PriorityQueue源码阅读

概述

PriorityQueue：用堆的基本原理实现的。堆就是一棵完全二叉树，**节点之间不是完全有序**，但是结点和父结点之间存在一定的关系

根据关系可以将优先队列区分为：最大堆、最小堆。最大堆就是父结点的值比子节点的值要大，那么根节点一定是值最大的结点。最小堆同理。如果逐个出队，则会得到有序数组。

并且，优先队列在物理上用数组存储，主要是完全二叉树的特性决定的

- 实现了队列接口，队列的API均能使用
- 每个结点均有优先级，优先级最高的排在最前面；
- 为了保持一定顺序，**PriorityQueue要求要么元素实现Comparable接口，要么传递一个比较器Comparator。**
- PriorityQueue也是fast-fail的，即有modCount来处理并发的情况（尽量发现，并不能保证所有并发操作均正确）
- PriorityQueue中**不能存在null结点**
- PriorityQueue是**非线程安全的**

1. 类头

```
public class PriorityQueue<E> extends AbstractQueue<E>
    implements java.io.Serializable
```

继承自AbstractQueue，是一个抽象类：

```
public abstract class AbstractQueue<E> extends AbstractCollection<E>
    implements Queue<E>
```

AbstractQueue实现了队列接口，所以PriorityQueue实现了Queue接口

Queue接口定义的方法：

队列的主要功能有：**队尾添加元素、队头移除元素、查看队首元素**

```
boolean add(E e);        // 如果队列大小达到上限，插入失败，则会抛出异常
boolean offer(E e);      // 大小达到上限，插入失败，不会抛出异常
E remove();          // 如果队列为空，抛出异常
E poll();
E element();         // 如果队列为空，抛出异常
E peek();
```

——总结，有两组方法，一组针对边界条件会抛出异常，一组不会。我们常用的是：`offer(xxx)、poll()、peek()`

2. 静态变量 or 实例变量

```
private static final int DEFAULT_INITIAL_CAPACITY = 11;          //如果构造方法没有传参，默认容量为11

private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;     // 最大容量（-8是JVM可能会对数组头进行设置）
```

```
transient Object[] queue;                // 内部存储数据的数组

private int size = 0;            // 记录存储的元素个数，不等于queue.length（容量）

transient int modCount = 0;      // 记录修改次数，主要针对多线程（桶ArrayList等）
```

——**PriorityQueue就是用数组存储的，类似于LinkedList**

```
private final Comparator<? super E> comparator; // 比较器，在构造方法中传入，如果没有就用默认序，元素之间比较
```

——**PriorityQueue的比较器是可以传入的，并且只能赋值一次。**如果没有传入，那么就按照元素之间的compareTo进行比较，**默认是最小堆**

3. 构造方法

1. 默认构造方法

```
public PriorityQueue() {
    this(DEFAULT_INITIAL_CAPACITY, null);
}
```

如果没有设定大小，就默认给11.

2. 一参构造方法

```
public PriorityQueue(int initialCapacity) {
    this(initialCapacity, null);
}
```

```
public PriorityQueue(Comparator<? super E> comparator) {
    this(DEFAULT_INITIAL_CAPACITY, comparator);
}
```

——本质山都是去调用二参构造方法。

3. 二参构造方法

```
public PriorityQueue(int initialCapacity, Comparator<? super E> comparator) {
    // Note: This restriction of at least one is not actually needed,
    // but continues for 1.5 compatibility
    if (initialCapacity < 1)
        throw new IllegalArgumentException();           // 容量至少要>=1
    this.queue = new Object[initialCapacity];
    this.comparator = comparator;
}
```

——主要就是赋值：创建指定长度的数组、赋值指定的比较器。

4. 传递容器对象构造方法

如果传递的是：集合对象

```
public PriorityQueue(Collection<? extends E> c) {
    if (c instanceof SortedSet<?>) {                // 先看是否是排序的集合
        SortedSet<? extends E> ss = (SortedSet<? extends E>) c;
        this.comparator = (Comparator<? super E>) ss.comparator();  // 获得比较器
        initElementsFromCollection(ss);
    }
    else if (c instanceof PriorityQueue<?>) {           // 是否是优先队列对象
        PriorityQueue<? extends E> pq = (PriorityQueue<? extends E>) c;
        this.comparator = (Comparator<? super E>) pq.comparator();  // 获取其比较器
        initFromPriorityQueue(pq);
    }
    else {
        this.comparator = null;
        initFromCollection(c);
    }
}
```

理解：

1. 如果传递的集合是SortSet，即本身就是有序的，那么一定有比较器，则获得其比较器
2. 如果传递的集合是PriorityQueue的，获得其比较器
3. 如果传递的集合是普通的Collection对象

