

阻塞队列介绍
Queue接口
BlockingQueue接口
BlockingQueue常用方法示例
阻塞队列特性
阻塞
是否有界
应用场景
常见阻塞队列
ArrayBlockingQueue
ArrayBlockingQueue使用
ArrayBlockingQueue的原理
数据结构
入队put方法
出队take方法
LinkedBlockingQueue
LinkedBlockingQueue使用
LinkedBlockingQueue的原理
数据结构
构造器
入队put方法
出队take方法

LinkedBlockingQueue与ArrayBlockingQueue对比

SynchronousQueue

应用场景

SynchronousQueue使用

PriorityBlockingQueue

应用场景

PriorityBlockingQueue使用

如何构造优先级队列

DelayQueue

DelayQueue使用

DelayQueue的原理

数据结构

入队put方法

出队take方法

如何选择适合的阻塞队列

线程池对于阻塞队列的选择

选择策略

阻塞队列介绍

Queue接口

```
1 public interface Queue<E> extends Collection<E> {
2     //添加一个元素，添加成功返回true，如果队列满了，就会抛出异常
3     boolean add(E e);
4     //添加一个元素，添加成功返回true，如果队列满了，返回false
5     boolean offer(E e);
6     //返回并删除队首元素，队列为空则抛出异常
```

```
7     E remove();
8     //返回并删除队首元素，队列为空则返回null
9     E poll();
10    //返回队首元素，但不移除，队列为空则抛出异常
11    E element();
12    //获取队首元素，但不移除，队列为空则返回null
13    E peek();
```

BlockingQueue接口

BlockingQueue 继承了 Queue 接口，是队列的一种。Queue 和 BlockingQueue 都是在 Java 5 中加入的。**阻塞队列（BlockingQueue）**是一个在队列基础上又支持了两个附加操作的队列，常用解耦。两个附加操作：

- 支持阻塞的插入方法put: 队列满时，队列会阻塞插入元素的线程，直到队列不满。
- 支持阻塞的移除方法take: 队列空时，获取元素的线程会等待队列变为非空

```
public interface BlockingQueue<E> extends Queue<E> {
    /**...*/
    boolean add( @NotNull E e);

    /**...*/
    boolean offer( @NotNull E e);

    /**...*/
    void put( @NotNull E e) throws InterruptedException;

    /**...*/
    boolean offer(E e, long timeout, @NotNull TimeUnit unit)
        throws InterruptedException;

    /**...*/
    @NotNull E take() throws InterruptedException;

    /**...*/
    @Nullable E poll(long timeout, @NotNull TimeUnit unit)
        throws InterruptedException;

    /**...*/
    int remainingCapacity();

    /**...*/
    boolean remove(Object o);
```

BlockingQueue和JDK集合包中的Queue接口兼容，同时在其基础上增加了阻塞功能。

入队：

- (1) offer(E e): 如果队列没满，返回true，如果队列已满，返回false（不阻塞）
- (2) offer(E e, long timeout, TimeUnit unit): 可以设置阻塞时间，如果队列已满，则进行阻塞。超过阻塞时间，则返回false
- (3) **put(E e): 队列没满的时候是正常的插入，如果队列已满，则阻塞，直至队列空出位置**

出队：

- (1) poll(): 如果有数据，出队，如果没有数据，返回null（不阻塞）
- (2) poll(long timeout, TimeUnit unit): 可以设置阻塞时间，如果没有数据，则阻塞，超过阻塞时间，则返回null
- (3) **take(): 队列里有数据会正常取出数据并删除；但是如果队列里无数据，则阻塞，直到队列里有数据**

BlockingQueue常用方法示例

当队列满了无法添加元素，或者是队列空了无法移除元素时：

- 1. 抛出异常：add、remove、element
- 2. 返回结果但不抛出异常：offer、poll、peek
- 3. 阻塞：put、take

方法	抛出异常	返回特定值	阻塞	阻塞特定时间
入队	add(e)	offer(e)	put(e)	offer(e, time, unit)
出队	remove()	poll()	take()	poll(time, unit)
获取队首元素	element()	peek()	不支持	不支持

```
1 public class BlockingQueueTest {
2
3     public static void main(String[] args) {
4
5         addTest();
6
7     }
8
9     /**
10      * add 方法是往队列里添加一个元素，如果队列满了，就会抛出异常来提示队列已满。
11      */
12     private static void addTest() {
13         BlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<Integer>(2);
14         System.out.println(blockingQueue.add(1));
15         System.out.println(blockingQueue.add(2));
16         System.out.println(blockingQueue.add(3));
17     }
18
19     /**
20      * remove 方法的作用是删除元素并返回队列的头节点，如果删除的队列是空的， remove 方法就会抛出
21      */
22     private static void removeTest() {
23         ArrayBlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<Integer>(2);
24         blockingQueue.add(1);
25         blockingQueue.add(2);
26
27         System.out.println(blockingQueue.remove());
```

```
27         System.out.println(blockingQueue.remove());
28         System.out.println(blockingQueue.remove());
29     }
30
31     /**
32      * element 方法是返回队列的头部节点，但是并不删除。如果队列为空，抛出异常
33      */
34     private static void elementTest() {
35         ArrayBlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<Integer>(2);
36         blockingQueue.element();
37     }
38
39     /**
40      * offer 方法用来插入一个元素。如果添加成功会返回 true，而如果队列已经满了，返回false
41      */
42     private static void offerTest(){
43         ArrayBlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<Integer>(2);
44         System.out.println(blockingQueue.offer(1));
45         System.out.println(blockingQueue.offer(2));
46         System.out.println(blockingQueue.offer(3));
47     }
48
49     /**
50      * poll 方法作用也是移除并返回队列的头节点。 如果队列为空，返回null
51      */
52     private static void pollTest() {
53         ArrayBlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<Integer>(3);
54         blockingQueue.offer(1);
55         blockingQueue.offer(2);
56         blockingQueue.offer(3);
57         System.out.println(blockingQueue.poll());
58         System.out.println(blockingQueue.poll());
59         System.out.println(blockingQueue.poll());
60         System.out.println(blockingQueue.poll());
61     }
62
63     /**
64      * peek 方法返回队列的头元素但并不删除。 如果队列为空，返回null
65      */
66     private static void peekTest() {
```

```

67     ArrayBlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<Integer>(2);
68     System.out.println(blockingQueue.peek());
69 }
70
71 /**
72  * put 方法的作用是插入元素。如果队列已满就无法继续插入,阻塞插入线程,直至队列空出位置
73  */
74 private static void putTest(){
75     BlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<Integer>(2);
76     try {
77         blockingQueue.put(1);
78         blockingQueue.put(2);
79         blockingQueue.put(3);
80     } catch (InterruptedException e) {
81         e.printStackTrace();
82     }
83 }
84
85 /**
86  * take 方法的作用是获取并移除队列的头结点。如果队列表里无数据,则阻塞,直到队列表里有数据
87  */
88 private static void takeTest(){
89     BlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<Integer>(2);
90     try {
91         blockingQueue.take();
92     } catch (InterruptedException e) {
93         e.printStackTrace();
94     }
95 }
96

