

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

```
1 //队列的默认初始容量为11
2 private static final int DEFAULT_INITIAL_CAPACITY = 11;
3
4 //使用数组来存储元素
5 transient Object[] queue;
6
7 //队列中的元素个数
8 private int size = 0;
9
10 //元素排序使用的比较器，如果 comparator 为 null，则使用自然排序
11 private final Comparator<? super E> comparator;
12
13 //记录队列结构修改的次数
14 transient int modCount = 0;
15
16 //数组的最大长度
17 private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
18
```

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。

```
1 public PriorityQueue() {
2     //使用默认的初始容量11，插入的元素为自然排序
3     this(DEFAULT_INITIAL_CAPACITY, null);
4 }
```

```
4     }
```

- 指定初始容量的构造方法

```
1     public PriorityQueue(int initialCapacity) {  
2         //指定初始容量为initialCapacity，插入的元素使用自然排序  
3         this(initialCapacity, null);  
4     }
```

- 指定比较器的构造方法

```
1     public PriorityQueue(Comparator<? super E>comparator){  
2         //使用指定的比较器和初始容量，初始容量为11  
3         this(DEFAULT_INITIAL_CAPACITY, comparator);  
4     }
```

- 指定初始容量和比较器的构造方法

```
1     public PriorityQueue  
2         (int initialCapacity, Comparator<? super E>comparator){  
3         //指定的容量          指定的比较器  
4         //如果指定的初始容量小于1，就使用默认的11  
5         if (initialCapacity < 1)  
6             throw new IllegalArgumentException();  
7         this.queue = new Object[initialCapacity];  
8         this.comparator = comparator;  
9     }
```

- 使用给定集合构造队列的方法

```
1     public PriorityQueue(Collection<? extends E> c) {  
2         if (c instanceof SortedSet<?>) {  
3             SortedSet<? extends E> ss = (SortedSet<? extends E>) c;  
4             this.comparator = (Comparator<? super E>) ss.comparator();  
5             initElementsFromCollection(ss);  
6         }  
7         else if (c instanceof PriorityQueue<?>) {  
8             PriorityQueue<? extends E> pq = (PriorityQueue<? extends E>) c;  
9             this.comparator = (Comparator<? super E>) pq.comparator();  
10            initFromPriorityQueue(pq);  
11        }  
12        else{  
13            this.comparator = null;  
14            initFromCollection(c);  
15        }  
16    }
```

- 使用给定优先级队列构造队列的方法

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

```
5     }
```

- 使用给定SortedSet构造队列的方法

```
1 //使用给定SortedSet构造队列的方法
2 public PriorityQueue(SortedSet<? extends E> c) {
3     this.comparator = (Comparator<? super E>) c.comparator();
4     initElementsFromCollection(c);
5 }
```

3.其常用方法

1.扩容 (grow)

```
1 private void grow(int minCapacity) {
2     int oldCapacity = queue.length;
3     // Double size if small; else grow by 50%
4     //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
5     //如果比64大，就扩容1.5倍
6     int newCapacity = oldCapacity + ((oldCapacity < 64) ?
7                                     (oldCapacity + 2) :
8                                     (oldCapacity >> 1));
9     // overflow-conscious code
10    //如果新的长的比规定的最大长度还要长
11    //就是不要让他扩的太狠了，造成浪费
12    if (newCapacity - MAX_ARRAY_SIZE > 0)
13        newCapacity = hugeCapacity(minCapacity);
14    //queue扩容为newCapacity的大小
15    queue = Arrays.copyOf(queue, newCapacity);
16 }
17 //返回扩容后数组的长度
18 private static int hugeCapacity(int minCapacity) {
19     //超过数组的最大容量
20     //两种情况:1.长度小于0    2.太大了，比MAX_ARRAY_SIZE-8还要大
21     //长度小于0，就抛出一个异常
22     if (minCapacity < 0) // overflow
23         throw new OutOfMemoryError();
24     //太长了的话就看看和规定的数组比较
```

```

25         //要是比规定的大就返回 Integer.MAX_VALUE
26         //要是比规定的小就返回 MAX_ARRAY_SIZE
27         return (minCapacity > MAX_ARRAY_SIZE) ?
28             Integer.MAX_VALUE :
29             MAX_ARRAY_SIZE;
30     }

