# jdk8中数据结构

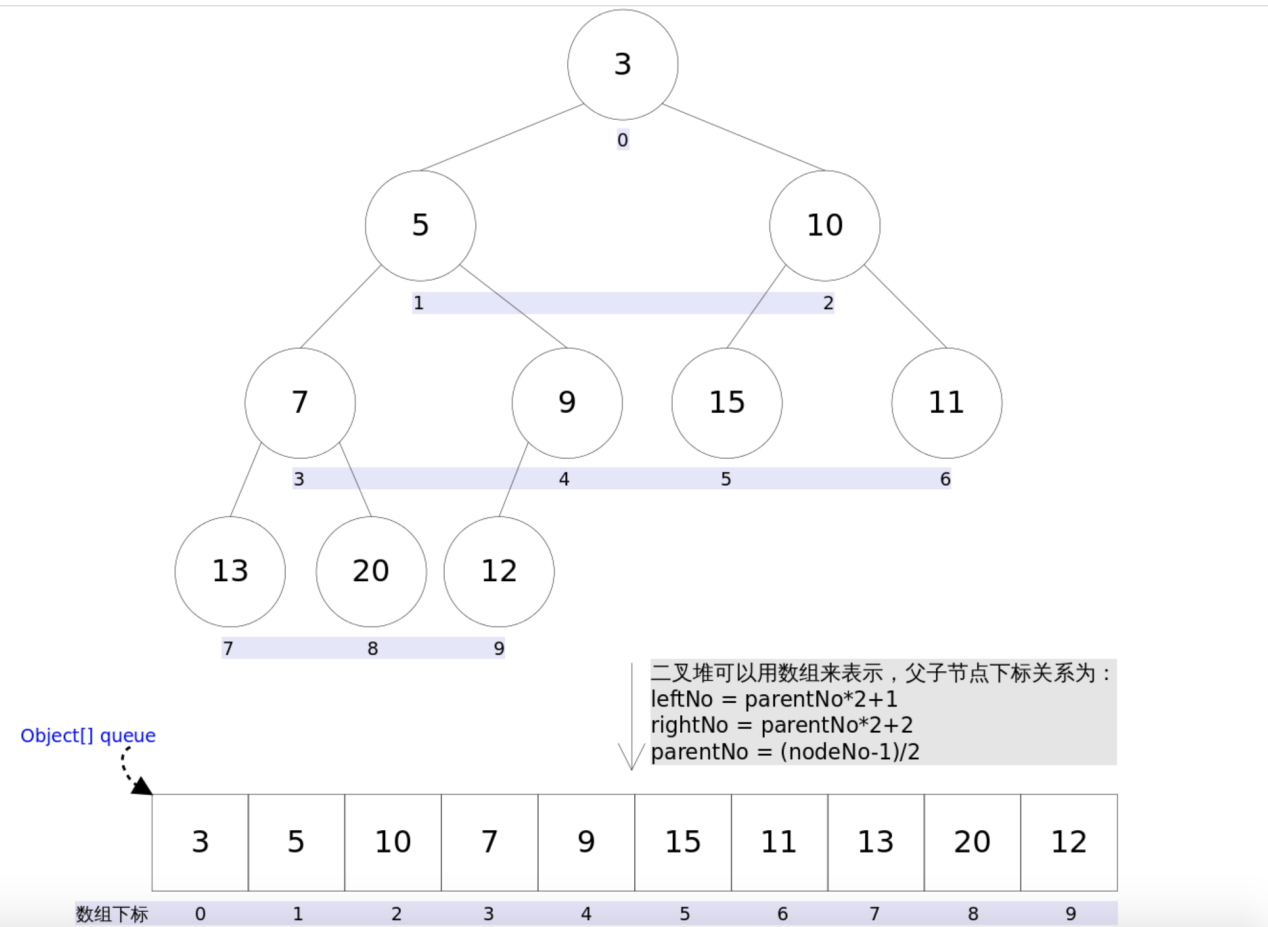
# 对列

## PriorityQueue(优先对列)：底层是数组

特性：有序，父结点大于小于左右孩子。

PriorityQueue的peek()和element操作是常数时间，add(), offer(), 无参数的remove()以及poll()方法的时间复杂度都是log(N)

