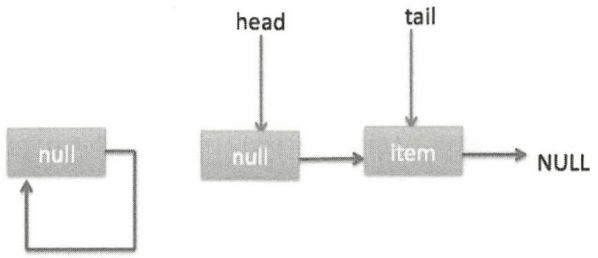


图 7-11

需要注意的是，这里自引用的节点会被垃圾回收掉。

可见，offer 操作中的关键步骤是代码（5），通过原子 CAS 操作来控制某时只有一个线程可以追加元素到队列末尾。进行 CAS 竞争失败的线程会通过循环一次次尝试进行 CAS 操作，直到 CAS 成功才会返回，也就是通过使用无限循环不断进行 CAS 尝试方式来替代阻塞算法挂起调用线程。相比阻塞算法，这是使用 CPU 资源换取阻塞所带来的开销。

2. add 操作

add 操作是在链表末尾添加一个元素，其实在内部调用的还是 offer 操作。

```
public boolean add(E e) {
```

仅供非商业用途或交流学习使用


```
        return offer(e);
    }
```

3. poll 操作

poll 操作是在队列头部获取并移除一个元素，如果队列为空则返回 null。下面看看它的实现原理。

```
public E poll() {
    //(1) goto标记
    restartFromHead:

    //(2)无限循环
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {

            //(3)保存当前节点值
            E item = p.item;

            //(4)当前节点有值则CAS变为null
            if (item != null && p.casItem(item, null)) {
                //(5)CAS成功则标记当前节点并从链表中移除
                if (p != h)
                    updateHead(h, ((q = p.next) != null) ? q : p);
                return item;
            }
            //(6)当前队列为空则返回null
            else if ((q = p.next) == null) {
                updateHead(h, p);
                return null;
            }
            //(7)如果当前节点被自引用了，则重新寻找新的队列头节点
            else if (p == q)
                continue restartFromHead;
            else//(8)
                p = q;
        }
    }
}
    final void updateHead(Node<E> h, Node<E> p) {
        if (h != p && casHead(h, p))
            h.lazySetNext(h);
    }
```

仅供非商业用途或交流学习使用

同样，也结合图来讲解代码执行逻辑。

I. poll 操作是从队头获取元素，所以代码（2）内层循环是从 head 节点开始迭代，代码（3）获取当前队列头的节点，队列一开始为空时队列状态如图 7-12 所示。

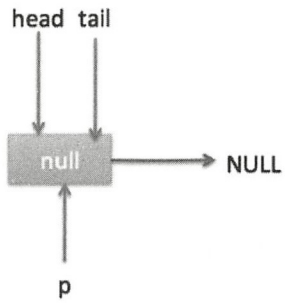


图 7-12

由于 head 节点指向的是 item 为 null 的哨兵节点，所以会执行到代码（6），假设这个过程中没有线程调用 offer 方法，则此时 q 等于 null，这时候队列状态如图 7-13 所示。

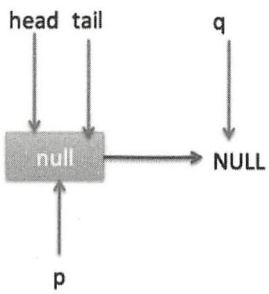


图 7-13

所以会执行 updateHead 方法，由于 h 等于 p 所以没有设置头节点，poll 方法直接返回 null。

II. 假设执行到代码（6）时已经有其他线程调用了 offer 方法并成功添加一个元素到队列，这时候 q 指向的是新增元素的节点，此时队列状态如图 7-14 所示。

仅供非商业用途或交流学习使用

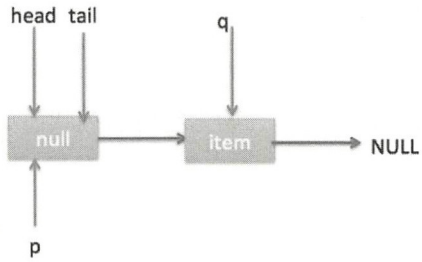


图 7-14

所以代码（6）判断的结果为 false，然后会转向执行代码（7），而此时 p 不等于 q，所以转向执行代码（8），执行的结果是 p 指向了节点 q，此时队列状态如图 7-15 所示。

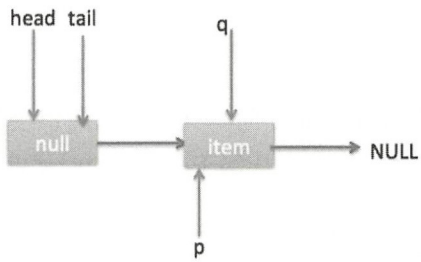


图 7-15

然后程序转向执行代码（3），p 现在指向的元素值不为 null，则执行 p.casItem(item, null) 通过 CAS 操作尝试设置 p 的 item 值为 null，如果此时没有其他线程进行 poll 操作，则 CAS 成功会执行代码（5），由于此时 p!=h 所以设置头节点为 p，并设置 h 的 next 节点为 h 自己，poll 然后返回被从队列移除的节点值 item。此时队列状态如图 7-16 所示。

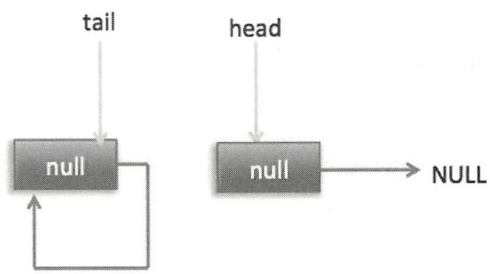


图 7-16

仅供非商业用途或交流学习使用

这个状态就是在讲解 offer 操作时，offer 代码的执行路径（7）的状态。

III. 假如现在一个线程调用了 poll 操作，则在执行代码（4）时队列状态如图 7-17 所示。

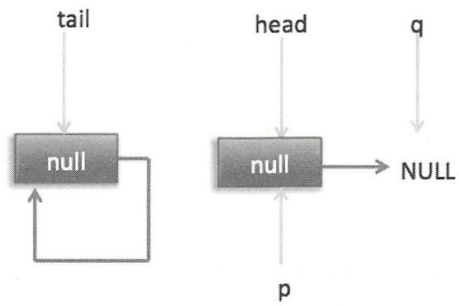


图 7-17

这时候执行代码（6）返回 null。

IV. 现在 poll 的代码还有分支（7）没有执行过，那么什么时候会执行呢？下面来看看。假设线程 A 执行 poll 操作时当前队列状态如图 7-18 所示。

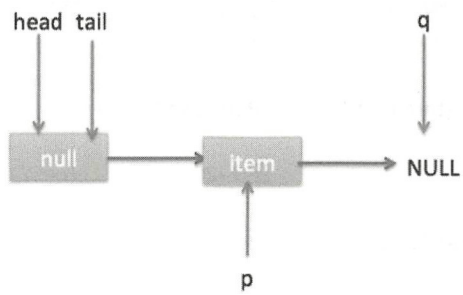


图 7-18

那么执行 p.casItem(item, null) 通过 CAS 操作尝试设置 p 的 item 值为 null，假设 CAS 设置成功则标记该节点并从队列中将其移除，此时队列状态如图 7-19 所示。

仅供非商业用途或交流学习使用

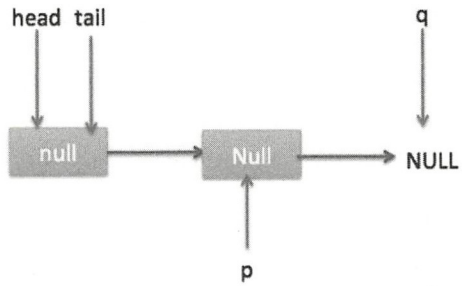


图 7-19

然后，由于 p!=h, 所以会执行 updateHead 方法，假如线程 A 执行 updateHead 前另外一个线程 B 开始 poll 操作，这时候线程 B 的 p 指向 head 节点，但是还没有执行到代码（6），这时候队列状态如图 7-20 所示

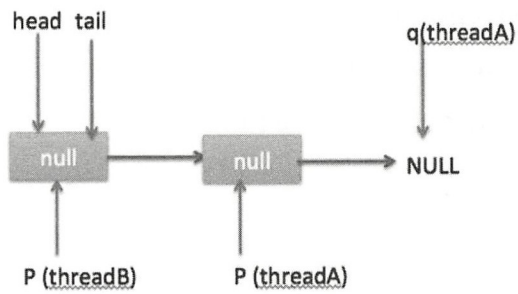


图 7-20

然后线程 A 执行 updateHead 操作，执行完毕后线程 A 退出，这时候队列状态如图 7-21 所示。

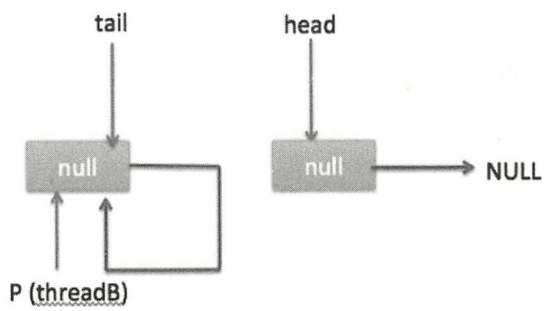
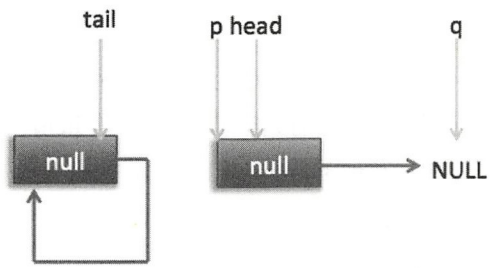