4. 辅助方法

辅助方法，主要针对的是初始时，不同的传参来构造堆的方法：

传入优先队列，来构造优先队列

操作：直接复制一个新的数组即可

```
private void initFromPriorityQueue(PriorityQueue<? extends E> c) {
    if (c.getClass() == PriorityQueue.class) {      // 再次判断传参是否是否是优先队列
类型
        this.queue = c.toArray();        // 直接复制一份新的数组
        this.size = c.size();       // 大小也跟着
    } else {
        initFromCollection(c);      // 如果不是优先队列类型，那么需要调用普通的
collection建堆方式
    }
}
```

传入普通集合类对象，来构造优先级队列

操作：先获得存储数据的数组，然后再建堆

```
private void initFromCollection(Collection<? extends E> c) {
    initElementsFromCollection(c);
    heapify();
}
```

```
private void initElementsFromCollection(Collection<? extends E> c) {
    Object[] a = c.toArray();           // 先获得数组
    // If c.toArray incorrectly doesn't return Object[], copy it.
    if (a.getClass() != Object[].class)         // 而如果a不是object类型的数组（是Arrays.asList().toArray()的）
        a = Arrays.copyOf(a, a.length, Object[].class);     // 需要指定元素类型复制一次
    int len = a.length;
    if (len == 1 || this.comparator != null)    // 如果只有一个元素，或者存在的比较器，那么不能存在元素为null
        for (int i = 0; i < len; i++)
            if (a[i] == null)
                throw new NullPointerException();
    this.queue = a;
    this.size = a.length;
}
```

——如果有比较器，那么元素不能为null；如果只有一个元素，那么该元素也不能为null

建堆的过程：

```
private void heapify() {
    for (int i = (size >>> 1) - 1; i >= 0; i--) // 从最后一个非叶子结点开始构建堆
        siftDown(i, (E) queue[i]);      // 向下调整
}
```

```
private void siftDown(int k, E x) {
    if (comparator != null)             // 如果有比较器就用比较器比较
        siftDownUsingComparator(k, x);
    else
        siftDownComparable(k, x);       // 如果没有，就用元素的自带的compareTo比较
}
```

向下找到适合k的插入位置：

```
private void siftDownUsingComparator(int k, E x) {
    int half = size >>> 1;
    while (k < half) {          // 超过一半，说明已经到叶子结点，迭代可以结束了
        int child = (k << 1) + 1;           // 2k+1，表示k结点的左孩子
        Object c = queue[child];
        int right = child + 1;      // 右孩子结点的index
        if (right < size &&                 // 如果存在右孩子，且右孩子优先级更高（最小堆中就是值更小）
            comparator.compare((E) c, (E) queue[right]) > 0)
            c = queue[child = right];           // 那么待交换的就是做孩子
        if (comparator.compare(x, (E) c) <= 0)      // 和孩子中的最高优先级进行比较，如果x优先级更高循环结束
            break;
        queue[k] = c;   // 将优先级更高的结点作为父结点
        k = child;
    }
    queue[k] = x;
}

private void siftDownUsingComparator(int k, E x) {....} // 同理
```

流程：

1. 根据index，去获取其左右孩子，和左右孩子中优先级更高的结点（最小堆中就是值更小）的结点进行比较，如果根的优先级更高，那么找到合适的插入点了；否则就将根和那个孩子互换，然后以孩子结点为起点，再次向下遍历，直到找到合适的位置 or 已经到了叶子结点

总体建堆的思路，就是从最后一个非叶子结点开始建堆，每个结点的建堆方式都是向下找合适的插入点；最后一个非叶子结点建好后，向上一直到根节点遍历。

向上调整：出现在插入新结点的时候调用

```
private void siftUp(int k, E x) {
    if (comparator != null)
        siftUpUsingComparator(k, x);
    else
        siftUpComparable(k, x);
}

private void siftUpComparable(int k, E x) {
    Comparable<? super E> key = (Comparable<? super E>) x;      // 看x是否实现了Comparable接口
    while (k > 0) {
        int parent = (k - 1) >>> 1;     // 找到父结点
        Object e = queue[parent];
        if (key.compareTo((E) e) >= 0)          // 如果比父结点优先级低（最小堆中，比父结点值更大），就是合适的位置
            break;
        queue[k] = e;       // 比父结点优先级更高（比父结点更小），那么和父结点互换
        k = parent;     // 以父结点为起点向上再调整
    }
    queue[k] = key;
}

private void siftUpUsingComparator(int k, E x) {}
```

流程：

1. 根据index，去获取父结点，如果比父结点小（最小堆中），那么将父结点换到index上，然后从父结点（(index-1)/2）开始再次向上遍历，直到找到合适的插入位置 or 到根节点