通过完全二叉树实现的小顶堆(任意一个非叶子节点的权值，都不大于器左右子结点的权值)。底层是数组



### 1 字段queue

底层数组对象

|  |
| --- |
| */\*\*  \* Priority queue represented as a balanced binary heap: the two  \* children of queue[n] are queue[2\*n+1] and queue[2\*(n+1)]. The  \* priority queue is ordered by comparator, or by the elements'  \* natural ordering, if comparator is null: For each node n in the  \* heap and each descendant d of n, n <= d. The element with the  \* lowest value is in queue[0], assuming the queue is nonempty.  \*/*  *// 优先对列是平衡二叉堆，queue[n]的两个孩子节点分别是queue[2\*n+1],*  *// queue[2\*(n+1)]。优先对列通过比较器使其有序，或者按自然顺序。* transient Object[] queue; // non-private to simplify nested class access |

### 2 方法add和offer：入对列

|  |
| --- |
| public boolean add(E e) {  return offer(e); }  public boolean offer(E e) {  if (e == null)  throw new NullPointerException();  modCount++;  int i = size;  if (i >= queue.length)  // 扩容操作  grow(i + 1);  size = i + 1;  if (i == 0)  queue[0] = e;  else  // 进行调整  siftUp(i, e);  return true; }  private void siftUp(int k, E x) {  if (comparator != null)  // 通过比较器  siftUpUsingComparator(k, x);  else  // 按照自然顺序  siftUpComparable(k, x); }  private void siftUpComparable(int k, E x) {  Comparable<? super E> key = (Comparable<? super E>) x;  while (k > 0) {  // 无符号右移1位，相当于除以2，计算出父结点的索引  int parent = (k - 1) >>> 1;  // 父结点的元素  Object e = queue[parent];  // 如果果key（即目标元素x）刚好大于等于parent，就退出循环  if (key.compareTo((E) e) >= 0)  break;  // 走到这里说明，key(即目标元素x)，小于e(parent),那么就需要交换  queue[k] = e;  // 以父结点索引继续向上寻找  k = parent;  }  queue[k] = key; }  这个方法和上个方法一样的，只是比较器变成立自定义的比较器  private void siftUpUsingComparator(int k, E x) {  while (k > 0) {  int parent = (k - 1) >>> 1;  Object e = queue[parent];  if (comparator.compare(x, (E) e) >= 0)  break;  queue[k] = e;  k = parent;  }  queue[k] = x; }  如下图 |

### 3 poll 出队方法

|  |
| --- |
| public E poll() {  if (size == 0)  return null;  int s = --size;  modCount++;  // 取出对列头部元素  E result = (E) queue[0];  // 取出对列尾部元素  E x = (E) queue[s];  queue[s] = null;  if (s != 0)  // 调整对列  siftDown(0, x);  return result; }  private void siftDown(int k, E x) {  if (comparator != null)  siftDownUsingComparator(k, x);  else  siftDownComparable(k, x); }  private void siftDownComparable(int k, E x) {  Comparable<? super E> key = (Comparable<? super E>)x;  // 求出2分之一索引  int half = size >>> 1; // loop while a non-leaf  while (k < half) {  // 这是求出左孩子的索引，假设左孩子是最小的那个  int child = (k << 1) + 1; // assume left child is least  // 左孩子元素  Object c = queue[child];  // 右孩子  int right = child + 1;  // 若是左孩子大于右孩子  // 那就将右孩子的值赋给c  if (right < size &&  ((Comparable<? super E>) c).compareTo((E) queue[right]) > 0)  c = queue[child = right];  if (key.compareTo((E) c) <= 0)  break;  // 左孩子(实际是左右孩子中小的那个值)移到父结点  queue[k] = c;  // 在以小的那个索引开始往下递归，逐层比对左右孩子，将小的那个提升到父结点位置  k = child;  }  queue[k] = key; } |

核心方法就是siftUp和siftDown。

## DelayQueue(延迟对列)：底层是PriorityQueue

### 字段

上述比较特殊的是 leader 字段，我们之前已经说过，DelayQueue 每次只会出队一个过期的元素，如果队首元素没有过期，就会阻塞出队线程，让线程在 available 这个条件队列上无限等待。