```

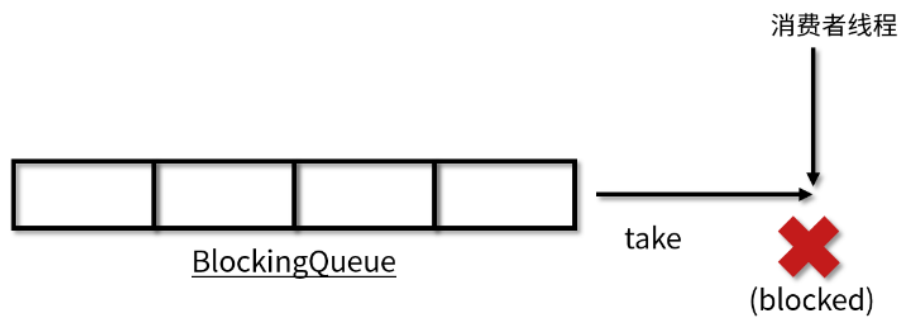
阻塞队列特性

阻塞

阻塞队列区别于其他类型的队列的最主要的特点就是“阻塞”这两个字,所以下面重点介绍阻塞功能:阻塞功能使得生产者和消费者两端的能力得以平衡,当有任何一端速度过快时,阻塞队列便会把过快的速度给降下来。实现阻塞最重要的两个方法是 take 方法和 put 方法。

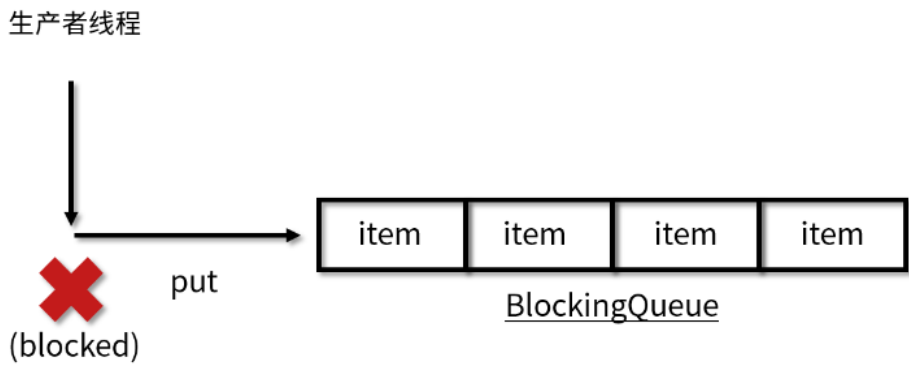
take 方法

take 方法的功能是获取并移除队列的头结点，通常在队列里有数据的时候是可以正常移除的。可是一旦执行 take 方法的时候，队列里无数据，则阻塞，直到队列里有数据。一旦队列里有数据了，就会立刻解除阻塞状态，并且取到数据。过程如图所示：



put 方法

put 方法插入元素时，如果队列没有满，那就和普通的插入一样是正常的插入，但是如果队列已满，那么就无法继续插入，则阻塞，直到队列里有了空闲空间。如果后续队列有了空闲空间，比如消费者消费了一个元素，那么此时队列就会解除阻塞状态，并把需要添加的数据添加到队列中。过程如图所示：



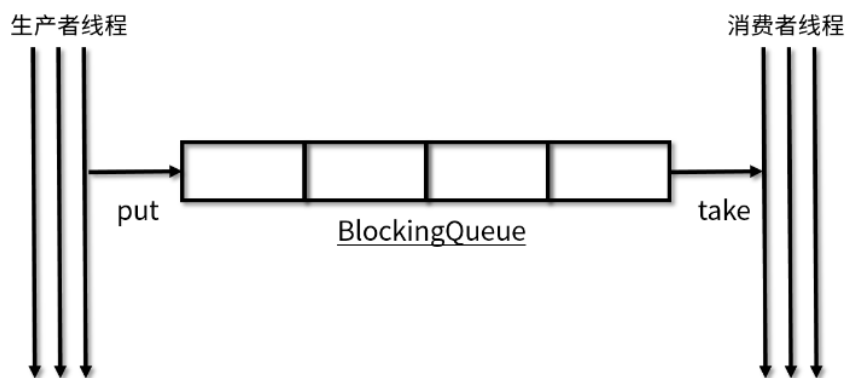
思考：阻塞队列是否有容量限制？

是否有界

阻塞队列还有一个非常重要的属性，那就是容量的大小，分为有界和无界两种。无界队列意味着里面可以容纳非常多的元素，例如 `LinkedBlockingQueue` 的上限是 `Integer.MAX_VALUE`，是非常大的一个数，可以近似认为是无限容量，因为我们几乎无法把这个容量装满。但是有的阻塞队列是有界的，例如 `ArrayBlockingQueue` 如果容量满了，也不会扩容，所以一旦满了就无法再往里放数据了。

应用场景

`BlockingQueue` 是线程安全的，我们在很多场景下都可以利用线程安全的队列来优雅地解决我们业务自身的线程安全问题。比如说，使用生产者/消费者模式的时候，我们生产者只需要往队列里添加元素，而消费者只需要从队列里取出它们就可以了，如图所示：



因为阻塞队列是线程安全的，所以生产者和消费者都可以是多线程的，不会发生线程安全问题。生产者/消费者直接使用线程安全的队列就可以，而不需要自己去考虑更多的线程安全问题。这也就意味着，考虑锁等线程安全问题的重任从“你”转移到了“队列”上，降低了我们开发的难度和工作量。

