PriorityQueue源码分析

授之以鱼不如授之以渔

- 知识关系网，过遍数（思维导图+人类遗忘曲线）

语言语法特性

- 有界类型：让自己类的代码可以调用泛型类型的方法。
  - 泛型类型不可以调用方法，因为不知道是什么类型。如果需要使用某个类的方法，则需要给定类型的范围。
  - class MyGenericClassBounded<MyType extends GrandParent>{}
- 泛型的协变和逆变：让参数和返回值等引用类型的泛型类型更灵活。
  - <? super E>
  - <? extends E>
  - Java泛型对协变和逆变的支持是为了支持范围更广的参数类型。
  - 协变和逆变是针对引用类型而言的，可以用在返回值类型，参数类型，等引用类型上。创建对象的时候，不能使用协变和逆变。
  - 用法：写入使用逆变，读取使用协变。

```
//协变语法如下，意思就是这个参数可以接受的List引用的泛型类型为Parent或者其子类
public static void extMethod(List<? extends Parent> extParam){}

//同样的道理，我们也可以创建协变的引用，让它可以接受的List引用的泛型类型为Parent或者其子类
List<? extends Parent> g2ListExt = null;
//下面会出错
g2ListExt = new ArrayList<Chirdren>();
g2ListExt.add(new GrandParent());
g2ListExt.add(new Parent());
```

```
g2ListExt.add(new Chirdren());

//逆变和协变正好相反，允许的类型为Parent或者其父类
List<? super Parent> g2ListSup = null;
//这代码可行
g2ListSup = new ArrayList<Parent>();
g2ListSup = new ArrayList<GrandParent>();

//但是同样的原因，无法让具体的类型满足其参数要求，甚至是Object
g2ListExt.add(new GrandParent());
g2ListExt.add(new Parent());
g2ListExt.add(new Chirdren());
g2ListExt.add(new Object());

//无论是协变还是逆变，都只能用在引用上，而不能在创建对象时使用其做为泛型参数
List<? extends Parent> g2ListExt = new ArrayList<? extends Parent>()
;
List<? super Parent> g2ListExt = new ArrayList<? super Parent>();
```

- 个人总结：由于java的泛型实现是通过类型擦除的，所以无论是普通的泛型，还是协变还是逆变，一旦引用地址确定类型，其使用过程类型就不变了，除非换个引用地址。

计算机基础知识

- Java标准库提供了基于二叉堆实现的PriorityQueue。具体是一个完全二叉树（小顶堆）数据结构，是通过数组来存储的。

数据结构与算法

- https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/PriorityQueue.html
- PriorityQueue实现不是Synchronized的，多线程不要同时修改PriorityQueue，如果要同时修改queue，需要使用线程安全的PriorityBlockingQueue。
- offer，poll，remove，add等入队和出队方法是O(log(n))时间复杂度。remove(Object)和contains（Object）是O(n)时间复杂度。peek、element和size是O(1)时间复杂度。
- 优先队列实现机制：
  - 1、Heap（Binary，Binomial，Fibonacci）
  - 2、Binary Search Tree

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[8][9]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(\log n)$	$O(\log n)$[c]
Fibonacci[8][10]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Pairing[11]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$o(\log n)$[b][d]	$\Theta(1)$
Brodal[14][e]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[16]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Strict Fibonacci[17]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2-3 heap[18]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$\Theta(1)$	?

- PriorityQueue是用数组存储元素的，当数组空间不足需要扩容时，进行了判断：如果旧数组长度小于64，就扩大为原来的2倍长，若大于64，就扩大为原来的1.5倍长。

```
    /**
     * Increases the capacity of the array.
     *
     * @param minCapacity the desired minimum capacity
     */
    private void grow(int minCapacity) {
        int oldCapacity = queue.length;
        // Double size if small; else grow by 50%
        int newCapacity = ArraysSupport.newLength(oldCapacity,
                minCapacity - oldCapacity, /* minimum growth */
                oldCapacity < 64 ? oldCapacity + 2 : oldCapacity >>
1
                                           /* preferred growth */);
        queue = Arrays.copyOf(queue, newCapacity);
    }
```

- PriorityQueue默认是一个小顶堆，插入数据时进行堆的调整。

```
private static <T> void siftUpComparable(int k, T x, Object[] es) {
        Comparable<? super T> key = (Comparable<? super T>) x;
        while (k > 0) {
            int parent = (k - 1) >>> 1;
            Object e = es[parent];
            if (key.compareTo((T) e) >= 0)
```

```
                break;
            es[k] = e;
            k = parent;
        }
        es[k] = key;
    }
```

- PriorityQueue通过传入自定义的Comparator函数来实现大顶堆。

```
private static <T> void siftUpUsingComparator(
        int k, T x, Object[] es, Comparator<? super T> cmp) {
        while (k > 0) {
            int parent = (k - 1) >>> 1;
            Object e = es[parent];
            if (cmp.compare(x, (T) e) >= 0)
                break;
            es[k] = e;
            k = parent;
        }
        es[k] = x;
    }
```

设计模式

- 组合
- 迭代器模式
  - foreach循环、迭代器循环，本质上属于一种，都可以看作迭代器遍历。
  - 迭代器模式封装集合内部的复杂数据结构，开发者不需要了解如何遍历，直接使用容器提供的迭代器即可。
  - 迭代器模式将集合对象的遍历操作从集合类中拆分出来，放到迭代器类中，让两者的职责更加单一。
  - 迭代器模式让添加的遍历算法更加容易，更符合开闭原则。除此之外，因为迭代器都实现自相同的接口，在开发中，基于接口而非实现编程，替换迭代器也变得更加容易。

```

List<String> names = new ArrayList<>();
names.add("xzg");
names.add("wang");
```

```java
names.add("zheng");

// 第一种遍历方式：for循环
for (int i = 0; i < names.size(); i++) {
  System.out.print(names.get(i) + ",");
}

// 第二种遍历方式：foreach循环
for (String name : names) {
  System.out.print(name + ",")
}

// 第三种遍历方式：迭代器遍历
Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
  System.out.print(iterator.next() + ",");//Java中的迭代器接口是第二种定义方式，next()既移动游标又返回数据
}
```