为了提升性能，DelayQueue 并不会让所有出队线程都无限等待，而是用 leader 保存了第一个尝试出队的线程，该线程的等待时间是队首元素的剩余有效期。这样，一旦 leader 线程被唤醒（此时队首元素也失效了），就可以出队成功，然后唤醒一个其它在 available 条件队列上等待的线程。之后，会重复上一步，新唤醒的线程可能取代成为新的 leader 线程。这样，就避免了无效的等待，提升了性能。这其实是一种名为 Leader-Follower pattern 的多线程设计模式

|  |
| --- |
| // 同步锁  private final transient ReentrantLock lock = new ReentrantLock(); // 优先对列，元素是存放在优先对列中的  private final PriorityQueue<E> q = new PriorityQueue<E>();  */\*\*  \* Thread designated to wait for the element at the head of  \* the queue. This variant of the Leader-Follower pattern  \* (http://www.cs.wustl.edu/~schmidt/POSA/POSA2/) serves to  \* minimize unnecessary timed waiting. When a thread becomes  \* the leader, it waits only for the next delay to elapse, but  \* other threads await indefinitely. The leader thread must  \* signal some other thread before returning from take() or  \* poll(...), unless some other thread becomes leader in the  \* interim. Whenever the head of the queue is replaced with  \* an element with an earlier expiration time, the leader  \* field is invalidated by being reset to null, and some  \* waiting thread, but not necessarily the current leader, is  \* signalled. So waiting threads must be prepared to acquire  \* and lose leadership while waiting.  \*/*  *// 被阻塞的当前线程，若是null，则没有阻塞，若不是null，则没被阻塞* private Thread leader = null;  */\*\*  \* Condition signalled when a newer element becomes available  \* at the head of the queue or a new thread may need to  \* become leader.  \*/* private final Condition available = lock.newCondition(); |

### 入队put和offer

|  |
| --- |
| public void put(E e) {  offer(e); }  public boolean offer(E e) {  final ReentrantLock lock = this.lock;  lock.lock();  try {  q.offer(e);  // 如果入队元素在对首，则唤醒一个出队线程  if (q.peek() == e) {  leader = null;  // 唤醒出队线程  available.signal();  }  return true;  } finally {  lock.unlock();  } } |

### 3.出队poll

|  |
| --- |
| // 没有阻塞的出队  public E poll() {  final ReentrantLock lock = this.lock;  lock.lock();  try {  E first = q.peek();  //   if (first == null || first.getDelay(*NANOSECONDS*) > 0)  return null;  else  return q.poll();  } finally {  lock.unlock();  } } |

### 4.take

|  |
| --- |
| public E take() throws InterruptedException {  final ReentrantLock lock = this.lock;  lock.lockInterruptibly();  try {  // 死循环知道对首元素出队  for (;;) {  E first = q.peek();  // 对搜元素是空，说明对列是空的，需要等待入队  if (first == null)  available.await();  else {  // 不是空，看看任务的时间有没有到期  long delay = first.getDelay(*NANOSECONDS*);  if (delay <= 0)  // 小于0，到期了就可以出队  return q.poll();  first = null; // don't retain ref while waiting  // 程序走到这里，说明，对首的过期时间还没到  if (leader != null)  // leader 不是空，说明这个线程就是要出队的线程  available.await();  else {  Thread thisThread = Thread.*currentThread*();  leader = thisThread;  try {  // 阻塞leader线程直到delay时间后  available.awaitNanos(delay);  } finally {  if (leader == thisThread)  leader = null;  }  }  }  }  } finally {  if (leader == null && q.peek() != null)  // 当前线程成功出队后，唤醒其他出队的线程，接着出队  available.signal();  lock.unlock();  } } |

### 5.peek

|  |
| --- |
| public E peek() {  final ReentrantLock lock = this.lock;  lock.lock();  try {  // 查看对首元素  return q.peek();  } finally {  lock.unlock();  } } |