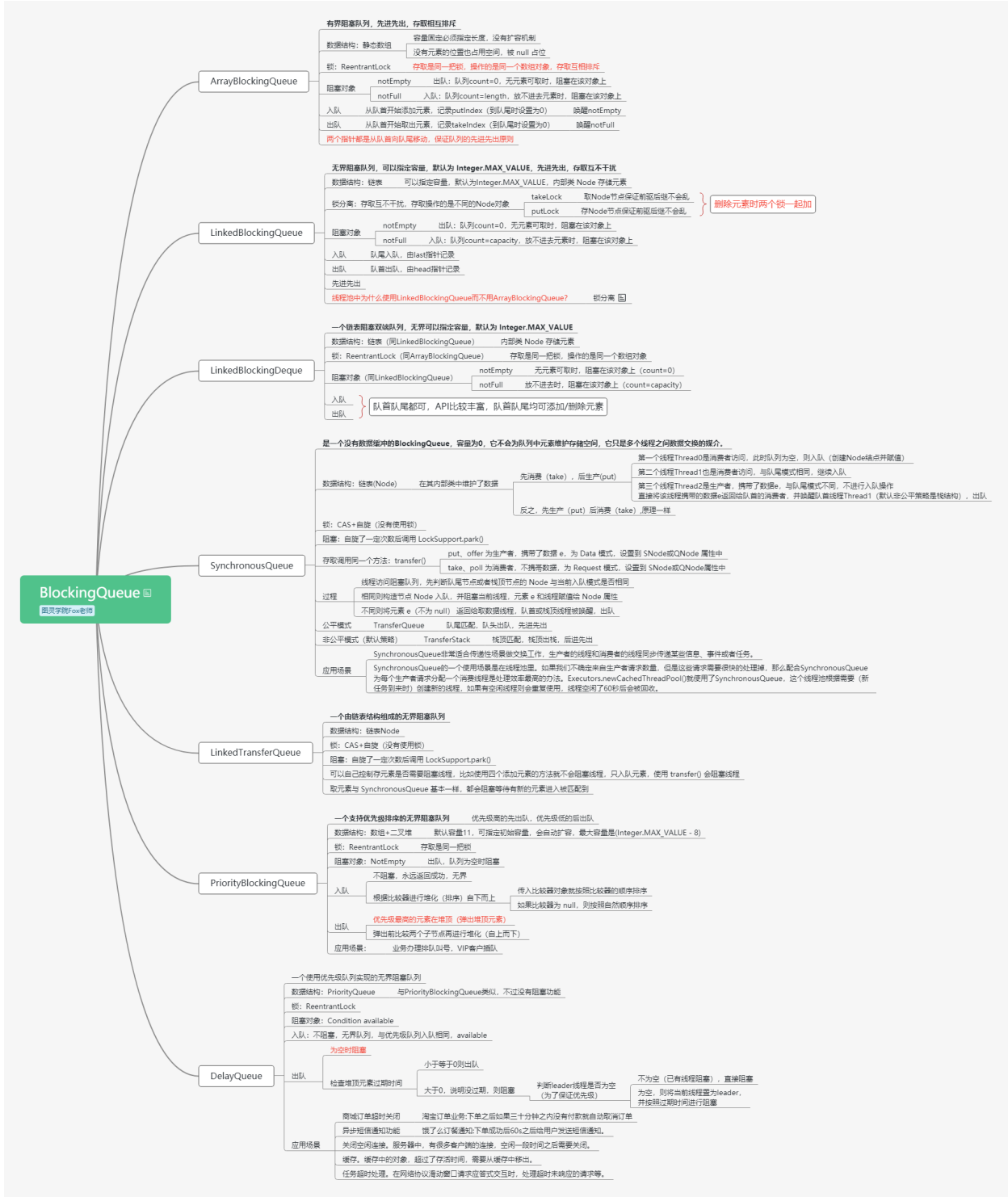
同时，队列它还能起到一个隔离的作用。比如说我们开发一个银行转账的程序，那么生产者线程不需要关心具体的转账逻辑，只需要把转账任务，如账户和金额等信息放到队列中就可以，而不需要去关心银行这个类如何实现具体的转账业务。而作为银行这个类来讲，它会去从队列里取出来将要执行的具体的任务，再去通过自己的各种方法来完成本次转账。这样就实现了具体任务与执行任务类之间的解耦，任务被放在了阻塞队列中，而负责放任务的线程是无法直接访问到我们银行具体实现转账操作的对象的，实现了隔离，提高了安全性。

常见阻塞队列

BlockingQueue 接口的实现类都被放在了 juc 包中，它们的区别主要体现在存储结构上或对元素操作上的不同，但是对于take与put操作的原理，却是类似的。

队列	描述
ArrayBlockingQueue	基于数组结构实现的一个有界阻塞队列
LinkedBlockingQueue	基于链表结构实现的一个无界阻塞队列，指定容量为有界阻塞队列
PriorityBlockingQueue	支持按优先级排序的无界阻塞队列
DelayQueue	基于优先级队列（PriorityBlockingQueue）实现的无界阻塞队列
SynchronousQueue	不存储元素的阻塞队列
LinkedTransferQueue	基于链表结构实现的一个无界阻塞队列
LinkedBlockingDeque	基于链表结构实现的一个双端阻塞队列

<https://www.processon.com/view/link/618ce3941e0853689b0818e2>



ArrayBlockingQueue

ArrayBlockingQueue是最典型的有界阻塞队列，其内部是用数组存储元素的，初始化时需要指定容量大小，利用 ReentrantLock 实现线程安全。

在生产者-消费者模型中使用时，如果生产速度和消费速度基本匹配的情况下，使用 ArrayBlockingQueue是个不错选择；当如果生产速度远远大于消费速度，则会导致队列填满，大量生产线程被阻塞。

使用独占锁ReentrantLock实现线程安全，入队和出队操作使用同一个锁对象，也就是只能有一个线程可以进行入队或者出队操作；这也就意味着生产者和消费者无法并行操作，在高并发场景下会成为性能瓶颈。



ArrayBlockingQueue使用

```
1 BlockingQueue queue = new ArrayBlockingQueue(1024);
2 queue.put("1"); //向队列中添加元素
```

ArrayBlockingQueue的原理

数据结构

利用了Lock锁的Condition通知机制进行阻塞控制。

核心：一把锁，两个条件

```
1 //数据元素数组
2 final Object[] items;
3 //下一个待取出元素索引
4 int takeIndex;
5 //下一个待添加元素索引
6 int putIndex;
7 //元素个数
8 int count;
9 //内部锁
10 final ReentrantLock lock;
11 //消费者
12 private final Condition notEmpty;
13 //生产者
14 private final Condition notFull;
15
16 public ArrayBlockingQueue(int capacity) {
17     this(capacity, false);
18 }
19 public ArrayBlockingQueue(int capacity, boolean fair) {
20     ...
21     lock = new ReentrantLock(fair); //公平，非公平
```

```
22     notEmpty = lock.newCondition();
23     notFull = lock.newCondition();
24 }
```

