

java对象的比较

【本节目标】

1. Java中对象的比较
2. 集合框架中PriorityQueue的比较方式
3. 模拟实现PriorityQueue

1. PriorityQueue中插入对象

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。

```
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 对象比较的问题

```
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equals

```
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;
        }

        // o如果是null对象，或者o不是Card的子类
        if (o == null || !(o instanceof Card)) {
            return false;
        }

        // 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
        Card c = (Card)o;
        return rank == c.rank

        && suit.equals(c.suit);
    }
}
```

```
}  
}
```

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Comparable接口类的比较

Comparable是JDK提供的泛型的比较接口类，源码实现具体如下：

```
public interface Comparable<E> {  
    // 返回值:  
    // < 0: 表示 this 指向的对象小于 o 指向的对象  
    // == 0: 表示 this 指向的对象等于 o 指向的对象  
    // > 0: 表示 this 指向的对象大于 o 指向的对象  
    int compareTo(E o);  
}
```

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

```
public class Card implements Comparable<Card> {  
    public int rank; // 数值  
    public String suit; // 花色  
  
    public Card(int rank, String suit) {  
        this.rank = rank;  
        this.suit = suit;  
    }  
  
    // 根据数值比较，不管花色  
    // 这里我们认为 null 是最小的  
    @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
}
}

```

Comparable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 指向的对象大于 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：Comparable和Comparator两种方式。

1. Comparable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Comparable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. 使用PriorityQueue创建大小堆，解决TOPK问题

```

//使用比较器创建小根堆
class LessIntComp implements Comparator<Integer>{
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1 - o2;
    }
}

//使用比较器创建大根堆
class GreaterIntComp implements Comparator<Integer>{
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1;
    }
}

public class TestDemo<E> {

    //求最小的K个数，通过比较器创建大根堆
    public static int[] smallestK(int[] array, int k) {
        if(k <= 0) {
            return new int[k];
        }
        GreaterIntComp greaterCmp = new GreaterIntComp();
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(greaterCmp);
        //先将前K个元素，创建大根堆
        for(int i = 0; i < k; i++) {
            maxHeap.offer(array[i]);
        }
        //从第K+1个元素开始，每次和堆顶元素比较
        for (int i = k; i < array.length; i++) {
            int top = maxHeap.peek();
            if(array[i] < top) {
                maxHeap.poll();
                maxHeap.offer(array[i]);
            }
        }
    }

    //取出前K个

```



```
int[] ret = new int[k];
for (int i = 0; i < k; i++) {
    int val = maxHeap.poll();
    ret[i] = val;
}
return ret;
}

public static void main(String[] args) {
    int[] array = {4,1,9,2,8,0,7,3,6,5};
    int[] ret = smallestK(array,3);
    System.out.println(Arrays.toString(ret));
}
}
```