5. 扩容

扩容，整体和ArrayList一致，稍微分的更细一点：

- 旧容量 < 64，那么扩容到2倍+2
- 旧容量 > 64，那么扩容到1.5倍

上限扩容也一样，如果最小长度已经超过上限，那么就给Integer.MAX_VALUE，下一次还需要扩容就会溢出

```
private void grow(int minCapacity) {
    int oldCapacity = queue.length;     // 旧长度
    // 如果容量小于64，那么扩容是2倍+2；如果大于64，那么是扩容到1.5倍
    int newCapacity = oldCapacity + ((oldCapacity < 64) ?
                                     (oldCapacity + 2) :
                                     (oldCapacity >> 1));
    // 如果超过上限
    if (newCapacity - MAX_ARRAY_SIZE > 0)
```

```
            newCapacity = hugeCapacity(minCapacity);
        queue = Arrays.copyOf(queue, newCapacity);       // 将数组复制过去
    }

    // 上限容量还需要扩容
    private static int hugeCapacity(int minCapacity) {
        if (minCapacity < 0) // 出现负数，说明已经溢出了，报错
            throw new OutOfMemoryError();
        return (minCapacity > MAX_ARRAY_SIZE) ?          // 如果需要的最小容量超过上限，就
    给整型的最大值
            Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
    }
```

6. Queue的3个方法的实现

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

public boolean offer(E e) {
    if (e == null)                    // 不能有null结点
        throw new NullPointerException();
    modCount++;
    int i = size;              // 先添加到最后
    if (i >= queue.length)             // 超过容量，则需要扩容（5）
        grow(i + 1);
    size = i + 1;
    if (i == 0)                // 如果堆为空，就是添加第一个元素，那么直接赋值即可
        queue[0] = e;
    else
        siftUp(i, e);          // 否则需要向上调整
    return true;
}
```

注意：Queue默认的add方法，如果达到上限，是会抛出异常的，但是**PriorityQueue的add方法并不会抛出异常。**

∵每次插入都会去判断是否需要扩容，所以如果达到上限，grow方法会抛出异常

```
public E peek() {
    return (size == 0) ? null : (E) queue[0];
}
```

（element方法是直接继承了AbstractQueue的）

```
public E element() {
    E x = peek();
    if (x != null)
        return x;
    else
        throw new NoSuchElementException();      // 如果返回值是null，说明堆为空，抛出
    异常
}
```

弹出堆顶

```
public E poll() {
    if (size == 0)
        return null;
    int s = --size;
    modCount++;
    E result = (E) queue[0];
    E x = (E) queue[s];      // 将最后一个元素的值取出
    queue[s] = null;         // 最后一个结点清空
    if (s != 0)
        siftDown(0, x);      // 向下调整
    return result;
}
```

（remove方法是直接继承了AbstractQueue的）

```
public E remove() {
    E x = poll();
    if (x != null)
        return x;
    else
        throw new NoSuchElementException();      // 如果弹出失败，那么抛出异常
}
```

7. 其他方法

7.1 查

获得指定元素的下标：

```
private int indexOf(Object o) {
    if (o != null) {
        for (int i = 0; i < size; i++)       // 数组遍历，找到第一个匹配的即可
            if (o.equals(queue[i]))
                return i;
    }
    return -1;
}
```

看该结点是否存在

```
public boolean contains(Object o) {
    return indexOf(o) != -1;
}
```

求元素个数

```
public int size() {
    return size;
}
```

7.2 删

删除指定元素

```
public boolean remove(Object o) {
    int i = indexOf(o);             // 找到该元素的位置
    if (i == -1)
        return false;
    else {
        removeAt(i);
        return true;
    }
}

private E removeAt(int i) {
    // assert i >= 0 && i < size;
    modCount++;
    int s = --size;
    if (s == i) // 如果删除的是最后一个元素，直接清空该内容即可
        queue[i] = null;
    else {
        E moved = (E) queue[s];
        queue[s] = null;
        siftDown(i, moved);     // 将最后一个结点从被删除出开始向下找合适的插入位置
        if (queue[i] == moved) {    // 如果还是在原地，说明需要向上调整
            siftUp(i, moved);
            if (queue[i] != moved)      // 如果发生了更新，那么返回最后的结点
                return moved;
        }
    }
    return null;
}
```

流程：

1. 先找到删除的结点位置（遍历数组）
2. 然后获取最后一个结点，从被删除的index开始，先向下遍历，如果index的内容发生变化，即不等于moved，那么说明其向下遍历找到了合适的点；如果index未发生变化，即等于moved，那么说明其需要向上遍历，然后就向上遍历找合适的点
3. 如果向上遍历找到合适的点，那么返回该结点；如果只需要向下遍历，那么返回null