图 7-21

仅供非商业用途或交流学习使用

然后线程 B 继续执行代码（6），q=p.next，由于该节点是自引用节点，所以 p==q，所以会执行代码（7）跳到外层循环 restartFromHead，获取当前队列头 head，现在的状态如图 7-22 所示。



如图 7-22

总结：poll 方法在移除一个元素时，只是简单地使用 CAS 操作把当前节点的 item 值设置为 null，然后通过重新设置头节点将该元素从队列里面移除，被移除的节点就成了孤立节点，这个节点会在垃圾回收时被回收掉。另外，如果在执行分支中发现头节点被修改了，要跳到外层循环重新获取新的头节点。

4. peek 操作

peek 操作是获取队列头部一个元素（只获取不移除），如果队列为空则返回 null。下面看下其实现原理。

```
public E peek() {
   //(1)
    restartFromHead:
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {
            //(2)
            E item = p.item;
            //(3)
            if (item != null || (q = p.next) == null) {
                updateHead(h, p);
                return item;
            }
            //(4)
            else if (p == q)
                continue restartFromHead;
            else
```

```
                //(5)
                    p = q;
            }
        }
    }
```

Peek 操作的代码结构与 poll 操作类似，不同之处在于代码（3）中少了 castItem 操作。其实这很正常，因为 peek 只是获取队列头元素值，并不清空其值。根据前面的介绍我们知道第一次执行 offer 后 head 指向的是哨兵节点（也就是 item 为 null 的节点），那么第一次执行 peek 时在代码（3）中会发现 item==null，然后执行 q=p.next，这时候 q 节点指向的才是队列里面第一个真正的元素，或者如果队列为 null 则 q 指向 null。

当队列为空时队列状态如图 7-23 所示。

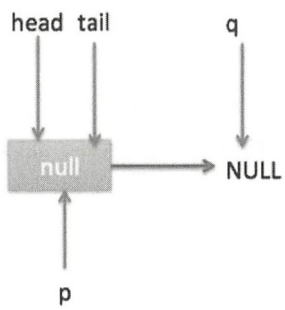


图 7-23

这时候执行 updateHead，由于 h 节点等于 p 节点，所以不进行任何操作，然后 peek 操作会返回 null。

当队列中至少有一个元素时（这里假设只有一个），队列状态如图 7-24 所示。

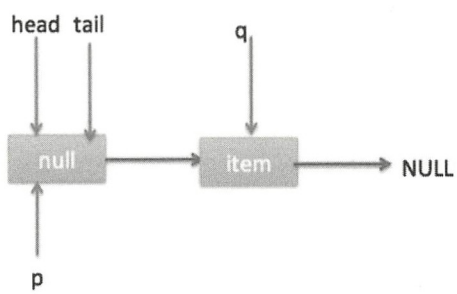


图 7-24

这时候执行代码(5),p 指向了 q 节点,然后执行代码(3),此时队列状态如图 7-25 所示。

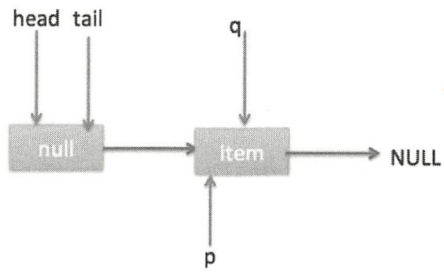


图 7-25

执行代码（3）时发现 item 不为 null，所以执行 updateHead 方法，由于 h!=p, 所以设置头节点，设置后队列状态如图 7-26 所示。

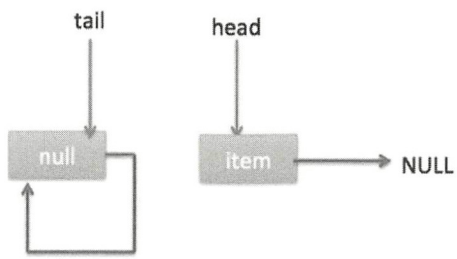


图 7-26

也就是剔除了哨兵节点。

总结：peek 操作的代码与 poll 操作类似，只是前者只获取队列头元素但是并不从队列里将它删除，而后者获取后需要从队列里面将它删除。另外，在第一次调用 peek 操作时，会删除哨兵节点，并让队列的 head 节点指向队列里面第一个元素或者 null。

5. size 操作

计算当前队列元素个数，在并发环境下不是很有用，因为 CAS 没有加锁，所以从调用 size 函数到返回结果期间有可能增删元素，导致统计的元素个数不精确。

```
public int size() {
    int count = 0;
    for (Node<E> p = first(); p != null; p = succ(p))
```

```
        if (p.item != null)
            // 最大值Integer.MAX_VALUE
            if (++count == Integer.MAX_VALUE)
                break;
    return count;
}

//获取第一个队列元素（哨兵元素不算），没有则为null
Node<E> first() {
    restartFromHead:
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {
            boolean hasItem = (p.item != null);
            if (hasItem || (q = p.next) == null) {
                updateHead(h, p);
                return hasItem ? p : null;
            }
            else if (p == q)
                continue restartFromHead;
            else
                p = q;
        }
    }
}

//获取当前节点的next元素，如果是自引入节点则返回真正的头节点
final Node<E> succ(Node<E> p) {
    Node<E> next = p.next;
    return (p == next) ? head : next;
}
```

6. remove 操作

如果队列里面存在该元素则删除该元素，如果存在多个则删除第一个，并返回 true，否则返回 false。

```
public boolean remove(Object o) {

    //为空，则直接返回false
    if (o == null) return false;
    Node<E> pred = null;
    for (Node<E> p = first(); p != null; p = succ(p)) {
        E item = p.item;
```

```
            //相等则使用CAS设置为null,同时一个线程操作成功，失败的线程循环查找队列中是否有匹配的其他元素。
            if (item != null &&
                o.equals(item) &&
                p.casItem(item, null)) {

                //获取next元素
                Node<E> next = succ(p);

                //如果有前驱节点，并且next节点不为空则链接前驱节点到next节点
                if (pred != null && next != null)
                    pred.casNext(p, next);
                return true;
            }
            pred = p;
        }
        return false;
    }
```

7. contains 操作

判断队列里面是否含有指定对象，由于是遍历整个队列，所以像 size 操作一样结果也不是那么精确，有可能调用该方法时元素还在队列里面，但是遍历过程中其他线程才把该元素删除了，那么就会返回 false。

```
public boolean contains(Object o) {
    if (o == null) return false;
    for (Node<E> p = first(); p != null; p = succ(p)) {
        E item = p.item;
        if (item != null && o.equals(item))
            return true;
    }
    return false;
}
```

7.1.3　小结

ConcurrentLinkedQueue 的底层使用单向链表数据结构来保存队列元素，每个元素被包装成一个 Node 节点。队列是靠头、尾节点来维护的，创建队列时头、尾节点指向一个 item 为 null 的哨兵节点。第一次执行 peek 或者 first 操作时会把 head 指向第一个真正的队列元素。由于使用非阻塞 CAS 算法，没有加锁，所以在计算 size 时有可能进行了 offer、

poll 或者 remove 操作，导致计算的元素个数不精确，所以在并发情况下 size 函数不是很有用。

如图 7-27 所示，入队、出队都是操作使用 volatile 修饰的 tail、head 节点，要保证在多线程下出入队线程安全，只需要保证这两个 Node 操作的可见性和原子性即可。由于 volatile 本身可以保证可见性，所以只需要保证对两个变量操作的原子性即可。

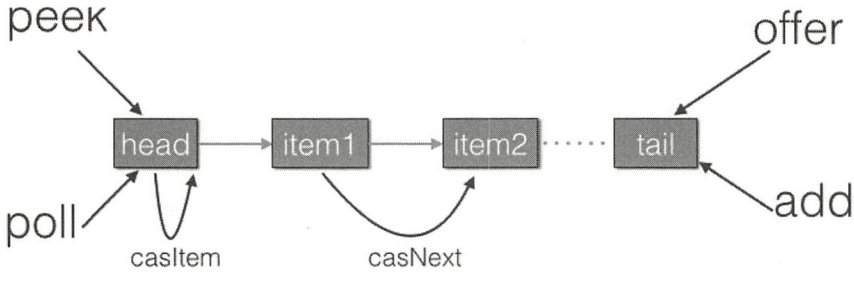


图 7-27

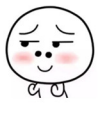
offer 操作是在 tail 后面添加元素，也就是调用 tail.casNext 方法，而这个方法使用的是 CAS 操作，只有一个线程会成功，然后失败的线程会循环，重新获取 tail，再执行 casNext 方法。poll 操作也通过类似 CAS 的算法保证出队时移除节点操作的原子性。

7.2 LinkedBlockingQueue 原理探究

前面介绍了使用 CAS 算法实现的非阻塞队列 ConcurrentLinkedQueue，下面我们来介绍使用独占锁实现的阻塞队列 LinkedBlockingQueue。

