PriorityQueue 的源码分析

PriorityQueue 的特点：

- PriorityQueue 是一个无界队列，没有容量限制。
- PriorityQueue 不允许插入空元素，如果提供了比较器，插入的元素就按照比较器排序。否则，按照自然顺序来排序。
- PriorityQueue 内部没有实现同步，是线程不安全的队列。

下面从三个方面分析其源码：

1.其内部类的成员属性

2.其构造方法

3.其常用的方法：公共方法 、插入方法、移除方法

1.其内部类的成员属性

```
    //队列的默认初始容量为11
     private static final int DEFAULT_INITIAL_CAPACITY = 11;

    //使用数组来存储元素
     transient Object[] queue;

     //队列中的元素个数
     private int size = 0;

     //元素排序使用的比较器，如果 comparator 为 null，则使用自然排序
     private final Comparator<? super E> comparator;

     //记录队列结构修改的次数
     transient int modCount = 0;

    //数组的最大长度
    private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

```

2.其构造方法

- 默认无参构造方法，既不指定队列的初始容量，也不指定比较器。

```
    public PriorityQueue() {
        //使用默认的初始容量11，插入的元素为自然排序
        this(DEFAULT_INITIAL_CAPACITY, null);
```

```
    }
```

- 指定初始容量的构造方法

```
    public PriorityQueue(int initialCapacity) {
        //指定初始容量为initialCapacity，插入的元素使用自然排序
        this(initialCapacity, null);
    }
```

- 指定比较器的构造方法

```
    public PriorityQueue(Comparator<?super E>comparator){
        //使用指定的比较器和初始容量，初始容量为11
        this(DEFAULT_INITIAL_CAPACITY, comparator);
    }
```

- 指定初始容量和比较器的构造方法

```
     public PriorityQueue
        (int initialCapacity, Comparator<? super E>comparator){
        //指定的容量            指定的比较器
        //如果指定的初始容量小于1，就使用默认的11
        if (initialCapacity < 1)
            throw new IllegalArgumentException();
        this.queue = new Object[initialCapacity];
        this.comparator = comparator;
        }
```

- 使用给定集合构造队列的方法

```
 public PriorityQueue(Collection<? extends E> c) {
     if (c instanceof SortedSet<?>) {
            SortedSet<? extends E> ss = (SortedSet<? extends E>) c;
            this.comparator = (Comparator<? super E>) ss.comparator();
            initElementsFromCollection(ss);
        }
     else if (c instanceof PriorityQueue<?>) {
       PriorityQueue<? extends E> pq = (PriorityQueue<? extends E>) c;
            this.comparator = (Comparator<? super E>) pq.comparator();
            initFromPriorityQueue(pq);
        }
    else{
            this.comparator = null;
            initFromCollection(c);
        }
    }
```

- 使用给定优先级队列构造队列的方法

```
//使用给定优先级队列构造队列的方法
    public PriorityQueue(PriorityQueue<? extends E> c) {
        this.comparator = (Comparator<? super E>) c.comparator();
        initFromPriorityQueue(c);
```

```
    }
```

- 使用给定SortedSet构造队列的方法

```
//使用给定SortedSet构造队列的方法
  public PriorityQueue(SortedSet<? extends E> c) {
      this.comparator = (Comparator<? super E>) c.comparator();
      initElementsFromCollection(c);

```

3.其常用方法

1.扩容（grow）

```
private void grow(int minCapacity) {
       int oldCapacity = queue.length;
       // Double size if small; else grow by 50%
       //判断当前容量和64的大小关系，如果比64小，就扩容2倍，之后在加2
       //如果比64大，就扩容1.5倍
       int newCapacity = oldCapacity + ((oldCapacity < 64) ?
                                        (oldCapacity + 2) :
                                        (oldCapacity >> 1));
       // overflow-conscious code
       //如果新的长的比规定的最大长度还要长
       //就是不要让他扩的太狠了，造成浪费
       if (newCapacity - MAX_ARRAY_SIZE > 0)
           newCapacity = hugeCapacity(minCapacity);
        //queue扩容为newCapacity的大小
        queue = Arrays.copyOf(queue, newCapacity);
  }
   //返回扩容后数组的长度
   private static int hugeCapacity(int minCapacity) {
       //超过数组的最大容量
       //两种情况:1.长度小于0    2.太大了，比MAX_ARRAY_SIZE-8还要大
       //长度小于0，就抛出一个异常
       if (minCapacity < 0) // overflow
           throw new OutOfMemoryError();
       //太长了的话就看看和规定的数组比较
```

```
        //要是比规定的大就返回 Integer.MAX_VALUE
        //要是比规定的小就返回 MAX_ARRAY_SIZE
        return (minCapacity > MAX_ARRAY_SIZE) ?
            Integer.MAX_VALUE :
            MAX_ARRAY_SIZE;
    }
```

