问题

（1）DelayQueue是阻塞队列吗?

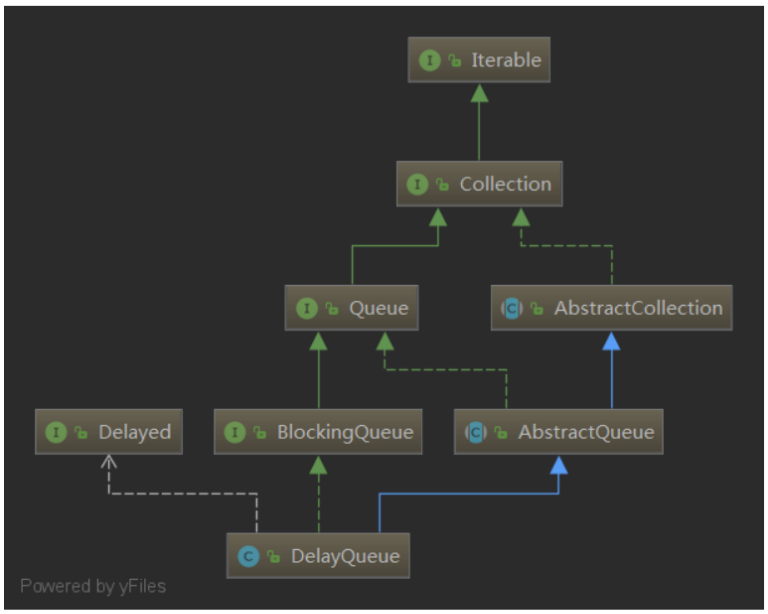
（2）DelayQueue的实现方式?

（3）DelayQueue主要用于什么场景?

简介

DelayQueue是java并发包下的延时阻塞队列，常用于实现定时任务。

继承体系



从继承体系可以看到，DelayQueue实现了BlockingQueue，所以它是一个阻塞队列。

另外，DelayQueue还组合了一个叫做Delayed的接口，DelayQueue中存储的所有元素必须实现Delayed接口。

那么，Delayed是什么呢?

```
1.  public interface Delayed extends Comparable<Delayed> {
2.
3.      long getDelay(TimeUnit unit);
4.  }
5.
```

Delayed是一个继承自Comparable的接口，并且定义了一个getDelay()方法，用于表示还有多少时间到期，到期了应返回小于等于0的数值。

源码分析

主要属性

```
1.  // 用于控制并发的锁
2.  private final transient ReentrantLock lock = new ReentrantLock();
3.  // 优先级队列
4.  private final PriorityQueue<E> q = new PriorityQueue<E>();
5.  // 用于标记当前是否有线程在排队（仅用于取元素时）
6.  private Thread leader = null;
7.  // 条件，用于表示现在是否有可取的元素
8.  private final Condition available = lock.newCondition();
9.
```

从属性我们可以知道，延时队列主要使用优先级队列来实现，并辅以重入锁和条件来控制并发安全。

因为优先级队列是无界的，所以这里只需要一个条件就可以了。

还记得优先级队列吗？点击链接直达【死磕 java集合之PriorityQueue源码分析】

主要构造方法

```
    public DelayQueue() {}

    public DelayQueue(Collection<? extends E> c) {
        this.addAll(c);
    }

```

构造方法比较简单，一个默认构造方法，一个初始化添加集合c中所有元素的构造方法。

入队

因为DelayQueue是阻塞队列，且优先级队列是无界的，所以入队不会阻塞不会超时，因此它的四个入队方法是一样的。

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

    public void put(E e) {
        offer(e);
    }

    public boolean offer(E e, long timeout, TimeUnit unit) {
        return offer(e);
    }

    public boolean offer(E e) {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            q.offer(e);
            if (q.peek() == e) {
                leader = null;
                available.signal();
            }
            return true;
        } finally {
            lock.unlock();
        }
    }

```

入队方法比较简单：

（1）加锁；

（2）添加元素到优先级队列中；

（3）如果添加的元素是堆顶元素，就把leader置为空，并唤醒等待在条件available上的线程；

（4）解锁；

出队

因为DelayQueue是阻塞队列，所以它的出队有四个不同的方法，有抛出异常的，有阻塞的，有不阻塞的，有超时的。

我们这里主要分析两个，poll()和take()方法。

```
    public E poll() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            E first = q.peek();
            if (first == null || first.getDelay(NANOSECONDS) > 0)
                return null;
            else
                return q.poll();
        } finally {
            lock.unlock();
        }
    }

```

poll()方法比较简单：

（1）加锁；

（2）检查第一个元素，如果为空或者还没到期，就返回null；

（3）如果第一个元素到期了就调用poll()弹出第一个元素；

（4）解锁。

