问题

（1）PriorityBlockingQueue的实现方式?

（2）PriorityBlockingQueue是否需要扩容?

（3）PriorityBlockingQueue是怎么控制并发安全的?

简介

PriorityBlockingQueue是java并发包下的优先级阻塞队列，它是线程安全的，如果让你来实现你会怎么实现它呢?

还记得我们前面介绍过的PriorityQueue吗？点击链接直达【死磕 java集合之PriorityQueue源码分析】

还记得优先级队列一般使用什么来实现吗？点击链接直达【拜托，面试别再问我堆（排序）了！】

源码分析

主要属性

```
    // 默认容量为11
    private static final int DEFAULT_INITIAL_CAPACITY = 11;
    // 最大数组大小
    private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
    // 存储元素的地方
    private transient Object[] queue;
    // 元素个数
    private transient int size;
    // 比较器
    private transient Comparator<? super E> comparator;
    // 重入锁
    private final ReentrantLock lock;
    // 非空条件
    private final Condition notEmpty;
    // 扩容的时候使用的控制变量，CAS更新这个值，谁更新成功了谁扩容，其它线程让出CPU
    private transient volatile int allocationSpinLock;
    // 不阻塞的优先级队列，非存储元素的地方，仅用于序列化/反序列化时
    private PriorityQueue<E> q;

```

（1）依然是使用一个数组来使用元素;

（2）使用一个锁加一个notEmpty条件来保证并发安全;

（3）使用一个变量的CAS操作来控制扩容;

为啥没有notFull条件呢?

主要构造方法

```
    // 默认容量为11
    public PriorityBlockingQueue() {
        this(DEFAULT_INITIAL_CAPACITY, null);
    }
    // 传入初始容量
    public PriorityBlockingQueue(int initialCapacity) {
        this(initialCapacity, null);
```

```
    }
    // 传入初始容量和比较器
    // 初始化各变量
    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];
    }

```

入队

每个阻塞队列都有四个方法，我们这里只分析一个offer(E e)方法：

```

    public boolean offer(E e) {
        // 元素不能为空
        if (e == null)
            throw new NullPointerException();
        final ReentrantLock lock = this.lock;
        // 加锁
        lock.lock();
        int n, cap;
        Object[] array;
        // 判断是否需要扩容，即元素个数达到了数组容量
        while ((n = size) >= (cap = (array = queue).length))
            tryGrow(array, cap);
```

```
        try {
            Comparator<? super E> cmp = comparator;
            // 根据是否有比较器选择不同的方法
            if (cmp == null)
                siftUpComparable(n, e, array);
            else
                siftUpUsingComparator(n, e, array, cmp);
            // 插入元素完毕，元素个数加1
            size = n + 1;
            // 唤醒notEmpty条件
            notEmpty.signal();
        } finally {
            // 解锁
            lock.unlock();
        }
        return true;
    }

    private static <T> void siftUpComparable(int k, T x, Object[] array) {
        Comparable<? super T> key = (Comparable<? super T>) x;
        while (k > 0) {
            // 取父节点
            int parent = (k - 1) >>> 1;
            // 父节点的元素值
            Object e = arra截图(Alt + A)
            // 如果key大于父节点，堆化结束
            if (key.compareTo((T) e) >= 0)
                break;
            // 否则，交换二者的位置，继续下一轮比较
            array[k] = e;
            k = parent;
        }
        // 找到了应该放的位置，放入元素
```

```
        array[k] = key;
    }

```

入队的整个操作跟PriorityQueue几乎一致：

（1）加锁；

（2）判断是否需要扩容；

（3）添加元素并做自下而上的堆化；

（4）元素个数加1并唤醒notEmpty条件，唤醒取元素的线程；

（5）解锁；

扩容

```
    private void tryGrow(Object[] array, int oldCap) {
        // 先释放锁，因为是从offer()方法的锁内部过来的
        // 这里先释放锁，使用allocationSpinLock变量控制扩容的过程
        // 防止阻塞的线程过多
        lock.unlock(); // must release and then re-acquire main lock
        Object[] newArray = null;
        // CAS更新allocationSpinLock变量为1的线程获得扩容资格
        if (allocationSpinLock == 0 &&
            UNSAFE.compareAndSwapInt(this, allocationSpinLockOffset,
                                     0, 1)) {
            try {
                // 旧容量小于64则翻倍，旧容量大于64则增加一半
                int newCap = oldCap + ((oldCap < 64) ?
                                       (oldCap + 2) : // grow faster if small
                                       (oldCap >> 1));
                // 判断新容量是否溢出
                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;
                }
                // 创建新数组
                if (newCap > oldCap && queue == array)
                    newArray = new Object[newCap];
            } finally {
                // 相当于解锁
                allocationSpinLock = 0;
            }
        }
        // 只有进入了上面条件的才会满足这个条件
        // 意思是让其它线程让出CPU
        if (newArray == null) // back off if another thread is allocating
            Thread.yield();
        // 再次加锁
        lock.lock();
        // 判断新数组创建成功并且旧数组没有被替换过
        if (newArray != null && queue == array) {
            // 队列赋值为新数组
            queue = newArray;
            // 并拷贝旧数组元素到新数组中
            System.arraycopy(array, 0, newArray, 0, oldCap);
        }
    }