## ArrayBlockingQueue(有界阻塞对列)：底层是数组

ArrayBlockingQueue是数组实现的线程安全的有界的阻塞对列。FIFO(先进先出)，底层是通过数组实现的

1. ArrayBlockingQueue继承于AbstractQueue，并且它实现了BlockingQueue接口。  
    2. ArrayBlockingQueue内部是通过Object[]数组保存数据的，也就是说ArrayBlockingQueue本质上是通过数组实现的。ArrayBlockingQueue的大小，即数组的容量是创建ArrayBlockingQueue时指定的。  
    3. ArrayBlockingQueue与ReentrantLock是组合关系，ArrayBlockingQueue中包含一个ReentrantLock对象(lock)。ReentrantLock是可重入的互斥锁，ArrayBlockingQueue就是根据该互斥锁实现“多线程对竞争资源的互斥访问”。而且，ReentrantLock分为公平锁和非公平锁，关于具体使用公平锁还是非公平锁，在创建ArrayBlockingQueue时可以指定；而且，ArrayBlockingQueue默认会使用非公平锁。  
    4. ArrayBlockingQueue与Condition是组合关系，ArrayBlockingQueue中包含两个Condition对象(notEmpty和notFull)。而且，Condition又依赖于ArrayBlockingQueue而存在，通过Condition可以实现对ArrayBlockingQueue的更精确的访问 -- (01)若某线程(线程A)要取数据时，数组正好为空，则该线程会执行notEmpty.await()进行等待；当其它某个线程(线程B)向数组中插入了数据之后，会调用notEmpty.signal()唤醒“notEmpty上的等待线程”。此时，线程A会被唤醒从而得以继续运行。(02)若某线程(线程H)要插入数据时，数组已满，则该线程会它执行notFull.await()进行等待；当其它某个线程(线程I)取出数据之后，会调用notFull.signal()唤醒“notFull上的等待线程”。此时，线程H就会被唤醒从而得以继续运行。

### 1.字段

|  |
| --- |
| */\*\* The queued items \*/*  *存放元素的数组* final Object[] items;  */\*\* items index for next take, poll, peek or remove \*/*  *// 下一个出队列的索引* int takeIndex;  */\*\* items index for next put, offer, or add \*/*  *// 下一个如对列的索引* int putIndex;  */\*\* Number of elements in the queue \*/*  *// 元素个数* int count;  /\*  \* Concurrency control uses the classic two-condition algorithm  \* found in any textbook.  \*/  */\*\* Main lock guarding all access \*/*  *// 同步的lock锁* final ReentrantLock lock;  */\*\* Condition for waiting takes \*/*  *// 阻塞用的条件对象：当空的时候，用于阻塞出队的线程* private final Condition notEmpty;  */\*\* Condition for waiting puts \*/*  *// 阻塞用的添加对象：当满的时候，用于阻塞入队的线程* private final Condition notFull; |

### 方法add和offer

|  |
| --- |
| public boolean add(E e) {  // 调用父类的add  return super.add(e); }  // 父类add  public boolean add(E e) {  // 实际调用offer  if (offer(e))  return true;  else  // 对列满了会抛出异常  throw new IllegalStateException("Queue full"); }  public boolean offer(E e) {  *checkNotNull*(e);  final ReentrantLock lock = this.lock;  lock.lock();  try {  if (count == items.length)  // 满了，就不添加了  return false;  else {  // 入队  enqueue(e);  return true;  }  } finally {  lock.unlock();  } }  private void enqueue(E x) {  // assert lock.getHoldCount() == 1;  // assert items[putIndex] == null;  final Object[] items = this.items;  items[putIndex] = x;  if (++putIndex == items.length)  putIndex = 0;  count++;  // 唤醒消费线程进行出队操作  notEmpty.signal(); } |

### put 方法

|  |
| --- |
| public void put(E e) throws InterruptedException {  *checkNotNull*(e);  final ReentrantLock lock = this.lock;  lock.lockInterruptibly();  try {  while (count == items.length)  // 满了，对入队操作进行阻塞  notFull.await();  // 入队  enqueue(e);  } finally {  lock.unlock();  } } |

### poll方法

|  |
| --- |
| public E poll() {  final ReentrantLock lock = this.lock;  lock.lock();  try {  return (count == 0) ? null : dequeue();  } finally {  lock.unlock();  } }  private E dequeue() {  // assert lock.getHoldCount() == 1;  // assert items[takeIndex] != null;  final Object[] items = this.items;  @SuppressWarnings("unchecked")  // 将最先入队的进行出队，其实就是从索引0开始出队，因为最先入队的索引是从  // 0递增上去的  E x = (E) items[takeIndex];  items[takeIndex] = null;  if (++takeIndex == items.length)  // 若是出对完了，将出队索引归零  takeIndex = 0;  count--;  if (itrs != null)  // 看不懂 干啥  itrs.elementDequeued();  notFull.signal();  return x; } |