入队put方法

```
1  public void put(E e) throws InterruptedException {
2      //检查是否为空
3      checkNotNull(e);
4      final ReentrantLock lock = this.lock;
5      //加锁，如果线程中断抛出异常
6      lock.lockInterruptibly();
7      try {
8          //阻塞队列已满，则将生产者挂起，等待消费者唤醒
9          //设计注意点： 用while不用if是为了防止虚假唤醒
10         while (count == items.length)
11             notFull.await(); //队列满了，使用notFull等待（生产者阻塞）
12         // 入队
13         enqueue(e);
14     } finally {
15         lock.unlock(); // 唤醒消费者线程
16     }
17 }
18
19 private void enqueue(E x) {
20     final Object[] items = this.items;
21     //入队 使用的putIndex
22     items[putIndex] = x;
23     if (++putIndex == items.length)
24         putIndex = 0; //设计的精髓： 环形数组，putIndex指针到数组尽头了，返回头部
25     count++;
26     //notEmpty条件队列转同步队列，准备唤醒消费者线程，因为入队了一个元素，肯定不为空了
27     notEmpty.signal();
28 }
```

出队take方法

```
1  public E take() throws InterruptedException {
```

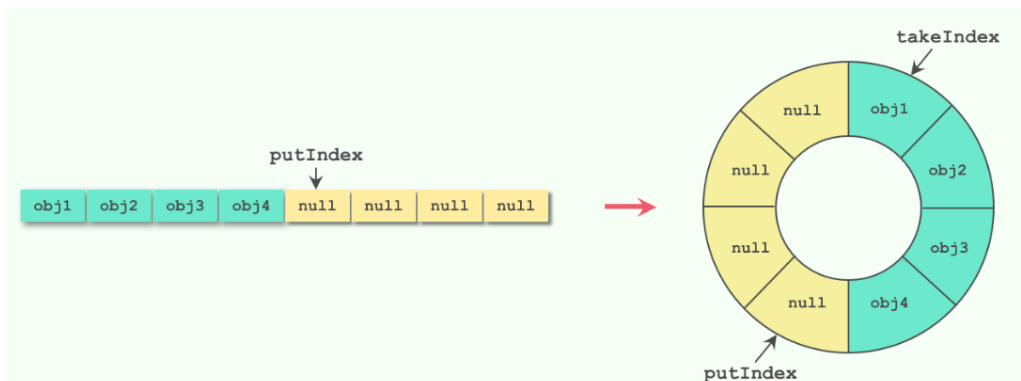
```

2    final ReentrantLock lock = this.lock;
3    //加锁，如果线程中断抛出异常
4    lock.lockInterruptibly();
5    try {
6        //如果队列为空，则消费者挂起
7        while (count == 0)
8            notEmpty.await();
9        //出队
10       return dequeue();
11    } finally {
12        lock.unlock();// 唤醒生产者线程
13    }
14 }

15 private E dequeue() {
16     final Object[] items = this.items;
17     @SuppressWarnings("unchecked")
18     E x = (E) items[takeIndex]; //取出takeIndex位置的元素
19     items[takeIndex] = null;
20     if (++takeIndex == items.length)
21         takeIndex = 0; //设计的精髓： 环形数组，takeIndex 指针到数组尽头了，返回头部
22     count--;
23     if (itrs != null)
24         itrs.elementDequeued();
25     //notFull条件队列转同步队列，准备唤醒生产者线程，此时队列有空位
26     notFull.signal();
27     return x;
28 }

```

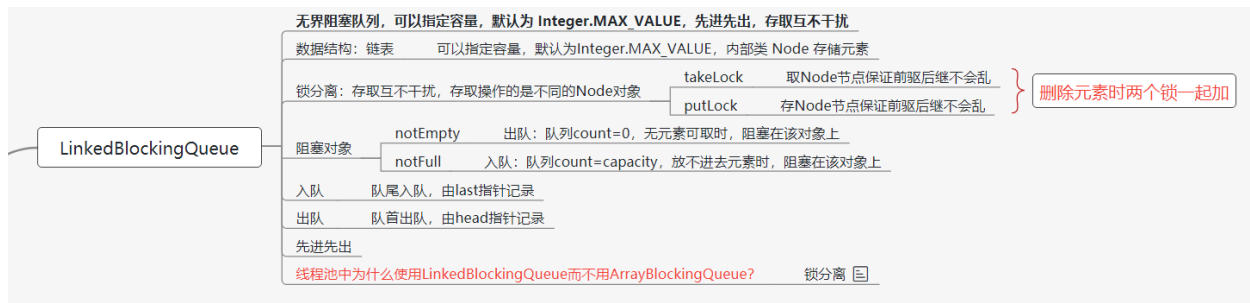
思考：为什么对数组操作要设计成双指针？



LinkedBlockingQueue

LinkedBlockingQueue是一个基于链表实现的阻塞队列，默认情况下，该阻塞队列的大小为Integer.MAX_VALUE，由于这个数值特别大，所以 LinkedBlockingQueue 也被称作无界队列，代表它几乎没有界限，队列可以随着元素的添加而动态增长，但是如果没有剩余内存，则队列将抛出OOM错误。所以为了避免队列过大造成机器负载或者内存爆满的情况出现，我们在使用的时候建议手动传一个队列的大小。

LinkedBlockingQueue内部由单链表实现，只能从head取元素，从tail添加元素。LinkedBlockingQueue采用两把锁的锁分离技术实现入队出队互不阻塞，添加元素和获取元素都有独立的锁，也就是说LinkedBlockingQueue是读写分离的，读写操作可以并行执行。



LinkedBlockingQueue使用

```
1 //指定队列的大小创建有界队列
2 BlockingQueue<Integer> boundedQueue = new LinkedBlockingQueue<>(100);
3 //无界队列
4 BlockingQueue<Integer> unboundedQueue = new LinkedBlockingQueue<>();
```