```
boolean removeEq(Object o) {
    for (int i = 0; i < size; i++) {
        if (o == queue[i]) {
            removeAt(i);
            return true;
        }
    }
    return false;
}
```

如果遇到了值相等的元素，就删除，具体删除操作同上。

清空数组：

```
public void clear() {
    modCount++;
    for (int i = 0; i < size; i++)
        queue[i] = null;
    size = 0;
}
```

7.3 toArray：转换成数组

就是将存储数据的数组复制一份出来，注意是**浅拷贝**

```
public Object[] toArray() {
    return Arrays.copyOf(queue, size);
}
```

转换成数组，该数组是给定的

```
public <T> T[] toArray(T[] a) {
    final int size = this.size;
    if (a.length < size)            // 如果给定数组长度不够的话，那么创建一个新的数组赋值
        // Make a new array of a's runtime type, but my contents:
        return (T[]) Arrays.copyOf(queue, size, a.getClass());
    System.arraycopy(queue, 0, a, 0, size); // 如果给定数组长度够的话，那么就用给定的数
组复制
    if (a.length > size)
        a[size] = null;     // 数组后面全部给null（可能之前有内容）
    return a;
}
```

8. 迭代器

之前看ArrayList、LinkedList都是有迭代器，而PriorityQueue也有迭代器。

迭代器的作用：

迭代器模式：用于遍历集合类的标准访问方法。它可以把**访问逻辑从不同类型的集合类中抽象出**来，从而避免向客户端暴露集合的内部结构

如果没有迭代器，那么如果要访问ArrayList可以用`for (int i=0; i<array.size(); i++) {...}`；访问LinkedList可以用`while ((e=e.next())!=null) {...}`，这样必须事先知道集合的内部结构，代码无法复用，如果进行替换，那么代码需要重写。

Iterator模式总是用同一种逻辑来遍历集合：

```
for(Iterator it = c.iterator(); it.hasNext();){
    Object obj = it.next();
}

// 之后优化成：更为简单，但是隐蔽了其实现的本质
for(Type num: c){
    ....
}
```

所有的内部状态（如当前元素位置，是否有下一个元素）都由Iterator来维护，能够遍历集合，且对于所有集合类方法都一样，代码解耦合。

遍历过程中如果发生了删除操作，并不能保证顺序按照原来的样子

```
private final class Itr implements Iterator<E> {
    private int cursor = 0;          // 指向下一个要访问的结点
    private int lastRet = -1;        // 上一个返回的结点

    private ArrayDeque<E> forgetMeNot = null;    // 主要是涉及到删除，删除之后需要重建堆，那么可能存在部分结点被忽略
    private E lastRetElt = null;     // 记录上一次被删除的结点

    private int expectedModCount = modCount;         // 记录遍历过程中是否发生结构性变化（外部方法导致的）

    public boolean hasNext() {                   // 看是否存在下一个结点
        return cursor < size ||
            (forgetMeNot != null && !forgetMeNot.isEmpty());
    }

    public E next() {
        if (expectedModCount != modCount)
            throw new ConcurrentModificationException();
        if (cursor < size)
            return (E) queue[lastRet = cursor++];    // 先遍历完所有结点，再遍历那些因为删除而被提到前面的结点
        if (forgetMeNot != null) {       // 说明发生了删除，且结点发生了变化
            lastRet = -1;
            lastRetElt = forgetMeNot.poll();     // 里面的结点出队
            if (lastRetElt != null)
                return lastRetElt;
        }
        throw new NoSuchElementException();
    }

    public void remove() {
        if (expectedModCount != modCount)
            throw new ConcurrentModificationException();
        if (lastRet != -1) {         // 之前都在正常遍历，这一次需要删除当前结点
            E moved = PriorityQueue.this.removeAt(lastRet); // 调用外部的removeAt方法删除
            lastRet = -1;        // lastRet表示本次循环进行了删除
            if (moved == null)       // 如果返回值是null，说明堆结构没有发生变化（删除是最后一个结点/结点没有移动）
                cursor--;        // 只需要更新cursor即可
            else {               // 如果返回了新插入了的结点，那么说明数组发生了变化，所以需要将该结点加入到队列中
                if (forgetMeNot == null)         // 如果没有新建则需要新建
                    forgetMeNot = new ArrayDeque<>();
                forgetMeNot.add(moved);
            }
        } else if (lastRetElt != null) {    // lastRet=-1 且 lastRetElt != null，表示遍历到最后，就看因为删除被忽略的结点
            PriorityQueue.this.removeEq(lastRetElt);         // 在优先级队列章将其删除
            lastRetElt = null;
```

```
        } else {            // lastRet=-1 且 lastRetElt = null，已经遍历完了，没有可删除的
            throw new IllegalStateException();
        }
        expectedModCount = modCount;
    }
}
```

参考：https://www.liaoxuefeng.com/article/895885644922112