7.2.1 类图结构

同样首先看一下 LinkedBlockingQueue 的类图结构，以便从全局对 LinkedBlockingQueue 有个直观的了解，如图 7-28 所示。



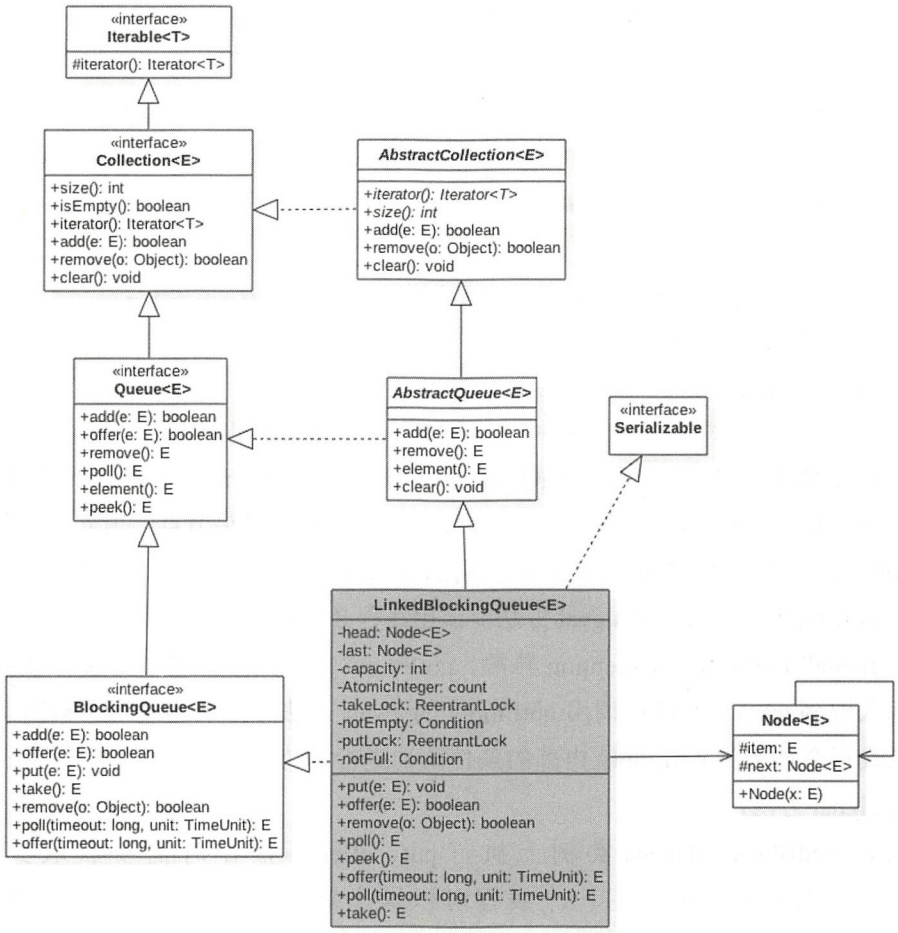
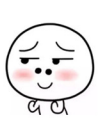


图 7-28

由类图可以看到，LinkedBlockingQueue 也是使用单向链表实现的，其也有两个 Node，分别用来存放首、尾节点，并且还有一个初始值为 0 的原子变量 count，用来记录队列元素个数。另外还有两个 ReentrantLock 的实例，分别用来控制元素入队和出队的原子性，其中 takeLock 用来控制同时只有一个线程可以从队列头获取元素，其他线程必须等待，putLock 控制同时只能有一个线程可以获取锁，在队列尾部添加元素，其他线程必须等待。另外，notEmpty 和 notFull 是条件变量，它们内部都有一个条件队列用来存放进队和出队时被阻塞的线程，其实这是生产者—消费者模型。如下是独占锁的创建代码。



```
/** 执行take、 poll等操作时需要获取该锁 */
private final ReentrantLock takeLock = new ReentrantLock();

/** 当队列为空时，执行出队操作（比如take）的线程会被放入这个条件队列进行等待 */
private final Condition notEmpty = takeLock.newCondition();

/** 执行put、 offer等操作时需要获取该锁*/
private final ReentrantLock putLock = new ReentrantLock();

/**当队列满时，执行进队操作（比如put)的线程会被放入这个条件队列进行等待 */
private final Condition notFull = putLock.newCondition();

/** 当前队列元素个数 */
private final AtomicInteger count = new AtomicInteger(0);
```

- 当调用线程在LinkedBlockingQueue实例上执行take、poll等操作时需要获取到takeLock锁，从而保证同时只有一个线程可以操作链表头节点。另外由于条件变量notEmpty内部的条件队列的维护使用的是takeLock的锁状态管理机制，所以在调用notEmpty的await和signal方法前调用线程必须先获取到takeLock锁，否则会抛出IllegalMonitorStateException异常。notEmpty内部则维护着一个条件队列，当线程获取到takeLock锁后调用notEmpty的await方法时，调用线程会被阻塞，然后该线程会被放到notEmpty内部的条件队列进行等待，直到有线程调用了notEmpty的signal方法。
- 在LinkedBlockingQueue实例上执行put、offer等操作时需要获取到putLock锁，从而保证同时只有一个线程可以操作链表尾节点。同样由于条件变量notFull内部的条件队列的维护使用的是putLock的锁状态管理机制，所以在调用notFull的await和signal方法前调用线程必须先获取到putLock锁，否则会抛出IllegalMonitorStateException异常。notFull内部则维护着一个条件队列，当线程获取到putLock锁后调用notFull的await方法时，调用线程会被阻塞，然后该线程会被放到notFull内部的条件队列进行等待，直到有线程调用了notFull的signal方法。

如下是LinkedBlockingQueue的无参构造函数的代码。

```
public static final int   MAX_VALUE = 0x7fffffff;

public LinkedBlockingQueue() {
    this(Integer.MAX_VALUE);
}
```

```
public LinkedBlockingQueue(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
    //初始化首、尾节点,让它们指向哨兵节点
    last = head = new Node<E>(null);
}
```

由该代码可知，默认队列容量为 0x7fffffff，用户也可以自己指定容量，所以从一定程度上可以说 LinkedBlockingQueue 是有界阻塞队列。

7.2.2　LinkedBlockingQueue 原理介绍

本节讲解 LinkedBlockingQueue 的几个重要方法。

1. offer 操作

向队列尾部插入一个元素，如果队列中有空闲则插入成功后返回 true，如果队列已满则丢弃当前元素然后返回 false。如果 e 元素为 null 则抛出 NullPointerException 异常。另外，该方法是非阻塞的。

```
public boolean offer(E e) {

    //(1)为空元素则抛出空指针异常
    if (e == null) throw new NullPointerException();

    //(2) 如果当前队列满则丢弃将要放入的元素，然后返回false
    final AtomicInteger count = this.count;
    if (count.get() == capacity)
        return false;

    //(3) 构造新节点，获取putLock独占锁
    int c = -1;
    Node<E> node = new Node<E>(e);
    final ReentrantLock putLock = this.putLock;
    putLock.lock();
    try {
        //(4)如果队列不满则进队列，并递增元素计数
        if (count.get() < capacity) {
            enqueue(node);
            c = count.getAndIncrement();
            //(5)
```

```
                if (c + 1 < capacity)
                    notFull.signal();
            }
        } finally {
            //(6)释放锁
            putLock.unlock();
        }
        //(7)
        if (c == 0)
            signalNotEmpty();
        //(8)
        return c >= 0;
    }

private void enqueue(Node<E> node) {
 last = last.next = node;
}
```

代码（2）判断如果当前队列已满则丢弃当前元素并返回 false。

代码（3）获取到 putLock 锁，当前线程获取到该锁后，则其他调用 put 和 offer 操作的线程将会被阻塞（阻塞的线程被放到 putLock 锁的 AQS 阻塞队列）。

代码（4）这里重新判断当前队列是否满，这是因为在执行代码（2）和获取到 putLock 锁期间可能其他线程通过 put 或者 offer 操作向队列里面添加了新元素。重新判断队列确实不满则新元素入队，并递增计数器。

代码（5）判断如果新元素入队后队列还有空闲空间，则唤醒 notFull 的条件队列里面因为调用了 notFull 的 await 操作（比如执行 put 方法而队列满了的时候）而被阻塞的一个线程，因为队列现在有空闲所以这里可以提前唤醒一个入队线程。

代码（6）则释放获取的 putLock 锁，这里要注意，锁的释放一定要在 finally 里面做，因为即使 try 块抛出异常了，finally 也是会被执行到。另外释放锁后其他因为调用 put 操作而被阻塞的线程将会有一个获取到该锁。

代码（7）中的 c==0 说明在执行代码（6）释放锁时队列里面至少有一个元素，队列里面有元素则执行 signalNotEmpty 操作，signalNotEmpty 的代码如下。

```
    private void signalNotEmpty() {
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lock();
```

仅供非商业用途或交流学习使用


```
        try {
            notEmpty.signal();
        } finally {
            takeLock.unlock();
        }
    }
```

该方法的作用就是激活 notEmpty 的条件队列中因为调用 notEmpty 的 await 方法（比如调用 take 方法并且队列为空的时候）而被阻塞的一个线程，这也说明了调用条件变量的方法前要获取对应的锁。

综上可知，offer 方法通过使用 putLock 锁保证了在队尾新增元素操作的原子性。另外，调用条件变量的方法前一定要记得获取对应的锁，并且注意进队时只操作队列链表的尾节点。