2.添加元素（add/offer）

```
//执行offer
public boolean add(E e) {
        return offer(e);
    }
//
public boolean offer(E e) {
        //判断下添加的元素是否为空
        if (e == null)
            throw new NullPointerException();
        modCount++;
        //判断是否需要扩容
        int i = size;
        if (i >= queue.length)
            grow(i + 1);
        size = i + 1;
        //判断是否添加的是第一个元素
        if (i == 0)
            queue[0] = e;
        else
        //排序
            siftUp(i, e);
        return true;
    }
```

比较器排序

这里主要应用的是堆排序

```

    //向上调整构造最小堆
    private void siftUp(int k, E x) {
        if (comparator != null)
            siftUpUsingComparator(k, x);
        else
            siftUpComparable(k, x);
    }

```

```
    //使用自然排序向上调整构造最小堆
    private void siftUpComparable(int k, E x) {
        Comparable<? super E> key = (Comparable<? super E>) x;
        while (k > 0) {
            //计算第k个节点的父节点的下标
            int parent = (k - 1) >>> 1;
            Object e = queue[parent];
            //如果x大于第k个节点的父节点，说明已是最小堆，直接跳出循环
            if (key.compareTo((E) e) >= 0)
                break;
            //交换第k个节点与父节点的元素
            queue[k] = e;
            k = parent;
        }
        queue[k] = key;
    }

    //使用比较器向上调整构造最小堆
    private void siftUpUsingComparator(int k, E x) {
        while (k > 0) {
           //计算第k个节点的父节点的下标
            int parent = (k - 1) >>> 1;
            Object e = queue[parent];
            //如果x大于第k个节点的父节点，说明已是最小堆，直接跳出循环
            if (comparator.compare(x, (E) e) >= 0)
                break;
            //交换第k个节点与父节点的元素
            queue[k] = e;
            k = parent;
        }
        queue[k] = x;
    }

    //向下调整构造最小堆
    private void siftDown(int k, E x) {
        if (comparator != null)
             //比较器不为null，使用比较器构造最小堆
            siftDownUsingComparator(k, x);
        else
            //比较器为null使用自然排序构造最小堆
            siftDownComparable(k, x);
    }

    //使用自然排序向下调整构造最小堆
    private void siftDownComparable(int k, E x) {
        Comparable<? super E> key = (Comparable<? super E>)x;
        int half = size >>> 1;        // loop while a non-leaf
        while (k < half) {
             //计算第k个节点的左孩子节点下标，第k个节点的左孩子节点为2k+1
```

```
            int child = (k << 1) + 1; // assume left child is least
            Object c = queue[child];
            //计算第k个节点的右孩子节点下标，第k个节点的右孩子节点为2k+2
            int right = child + 1;
            //获取左右孩子中最小的节点
            if (right < size &&
                ((Comparable<? super E>) c).compareTo((E) queue[right]) >
0)
                c = queue[child = right];
            //如果key小于c，说明已经是最小堆，直接跳出循环
            if (key.compareTo((E) c) <= 0)
                break;
            //将左右孩子中最小的节点放在第k个节点位置
            queue[k] = c;
            k = child; //此处k重新赋值
        }
        //将元素x放在第k个节点位置
        queue[k] = key;
    }

    //使用比较器排序向下调整构造最小堆
    private void siftDownUsingComparator(int k, E x) {
        int half = size >>> 1;
        while (k < half) {
             //计算第k个节点的左孩子节点下标，第k个节点的左孩子节点为2k+1
            int child = (k << 1) + 1;
            Object c = queue[child];
            //计算第k个节点的右孩子节点下标，第k个节点的右孩子节点为2k+2
            int right = child + 1;
             //获取左右孩子中最小的节点
            if (right < size &&
                comparator.compare((E) c, (E) queue[right]) > 0)
                c = queue[child = right];
             //如果x小于c，说明已经是最小堆，直接跳出循环
            if (comparator.compare(x, (E) c) <= 0)
                break;
            //将左右孩子中最小的节点放在第k个节点位置
            queue[k] = c;
            k = child;   //此处k重新赋值
        }
         //将元素x放在第k个节点位置
        queue[k] = x;
    }

    //向下调整构建最小堆
    private void heapify() {
        for (int i = (size >>> 1) - 1; i >= 0; i--)
             //向下调整构建最小堆
            siftDown(i, (E) queue[i]);
```

```
107     }
108
```