```
    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            for (;;) {
                // 堆顶元素
                E first = q.peek();
                // 如果堆顶元素为空，说明队列中还没有元素，直接阻塞等待
                if (first == null)
                    available.await();
                else {
                    // 堆顶元素的到期时间
                    long delay = first.getDelay(NANOSECONDS);
                    // 如果小于0说明已到期，直接调用poll()方法弹出堆顶元素
                    if (delay <= 0)
                        return q.poll();

                    // 如果delay大于0 ，则下面要阻塞了

                    // 将first置为空方便gc，因为有可能其它元素弹出了这个元素
                    // 这里还持有着引用不会被清理
                    first = null; // don't retain ref while waiting
```

```
                    // 如果前面有其它线程在等待，直接进入等待
                    if (leader != null)
                        available.await();
                    else {
                        // 如果leader为null，把当前线程赋值给它
                        Thread thisThread = Thread.currentThread();
                        leader = thisThread;
                        try {
                            // 等待delay时间后自动醒过来
                            // 醒过来后把leader置空并重新进入循环判断堆顶元素是否到期
                            // 这里即使醒过来后也不一定能获取到元素
                            // 因为有可能其它线程先一步获取了锁并弹出了堆顶元素
                            // 条件锁的唤醒分成两步，先从Condition的队列里出队
                            // 再入队到AQS的队列中，当其它线程调用LockSupport.unpark(t)的时候才会真正唤醒
                            // 关于AQS我们后面会讲的^^
                            available.awaitNanos(delay);
                        } finally {
                            // 如果leader还是当前线程就把它置为空，让其它线程有机会获取元素
                            if (leader == thisThread)
                                leader = null;
                        }
                    }
                }
            }
        } finally {
            // 成功出队后，如果leader为空且堆顶还有元素，就唤醒下一个等待的线程
            if (leader == null && q.peek() != null)
                // signal()只是把等待的线程放到AQS的队列里面，并不是真正的唤醒
                available.signal();
            // 解锁，这才是真正的唤醒
            lock.unlock();
        }
    }

```

take()方法稍微要复杂一些：

（1）加锁；

（2）判断堆顶元素是否为空，为空的话直接阻塞等待；

（3）判断堆顶元素是否到期，到期了直接poll()出元素；

（4）没到期，再判断前面是否有其它线程在等待，有则直接等待；

（5）前面没有其它线程在等待，则把自己当作第一个线程等待delay时间后唤醒，再尝试获取元素；

（6）获取到元素之后再唤醒下一个等待的线程；

（7）解锁；

使用方法

说了那么多，是不是还是不知道怎么用呢？那怎么能行，请看下面的案例：

```
public class DelayQueueTest {
    public static void main(String[] args) {
        DelayQueue<Message> queue = new DelayQueue<>();

        long now = System.currentTimeMillis();

        // 启动一个线程从队列中取元素
        new Thread(()->{
```

```
                while (true) {
                    try {
                        // 将依次打印1000，2000，5000，7000，8000
                        System.out.println(queue.take().deadline - now);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }).start();

            // 添加5个元素到队列中
            queue.add(new Message(now + 5000));
            queue.add(new Message(now + 8000));
            queue.add(new Message(now + 2000));
            queue.add(new Message(now + 1000));
            queue.add(new Message(now + 7000));
        }
    }

    class Message implements Delayed {
        long deadline;

        public Message(long deadline) {
            this.deadline = deadline;
        }

        @Override
        public long getDelay(TimeUnit unit) {
            return deadline - System.currentTimeMillis();
        }

        @Override
        public int compareTo(Delayed o) {
            return (int) (getDelay(TimeUnit.MILLISECONDS) - o.getDelay(TimeUnit.MILLISECONDS));
        }

        @Override
        public String toString() {
            return String.valueOf(deadline);
        }
    }
```

是不是很简单，越早到期的元素越先出队。

总结

（1）DelayQueue是阻塞队列；

（2）DelayQueue内部存储结构使用优先级队列；

（3）DelayQueue使用重入锁和条件来控制并发安全；

（4）DelayQueue常用于定时任务；

彩蛋

java中的线程池实现定时任务是直接用的DelayQueue吗？

当然不是，ScheduledThreadPoolExecutor中使用的是它自己定义的内部类DelayedWorkQueue，其实里面的实现逻辑基本都是一样的，只不过DelayedWorkQueue里面没有使用现在的PriorityQueue，而是使用数组又实现了一遍优先级队列，本质上没有什么区别。