2. put 操作

向队列尾部插入一个元素，如果队列中有空闲则插入后直接返回，如果队列已满则阻塞当前线程，直到队列有空闲插入成功后返回。如果在阻塞时被其他线程设置了中断标志，则被阻塞线程会抛出 InterruptedException 异常而返回。另外，如果 e 元素为 null 则抛出 NullPointerException 异常。

put 操作的代码结构与 offer 操作类似，代码如下。

```
public void put(E e) throws InterruptedException {
        //(1)如果为空元素则抛出空指针异常
        if (e == null) throw new NullPointerException();
        //(2) 构建新节点，并获取独占锁putLock
        int c = -1;
        Node<E> node = new Node<E>(e);
        final ReentrantLock putLock = this.putLock;
        final AtomicInteger count = this.count;
        putLock.lockInterruptibly();
        try {
            //(3)如果队列满则等待
            while (count.get() == capacity) {
                notFull.await();
            }
            //(4)进队列并递增计数
            enqueue(node);
            c = count.getAndIncrement();
```

仅供非商业用途或交流学习使用


```
            //(5)
            if (c + 1 < capacity)
                notFull.signal();
        } finally {
            //(6)
            putLock.unlock();
        }
        //(7)
        if (c == 0)
            signalNotEmpty();
    }
```

在代码（2）中使用 putLock.lockInterruptibly() 获取独占锁，相比在 offer 方法中获取独占锁的方法这个方法可以被中断。具体地说就是当前线程在获取锁的过程中，如果被其他线程设置了中断标志则当前线程会抛出 InterruptedException 异常，所以 put 操作在获取锁的过程中是可被中断的。

代码（3）判断如果当前队列已满，则调用 notFull 的 await() 方法把当前线程放入 notFull 的条件队列，当前线程被阻塞挂起后会释放获取到的 putLock 锁。由于 putLock 锁被释放了，所以现在其他线程就有机会获取到 putLock 锁了。

另外代码（3）在判断队列是否为空时为何使用 while 循环而不是 if 语句？这是考虑到当前线程被虚假唤醒的问题，也就是其他线程没有调用 notFull 的 singal 方法时 notFull.await() 在某种情况下会自动返回。如果使用 if 语句那么虚假唤醒后会执行代码（4）的元素入队操作，并且递增计数器，而这时候队列已经满了，从而导致队列元素个数大于队列被设置的容量，进而导致程序出错。而使用 while 循环时，假如 notFull.await() 被虚假唤醒了，那么再次循环检查当前队列是否已满，如果是则再次进行等待。

3. poll 操作

从队列头部获取并移除一个元素，如果队列为空则返回 null，该方法是不阻塞的。

```
public E poll() {
    //(1)队列为空则返回null
    final AtomicInteger count = this.count;
    if (count.get() == 0)
        return null;
    //(2)获取独占锁
    E x = null;
```

仅供非商业用途或交流学习使用


```
        int c = -1;
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lock();
        try {
            //(3)队列不空则出队并递减计数
            if (count.get() > 0) {//3.1
                x = dequeue();//3.2
                c = count.getAndDecrement();//3.3
                //(4)
                if (c > 1)
                    notEmpty.signal();
            }
        } finally {
            //(5)
            takeLock.unlock();
        }
        //(6)
        if (c == capacity)
            signalNotFull();
        //(7)返回
        return x;
    }

    private E dequeue() {
        Node<E> h = head;
        Node<E> first = h.next;
        h.next = h; // help GC
        head = first;
        E x = first.item;
        first.item = null;
        return x;
    }
```

代码（1）判断如果当前队列为空，则直接返回 null。

代码（2）获取独占锁 takeLock，当前线程获取该锁后，其他线程在调用 poll 或者 take 方法时会被阻塞挂起。

代码（3）判断如果当前队列不为空则进行出队操作，然后递减计数器。这里需要思考，如何保证执行代码 3.1 时队列不空，而执行代码 3.2 时也一定不会空呢？毕竟这不是原子性操作，会不会出现代码 3.1 判断队列不为空，但是执行代码 3.2 时队列为空了呢？那么我们看在执行到代码 3.2 前在哪些地方会修改 count 的计数。由于当前线程已经拿到

仅供非商业用途或交流学习使用

了 takeLock 锁，所以其他调用 poll 或者 take 方法的线程不可能会走到修改 count 计数的地方。其实这时候如果能走到修改 count 计数的地方是因为其他线程调用了 put 和 offer 操作，由于这两个操作不需要获取 takeLock 锁而获取的是 putLock 锁，但是在 put 和 offer 操作内部是增加 count 计数值的，所以不会出现上面所说的情况。其实只需要看在哪些地方递减了 count 计数值即可，只有递减了 count 计数值才会出现上面说的，执行代码 3.1 时队列不空，而执行代码 3.2 时队列为空的情况。我们查看代码，只有在 poll、take 或者 remove 操作的地方会递减 count 计数值，但是这三个方法都需要获取到 takeLock 锁才能进行操作，而当前线程已经获取了 takeLock 锁，所以其他线程没有机会在当前情况下递减 count 计数值，所以看起来代码 3.1、3.2 不是原子性的，但是它们是线程安全的。

代码（4）判断如果 c>1 则说明当前线程移除掉队列里面的一个元素后队列不为空（c 是删除元素前队列元素个数），那么这时候就可以激活因为调用 take 方法而被阻塞到 notEmpty 的条件队列里面的一个线程。

代码（6）说明当前线程移除队头元素前当前队列是满的，移除队头元素后当前队列至少有一个空闲位置，那么这时候就可以调用 signalNotFull 激活因为调用 put 方法而被阻塞到 notFull 的条件队列里的一个线程，signalNotFull 的代码如下。

```
private void signalNotFull() {
    final ReentrantLock putLock = this.putLock;
    putLock.lock();
    try {
        notFull.signal();
    } finally {
        putLock.unlock();
    }
}
```

poll 代码逻辑比较简单，值得注意的是，获取元素时只操作了队列的头节点。

4. peek 操作

获取队列头部元素但是不从队列里面移除它，如果队列为空则返回 null。该方法是不阻塞的。

```
public E peek() {
    //(1)
    if (count.get() == 0)
```

仅供非商业用途或交流学习使用


```
            return null;
        //(2)
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lock();
        try {
            Node<E> first = head.next;
            //(3)
            if (first == null)
                return null;
            else
            //(4)
                return first.item;
        } finally {
           //(5)
            takeLock.unlock();
        }
    }
```

peek 操作的代码也比较简单，这里需要注意的是，代码（3）这里还是需要判断 first 是否为 null，不能直接执行代码（4）。正常情况下执行到代码（2）说明队列不为空，但是代码（1）和（2）不是原子性操作，也就是在执行点（1）判断队列不空后，在代码（2）获取到锁前有可能其他线程执行了 poll 或者 take 操作导致队列变为空。然后当前线程获取锁后，直接执行代码（4）（first.item）会抛出空指针异常。

5. take 操作

获取当前队列头部元素并从队列里面移除它。如果队列为空则阻塞当前线程直到队列不为空然后返回元素，如果在阻塞时被其他线程设置了中断标志，则被阻塞线程会抛出 InterruptedException 异常而返回。

```
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    //(1)获取锁
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        //(2)当前队列为空则阻塞挂起
        while (count.get() == 0) {
            notEmpty.await();
```

仅供非商业用途或交流学习使用


```
            }
            //(3)出队并递减计数
            x = dequeue();
            c = count.getAndDecrement();
            //(4)
            if (c > 1)
                notEmpty.signal();
        } finally {
            //(5)
            takeLock.unlock();
        }
        //(6)
        if (c == capacity)
            signalNotFull();
        //(7)
        return x;
    }
```

在代码（1）中，当前线程获取到独占锁，其他调用 take 或者 poll 操作的线程将会被阻塞挂起。

代码（2）判断如果队列为空则阻塞挂起当前线程，并把当前线程放入 notEmpty 的条件队列。

代码（3）进行出队操作并递减计数。

代码（4）判断如果 c>1 则说明当前队列不为空，那么唤醒 notEmpty 的条件队列里面的一个因为调用 take 操作而被阻塞的线程。

代码（5）释放锁。

代码（6）判断如果 c == capacity 则说明当前队列至少有一个空闲位置，那么激活条件变量 notFull 的条件队列里面的一个因为调用 put 操作而被阻塞的线程。

6. remove 操作

删除队列里面指定的元素，有则删除并返回 true，没有则返回 false。

```
public boolean remove(Object o) {
    if (o == null) return false;

    //(1)双重加锁
```

仅供非商业用途或交流学习使用


```
    fullyLock();
    try {

        //(2)遍历队列找到则删除并返回true
        for (Node<E> trail = head, p = trail.next;
             p != null;
             trail = p, p = p.next) {
             //(3)
            if (o.equals(p.item)) {
                unlink(p, trail);
                return true;
            }
        }
        //(4)找不到则返回false
        return false;
    } finally {
        //(5)解锁
        fullyUnlock();
    }
}
```

代码（1）通过 fullyLock 获取双重锁，获取后，其他线程进行入队或者出队操作时就会被阻塞挂起。

```
void fullyLock() {
    putLock.lock();
    takeLock.lock();
}
```

代码(2)遍历队列寻找要删除的元素，找不到则直接返回 false，找到则执行 unlink 操作。unlik 操作的代码如下。

```
void unlink(Node<E> p, Node<E> trail) {

  p.item = null;
  trail.next = p.next;
  if (last == p)
      last = trail;
  //如果当前队列满，则删除后，也不忘记唤醒等待的线程
  if (count.getAndDecrement() == capacity)
      notFull.signal();
}
```