### take方法

|  |
| --- |
| // 响应中断  public E take() throws InterruptedException {  final ReentrantLock lock = this.lock;  lock.lockInterruptibly();  try {  while (count == 0)  // 没有了，就阻塞出队的操作，等待添加  notEmpty.await();  return dequeue();  } finally {  lock.unlock();  } } |

SynchronousQueue

## LinkedBlockingQueue(无界阻塞对列):底层是链表

LinkedBlockingQueue是一个基于已链接节点的，范围任意的blocking queue

此队列按FIFO（先进先出）排序元素

新元素插入到队列的尾部，并且队列获取操作会获得位于队列头部的元素

链接队列的吞吐量通常要高于基于数组的对列（ArrayBlockingQueue）,但是在大多数并发应用程序中，其可预知的性能要低

可选的容量范围构造方法参数作为防止队列过度扩展的一种方法，如果未指定容量，则等于Integer.MAX\_VALUE，除非插入节点会使队列超出容量，否则每次插入后会动态地创建链接节点

### 1 字段

|  |
| --- |
| */\*\*  \* Linked list node class  \*/*  *// 节点类：单向链表节点* static class Node<E> {  E item;   */\*\*  \* One of:  \* - the real successor Node  \* - this Node, meaning the successor is head.next  \* - null, meaning there is no successor (this is the last node)  \*/* Node<E> next;   Node(E x) { item = x; } }  */\*\* The capacity bound, or Integer.MAX\_VALUE if none \*/* private final int capacity;  */\*\* Current number of elements \*/*  *// 记录节点长度的原子类* private final AtomicInteger count = new AtomicInteger();  */\*\*  \* Head of linked list.  \* Invariant: head.item == null  \*/*  *// 头结点* transient Node<E> head;  */\*\*  \* Tail of linked list.  \* Invariant: last.next == null  \*/*  *// 为节点* private transient Node<E> last;  */\*\* Lock held by take, poll, etc \*/*  *// 出队列的锁* private final ReentrantLock takeLock = new ReentrantLock();  */\*\* Wait queue for waiting takes \*/*  *// 等待出队的条件对象，用于阻塞添加的操作* private final Condition notEmpty = takeLock.newCondition();  */\*\* Lock held by put, offer, etc \*/*  *// 如对列的锁* private final ReentrantLock putLock = new ReentrantLock();  */\*\* Wait queue for waiting puts \*/*  *// 等待对列入队条件对象，用于阻塞出队的操作* private final Condition notFull = putLock.newCondition(); |

### 2 方法offer

|  |
| --- |
| public boolean offer(E e) {  if (e == null) throw new NullPointerException();  final AtomicInteger count = this.count;  // capacity默认是Integer最大值  if (count.get() == capacity)  return false;  int c = -1;  // 构造新的结点  Node<E> node = new Node<E>(e);  final ReentrantLock putLock = this.putLock;  putLock.lock();  try {  if (count.get() < capacity) {  enqueue(node);  c = count.getAndIncrement();  if (c + 1 < capacity)  // 唤醒put的入队操作  notFull.signal();  }  } finally {  putLock.unlock();  }  if (c == 0)  // 唤醒一个出队的操作，  signalNotEmpty();  return c >= 0; }  // 入队，直接将新结点赋给last的next，然后再将last指向构造的新节点  private void enqueue(Node<E> node) {  // assert putLock.isHeldByCurrentThread();  // assert last.next == null;  last = last.next = node; }  private void signalNotEmpty() {  final ReentrantLock takeLock = this.takeLock;  takeLock.lock();  try {  notEmpty.signal();  } finally {  takeLock.unlock();  } } |

