java对象的比较

本节目标

> - 问题的提出
> - 元素的比较
> - Java中对象的比较
> - 集合框架中PriorityQueue的比较方式
> - 模拟实现PriorityQueue

1. 问题提出

上节课我们讲了优先级队列，**优先级队列在插入元素时有个要求：插入的元素不能是null或者元素之间必须要能够进行比较**，为了简单起见，我们只是插入了Integer类型，那优先级队列中能否插入自定义类型对象呢？

```
class Card {
    public int rank;    // 数值
    public String suit; // 花色

    public Card(int rank, String suit) {
        this.rank = rank;
        this.suit = suit;
    }
}

public class TestPriorityQueue {
    public static void TestPriorityQueue()
    {
        PriorityQueue<Card> p = new PriorityQueue<>();
        p.offer(new Card(1, "♠"));
        p.offer(new Card(2, "♠"));
    }

    public static void main(String[] args) {
        TestPriorityQueue();
    }
}
```

优先级队列底层使用堆，而向堆中插入元素时，为了满足堆的性质，必须要进行元素的比较，而此时Card是没有办法直接进行比较的，因此抛出异常。

```
Connected to the target VM, address: '127.0.0.1:17622', transport: 'socket'
Exception in thread "main" java.lang.ClassCastException: Card cannot be cast to java.lang.Comparable
    at java.util.PriorityQueue.siftUpComparable(PriorityQueue.java:653)
    at java.util.PriorityQueue.siftUp(PriorityQueue.java:648)
    at java.util.PriorityQueue.offer(PriorityQueue.java:345)
    at TestPriorityQueue.TestPriorityQueue3(TestPriorityQueue.java:86)
    at TestPriorityQueue.main(TestPriorityQueue.java:92)
Disconnected from the target VM, address: '127.0.0.1:17622', transport: 'socket'
```

2. 元素的比较

2.1 基本类型的比较

在Java中，基本类型的对象可以直接比较大小。

```java
public class TestCompare {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        System.out.println(a > b);
        System.out.println(a < b);
        System.out.println(a == b);

        char c1 = 'A';
        char c2 = 'B';
        System.out.println(c1 > c2);
        System.out.println(c1 < c2);
        System.out.println(c1 == c2);

        boolean b1 = true;
        boolean b2 = false;
        System.out.println(b1 == b2);
        System.out.println(b1 != b2);
    }
}
```

2.2 对象的比较

```java
class Card {
    public int rank;    // 数值
    public String suit; // 花色

    public Card(int rank, String suit) {
        this.rank = rank;
        this.suit = suit;
    }
}

public class TestPriorityQueue {

    public static void main(String[] args) {
```

```
        Card c1 = new Card(1, "♠");
        Card c2 = new Card(2, "♠");
        Card c3 = c1;

        //System.out.println(c1 > c2);   // 编译报错
        System.out.println(c1 == c2);  // 编译成功 ----> 打印false，因为c1和c2指向的是不同对象
        //System.out.println(c1 < c2);   // 编译报错
        System.out.println(c1 == c3);  // 编译成功 ----> 打印true，因为c1和c3指向的是同一个对象
    }
}
```

c1、c2和c3分别是Card类型的引用变量，上述代码在比较编译时：

c1 > c2 编译失败

c1== c2 编译成功

c1 < c2 编译失败

从编译结果可以看出，**Java中引用类型的变量不能直接按照 > 或者 < 方式进行比较。** 那为什么==可以比较?

因为：**对于用户实现自定义类型，都默认继承自Object类，而Object类中提供了equal方法，而==默认情况下调用的就是equal方法**，但是该方法的比较规则是：**没有比较引用变量引用对象的内容，而是直接比较引用变量的地址**，但有些情况下该种比较就不符合题意。

```
// Object中equal的实现，可以看到：直接比较的是两个引用变量的地址
 public boolean equals(Object obj) {
            return (this == obj);
        }
```

3. 对象的比较

有些情况下，需要比较的是对象中的内容，比如：向优先级队列中插入某个对象时，需要对按照对象中内容来调整堆，那该如何处理呢?