删除元素后，如果发现当前队列有空闲空间，则唤醒 notFull 的条件队列中的一个因

仅供非商业用途或交流学习使用

为调用 put 方法而被阻塞的线程。

代码（5）调用 fullyUnlock 方法使用与加锁顺序相反的顺序释放双重锁。

```
void fullyUnlock() {
    takeLock.unlock();
    putLock.unlock();
}
```

总结：由于 remove 方法在删除指定元素前加了两把锁，所以在遍历队列查找指定元素的过程中是线程安全的，并且此时其他调用入队、出队操作的线程全部会被阻塞。另外，获取多个资源锁的顺序与释放的顺序是相反的。

7. size 操作

获取当前队列元素个数。

```
public int size() {
    return count.get();
}
```

由于进行出队、入队操作时的 count 是加了锁的，所以结果相比 ConcurrentLinkedQueue 的 size 方法比较准确。这里考虑为何在 ConcurrentLinkedQueue 中需要遍历链表来获取 size 而不使用一个原子变量呢？这是因为使用原子变量保存队列元素个数需要保证入队、出队操作和原子变量操作是原子性操作，而 ConcurrentLinkedQueue 使用的是 CAS 无锁算法，所以无法做到这样。

7.2.3 小结

LinkedBlockingQueue 的内部是通过单向链表实现的，使用头、尾节点来进行入队和出队操作，也就是入队操作都是对尾节点进行操作，出队操作都是对头节点进行操作。

如图 7-29 所示，对头、尾节点的操作分别使用了单独的独占锁从而保证了原子性，所以出队和入队操作是可以同时进行的。另外对头、尾节点的独占锁都配备了一个条件队列，用来存放被阻塞的线程，并结合入队、出队操作实现了一个生产消费模型。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

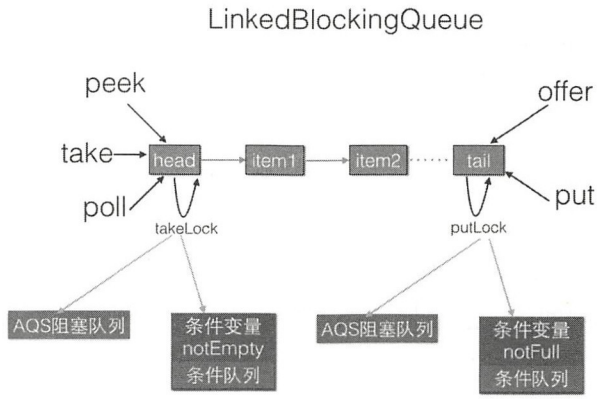


图 7-29

7.3 ArrayBlockingQueue 原理探究

上节介绍了使用有界链表方式实现的阻塞队列 LinkedBlockingQueue，本节来研究使用有界数组方式实现的阻塞队列 ArrayBlockingQueue 的原理。

7.3.1 类图结构

同样，为了能从全局一览 ArrayBlockingQueue 的内部构造，先来看它的类图，如图 7-30 所示。

由该图可以看出，ArrayBlockingQueue 的内部有一个数组 items，用来存放队列元素，putindex 变量表示入队元素下标，takeIndex 是出队下标，count 统计队列元素个数。从定义可知，这些变量并没有使用 volatile 修饰，这是因为访问这些变量都是在锁块内，而加锁已经保证了锁块内变量的内存可见性了。另外有个独占锁 lock 用来保证出、入队操作的原子性，这保证了同时只有一个线程可以进行入队、出队操作。另外，notEmpty、notFull 条件变量用来进行出、入队的同步。

仅供非商业用途或交流学习使用

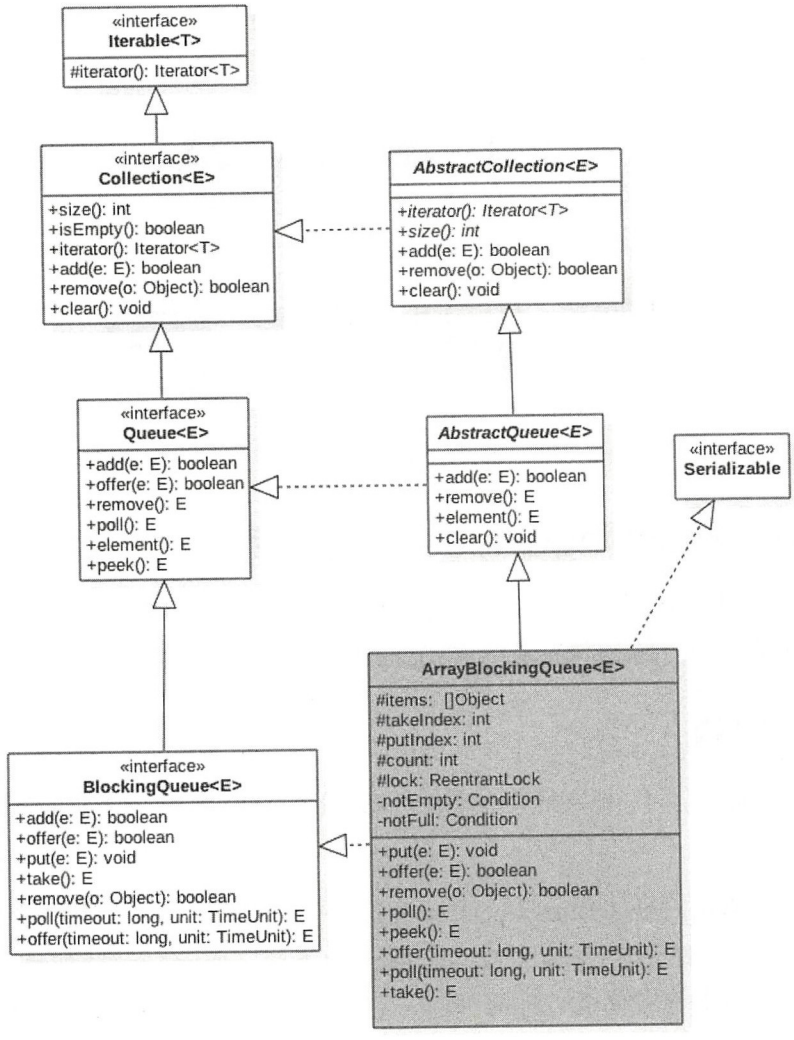


图 7-30

另外，由于 ArrayBlockingQueue 是有界队列，所以构造函数必须传入队列大小参数。构造函数的代码如下。

```
public ArrayBlockingQueue(int capacity) {
      this(capacity, false);
}
```

仅供非商业用途或交流学习使用

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}
```

由以上代码可知，在默认情况下使用 ReentrantLock 提供的非公平独占锁进行出、入队操作的同步。

7.3.2 ArrayBlockingQueue 原理介绍

本节主要讲解下面几个函数的原理，研究过 LinkedBlockingQueue 的实现后再看 ArrayBlockingQueue 的实现会感觉后者简单了很多。

1. offer 操作

向队列尾部插入一个元素，如果队列有空闲空间则插入成功后返回 true，如果队列已满则丢弃当前元素然后返回 false。如果 e 元素为 null 则抛出 NullPointerException 异常。另外，该方法是不阻塞的。

```
public boolean offer(E e) {
    //(1) e为null，则抛出NullPointerException异常
    checkNotNull(e);
    //(2) 获取独占锁
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        //(3) 如果队列满则返回false
        if (count == items.length)
            return false;
        else {
            //(4) 否则插入元素
            enqueue(e);
            return true;
        }
    } finally {
        lock.unlock();
    }
```

仅供非商业用途或交流学习使用

```
}
```

代码（2）获取独占锁，当前线程获取该锁后，其他入队和出队操作的线程都会被阻塞挂起而后被放入 lock 锁的 AQS 阻塞队列。

代码（3）判断如果队列满则直接返回 false，否则调用 enqueue 方法后返回 true，enqueue 的代码如下。

```
private void enqueue(E x) {
    //（6）元素入队
    final Object[] items = this.items;
    items[putIndex] = x;
    //（7）计算下一个元素应该存放的下标位置
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
    //(8)
    notEmpty.signal();
}
```

如上代码首先把当前元素放入 items 数组，然后计算下一个元素应该存放的下标位置，并递增元素个数计数器，最后激活 notEmpty 的条件队列中因为调用 take 操作而被阻塞的一个线程。这里由于在操作共享变量 count 前加了锁，所以不存在内存不可见问题，加过锁后获取的共享变量都是从主内存获取的，而不是从 CPU 缓存或者寄存器获取。

代码（5）释放锁，然后会把修改的共享变量值（比如 count 的值）刷新回主内存中，这样其他线程通过加锁再次读取这些共享变量时，就可以看到最新的值。

2．put 操作

向队列尾部插入一个元素，如果队列有空闲则插入后直接返回 true，如果队列已满则阻塞当前线程直到队列有空闲并插入成功后返回 true，如果在阻塞时被其他线程设置了中断标志，则被阻塞线程会抛出 InterruptedException 异常而返回。另外，如果 e 元素为 null 则抛出 NullPointerException 异常。

```
public void put(E e) throws InterruptedException {
    //(1)
    checkNotNull(e);
    final ReentrantLock lock = this.lock;

    //(2)获取锁（可被中断）
```

仅供非商业用途或交流学习使用

```
    lock.lockInterruptibly();
    try {

        //(3)如果队列满，则把当前线程放入notFull管理的条件队列
        while (count == items.length)
            notFull.await();