LinkedBlockingQueue的原理

数据结构

```
1 // 容量,指定容量就是有界队列
2 private final int capacity;
3 // 元素数量
4 private final AtomicInteger count = new AtomicInteger();
5 // 链表头 本身是不存储任何元素的,初始化时item指向null
6 transient Node<E> head;
7 // 链表尾
8 private transient Node<E> last;
9 // take锁 锁分离,提高效率
10 private final ReentrantLock takeLock = new ReentrantLock();
11 // notEmpty条件
```

```

12 // 当队列无元素时，take锁会阻塞在notEmpty条件上，等待其它线程唤醒
13 private final Condition notEmpty = takeLock.newCondition();
14 // put锁
15 private final ReentrantLock putLock = new ReentrantLock();
16 // notFull条件
17 // 当队列满了时，put锁会会阻塞在notFull上，等待其它线程唤醒
18 private final Condition notFull = putLock.newCondition();
19
20 //典型的单链表结构
21 static class Node<E> {
22     E item; //存储元素
23     Node<E> next; //后继节点    单链表结构
24     Node(E x) { item = x; }
25 }

```

构造器

```

1 public LinkedBlockingQueue() {
2     // 如果没传容量，就使用最大int值初始化其容量
3     this(Integer.MAX_VALUE);
4 }
5
6 public LinkedBlockingQueue(int capacity) {
7     if (capacity <= 0) throw new IllegalArgumentException();
8     this.capacity = capacity;
9     // 初始化head和last指针为空值节点
10    last = head = new Node<E>(null);
11 }

```

入队put方法

```

1 public void put(E e) throws InterruptedException {
2     // 不允许null元素
3     if (e == null) throw new NullPointerException();
4     int c = -1;
5     // 新建一个节点
6     Node<E> node = new Node<E>(e);
7     final ReentrantLock putLock = this.putLock;
8     final AtomicInteger count = this.count;
9     // 使用put锁加锁

```

```

10    putLock.lockInterruptibly();
11    try {
12        // 如果队列满了，就阻塞在notFull上等待被其它线程唤醒（阻塞生产者线程）
13        while (count.get() == capacity) {
14            notFull.await();
15        }
16        // 队列不满，就入队
17        enqueue(node);
18        c = count.getAndIncrement(); // 队列长度加1，返回原值
19        // 如果现队列长度小于容量，notFull条件队列转同步队列，准备唤醒一个阻塞在notFull条件上的线程
20        // 这里为啥要唤醒一下呢？
21        // 因为可能有很多线程阻塞在notFull这个条件上，而取元素时只有取之前队列是满的才会唤醒notFull
22        if (c + 1 < capacity)
23            notFull.signal();
24    } finally {
25        putLock.unlock(); // 真正唤醒生产者线程
26    }
27    // 如果原队列长度为0，现在加了一个元素后立即唤醒阻塞在notEmpty上的线程
28    if (c == 0)
29        signalNotEmpty();
30 }
31 private void enqueue(Node<E> node) {
32     // 直接加到last后面，last指向入队元素
33     last = last.next = node;
34 }
35 private void signalNotEmpty() {
36     final ReentrantLock takeLock = this.takeLock;
37     takeLock.lock(); // 加take锁
38     try {
39         notEmpty.signal(); // notEmpty条件队列转同步队列，准备唤醒阻塞在notEmpty上的线程
40     } finally {
41         takeLock.unlock(); // 真正唤醒消费者线程
42     }

```

出队take方法

```

1  public E take() throws InterruptedException {
2      E x;
3      int c = -1;
4      final AtomicInteger count = this.count;

```

```

5    final ReentrantLock takeLock = this.takeLock;
6    // 使用takeLock加锁
7    takeLock.lockInterruptibly();
8    try {
9        // 如果队列无元素，则阻塞在notEmpty条件上（消费者线程阻塞）
10       while (count.get() == 0) {
11           notEmpty.await();
12       }
13       // 否则，出队
14       x = dequeue();
15       c = count.getAndDecrement();//长度-1，返回原值
16       if (c > 1)// 如果取之前队列长度大于1，notEmpty条件队列转同步队列，准备唤醒阻塞在notEmpty
17           notEmpty.signal();
18   } finally {
19       takeLock.unlock(); // 真正唤醒消费者线程
20   }
21   // 为什么队列是满的才唤醒阻塞在notFull上的线程呢？
22   // 因为唤醒是需要加putLock的，这是为了减少锁的次数,所以，这里索性在放完元素就检测一下，未满足
23   // 这也是锁分离带来的代价
24   // 如果取之前队列长度等于容量（已满），则唤醒阻塞在notFull的线程
25   if (c == capacity)
26       signalNotFull();
27   return x;
28 }
29 private E dequeue() {
30     // head节点本身是不存储任何元素的
31     // 这里把head删除，并把head下一个节点作为新的值
32     // 并把其值置空，返回原来的值
33     Node<E> h = head;
34     Node<E> first = h.next;
35     h.next = h; // 方便GC
36     head = first;
37     E x = first.item;
38     first.item = null;
39     return x;
40 }
41 private void signalNotFull() {
42     final ReentrantLock putLock = this.putLock;
43     putLock.lock();
44     try {

```



```
45         notFull.signal();// notFull条件队列转同步队列，准备唤醒阻塞在notFull上的线程
46     } finally {
47         putLock.unlock(); // 解锁，这才会真正的唤醒生产者线程
48     }
```

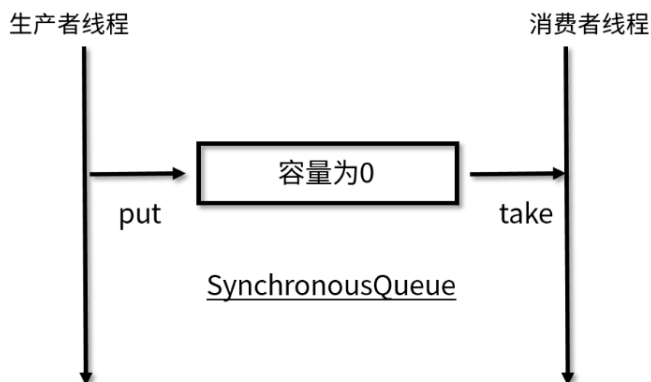
LinkedBlockingQueue与ArrayBlockingQueue对比

LinkedBlockingQueue是一个阻塞队列，内部由两个ReentrantLock来实现出入队列的线程安全，由各自的Condition对象的await和signal来实现等待和唤醒功能。它和ArrayBlockingQueue的不同点在于：