3.1 覆写基类的equal

```
public class Card {
    public int rank;    // 数值
    public String suit; // 花色

    public Card(int rank, String suit) {
        this.rank = rank;
        this.suit = suit;
    }

    @Override
    public boolean equals(Object o) {
        // 自己和自己比较
        if (this == o) {
            return true;
        }
```

```
16
17          // o如果是null对象，或者o不是Card的子类
18          if (o == null || !(o instanceof Card)) {
19              return false;
20          }
21
22          // 注意基本类型可以直接比较，但引用类型最好调用其equal方法
23          Card c = (Card)o;
24          return rank == c.rank
25              && suit.equals(c.suit);
26      }
27  }
```

注意： 一般覆写 equals 的套路就是上面演示的

1. 如果指向同一个对象，返回 true
2. 如果传入的为 null，返回 false
3. 如果传入的对象类型不是 Card，返回 false
4. 按照类的实现目标完成比较，例如这里只要花色和数值一样，就认为是相同的牌
5. 注意下调用其他引用类型的比较也需要 equals，例如这里的 suit 的比较

覆写基类equal的方式虽然可以比较，但缺陷是：**equal只能按照相等进行比较，不能按照大于、小于的方式进行比较。**

3.2 基于Comparble接口类的比较

Comparble是JDK提供的泛型的比较接口类，源码实现具体如下：

```
1  public interface Comparable<E> {
2      // 返回值:
3      // < 0: 表示 this 指向的对象小于 o 指向的对象
4      // == 0: 表示 this 指向的对象等于 o 指向的对象
5      // > 0: 表示 this 指向的对象等于 o 指向的对象
6      int compareTo(E o);
7  }
```

对用用户自定义类型，如果要想按照大小与方式进行比较时：**在定义类时，实现Comparble接口即可，然后在类中重写compareTo方法。**

```
1  public class Card implements Comparable<Card> {
2      public int rank;    // 数值
3      public String suit; // 花色
4
5      public Card(int rank, String suit) {
6          this.rank = rank;
7          this.suit = suit;
8      }
9
10     // 根据数值比较，不管花色
11     // 这里我们认为 null 是最小的
12     @Override
```

```
    public int compareTo(Card o) {
        if (o == null) {
            return 1;
        }
        return rank - o.rank;
    }

    public static void main(String[] args){
        Card p = new Card(1, "♠");
        Card q = new Card(2, "♠");
        Card o = new Card(1, "♠");
        System.out.println(p.compareTo(o));     // == 0, 表示牌相等
        System.out.println(p.compareTo(q));// < 0, 表示 p 比较小
        System.out.println(q.compareTo(p));// > 0, 表示 q 比较大
    }
}
```

Compareble是java.lang中的接口类，可以直接使用。

3.3 基于比较器比较

按照比较器方式进行比较，具体步骤如下：

- **用户自定义比较器类，实现Comparator接口**

```
public interface Comparator<T> {
    // 返回值:
    // < 0: 表示 o1 指向的对象小于 o2 指向的对象
    // == 0: 表示 o1 指向的对象等于 o2 指向的对象
    // > 0: 表示 o1 指向的对象等于 o2 指向的对象
    int compare(T o1, T o2);
}
```

注意：**区分Comparable和Comparator。**

- **覆写Comparator中的compare方法**

```
import java.util.Comparator;

class Card {
    public int rank;    // 数值
    public String suit; // 花色

    public Card(int rank, String suit) {
        this.rank = rank;
        this.suit = suit;
    }
}

class CardComparator implements Comparator<Card> {
    // 根据数值比较，不管花色

    // 这里我们认为 null 是最小的
```

```
    @Override
    public int compare(Card o1, Card o2) {
        if (o1 == o2) {
            return 0;
        }

        if (o1 == null) {
            return -1;
        }

        if (o2 == null) {
            return 1;
        }

        return o1.rank - o2.rank;
    }

    public static void main(String[] args){
        Card p = new Card(1, "♠");
        Card q = new Card(2, "♠");
        Card o = new Card(1, "♠");

        // 定义比较器对象
        CardComparator cmptor = new CardComparator();

        // 使用比较器对象进行比较
        System.out.println(cmptor.compare(p, o));       // == 0, 表示牌相等
        System.out.println(cmptor.compare(p, q));       // < 0, 表示 p 比较小
        System.out.println(cmptor.compare(q, p));       // > 0, 表示 q 比较大
    }
}
```

注意：Comparator是java.util 包中的泛型接口类，使用时必须导入对应的包。

3.4 三种方式对比

覆写的方法	说明
Object.equals	因为所有类都是继承自 Object 的，所以直接覆写即可，不过只能比较相等与否
Comparable.compareTo	需要手动实现接口，侵入性比较强，但一旦实现，每次用该类都有顺序，属于内部顺序
Comparator.compare	需要实现一个比较器对象，对待比较类的侵入性弱，但对算法代码实现侵入性强

4. 集合框架中PriorityQueue的比较方式

集合框架中的PriorityQueue底层使用堆结构，因此其内部的元素必须要能够比大小，PriorityQueue采用了：Comparble和Comparator两种方式。