### 3 方法put

|  |
| --- |
| public void put(E e) throws InterruptedException {  if (e == null) throw new NullPointerException();  // Note: convention in all put/take/etc is to preset local var  // holding count negative to indicate failure unless set.  int c = -1;  Node<E> node = new Node<E>(e);  final ReentrantLock putLock = this.putLock;  final AtomicInteger count = this.count;  putLock.lockInterruptibly();  try {  /\*  \* Note that count is used in wait guard even though it is  \* not protected by lock. This works because count can  \* only decrease at this point (all other puts are shut  \* out by lock), and we (or some other waiting put) are  \* signalled if it ever changes from capacity. Similarly  \* for all other uses of count in other wait guards.  \*/  while (count.get() == capacity) {  // 满了，阻塞，等待出队消费  notFull.await();  }  enqueue(node);  c = count.getAndIncrement();  if (c + 1 < capacity)  // 没满，唤醒被阻塞的put操作  notFull.signal();  } finally {  putLock.unlock();  }  if (c == 0)  // 唤醒出队操作  signalNotEmpty(); }  private void signalNotEmpty() {  final ReentrantLock takeLock = this.takeLock;  takeLock.lock();  try {  // 唤醒出队的操作  notEmpty.signal();  } finally {  takeLock.unlock();  } } |

### 4 poll

|  |
| --- |
| public E poll() {  final AtomicInteger count = this.count;  if (count.get() == 0)  return null;  E x = null;  int c = -1;  final ReentrantLock takeLock = this.takeLock;  takeLock.lock();  try {  if (count.get() > 0) {  x = dequeue();  c = count.getAndDecrement();  if (c > 1)  // 唤醒出队操作  notEmpty.signal();  }  } finally {  takeLock.unlock();  }  if (c == capacity)  // 换新入队的操作  signalNotFull();  return x; }  private void signalNotFull() {  final ReentrantLock putLock = this.putLock;  putLock.lock();  try {  // 唤醒入队的操作  notFull.signal();  } finally {  putLock.unlock();  } } |

### 5 方法take

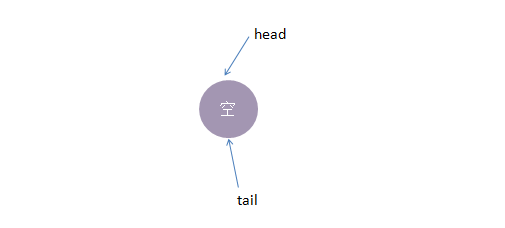
|  |
| --- |
| public E take() throws InterruptedException {  E x;  int c = -1;  final AtomicInteger count = this.count;  final ReentrantLock takeLock = this.takeLock;  takeLock.lockInterruptibly();  try {  while (count.get() == 0) {  // 阻塞出队操作，等待入队  notEmpty.await();  }  x = dequeue();  c = count.getAndDecrement();  if (c > 1)  // 唤醒入队操作  notEmpty.signal();  } finally {  takeLock.unlock();  }  if (c == capacity)  //唤醒入队操作  signalNotFull();  return x; }  private E dequeue() {  // assert takeLock.isHeldByCurrentThread();  // assert head.item == null;  // 出队，将head指向head的next  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; } |

## SynchronousQueue对列

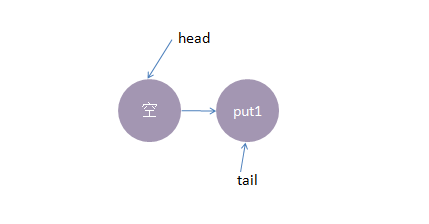
### 模型原理简介

SynchronousQueue 也是一个队列来的，但它的特别之处在于它内部没有容器，一个生产线程，当它生产产品（即put的时候），如果当前没有人想要消费产品(即当前没有线程执行take)，此生产线程必须阻塞，等待一个消费线程调用take操作，take操作将会唤醒该生产线程，同时消费线程会获取生产线程的产品（即数据传递），这样的一个过程称为一次配对过程(当然也可以先take后put,原理是一样的)

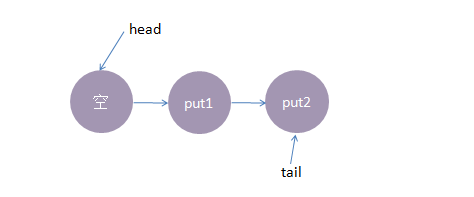
公平模式下，底层实现使用的是TransferQueue这个内部队列，它有一个head和tail指针，用于指向当前正在等待匹配的线程节点。  
初始化时，TransferQueue的状态如下：