- 队列大小有所不同，ArrayBlockingQueue是有界的初始化必须指定大小，而LinkedBlockingQueue可以是有界的也可以是无界的(Integer.MAX_VALUE)，对于后者而言，当添加速度大于移除速度时，在无界的情况下，可能会造成内存溢出等问题。
- **数据存储容器不同**，ArrayBlockingQueue采用的是数组作为数据存储容器，而LinkedBlockingQueue采用的则是以Node节点作为连接对象的链表。
- 由于ArrayBlockingQueue采用的是数组的存储容器，因此在插入或删除元素时不会产生或销毁任何额外的对象实例，而LinkedBlockingQueue则会生成一个额外的Node对象。这可能在长时间内需要高效并发地处理大批量数据的时候，对于GC可能存在较大影响。
- 两者的实现队列添加或移除的锁不一样，**ArrayBlockingQueue实现的队列中的锁是没有分离的**，即添加操作和移除操作采用的同一个ReentrantLock锁，而**LinkedBlockingQueue实现的队列中的锁是分离的**，其添加采用的是putLock，移除采用的则是takeLock，这样能大大提高队列的吞吐量，也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

SynchronousQueue

SynchronousQueue是一个没有数据缓冲的BlockingQueue，生产者线程对其的插入操作put必须等待消费者的移除操作take。



如图所示，SynchronousQueue 最大的不同之处在于，**它的容量为 0**，所以没有一个地方来暂存元素，导致**每次取数据都要先阻塞，直到有数据被放入**；同理，**每次放数据的时候也会阻塞，直到有消费者来取**。

需要注意的是，SynchronousQueue 的容量不是 1 而是 0，因为 SynchronousQueue 不需要去持有元素，它所做的就是直接传递（direct handoff）。由于每当需要传递的时候，SynchronousQueue 会把元素直接从生产者传给消费者，在此期间并不需要做存储，所以如果运用得当，它的效率是很高的。

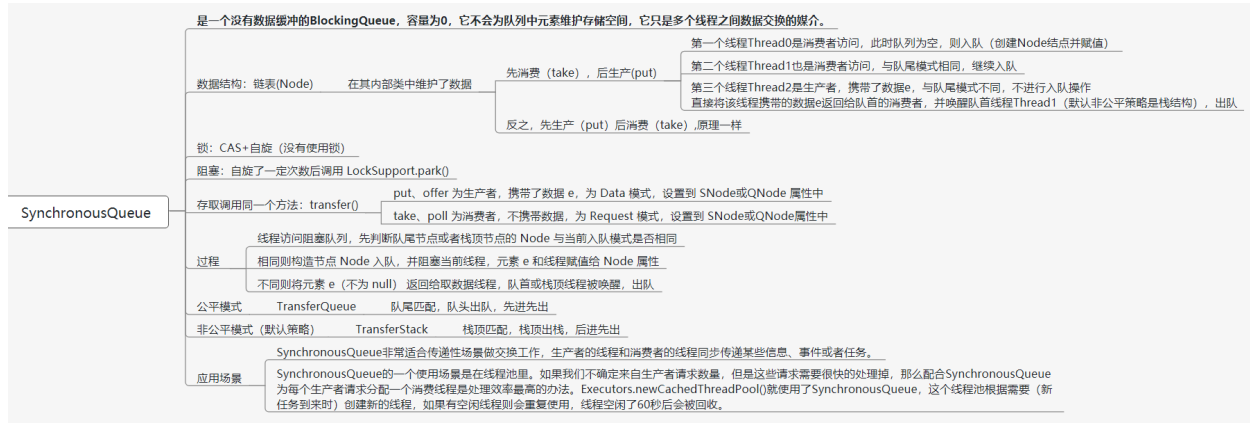
应用场景

SynchronousQueue 非常适合传递性场景做交换工作，生产者的线程和消费者的线程同步传递某些信息、事件或者任务。

SynchronousQueue 的一个使用场景是在线程池里。如果我们不确定来自生产者请求数量，但是这些请求需要很快的处理掉，那么配合 SynchronousQueue 为每个生产者请求分配一个消费线程是处理效率最高的办法。Executors.newCachedThreadPool() 就使用了 SynchronousQueue，这个线程池根据需要（新任务到来时）创建新的线程，如果有空闲线程则会重复使用，线程空闲了 60 秒后会被回收。

SynchronousQueue 使用

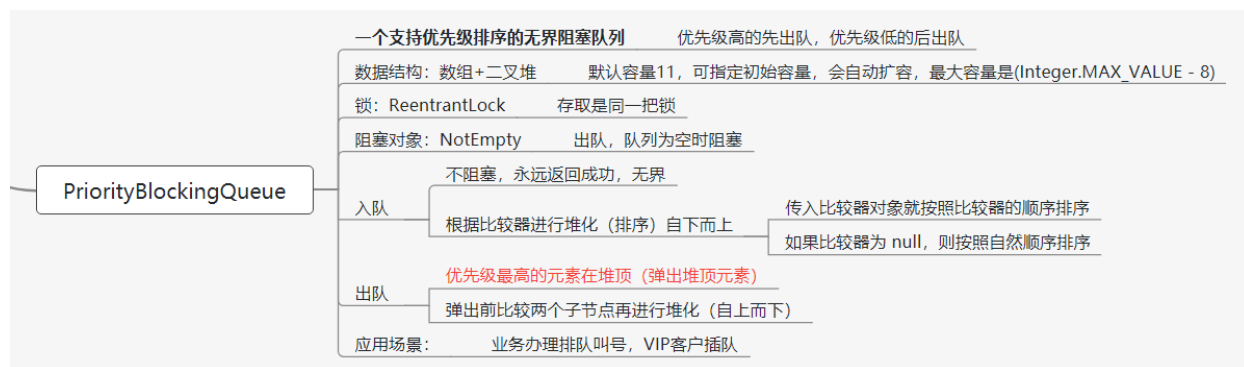
```
1 BlockingQueue<Integer> synchronousQueue = new SynchronousQueue<>();
```



PriorityBlockingQueue

PriorityBlockingQueue是一个无界的基于数组的优先级阻塞队列，数组的默认长度是11，虽然指定了数组的长度，但是可以无限的扩充，直到资源消耗尽为止，每次出队都返回优先级别最高的或者最低的元素。默认情况下元素采用自然顺序升序排序，当然我们也可以通过构造函数来指定Comparator来对元素进行排序。需要注意的是PriorityBlockingQueue不能保证同优先级元素的顺序。