1. Comparble是默认的内部比较方式，如果用户插入自定义类型对象时，该类对象必须要实现Comparble接口，并覆写compareTo方法
2. 用户也可以选择使用比较器对象，如果用户插入自定义类型对象时，必须要提供一个比较器类，让该类实现Comparator接口并覆写compare方法。

```
// JDK中PriorityQueue的实现：
public class PriorityQueue<E> extends AbstractQueue<E>
    implements java.io.Serializable {

    // ...

    // 默认容量
    private static final int DEFAULT_INITIAL_CAPACITY = 11;

    // 内部定义的比较器对象，用来接收用户实例化PriorityQueue对象时提供的比较器对象
    private final Comparator<? super E> comparator;

    // 用户如果没有提供比较器对象，使用默认的内部比较，将comparator置为null
    public PriorityQueue() {
        this(DEFAULT_INITIAL_CAPACITY, null);
    }

    // 如果用户提供了比较器，采用用户提供的比较器进行比较
    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();
        this.queue = new Object[initialCapacity];
        this.comparator = comparator;
    }

    // ...
    // 向上调整：
    // 如果用户没有提供比较器对象，采用Comparable进行比较
    // 否则使用用户提供的比较器对象进行比较
    private void siftUp(int k, E x) {
        if (comparator != null)
            siftUpUsingComparator(k, x);
        else
            siftUpComparable(k, x);
    }

   // 使用Comparable
    @SuppressWarnings("unchecked")
    private void siftUpComparable(int k, E x) {
        Comparable<? super E> key = (Comparable<? super E>) x;
        while (k > 0) {
            int parent = (k - 1) >>> 1;
            Object e = queue[parent];
```

```
            if (key.compareTo((E) e) >= 0)
                break;
            queue[k] = e;
            k = parent;
        }
        queue[k] = key;
    }

    // 使用用户提供的比较器对象进行比较
    @SuppressWarnings("unchecked")
    private void siftUpUsingComparator(int k, E x) {
        while (k > 0) {
            int parent = (k - 1) >>> 1;
            Object e = queue[parent];
            if (comparator.compare(x, (E) e) >= 0)
                break;
            queue[k] = e;
            k = parent;
        }
        queue[k] = x;
    }

}
```

