问题

（1）什么是优先级队列?

（2）怎么实现一个优先级队列?

（3）PriorityQueue是线程安全的吗?

（4）PriorityQueue就有序的吗?

简介

优先级队列，是0个或多个元素的集合，集合中的每个元素都有一个权重值，每次出队都弹出优先级最大或最小的元素。

一般来说，优先级队列使用堆来实现。

还记得堆的相关知识吗？链接直达【拜托，面试别再问我堆（排序）了！】。

那么Java里面是如何通过“堆”这个数据结构来实现优先级队列的呢?

让我们一起来学习吧。

源码分析

主要属性

```
1.          // 默认容量
2.          private static final int DEFAULT_INITIAL_CAPACITY = 11;
3.          // 存储元素的地方
4.          transient Object[] queue; // non-private to simplify nested class access
5.          // 元素个数
6.          private int size = 0;
7.          // 比较器
8.          private final Comparator<? super E> comparator;
9.          // 修改次数
10.         transient int modCount = 0; // non-private to simplify nested class access
11.
12.
```

（1）默认容量是11；

（2）queue，元素存储在数组中，这跟我们之前说的堆一般使用数组来存储是一致的；

（3）comparator，比较器，在优先级队列中，也有两种方式比较元素，一种是元素的自然顺序，一种是通过比较器来比较；

（4）modCount，修改次数，有这个属性表示PriorityQueue也是fast-fail的；

不知道fast-fail的，查看这篇文章的*彩蛋*部分：【死磕 java集合之HashSet源码分析】。

入队

入队有两个方法，add(E e)和offer(E e)，两者是一致的，add(E e)也是调用的offer(E e)。

```java
    public boolean add(E e) {
        return offer(e);
    }

    public boolean offer(E e) {
        // 不支持null元素
        if (e == null)
            throw new NullPointerException();
        modCount++;
        // 取size
        int i = size;
        // 元素个数达到最大容量了，扩容
        if (i >= queue.length)
            grow(i + 1);
        // 元素个数加1
        size = i + 1;
        // 如果还没有元素
        // 直接插入到数组第一个位置
        // 这里跟我们之前讲堆不一样了
        // java里面是从0开始的
        // 我们说的堆是从1开始的
        if (i == 0)
            queue[0] = e;
        else
            // 否则，插入元素到数组size的位置，也就是最后一个元素的下一位
            // 注意这里的size不是数组大小，而是元素个数
            // 然后，再做自下而上的堆化
            siftUp(i, e);
        return true;
    }
```

```java

    private void siftUp(int k, E x) {
        // 根据是否有比较器，使用不同的方法
        if (comparator != null)
            siftUpUsingComparator(k, x);
        else
            siftUpComparable(k, x);
    }

    @SuppressWarnings("unchecked")
    private void siftUpComparable(int k, E x) {
        Comparable<? super E> key = (Comparable<? super E>) x;
        while (k > 0) {
            // 找到父节点的位置
            // 因为元素是从0开始的，所以减1之后再除以2
            int parent = (k - 1) >>> 1;
            // 父节点的值
            Object e = queue[parent];
            // 比较插入的元素与父节点的值
            // 如果比父节点大，则跳出循环
            // 否则交换位置
            if (key.compareTo((E) e) >= 0)
                break;
            // 与父节点交换位置
            queue[k] = e;
            // 现在插入的元素位置移到了父节点的位置
            // 继续与父节点再比较
            k = parent;
        }
        // 最后找到应该插入的位置，放入元素
        queue[k] = key;
    }
```

```
63.
```

（1）入队不允许null元素；

（2）如果数组不够用了，先扩容；

（3）如果还没有元素，就插入下标0的位置；

（4）如果有元素了，就插入到最后一个元素往后的一个位置（实际并没有插入哈）；

（5）自下而上堆化，一直往上跟父节点比较；

（6）如果比父节点小，就与父节点交换位置，直到出现比父节点大为止；

（7）由此可见，PriorityQueue是一个小顶堆。

扩容

```
1.     private void grow(int minCapacity) {
2.         // 旧容量
3.         int oldCapacity = queue.length;
4.         // Double size if small; else grow by 50%
5.         // 旧容量小于64时，容量翻倍
6.         // 旧容量大于等于64，容量只增加旧容量的一半
7.         int newCapacity = oldCapacity + ((oldCapacity < 64) ?
8.                                          (oldCapacity + 2) :
9.                                          (oldCapacity >> 1));
10.        // overflow-conscious code
11.        // 检查是否溢出
12.        if (newCapacity - MAX_ARRAY_SIZE > 0)
13.            newCapacity = hugeCapacity(minCapacity);
```

```
14.
15.        // 创建出一个新容量大小的新数组并把旧数组元素拷贝过去
16.        queue = Arrays.copyOf(queue, newCapacity);
17.    }
18.
19.    private static int hugeCapacity(int minCapacity) {
20.        if (minCapacity < 0) // overflow
21.            throw new OutOfMemoryError();
22.        return (minCapacity > MAX_ARRAY_SIZE) ?
23.            Integer.MAX_VALUE :
24.            MAX_ARRAY_SIZE;
25.    }
26.
```