        //(4)插入元素
        enqueue(e);
    } finally {
        //(5)
        lock.unlock();
    }
}
```

在代码（2）中，在获取锁的过程中当前线程被其他线程中断了，则当前线程会抛出InterruptedException 异常而退出。

代码（3）判断如果当前队列已满，则把当前线程阻塞挂起后放入 notFull 的条件队列，注意这里也是使用了 while 循环而不是 if 语句。

代码（4）判断如果队列不满则插入当前元素，此处不再赘述。

3．poll 操作

从队列头部获取并移除一个元素，如果队列为空则返回 null，该方法是不阻塞的。

```
public E poll() {
    //(1)获取锁
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        //(2)当前队列为空则返回null,否则调用dequeue()获取
        return (count == 0) ? null : dequeue();
    } finally {
        //(3)释放锁
        lock.unlock();
    }
}
```

代码（1）获取独占锁。

代码（2）判断如果队列为空则返回 null，否则调用 dequeue() 方法。dequeue 方法的

仅供非商业用途或交流学习使用

代码如下。

```
private E dequeue() {
    final Object[] items = this.items;

    //(4)获取元素值
    @SuppressWarnings("unchecked")
    E x = (E) items[takeIndex];
    //(5)数组中的值为null
    items[takeIndex] = null;

    //(6)队头指针计算，队列元素个数减1
   if (++takeIndex == items.length)
            takeIndex = 0;
    count--;

    //(7)发送信号激活notFull条件队列里面的一个线程
    notFull.signal();
    return x;
}
```

由以上代码可知，首先获取当前队头元素并将其保存到局部变量，然后重置队头元素为 null，并重新设置队头下标，递减元素计数器，最后发送信号激活 notFull 的条件队列里面一个因为调用 put 方法而被阻塞的线程。

4. take 操作

获取当前队列头部元素并从队列里面移除它。如果队列为空则阻塞当前线程直到队列不为空然后返回元素，如果在阻塞时被其他线程设置了中断标志，则被阻塞线程会抛出 InterruptedException 异常而返回。

```
public E take() throws InterruptedException {
    //(1)获取锁
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {

        //(2)队列为空，则等待，直到队列中有元素
        while (count == 0)
            notEmpty.await();
        //(3)获取队头元素
        return dequeue();
```

仅供非商业用途或交流学习使用

```
    } finally {
        //(4) 释放锁
        lock.unlock();
    }
}
```

take 操作的代码也比较简单，与 poll 相比只是代码（2）不同。在这里，如果队列为空则把当前线程挂起后放入 notEmpty 的条件队列，等其他线程调用 notEmpty.signal() 方法后再返回。需要注意的是，这里也是使用 while 循环进行检测并等待而不是使用 if 语句。

5. peek 操作

获取队列头部元素但是不从队列里面移除它，如果队列为空则返回 null，该方法是不阻塞的。

```
public E peek() {
    //(1)获取锁
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        //(2)
        return itemAt(takeIndex);
    } finally {
       //(3)
        lock.unlock();
    }
}

 @SuppressWarnings("unchecked")
final E itemAt(int i) {
        return (E) items[i];
}
```

peek 的实现更简单，首先获取独占锁，然后从数组 items 中获取当前队头下标的值并返回，在返回前释放获取的锁。

6. size 操作

计算当前队列元素个数。

```
public int size() {
    final ReentrantLock lock = this.lock;
```

仅供非商业用途或交流学习使用

```
        lock.lock();
        try {
            return count;
        } finally {
            lock.unlock();
        }
    }
```

size 操作比较简单，获取锁后直接返回 count，并在返回前释放锁。也许你会问，这里又没有修改 count 的值，只是简单地获取，为何要加锁呢？其实如果 count 被声明为 volatile 的这里就不需要加锁了，因为 volatile 类型的变量保证了内存的可见性，而 ArrayBlockingQueue 中的 count 并没有被声明为 volatile 的，这是因为 count 操作都是在获取锁后进行的，而获取锁的语义之一是，获取锁后访问的变量都是从主内存获取的，这保证了变量的内存可见性。

7.3.3 小结

如图 7-31 所示，ArrayBlockingQueue 通过使用全局独占锁实现了同时只能有一个线程进行入队或者出队操作，这个锁的粒度比较大，有点类似于在方法上添加 synchronized 的意思。其中 offer 和 poll 操作通过简单的加锁进行入队、出队操作，而 put、take 操作则使用条件变量实现了，如果队列满则等待，如果队列空则等待，然后分别在出队和入队操作中发送信号激活等待线程实现同步。另外，相比 LinkedBlockingQueue，ArrayBlockingQueue 的 size 操作的结果是精确的，因为计算前加了全局锁。

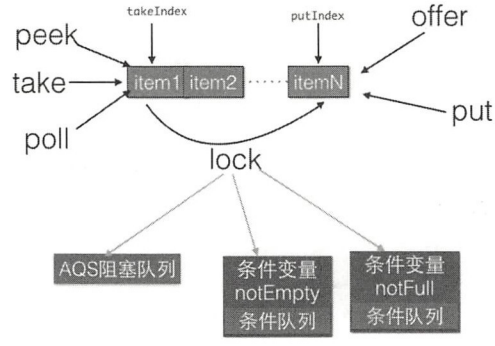


图 7-31

仅供非商业用途或交流学习使用

7.4　PriorityBlockingQueue 原理探究

7.4.1　介绍

PriorityBlockingQueue 是带优先级的无界阻塞队列，每次出队都返回优先级最高或者最低的元素。其内部是使用平衡二叉树堆实现的，所以直接遍历队列元素不保证有序。默认使用对象的 compareTo 方法提供比较规则，如果你需要自定义比较规则则可以自定义 comparators。

7.4.2　PriorityBlockingQueue 类图结构

下面首先通过类图结构（见图 7-32）来从全局了解 PriorityBlockingQueue 的原理。

由图 7-32 可知，PriorityBlockingQueue 内部有一个数组 queue，用来存放队列元素，size 用来存放队列元素个数。allocationSpinLock 是个自旋锁，其使用 CAS 操作来保证同时只有一个线程可以扩容队列，状态为 0 或者 1，其中 0 表示当前没有进行扩容，1 表示当前正在扩容。

由于这是一个优先级队列，所以有一个比较器 comparator 用来比较元素大小。lock 独占锁对象用来控制同时只能有一个线程可以进行入队、出队操作。notEmpty 条件变量用来实现 take 方法阻塞模式。这里没有 notFull 条件变量是因为这里的 put 操作是非阻塞的，为啥要设计为非阻塞的，是因为这是无界队列。

在如下构造函数中，默认队列容量为 11，默认比较器为 null，也就是使用元素的 compareTo 方法进行比较来确定元素的优先级，这意味着队列元素必须实现了 Comparable 接口。

```
private static final int DEFAULT_INITIAL_CAPACITY = 11;


public PriorityBlockingQueue() {
        this(DEFAULT_INITIAL_CAPACITY, null);
    }

    public PriorityBlockingQueue(int initialCapacity) {
        this(initialCapacity, null);
    }
```

```
public PriorityBlockingQueue(int initialCapacity,
                             Comparator<? super E> comparator) {
    if (initialCapacity < 1)
        throw new IllegalArgumentException();
    this.lock = new ReentrantLock();
    this.notEmpty = lock.newCondition();
    this.comparator = comparator;
    this.queue = new Object[initialCapacity];
}
```

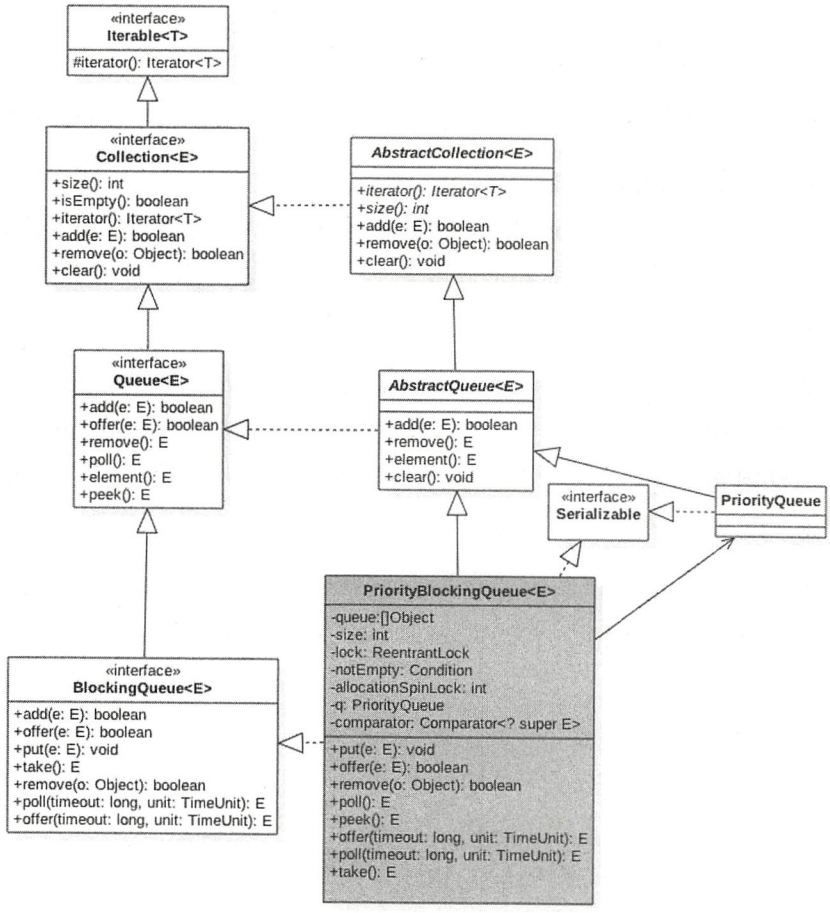


图 7-32

7.4.3　原理介绍

1. offer 操作

offer 操作的作用是在队列中插入一个元素，由于是无界队列，所以一直返回 true。如下是 offer 函数的代码。