优先级队列PriorityQueue：队列中每个元素都有一个优先级，出队的时候，优先级最高的先出。



应用场景

电商抢购活动，会员级别高的用户优先抢购到商品

银行办理业务，vip客户插队

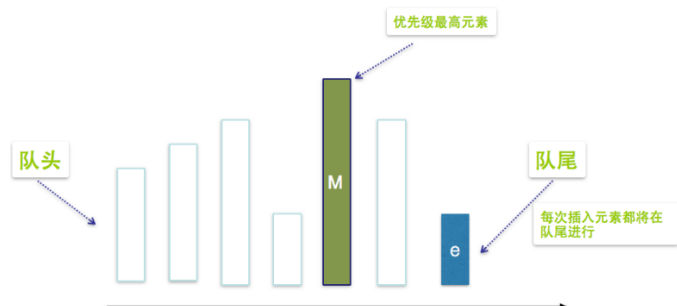
PriorityBlockingQueue使用

```
1 //创建优先级阻塞队列 Comparator为null,自然排序
2 PriorityBlockingQueue<Integer> queue=new PriorityBlockingQueue<Integer>(5);
3 //自定义Comparator
4 PriorityBlockingQueue queue=new PriorityBlockingQueue<Integer>(
5     5, new Comparator<Integer>() {
6         @Override
7         public int compare(Integer o1, Integer o2) {
8             return o2-o1;
9         }
10    })
```

思考：如何实现一个优先级队列？

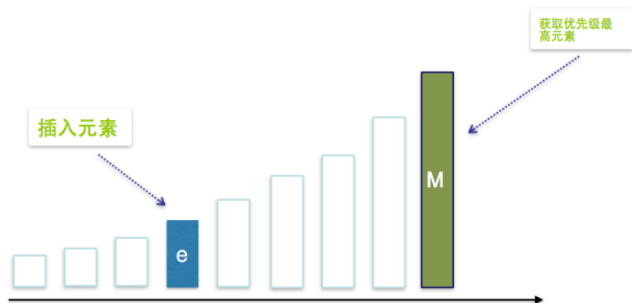
如何构造优先级队列

使用普通线性数组(无序)来表示优先级队列



- 执行插入操作时，直接将元素插入到数组末端，需要的成本为 $O(1)$ ，
- 获取优先级最高元素，我们需要遍历整个线性队列，匹配出优先级最高元素，需要的成本为 $O(n)$
- 删除优先级最高元素，我们需要两个步骤，第一找出优先级最高元素，第二步删除优先级最高元素，然后将后面的元素依次迁移，填补空缺，需要的成本为 $O(n)+O(n)=O(n)$

使用一个按顺序排列的有序向量实现优先级队列



- 获取优先级最高元素， $O(1)$
- 删除优先级最高元素， $O(1)$
- 插入一个元素，需要两个步骤，第一步我们需要找出要插的位置，这里我们可以使用二分查找，成本为 $O(\log n)$ ，第二步是插入元素之后，将其所有后继进行后移操作，成本为 $O(n)$ ，所有总成本为 $O(\log n) + O(n) = O(n)$

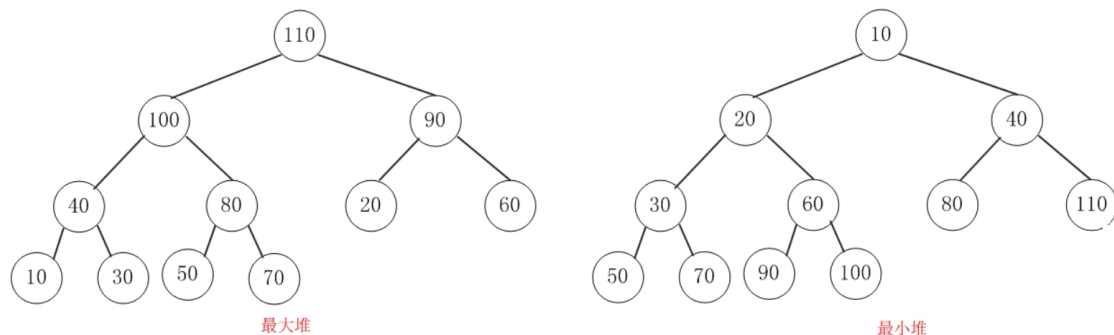
二叉堆

完全二叉树：除了最后一行，其他行都满的二叉树，而且最后一行所有叶子节点都从左向右开始排序。

二叉堆：完全二叉树的基础上，加以一定的条件约束的一种特殊的二叉树。根据约束条件的不同，二叉堆又可以分为两个类型：

大顶堆和小顶堆。

- 大顶堆（最大堆）：父结点的键值总是大于或等于任何一个子节点的键值；
- 小顶堆（最小堆）：父结点的键值总是小于或等于任何一个子节点的键值。



堆排序视频

链接：<https://pan.baidu.com/s/168rBwtzLsxQH9AJqZOtPPw>

提取码：e6oo

DelayQueue

DelayQueue 是一个支持延时获取元素的阻塞队列，内部采用优先队列 **PriorityQueue** 存储元素，同时元素必须实现 **Delayed** 接口；在创建元素时可以指定多久才可以从队列中获取当前元素，只有在延迟期满时才能从队列中提取元素。延迟队列的特点是：不是先进先出，而是会按照延迟时间的长短来排序，下一个即将执行的任务会排到队列的最前面。

它是无界队列，放入的元素必须实现 **Delayed** 接口，而 **Delayed** 接口又继承了 **Comparable** 接口，所以自然就拥有了比较和排序的能力，代码如下：

```

1 public interface Delayed extends Comparable<Delayed> {
2     //getDelay 方法返回的是“还剩下多长的延迟时间才会被执行”，
3     //如果返回 0 或者负数则代表任务已过期。
4     //元素会根据延迟时间的长短被放到队列的不同位置，越靠近队列头代表越早过期。
5     long getDelay(TimeUnit unit);
6 }

```



DelayQueue使用

```

1 DelayQueue<OrderInfo> queue = new DelayQueue<OrderInfo>();