（1）当数组比较小（小于64）的时候每次扩容容量翻倍；

（2）当数组比较大的时候每次扩容只增加一半的容量；

出队

出队有两个方法，remove()和poll()，remove()也是调用的poll()，只是没有元素的时候抛出异常。

```
1.     public E remove() {
2.         // 调用poll弹出队首元素
3.         E x = poll();
4.         if (x != null)
5.             // 有元素就返回弹出的元素
6.             return x;
7.         else
8.             // 没有元素就抛出异常
```

```
            throw new NoSuchElementException();
    }

    @SuppressWarnings("unchecked")
    public E poll() {
        // 如果size为0，说明没有元素
        if (size == 0)
            return null;
        // 弹出元素，元素个数减1
        int s = --size;
        modCount++;
        // 队列首元素
        E result = (E) queue[0];
        // 队列末元素
        E x = (E) queue[s];
        // 将队列末元素删除
        queue[s] = null;
        // 如果弹出元素后还有元素
        if (s != 0)
            // 将队列末元素移到队列首
            // 再做自上而下的堆化
            siftDown(0, x);
        // 返回弹出的元素
        return result;
    }

    private void siftDown(int k, E x) {
        // 根据是否有比较器，选择不同的方法
        if (comparator != null)
            siftDownUsingComparator(k, x);
        else
            siftDownComparable(k, x);
    }
```

```

    @SuppressWarnings("unchecked")
    private void siftDownComparable(int k, E x) {
        Comparable<? super E> key = (Comparable<? super E>)x;
        // 只需要比较一半就行了，因为叶子节点占了一半的元素
        int half = size >>> 1;        // loop while a non-leaf
        while (k < half) {
            // 寻找子节点的位置，这里加1是因为元素从0号位置开始
            int child = (k << 1) + 1; // assume left child is least
            // 左子节点的值
            Object c = queue[child];
            // 右子节点的位置
            int right = child + 1;
            if (right < size &&
                ((Comparable<? super E>) c).compareTo((E) queue[right]) > 0)
                // 左右节点取其小者
                c = queue[child = right];
            // 如果比子节点都小，则结束
            if (key.compareTo((E) c) <= 0)
                break;
            // 如果比最小的子节点大，则交换位置
            queue[k] = c;
            // 指针移到最小子节点的位置继续往下比较
            k = child;
        }
        // 找到正确的位置，放入元素
        queue[k] = key;
    }

```

（1）将队列首元素弹出；

（2）将队列末元素移到队列首；

（3）自上而下堆化，一直往下与最小的子节点比较；

（4）如果比最小的子节点大，就交换位置，再继续与最小的子节点比较；

（5）如果比最小的子节点小，就不用交换位置了，堆化结束；

（6）这就是堆中的删除堆顶元素；

取队首元素

取队首元素有两个方法，element()和peek()，element()也是调用的peek()，只是没取到元素时抛出异常。

```
1.    public E element() {
2.        E x = peek();
3.        if (x != null)
4.            return x;
5.        else
6.            throw new NoSuchElementException();
7.    }
8.    public E peek() {
9.        return (size == 0) ? null : (E) queue[0];
10.   }
11.
```

（1）如果有元素就取下标0的元素；

（3）如果没有元素就返回null，element()抛出异常；

总结

（1）PriorityQueue是一个小顶堆；

（2）PriorityQueue是非线程安全的；

（3）PriorityQueue不是有序的，只有堆顶存储着最小的元素；

（4）入队就是堆的插入元素的实现；

（5）出队就是堆的删除元素的实现；

（6）还不懂堆？看一看这篇文章【拜托，面试别再问我堆（排序）了！】。

彩蛋

（1）论Queue中的那些方法?

Queue是所有队列的顶级接口，它里面定义了一批方法，它们有什么区别呢?

操作	抛出异常	返回特定值
入队	add(e)	offer(e)——false
出队	remove()	poll()——null
检查	element()	peek()——null

（2）为什么PriorityQueue中的add(e)方法没有做异常检查呢?

因为PriorityQueue是无限增长的队列，元素不够用了会扩容，所以添加元素不会失败。