```
public boolean offer(E e) {

    if (e == null)
        throw new NullPointerException();

    //获取独占锁
    final ReentrantLock lock = this.lock;
    lock.lock();

    int n, cap;
    Object[] array;

    //(1)如果当前元素个数>=队列容量，则扩容
    while ((n = size) >= (cap = (array = queue).length))
        tryGrow(array, cap);


    try {
        Comparator<? super E> cmp = comparator;

        //(2)默认比较器为null
        if (cmp == null)
            siftUpComparable(n, e, array);
        else
            //(3)自定义比较器
            siftUpUsingComparator(n, e, array, cmp);

        //(9)将队列元素数增加1，并且激活notEmpty的条件队列里面的一个阻塞线程
        size = n + 1;
        notEmpty.signal();//激活因调用take()方法被阻塞的线程
    } finally {
        //释放独占锁
        lock.unlock();
    }
    return true;
}
```

如上代码的主流程比较简单，下面主要看看如何进行扩容和在内部建堆。首先看下面的扩容逻辑。

```
private void tryGrow(Object[] array, int oldCap) {
    lock.unlock(); //释放获取的锁
    Object[] newArray = null;

    //(4)CAS成功则扩容
    if (allocationSpinLock == 0 &&
        UNSAFE.compareAndSwapInt(this, allocationSpinLockOffset,
                                 0, 1)) {
        try {
            //oldGap<64则扩容,执行oldcap+2,否则扩容50%，并且最大为MAX_ARRAY_SIZE
            int newCap = oldCap + ((oldCap < 64) ?
                                   (oldCap + 2) : // grow faster if small
                                   (oldCap >> 1));
            if (newCap - MAX_ARRAY_SIZE > 0) {    // possible overflow
                int minCap = oldCap + 1;
                if (minCap < 0 || minCap > MAX_ARRAY_SIZE)
                    throw new OutOfMemoryError();
                newCap = MAX_ARRAY_SIZE;
            }
            if (newCap > oldCap && queue == array)
                newArray = new Object[newCap];
        } finally {
            allocationSpinLock = 0;
        }
    }

    //(5)第一个线程CAS成功后，第二个线程会进入这段代码，然后第二个线程让出CPU，尽量让第一个线程获取锁，但是这得不到保证。
    if (newArray == null) // back off if another thread is allocating
        Thread.yield();
    lock.lock();//(6)
    if (newArray != null && queue == array) {
        queue = newArray;
        System.arraycopy(array, 0, newArray, 0, oldCap);
    }
}
```

tryGrow 的作用是扩容。这里为啥在扩容前要先释放锁，然后使用 CAS 控制只有一个线程可以扩容成功？其实这里不先释放锁，也是可行的，也就是在整个扩容期间一直持有锁，但是扩容是需要花时间的，如果扩容时还占用锁那么其他线程在这个时候是不能进行

出队和入队操作的，这大大降低了并发性。所以为了提高性能，使用 CAS 控制只有一个线程可以进行扩容，并且在扩容前释放锁，让其他线程可以进行入队和出队操作。

spinlock 锁使用 CAS 控制只有一个线程可以进行扩容，CAS 失败的线程会调用 Thread.yield() 让出 CPU，目的是让扩容线程扩容后优先调用 lock.lock 重新获取锁，但是这得不到保证。有可能 yield 的线程在扩容线程扩容完成前已经退出，并执行代码（6）获取到了锁，这时候获取到锁的线程发现 newArray 为 null 就会执行代码（1）。如果当前数组扩容还没完毕，当前线程会再次调用 tryGrow 方法，然后释放锁，这又给扩容线程获取锁提供了机会，如果这时候扩容线程还没扩容完毕，则当前线程释放锁后又调用 yield 方法让出 CPU。所以当扩容线程进行扩容时，其他线程原地自旋通过代码（1）检查当前扩容是否完毕，扩容完毕后才退出代码（1）的循环。

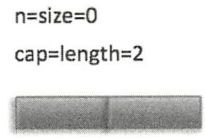
扩容线程扩容完毕后会重置自旋锁变量 allocationSpinLock 为 0，这里并没有使用 UNSAFE 方法的 CAS 进行设置是因为同时只可能有一个线程获取到该锁，并且 allocationSpinLock 被修饰为了 volatile 的。当扩容线程扩容完毕后会执行代码 (6) 获取锁，获取锁后复制当前 queue 里面的元素到新数组。

然后看下面的具体建堆算法。

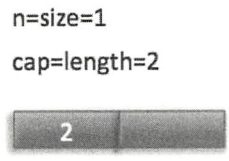
```
private static <T> void siftUpComparable(int k, T x, Object[] array) {
    Comparable<? super T> key = (Comparable<? super T>) x;

    //队列元素个数>0则判断插入位置，否则直接入队(7)
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        Object e = array[parent];
        if (key.compareTo((T) e) >= 0)
            break;
        array[k] = e;
        k = parent;
    }
    array[k] = key;(8)
}
```

下面用图来解释上面算法过程，假设队列初始化容量为 2，创建的优先级队列的泛型参数为 Integer。

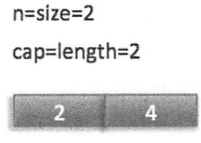
I. 首先调用队列的 offer(2) 方法，希望向队列插入元素 2，插入前队列状态如下所示：



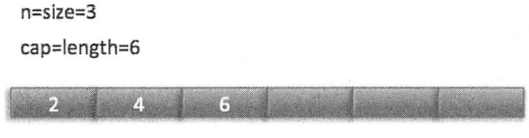
首先执行代码（1），从图中的变量值可知判断结果为 false，所以紧接着执行代码（2）。由于 k=n=size=0，所以代码（7）的判断结果为 false，因此会执行代码（8）直接把元素 2 入队。最后执行代码（9）将 size 的值加 1，这时候队列的状态如下所示：



II. 第二次调用队列的 offer(4) 时，首先执行代码（1），从图中的变量值可知判断结果为 false，所以执行代码（2）。由于 k=1，所以进入 while 循环，由于 parent=0;e=2;key=4;默认元素比较器使用元素的 compareTo 方法，可知 key>e，所以执行 break 退出 siftUpComparable 中的循环，然后把元素存到数组下标为 1 的地方。最后执行代码（9）将 size 的值加 1，这时候队列状态如下所示：



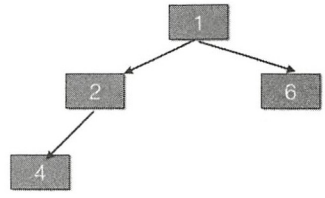
III. 第三次调用队列的 offer(6) 时，首先执行代码（1），从图中的变量值知道，这时候判断结果为 true，所以调用 tryGrow 进行数组扩容。由于 2<64，所以执行 newCap=2 + (2+2)=6，然后创建新数组并复制，之后调用 siftUpComparable 方法。由于 k=2>0，故进入 while 循环，由于 parent=0;e=2;key=6;key>e，所以执行 break 后退出 while 循环，并把元素 6 放入数组下标为 2 的地方。最后将 size 的值加 1，现在队列状态如下所示：



IV. 第四次调用队列的 offer(1) 时，首先执行代码（1），从图中的变量值知道，

这次判断结果为false，所以执行代码（2）。由于k=3，所以进入while循环，由于parent=1;e=4;key=1; key<e，所以把元素4复制到数组下标为3的地方。然后执行k=1，再次循环，发现e=2,key=1,key<e，所以复制元素2到数组下标1处，然后k=0退出循环。最后把元素1存放到下标为0的地方，现在的状态如下所示：

这时候二叉树堆的树形图如下所示：



由此可见，堆的根元素是1，也就是这是一个最小堆，那么当调用这个优先级队列的poll方法时，会依次返回堆里面值最小的元素。

2. poll操作

poll操作的作用是获取队列内部堆树的根节点元素，如果队列为空，则返回null。poll函数的代码如下。

```
public E poll() {
    final ReentrantLock lock = this.lock;
    lock.lock();//获取独占锁
    try {
        return dequeue();
    } finally {
        lock.unlock();//释放独占锁
    }
}
```

如以上代码所示，在进行出队操作时要先加锁，这意味着，当前线程在进行出队操作时，其他线程不能再进行入队和出队操作，但是前面在介绍offer函数时介绍过，这时候其他线程可以进行扩容。下面看下具体执行出队操作的dequeue方法的代码：

```
private E dequeue() {
```

```
    //队列为空，则返回null
    int n = size - 1;
    if (n < 0)
        return null;
    else {


        //(1)获取队头元素
        Object[] array = queue;
        E result = (E) array[0];

        //(2)获取队尾元素，并赋值为null
        E x = (E) array[n];
        array[n] = null;