```

DelayQueue的原理

数据结构

```

1 //用于保证队列操作的线程安全
2 private final transient ReentrantLock lock = new ReentrantLock();
3 // 优先级队列,存储元素, 用于保证延迟低的优先执行
4 private final PriorityQueue<E> q = new PriorityQueue<E>();
5 // 用于标记当前是否有线程在排队 (仅用于取元素时) leader 指向的是第一个从队列获取元素阻塞的线程
6 private Thread leader = null;
7 // 条件, 用于表示现在是否有可取的元素 当新元素到达, 或新线程可能需要成为leader时被通知
8 private final Condition available = lock.newCondition();
9
10 public DelayQueue() {}
11 public DelayQueue(Collection<? extends E> c) {
12     this.addAll(c);

```

入队put方法

```
1 public void put(E e) {
2     offer(e);
3 }
4 public boolean offer(E e) {
5     final ReentrantLock lock = this.lock;
6     lock.lock();
7     try {
8         // 入队
9         q.offer(e);
10        if (q.peek() == e) {
11            // 若入队的元素位于队列头部，说明当前元素延迟最小
12            // 将 leader 置空
13            leader = null;
14            // available条件队列转同步队列,准备唤醒阻塞在available上的线程
15            available.signal();
16        }
17        return true;
18    } finally {
19        lock.unlock(); // 解锁，真正唤醒阻塞的线程
20    }
21 }
```

出队take方法

```
1 public E take() throws InterruptedException {
2     final ReentrantLock lock = this.lock;
3     lock.lockInterruptibly();
4     try {
5         for (;;) {
6             E first = q.peek(); // 取出堆顶元素(最早过期的元素，但是不弹出对象)
7             if (first == null) // 如果堆顶元素为空，说明队列中还没有元素，直接阻塞等待
8                 available.await(); // 当前线程无限期等待，直到被唤醒，并且释放锁。
9             else {
10                long delay = first.getDelay(NANOSECONDS); // 堆顶元素的到期时间
11                if (delay <= 0) // 如果小于0说明已到期，直接调用poll()方法弹出堆顶元素
12                    return q.poll();
13
14                // 如果delay大于0，则下面要阻塞了
15                // 将first置为空方便gc
16            }
17        }
18    } finally {
19        lock.unlock();
20    }
21 }
```

```

16         first = null;
17         // 如果有线程争抢的Leader线程，则进行无限期等待。
18         if (leader != null)
19             available.await();
20         else {
21             // 如果leader为null，把当前线程赋值给它
22             Thread thisThread = Thread.currentThread();
23             leader = thisThread;
24             try {
25                 // 等待剩余等待时间
26                 available.awaitNanos(delay);
27             } finally {
28                 // 如果leader还是当前线程就把它置为空，让其它线程有机会获取元素
29                 if (leader == thisThread)
30                     leader = null;
31             }
32         }
33     }
34 }
35 } finally {
36     // 成功出队后，如果leader为空且堆顶还有元素，就唤醒下一个等待的线程
37     if (leader == null && q.peek() != null)
38         // available条件队列转同步队列,准备唤醒阻塞在available上的线程
39         available.signal();
40     // 解锁，真正唤醒阻塞的线程
41     lock.unlock();
42 }

```

1. 当获取元素时，先获取到锁对象。
2. 获取最早过期的元素，但是并不从队列中弹出元素。
3. 最早过期元素是否为空，如果为空则直接让当前线程无限期等待状态，并且让出当前锁对象。
4. 如果最早过期的元素不为空
5. 获取最早过期元素的剩余过期时间，如果已经过期则直接返回当前元素
6. 如果没有过期，也就是说剩余时间还存在，则先获取Leader对象，如果Leader已经有线程在处理，则当前线程进行无限期等待，如果Leader为空，则首先将Leader设置为当前线程，并且让当前线程等待剩余时间。
7. 最后将Leader线程设置为空
8. 如果Leader已经为空，并且队列有内容则唤醒一个等待的队列。

如何选择合适的阻塞队列

线程池对于阻塞队列的选择

线程池有很多种，不同种类的线程池会根据自己的特点，来选择适合自己的阻塞队列。

- FixedThreadPool (SingleThreadExecutor 同理) 选取的是 LinkedBlockingQueue
- CachedThreadPool 选取的是 SynchronousQueue
- ScheduledThreadPool (SingleThreadScheduledExecutor 同理) 选取的是延迟队列

选择策略

通常我们可以从以下 5 个角度考虑，来选择合适的阻塞队列：

功能

第 1 个需要考虑的就是功能层面，比如是否需要阻塞队列帮我们排序，如优先级排序、延迟执行等。如果有这个需要，我们就必须选择类似于 PriorityBlockingQueue 之类的有排序能力的阻塞队列。

容量

第 2 个需要考虑的是容量，或者说是否有存储的要求，还是只需要“直接传递”。在考虑这一点的时候，我们知道前面介绍的那几种阻塞队列，有的是容量固定的，如 ArrayBlockingQueue；有的默认是容量无限的，如 LinkedBlockingQueue；而有的里面没有任何容量，如 SynchronousQueue；而对于 DelayQueue 而言，它的容量固定就是 Integer.MAX_VALUE。所以不同阻塞队列的容量是千差万别的，**我们需要根据任务数量来推算出合适的容量**，从而去选取合适的 BlockingQueue。

能否扩容

第 3 个需要考虑的是能否扩容。因为有时我们并不能在初始的时候很好的准确估计队列的大小，因为业务可能有高峰期、低谷期。如果一开始就固定一个容量，可能无法应对所有的情况，也是不合适的，有可能需要动态扩容。如果我们需要动态扩容的话，那么就不能选择 ArrayBlockingQueue，因为它的容量在创建时就确定了，无法扩容。相反，PriorityBlockingQueue 即使在指定了初始容量之后，后续如果有需要，也可以自动扩容。所以**我们可以根据是否需要扩容来选取合适的队列**。

内存结构

第 4 个需要考虑的点就是内存结构。我们分析过 ArrayBlockingQueue 的源码，看到了它的内部结构是“数组”的形式。和它不同的是，LinkedBlockingQueue 的内部是用链表实现的，所以这里就需要我们考虑到，ArrayBlockingQueue 没有链表所需要的“节点”，空间利用率更高。所以如果我们对性能有要求可以从内存的结构角度去考虑这个问题。

性能

第 5 点就是从性能的角度去考虑。比如 LinkedBlockingQueue 由于拥有两把锁，它的操作粒度更细，在并发程度高的时候，相对于只有一把锁的 ArrayBlockingQueue 性能会更好。另外，SynchronousQueue 性能往往优于其他实现，因为它只需要“直接传递”，而不需要存储的过程。如果我们的场景需要直接传递的话，可以优先考虑 SynchronousQueue。