问题

（1）LinkedBlockingQueue的实现方式?

（2）LinkedBlockingQueue是有界的还是无界的队列?

（3）LinkedBlockingQueue相比ArrayBlockingQueue有什么改进?

简介

LinkedBlockingQueue是java并发包下一个以单链表实现的阻塞队列，它是线程安全的，至于它是不是有界的，请看下面的分析。

源码分析

主要属性

```
    // 容量
    private final int capacity;

    // 元素数量
    private final AtomicInteger count = new AtomicInteger();

    // 链表头
    transient Node<E> head;

    // 链表尾
    private transient Node<E> last;

    // take锁
    private final ReentrantLock takeLock = new ReentrantLock();

    // notEmpty条件
    // 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
    private final Condition notEmpty = takeLock.newCondition();

    // 放锁
    private final ReentrantLock putLock = new ReentrantLock();

    // notFull条件
    // 当队列满了时，put锁会会阻塞在notFull上，等待其它线程唤醒
    private final Condition notFull = putLock.newCondition();
```

（1）capacity，有容量，可以理解为LinkedBlockingQueue是有界队列

（2）head, last，链表头、链表尾指针

（3）takeLock，notEmpty，take锁及其对应的条件

（4）putLock, notFull，put锁及其对应的条件

（5）入队、出队使用两个不同的锁控制，锁分离，提高效率

内部类

```
    static class Node<E> {
        E item;

        Node<E> next;
```

```
        Node(E x) { item = x; }
    }
```

典型的单链表结构。

主要构造方法

```
    public LinkedBlockingQueue() {
        // 如果没传容量，就使用最大int值初始化其容量
        this(Integer.MAX_VALUE);
    }

    public LinkedBlockingQueue(int capacity) {
        if (capacity <= 0) throw new IllegalArgumentException();
        this.capacity = capacity;
        // 初始化head和last指针为空值节点
        last = head = new Node<E>(null);
    }
```

入队

入队同样有四个方法，我们这里只分析最重要的一个，put(E e)方法：

```
    public void put(E e) throws InterruptedException {
        // 不允许null元素
        if (e == null) throw new NullPointerException();
        int c = -1;
        // 新建一个节点
        Node<E> node = new Node<E>(e);
        final ReentrantLock putLock = this.putLock;
        final AtomicInteger count = this.count;
        // 使用put锁加锁
        putLock.lockInterruptibly();
        try {
            // 如果队列满了，就阻塞在notFull条件上
            // 等待被其它线程唤醒
            while (count.get() == capacity) {
                notFull.await();
            }
            // 队列不满了，就入队
            enqueue(node);
            // 队列长度加1
            c = count.getAndIncrement();
            // 如果现队列长度如果小于容量
            // 就再唤醒一个阻塞在notFull条件上的线程
            // 这里为啥要唤醒一下呢？
            // 因为可能有很多线程阻塞在notFull这个条件上的
            // 而取元素时只有取之前队列是满的才会唤醒notFull
            // 为什么队列满的才唤醒notFull呢？
            // 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，未满就唤醒其它notFull上的线程
            // 说白了，这也是锁分离带来的代价
            if (c + 1 < capacity)
                notFull.signal();
        } finally {
            // 释放锁
            putLock.unlock();
        }
        // 如果原队列长度为0，现在加了一个元素后立即唤醒notEmpty条件
```

```
        if (c == 0)
            signalNotEmpty();
    }

    private void enqueue(Node<E> node) {
        // 直接加到last后面
        last = last.next = node;
    }

    private void signalNotEmpty() {
        final ReentrantLock takeLock = this.takeLock;
        // 加take锁
        takeLock.lock();
        try {
            // 唤醒notEmpty条件
            notEmpty.signal();
        } finally {
            // 解锁
            takeLock.unlock();
        }
    }

```

（1）使用putLock加锁；

（2）如果队列满了就阻塞在notFull条件上；

（3）否则就入队；

（4）如果入队后元素数量小于容量，唤醒其它阻塞在notFull条件上的线程；

（5）释放锁；

（6）如果放元素之前队列长度为0，就唤醒notEmpty条件；

出队

出队同样也有四个方法，我们这里只分析最重要的那一个，take()方法：

```
    public E take() throws InterruptedException {
        E x;
        int c = -1;
        final AtomicInteger count = this.count;
        final ReentrantLock takeLock = this.takeLock;
        // 使用takeLock加锁
        takeLock.lockInterruptibly();
        try {
            // 如果队列无元素，则阻塞在notEmpty条件上
            while (count.get() == 0) {
                notEmpty.await();
            }
            // 否则，出队
            x = dequeue();
            // 获取出队前队列的长度
            c = count.getAndDecrement();
            // 如果取之前队列长度大于1，则唤醒notEmpty
            if (c > 1)
                notEmpty.signal();
        } finally {
            // 释放锁
            takeLock.unlock();
        }
        // 如果取之前队列长度等于容量
        // 则唤醒notFull
```

```
        if (c == capacity)
            signalNotFull();
        return x;
    }

    private E dequeue() {
        // head节点本身是不存储任何元素的
        // 这里把head删除，并把head下一个节点作为新的值
        // 并把其值置空，返回原来的值
        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;
    }

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

```

（1）使用takeLock加锁；

（2）如果队列空了就阻塞在notEmpty条件上；

（3）否则就出队；

（4）如果出队前元素数量大于1，唤醒其它阻塞在notEmpty条件上的线程；

（5）释放锁；

（6）如果取元素之前队列长度等于容量，就唤醒notFull条件；

总结

（1）LinkedBlockingQueue采用单链表的形式实现；

（2）LinkedBlockingQueue采用两把锁的锁分离技术实现入队出队互不阻塞；

（3）LinkedBlockingQueue是有界队列，不传入容量时默认为最大int值；

彩蛋

（1）LinkedBlockingQueue与ArrayBlockingQueue对比?

a）后者入队出队采用一把锁，导致入队出队相互阻塞，效率低下；

b）前才入队出队采用两把锁，入队出队互不干扰，效率较高；

c）二者都是有界队列，如果长度相等且出队速度跟不上入队速度，都会导致大量线程阻塞；

d）前者如果初始化不传入初始容量，则使用最大int值，如果出队速度跟不上入队速度，会导致队列特别长，占用大量内存；