原来实现优先级队列如此简单

前言

假如你设计的事件系统中有很多的事件，每个事件都定义了不同的权重值，系统需要优先处理权重较高的事件，这里你就需要使用到优先级队列，本篇我们一起来学习实现优先级队列的常用方式

队列API定义

在实现之前，首先我们需要先定义出优先级队的API，优先级队列是一种抽象的数据结构，我们依然可以基于前面我们使用到的队列API来修改；需要了解之前的队列的实现可以查看《面试的季节到了，老哥确定不来复习下数据结构吗》

```
public interface Queue<T> extends Iterable<T> {
    void enqueue(T item); //入队列

    T dequeue(); //出队列

    int size();

    boolean isEmpty();
}
```

其中的入队列enqueue和出队列dequeue是我们主要需要实现的方式，也是优先级队列的核心方法

初级版本的实现

队列*API*的抽象类

```
public abstract class AbstractQueue<T> implements Queue<T> {
    private Comparator<T> comparator;

    public AbstractQueue(Comparator<T> comparator) {
        this.comparator = comparator;
    }

    public boolean less(T a, T b) {
        return comparator.compare(a, b) < 0;
    }
```

```
    public void exch(T[] array, int i, int j) {
        T tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
    }
}
```

基于无序数组实现

实现优先级队列的最简单实现可以参考《面试的季节到了，老哥确定不来复习下数据结构吗》中栈的实现方式，enqueue和栈的push方式实现方式一致，dequeue可以参考选择排序的实现，循环一遍数组，找出最大值和数组最后一个元素交换，然后删除它；

```
public class DisorderPriorityQueue<T> extends AbstractQueue<T> {

    private T[] queue;
    private int size;

    public DisorderPriorityQueue(int max, Comparator<T> comparator) {
        super(comparator);
        this.queue = (T[]) new Object[max];
    }

    @Override
    public void enqueue(T item) {
        queue[size++] = item;
    }

    @Override
    public T dequeue() {
        int index = 0;
        for (int i = 1; i < size; i++) {
            if (less(queue[index], queue[i])) {
                index = i;
            }
        }
        size--;
        exch(queue, index, size);
        T data = queue[size];
        queue[size] = null;
        return data;
    }
```

```
    //省略其他函数
}
```

这里只实现了定长的优先级队列，如何实现自动扩容呢？也可以参考这篇文章《面试的季节到了，老哥确定不来复习下数据结构吗》；基于无序数组实现的enqueue时间复杂度是O(1)，dequeue时间复杂度是O(n)

基于有序数组实现

基于有序数组实现就是在入队的时候保证数组有序，那么在出队列的时候可以直接删掉最大值；插入的过程和插入排序类似的操作

```
public class OrderPriorityQueue<T> extends AbstractQueue<T> {

    private T[] queue;
    private int size;

    public OrderPriorityQueue(int max, Comparator<T> comparator) {
        super(comparator);
        this.queue = (T[]) new Object[max];
    }

    @Override
    public void enqueue(T item) {
        queue[size++] = item;
        for (int i = size - 1; i > 1 && less(queue[i], queue[i - 1]); i--) {
            exch(queue, i, i - 1);
        }
    }

    @Override
    public T dequeue() {
        size--;
        T data = queue[size];
        queue[size] = null;
        return data;
    }

    //省略其他函数
}
```

enqueue时间复杂度是O(n)，dequeue时间复杂度是O(1)

基于链表实现

基于链表的实现与上面的类似，有兴趣的可以自己实现

在《面试的季节到了，老哥确定不来复习下数据结构吗》中我们实现的栈和队列的操作都能够在常数时间内完成，但是优先级队列从上面的实现过程，我们发现初级版本的实现插入或删除最大值的操作最糟糕的情况会是线性时间。

二叉堆实现

二叉堆的定义

在二叉堆中，每个节点都将大于等于它的子节点，也成为堆有序；其中根节点是最大的节点。

二叉堆的表示：

重点：
在一个二叉堆中，位置k节点的父节点的位置为k/2,它的两个子节点的位置为2k和2k+1；基于这点，我们可以用数组来表示二叉堆，通过移动数组的下标来找到节点父节点和子节点

在元素进行插入和删除操作的过程中，会破坏堆有序，所以我们需要做一些操作来保证堆再次有序；主要有两种情况，当某个节点的优先级上升，我们需要**由下向上恢复堆有序（下沉）**；当某个节点优先级下降，我们需要**由上向下恢复堆有序（上浮）**

由上向下恢复堆有序（上浮）

```
private void swim(int k) {
    while (k > 0 && less(queue[k / 2], queue[k])) {
        exch(queue, k / 2, k);
        k = k / 2;
    }
}
```

根据当前的节点k找到父节点的位置k/2，比较当前节点和父节点，如果比父节点大就交换，直到找个比当前节点大的父节点或者已上浮到了根节点

由下向上恢复堆有序（下沉）

```
private void sink(int k) {
    while (2 * k <= size) {
        int i = 2 * k;
        if (less(queue[i], queue[i + 1])) {
            i++;
        }
        if (less(queue[i], queue[k])) {
            break;
```

```
            }
            exch(queue, i, k);
            k = i;
        }
    }
```

二叉堆实现优先级队列

- 入队操作：将新的元素添加到数组末尾，让新元素上浮到适合位置，增加堆的大小
- 出队操作：将最大的根节点删除，然后把最后一个元素放入到顶端，下层顶端元素到合适位置，减小堆大小

```
public class BinaryHeapPriorityQueue<T> extends AbstractQueue<T> {
    private T[] queue;
    private int size;

    public BinaryHeapPriorityQueue(int max, Comparator<T> comparator) {
        super(comparator);
        this.queue = (T[]) new Object[max + 1];
    }

    @Override
    public void enqueue(T item) {
        this.queue[++size] = item;
        this.swim(size);
    }

    @Override
    public T dequeue() {
        T max = this.queue[1];
        exch(this.queue, 1, size--);
        this.queue[size + 1] = null; //释放内存
        this.sink(1);
        return max;
    }

     //省略其他函数
}
```

注意：
由于我们为了方便计算父节点和子节点的索引位置，所以数组中的第一个位置是不会使用的；可以自己思考下使用第一个位置，那么子节点和父节点的位置应该如何计算?

基于堆的实现，入队和出队的时间复杂对都是logN，解决了初级版本实现的问题。

数组大小动态扩容和缩容依然可以参考之前栈的实现方式

文中所有源码已放入到了github仓库

https://github.com/silently9527/JavaCore

最后（点关注，不迷路）

文中或许会存在或多或少的不足、错误之处，有建议或者意见也非常欢迎大家在评论交流。

最后，**写作不易，请不要白嫖我哟**，希望朋友们可以**点赞评论关注**三连，因为这些就是我分享的全部动力来源🙏