        Comparator<? super E> cmp = comparator;
        if (cmp == null)//(3)
            siftDownComparable(0, x, array, n);
        else
            siftDownUsingComparator(0, x, array, n, cmp);
        size = n;//(4)
        return result;
    }
}
```

在如上代码中，如果队列为空则直接返回 null，否则执行代码（1）获取数组第一个元素作为返回值存放到变量 Result 中，这里需要注意，数组里面的第一个元素是优先级最小或者最大的元素，出队操作就是返回这个元素。 然后代码（2）获取队列尾部元素并存放到变量 x 中，且置空尾部节点，然后执行代码（3）将变量 x 插入到数组下标为 0 的位置，之后重新调整堆为最大或者最小堆，然后返回。这里重要的是，去掉堆的根节点后，如何使用剩下的节点重新调整一个最大或者最小堆。下面我们看下 siftDownComparable 的实现代码。

```
    private static <T> void siftDownComparable(int k, T x, Object[] array,
                                               int n) {
        if (n > 0) {
            Comparable<? super T> key = (Comparable<? super T>)x;
            int half = n >>> 1;           // loop while a non-leaf
            while (k < half) {
                int child = (k << 1) + 1; // assume left child is least
                Object c = array[child];(5)
```

```
                int right = child + 1; (6)
                if (right < n &&
                    ((Comparable<? super T>) c).compareTo((T) array[right]) > 0)(7)
                    c = array[child = right];
                if (key.compareTo((T) c) <= 0)(8)
                    break;
                array[k] = c;
                k = child;
            }
            array[k] = key;(9)
        }
    }
```

同样下面我们结合图来介绍上面调整堆的算法过程。接着上节队列的状态继续讲解，在上一节中队列元素序列为1、2、6、4。

I. 第一次调用队列的poll()方法时，首先执行代码（1）和代码（2），这时候变量size =4；n=3；result=1；x=4; 此时队列状态如下所示。

然后执行代码（3）调整堆后队列状态为

II. 第二次调用队列的poll()方法时，首先执行代码（1）和代码（2），这时候变量size =3；n=2；result=2；x=6; 此时队列状态为

然后执行代码（3）调整堆后队列状态为

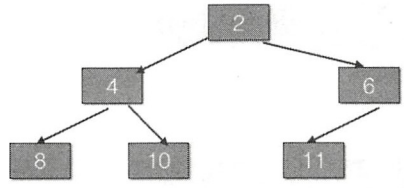
III. 第三次调用队列的poll()方法时，首先执行代码（1）和代码（2），这时候变量size =2；n=1；result=4；x=6; 此时队列状态为

然后执行代码（3）调整堆后队列状态为

IV．第四次直接返回元素 6。

下面重点说说 siftDownComparable 调整堆的算法。 首先介绍下堆调整的思路。由于队列数组第 0 个元素为树根，因此出队时要移除它。这时数组就不再是最小的堆了，所以需要调整堆。具体是从被移除的树根的左右子树中找一个最小的值来当树根，左右子树又会找自己左右子树里面那个最小值，这是一个递归过程，直到树叶节点结束递归。如果不太明白，没关系，下面我们结合图来说明，假如当前队列内容如下：

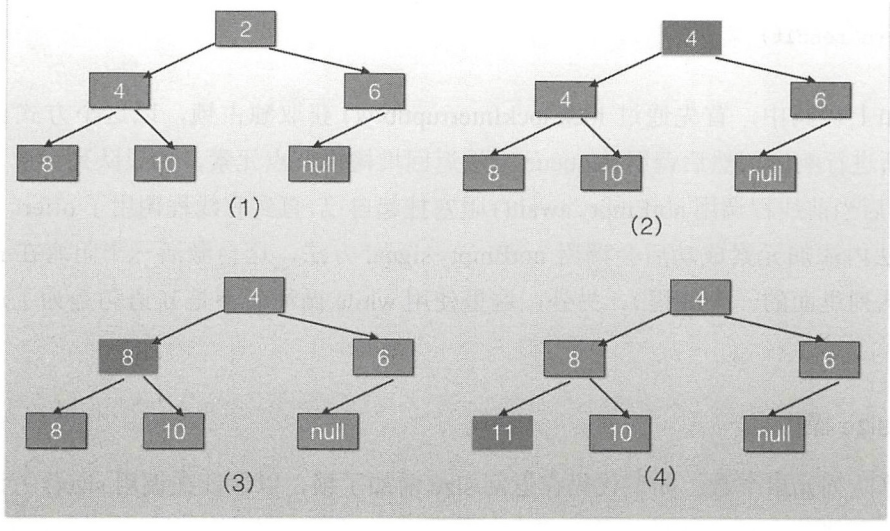
其对应的二叉堆树为：



这时候如果调用了 poll()，那么 result=2; x=11，并且队列末尾的元素被设置为 null，然后对于剩下的元素，调整堆的步骤如下图所示：

图（1）中树根的 leftChildVal = 4; rightChildVal = 6; 由于 4<6，所以 c=4。然后由于 11>4，也就是 key>c，所以使用元素 4 覆盖树根节点的值，现在堆对应的树如图（2）所示。

然后树根的左子树树根的左右孩子节点中的 leftChildVal = 8;rightChildVal = 10; 由于 8<10，所以 c=8。然后由于 11>8，也就是 key>c，所以元素 8 作为树根左子树的根节点，现在树的形状如图（3）所示。这时候判断是否 k<half，结果为 false，所以退出循环。然后把 x=11 的元素设置到数组下标为 3 的地方，这时候堆树如图（4）所示，至此调整堆完毕。siftDownComparable 返回的 result=2，所以 poll 方法也返回了。



3. put 操作

put 操作内部调用的是 offer 操作，由于是无界队列，所以不需要阻塞。

```
public void put(E e) {
    offer(e); // never need to block
}
```

4. take 操作

take 操作的作用是获取队列内部堆树的根节点元素，如果队列为空则阻塞，如以下代码所示。

```
public E take() throws InterruptedException {
    //获取锁，可被中断
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    E result;
    try {

        //如果队列为空，则阻塞，把当前线程放入notEmpty的条件队列
        while ( (result = dequeue()) == null)
            notEmpty.await();//阻塞当前线程
    } finally {
        lock.unlock();//释放锁
```

```
    }
    return result;
}
```

在如上代码中，首先通过 lock.lockInterruptibly() 获取独占锁，以这个方式获取的锁会对中断进行响应。然后调用 dequeue 方法返回堆树根节点元素，如果队列为空，则返回 false。然后当前线程调用 notEmpty.await() 阻塞挂起自己，直到有线程调用了 offer()方法(在 offer 方法内添加元素成功后会调用 notEmpty.signal 方法，这会激活一个阻塞在 notEmpty 的条件队列里面的一个线程)。另外，这里使用 while 循环而不是 if 语句是为了避免虚假唤醒。

5. size 操作

计算队列元素个数。如下代码在返回 size 前加了锁，以保证在调用 size() 方法时不会有其他线程进行入队和出队操作。另外，由于 size 变量没有被修饰为 volatie 的，所以这里加锁也保证了在多线程下 size 变量的内存可见性。

```
public int size() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return size;
    } finally {
        lock.unlock();
    }
}
```

7.4.4 案例介绍

下面我们通过一个案例来体会 PriorityBlockingQueue 的使用方法。在这个案例中，会把具有优先级的任务放入队列，然后从队列里面逐个获取优先级最高的任务来执行。

```
public class TestPriorityBlockingQueue {

    static class Task implements Comparable<Task> {

        public int getPriority() {
            return priority;
        }
```

```
        public void setPriority(int priority) {
            this.priority = priority;
        }

        public String getTaskName() {
            return taskName;
        }

        public void setTaskName(String taskName) {
            this.taskName = taskName;
        }

        private int priority = 0;

        private String taskName;

        @Override
        public int compareTo(Task o) {

            if (this.priority >= o.getPriority()) {
                return 1;
            } else {
                return -1;
            }
        }

        public void doSomeThing(){
            System.out.println(taskName + ":" + priority);
        }

    }

    public static void main(String[] args) {

        //创建任务，并添加到队列
        PriorityBlockingQueue<Task> priorityQueue = new
PriorityBlockingQueue<Task>();
        Random random = new Random();
        for(int i=0;i<10;++i){
            Task task = new Task();
            task.setPriority(random.nextInt(10));
            task.setTaskName("taskName" +i);
            priorityQueue.offer(task);
        }
```

```
        //取出任务执行
        while(!priorityQueue.isEmpty()){
            Task task = priorityQueue.poll();
            if(null != task){
                task.doSomeThing();
            }
        }

    }
}
```

如上代码首先创建了一个 Task 类，该类继承了 Comparable 方法并重写了 compareTo 方法，自定义了元素优先级比较规则。然后在 main 函数里面创建了一个优先级队列，并使用随机数生成器生成 10 个随机的有优先级的任务，并将它们添加到优先级队列。最后从优先级队列里面逐个获取任务并执行。运行上面代码，一个可能的输出如下所示。

```
taskName7:0
taskName6:1
taskName9:1
taskName1:2
taskName5:3
taskName0:3
taskName3:4
taskName8:5
taskName2:7
taskName4:7
```

从结果可知，任务执行的先后顺序和它们被放入队列的先后顺序没有关系，而是和它们的优先级有关系。

7.4.5 小结

PriorityBlockingQueue 队列在内部使用二叉树堆维护元素优先级，使用数组作为元素存储的数据结构，这个数组是可扩容的。当当前元素个数 >= 最大容量时会通过 CAS 算法扩容，出队时始终保证出队的元素是堆树的根节点，而不是在队列里面停留时间最长的元素。使用元素的 compareTo 方法提供默认的元素优先级比较规则，用户可以自定义优先级的比较规则。

如图 7-33 所示，PriorityBlockingQueue 类似于 ArrayBlockingQueue，在内部使用一