1、线程put1执行 put(1)操作，由于当前没有配对的消费线程，所以put1线程入队列，自旋一小会后睡眠等待，这时队列状态如下：



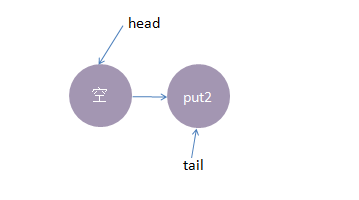
2、接着，线程put2执行了put(2)操作，跟前面一样，put2线程入队列，自旋一小会后睡眠等待，这时队列状态如下



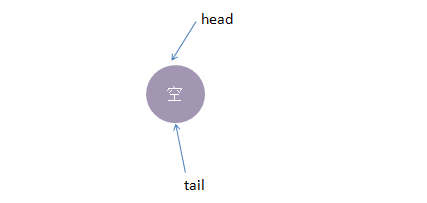
3、这时候，来了一个线程take1，执行了 take操作，由于tail指向put2线程，put2线程跟take1线程配对了(一put一take)，这时take1线程不需要入队，但是请注意了，这时候，要唤醒的线程并不是put2，而是put1。为何？ 大家应该知道我们现在讲的是公平策略，所谓公平就是谁先入队了，谁就优先被唤醒，我们的例子明显是put1应该优先被唤醒。至于读者可能会有一个疑问，明明是take1线程跟put2线程匹配上了，结果是put1线程被唤醒消费，怎么确保take1线程一定可以和次首节点(head.next)也是匹配的呢？其实大家可以拿个纸画一画，就会发现真的就是这样的。

公平策略总结下来就是：队尾匹配队头出队。

执行后put1线程被唤醒，take1线程的 take()方法返回了1(put1线程的数据)，这样就实现了线程间的一对一通信，这时候内部状态如下：

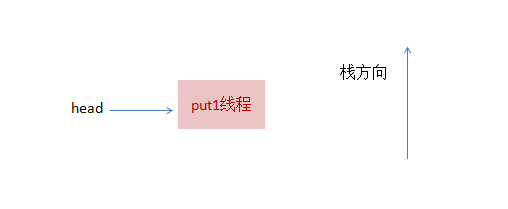


4、最后，再来一个线程take2，执行take操作，这时候只有put2线程在等候，而且两个线程匹配上了，线程put2被唤醒，  
take2线程take操作返回了2(线程put2的数据)，这时候队列又回到了起点，如下所示：

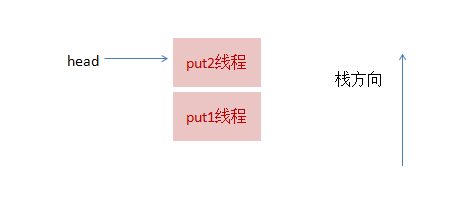


非公平模式下的模型：

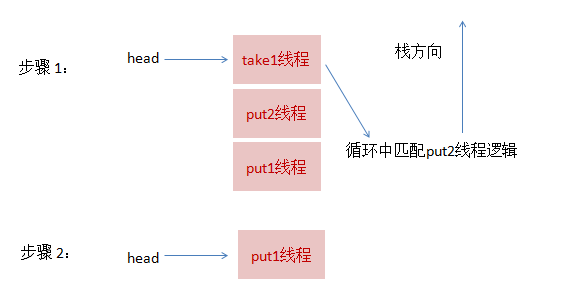
1、线程put1执行 put(1)操作，由于当前没有配对的消费线程，所以put1线程入栈，自旋一小会后睡眠等待，这时栈状态如下：



2、接着，线程put2再次执行了put(2)操作，跟前面一样，put2线程入栈，自旋一小会后睡眠等待，这时栈状态如下：



3、这时候，来了一个线程take1，执行了take操作，这时候发现栈顶为put2线程，匹配成功，但是实现会先把take1线程入栈，然后take1线程循环执行匹配put2线程逻辑，一旦发现没有并发冲突，就会把栈顶指针直接指向 put1线程



4、最后，再来一个线程take2，执行take操作，这跟步骤3的逻辑基本是一致的，take2线程入栈，然后在循环中匹配put1线程，最终全部匹配完毕，栈变为空，恢复初始状态，如下图所示