5. 模拟实现PriorityQueue

学生参考以下代码，自行模拟实现可以按照Comparble和比较器对象方式进行比较的通用PriorityQueue。

```
class LessIntComp implements Comparator<Integer>{
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1 - o2;
    }
}

class GreaterIntComp implements Comparator<Integer>{
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1;
    }
}

// 假设：创建的是小堆----泛型实现
public class MyPriorityQueue<E> {
    private Object[] hp;
    private int size = 0;
    private Comparator<? super E> comparator = null;

    // java8中：优先级队列的默认容量是11
    public MyPriorityQueue(Comparator<? super E> com) {
        hp = new Object[11];
        size = 0;
```

```
        comparator = com;
    }

    public MyPriorityQueue() {
        hp = new Object[11];
        size = 0;
        comparator = null;
    }

    // 按照指定容量设置大小
    public MyPriorityQueue(int capacity) {
        capacity = capacity < 1 ? 11 : capacity;
        hp = new Object[capacity];
        size = 0;
    }

    // 注意：没有此接口，给学生强调清楚
    // java8中：可以将一个集合中的元素直接放到优先级队列中
    public MyPriorityQueue(E[] array){
        // 将数组中的元素放到优先级队列底层的容器中
        hp = Arrays.copyOf(array, array.length);
        size = hp.length;
        // 对hp中的元素进行调整
        // 找到倒数第一个非叶子节点
        for(int root = ((size-2)>>1); root >= 0; root--){
            shiftDown(root);
        }
    }

    // 插入元素
    public void offer(E val){
        // 先检测是否需要扩容
        grow();

        // 将元素放在最后位置，然后向上调整
        hp[size] = val;
        size++;
        shiftUp(size-1);
    }

    // 删除元素：删除堆顶元素
    public void poll(){
        if(isEmpty()){
            return;
        }

        // 将堆顶元素与堆中最后一个元素进行交换
        swap((E[])hp, 0, size-1);

        // 删除最后一个元素
        size--;


        // 将堆顶元素向下调整
```

```
78          shiftDown(0);
79      }
80
81      public int size(){
82          return size;
83      }
84
85      public E peek(){
86          return (E)hp[0];
87      }
88
89      boolean isEmpty(){
90          return 0 == size;
91      }
92
93      // 向下调整
94      private void shiftDown(int parent){
95          if(null == comparator){
96              shiftDownWithcompareTo(parent);
97          }
98          else{
99              shiftDownWithComparetor(parent);
100         }
101     }
102
103     // 使用比较器比较
104     private void shiftDownWithComparetor(int parent){
105         // child作用：标记最小的孩子
106         // 因为堆是一个完全二叉树，而完全二叉树可能有左没有有
107         // 因此：默认情况下，让child标记左孩子
108         int child = parent * 2 + 1;
109
110         // while循环条件可以一直保证parent左孩子存在，但是不能保证parent的右孩子存在
111         while(child < size)
112         {
113             // 找parent的两个孩子中最小的孩子，用child进行标记
114             // 注意：parent的右孩子可能不存在
115             // 调用比较器来进行比较
116             if(child+1 < size && comparator.compare((E)hp[child+1], (E)hp[child]) < 0 ){
117                 child += 1;
118             }
119
120             // 如果双亲比较小的孩子还大，将双亲与较小的孩子交换
121             if(comparator.compare((E)hp[child], (E)hp[parent]) < 0) {
122                 swap((E[])hp, child, parent);
123
124                 // 小的元素往下移动，可能导致parent的子树不满足堆的性质
125                 // 因此：需要继续向下调整
126                 parent = child;
127                 child = child*2 + 1;
128             }
129             else{

130                 return;
```

```
131             }
132         }
133     }
134 
135     // 使用compareTo比较
136     private void shiftDownWithcompareTo(int parent){
137         // child作用：标记最小的孩子
138         // 因为堆是一个完全二叉树，而完全二叉树可能有左没有有
139         // 因此：默认情况下，让child标记左孩子
140         int child = parent * 2 + 1;
141 
142         // while循环条件可以一直保证parent左孩子存在，但是不能保证parent的右孩子存在
143         while(child < size)
144         {
145             // 找parent的两个孩子中最小的孩子，用child进行标记
146             // 注意：parent的右孩子可能不存在
147             // 向上转型，因为E的对象都实现了Comparable接口
148             if(child+1 < size && ((Comparable<? super E>)hp[child]).compareTo((E)hp[child])
    < 0){
149                 child += 1;
150             }
151 
152             // 如果双亲比较小的孩子还大，将双亲与较小的孩子交换
153             if(((Comparable<? super E>)hp[child]).compareTo((E)hp[parent]) < 0){
154                 swap((E[])hp, child, parent);
155 
156                 // 小的元素往下移动，可能导致parent的子树不满足堆的性质
157                 // 因此：需要继续向下调整
158                 parent = child;
159                 child = child*2 + 1;
160             }
161             else{
162                 return;
163             }
164         }
165     }
166 
167     // 向上调整
168     void shiftUp(int child){
169         if(null == comparator){
170             shiftUpWithCompareTo(child);
171         }
172         else{
173             shiftUpWithComparetor(child);
174         }
175     }
176 
177     void shiftUpWithComparetor(int child){
178         // 获取孩子节点的双亲
179         int parent = ((child-1)>>1);
180 
181         while(0 != child){
182             // 如果孩子比双亲还小，则不满足小堆的性质，交换
```

```
            if(comparator.compare((E)hp[child], (E)hp[parent]) < 0){
                swap((E[])hp, child, parent);
                child = parent;
                parent = ((child-1)>>1);
            }
            else{
                return;
            }
        }
    }

    void shiftUpWithCompareTo(int child){
        // 获取孩子节点的双亲
        int parent = ((child-1)>>1);

        while(0 != child){
            // 如果孩子比双亲还小，则不满足小堆的性质，交换
            if(((Comparable<? super E>)hp[child]).compareTo((E)hp[parent]) < 0){
                swap((E[])hp, child, parent);
                child = parent;
                parent = ((child-1)>>1);
            }
            else{
                return;
            }
        }
    }

    void swap(E[] hp, int i, int j){
        E temp = hp[i];
        hp[i] = hp[j];
        hp[j] = temp;
    }

    // 仿照JDK8中的扩容方式，注意还是有点点的区别，具体可以参考源代码
    void grow(){
        int oldCapacity = hp.length;
        if(size() >= oldCapacity){
            // Double size if small; else grow by 50%
            int newCapacity = oldCapacity + ((oldCapacity < 64) ?
                    (oldCapacity + 2) :
                    (oldCapacity >> 1));
            hp = Arrays.copyOf(hp, newCapacity);
        }
    }

    public static void main(String[] args) {
        int[] arr = {4,1,9,2,8,0,7,3,6,5};

        // 小堆---采用比较器创建小堆
        MyPriorityQueue<Integer> mq1 = new MyPriorityQueue(new LessIntComp());
        for(int e : arr){

            mq1.offer(e);
```

```
        }

        // 大堆---采用比较器创建大堆
        MyPriorityQueue<Integer> mq2 = new MyPriorityQueue(new GreaterIntComp());
        for(int e : arr){
            mq2.offer(e);
        }

        // 小堆--采用CompareTo比较创建小堆
        MyPriorityQueue<Integer> mq3 = new MyPriorityQueue();
        for(int e : arr){
            mq3.offer(e);
        }
    }
}
```