```

（1）解锁，解除offer()方法中加的锁；

（2）使用allocationSpinLock变量的CAS操作来控制扩容的过程；

（3）旧容量小于64则翻倍，旧容量大于64则增加一半；

（4）创建新数组；

（5）修改allocationSpinLock为0，相当于解锁；

（6）其它线程在扩容的过程中要让出CPU；

（7）再次加锁；

（8）新数组创建成功，把旧数组元素拷贝过来，并返回到offer()方法中继续添加元素操作；

出队

阻塞队列的出队方法也有四个，我们这里只分析一个take()方法：

```
    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        // 加锁
        lock.lockInterruptibly();
        E result;
        try {
            // 队列没有元素，就阻塞在notEmpty条件上
            // 出队成功，就跳出这个循环
            while ( (result = dequeue()) == null)
                notEmpty.await();
        } finally {
            // 解锁
            lock.unlock();
        }
        // 返回出队的元素
        return result;
    }

    private E dequeue() {
        // 元素个数减1
        int n = size - 1;
        if (n < 0)
            // 数组元素不足，返回null
            return null;
        else {
            Object[] array = queue;
            // 弹出堆顶元素
            E result = (E) array[0];
            // 把堆尾元素拿到堆顶
            E x = (E) array[n];
            array[n] = null;
            Comparator<? super E> cmp = comparator;
            // 并做自上而下的堆化
            if (cmp == null)
                siftDownComparable(0, x, array, n);
            else
                siftDownUsingComparator(0, x, array, n, cmp);
            // 修改size
            size = n;
            // 返回出队的元素
            return result;
        }
    }

```

```
45.     private static <T> void siftDownComparable(int k, T x, Object[] array,
46.                                                int n) {
47.         if (n > 0) {
48.             Comparable<? super T> key = (Comparable<? super T>)x;
49.             int half = n >>> 1;           // loop while a non-leaf
50.             // 只需要遍历到叶子节点就够了
51.             while (k < half) {
52.                 // 左子节点
53.                 int child = (k << 1) + 1; // assume left child is least
54.                 // 左子节点的值
55.                 Object c = array[child];
56.                 // 右子节点
57.                 int right = child + 1;
58.                 // 取左右子节点中最小的值
59.                 if (right < n &&
60.                     ((Comparable<? super T>) c).compareTo((T) array[right]) > 0)
61.                     c = array[child = right];
62.                 // key如果比左右子节点都小，则堆化结束
63.                 if (key.compareTo((T) c) <= 0)
64.                     break;
65.                 // 否则，交换key与左右子节点中最小的节点的位置
66.                 array[k] = c;
67.                 k = child;
68.             }
69.             // 找到了放元素的位置，放置元素
70.             array[k] = key;
71.         }
72.     }
73.
```

出队的过程与PriorityQueue基本类似：

（1）加锁；

（2）判断是否出队成功，未成功就阻塞在notEmpty条件上；

（3）出队时弹出堆顶元素，并把堆尾元素拿到堆顶；

（4）再做自上而下的堆化；

（5）解锁；

总结

（1）PriorityBlockingQueue整个入队出队的过程与PriorityQueue基本是保持一致的；

（2）PriorityBlockingQueue使用一个锁+一个notEmpty条件控制并发安全；

（3）PriorityBlockingQueue扩容时使用一个单独变量的CAS操作来控制只有一个线程进行扩容；

（4）入队使用自下而上的堆化；

（5）出队使用自上而下的堆化；

彩蛋

为什么PriorityBlockingQueue不需要notFull条件？

因为PriorityBlockingQueue在入队的时候如果没有空间了是会自动扩容的，也就不存在队列满了的状态，也就是不需要等待通知队列不满了可以放元素了，所以也就不需要notFull条件了。