```

2.添加元素 (add/offer)

```

1 //执行offer
2 public boolean add(E e) {
3     return offer(e);
4 }
5 //
6 public boolean offer(E e) {
7     //判断下添加的元素是否为空
8     if (e == null)
9         throw new NullPointerException();
10    modCount++;
11    //判断是否需要扩容
12    int i = size;
13    if (i >= queue.length)
14        grow(i + 1);
15    size = i + 1;
16    //判断是否添加的是第一个元素
17    if (i == 0)
18        queue[0] = e;
19    else
20        //排序
21        siftUp(i, e);
22    return true;
23 }

```

比较器排序

这里主要应用的是堆排序

```

1
2 //向上调整构造最小堆
3 private void siftUp(int k, E x) {
4     if (comparator != null)
5         siftUpUsingComparator(k, x);
6     else
7         siftUpComparable(k, x);
8 }
9

```

```

10 //使用自然排序向上调整构造最小堆
11 private void siftUpComparable(int k, E x) {
12     Comparable<? super E> key = (Comparable<? super E>) x;
13     while (k > 0) {
14         //计算第k个节点的父节点的下标
15         int parent = (k - 1) >>> 1;
16         Object e = queue[parent];
17         //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
18         if (key.compareTo((E) e) >= 0)
19             break;
20         //交换第k个节点与父节点的元素
21         queue[k] = e;
22         k = parent;
23     }
24     queue[k] = key;
25 }
26
27 //使用比较器向上调整构造最小堆
28 private void siftUpUsingComparator(int k, E x) {
29     while (k > 0) {
30         //计算第k个节点的父节点的下标
31         int parent = (k - 1) >>> 1;
32         Object e = queue[parent];
33         //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
34         if (comparator.compare(x, (E) e) >= 0)
35             break;
36         //交换第k个节点与父节点的元素
37         queue[k] = e;
38         k = parent;
39     }
40     queue[k] = x;
41 }
42
43 //向下调整构造最小堆
44 private void siftDown(int k, E x) {
45     if (comparator != null)
46         //比较器不为null，使用比较器构造最小堆
47         siftDownUsingComparator(k, x);
48     else
49         //比较器为null使用自然排序构造最小堆
50         siftDownComparable(k, x);
51 }
52
53 //使用自然排序向下调整构造最小堆
54 private void siftDownComparable(int k, E x) {
55     Comparable<? super E> key = (Comparable<? super E>)x;
56     int half = size >>> 1; // loop while a non-leaf
57     while (k < half) {
58         //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

```

```

59         int child = (k << 1) + 1; // assume left child is least
60         Object c = queue[child];
61         //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
62         int right = child + 1;
63         //获取左右孩子中最小的节点
64         if (right < size &&
65             ((Comparable<? super E>) c).compareTo((E) queue[right]) >
0)
66             c = queue[child = right];
67         //如果key小于c, 说明已经是最小堆, 直接跳出循环
68         if (key.compareTo((E) c) <= 0)
69             break;
70         //将左右孩子中最小的节点放在第k个节点位置
71         queue[k] = c;
72         k = child; //此处k重新赋值
73     }
74     //将元素x放在第k个节点位置
75     queue[k] = key;
76 }
77
78 //使用比较器排序向下调整构造最小堆
79 private void siftDownUsingComparator(int k, E x) {
80     int half = size >>> 1;
81     while (k < half) {
82         //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
83         int child = (k << 1) + 1;
84         Object c = queue[child];
85         //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
86         int right = child + 1;
87         //获取左右孩子中最小的节点
88         if (right < size &&
89             comparator.compare((E) c, (E) queue[right]) > 0)
90             c = queue[child = right];
91         //如果x小于c, 说明已经是最小堆, 直接跳出循环
92         if (comparator.compare(x, (E) c) <= 0)
93             break;
94         //将左右孩子中最小的节点放在第k个节点位置
95         queue[k] = c;
96         k = child; //此处k重新赋值
97     }
98     //将元素x放在第k个节点位置
99     queue[k] = x;
100 }
101
102 //向下调整构建最小堆
103 private void heapify() {
104     for (int i = (size >>> 1) - 1; i >= 0; i--)
105         //向下调整构建最小堆
106         siftDown(i, (E) queue[i]);

```

