(76) 并发容器 - 各种队列 / 计算机程序的思维逻辑

原创 2017-02-27 老马 老马说编程

本节，我们来探讨Java并发包中的各种队列。Java并发包提供了丰富的队列类，可以简单分为：

- 无锁非阻塞并发队列：ConcurrentLinkedQueue和ConcurrentLinkedDeque
- 普通阻塞队列：基于数组的ArrayBlockingQueue，基于链表的LinkedBlockingQueue和LinkedBlockingDeque
- 优先级阻塞队列：PriorityBlockingQueue
- 延时阻塞队列：DelayQueue
- 其他阻塞队列：SynchronousQueue和LinkedTransferQueue

这些队列迭代都不会抛出ConcurrentModificationException，都是弱一致的，后面就不单独强调了。下面，我们来简要探讨每类队列的用途、用法和基本实现原理。

无锁非阻塞并发队列

有两个无锁非阻塞队列：ConcurrentLinkedQueue和ConcurrentLinkedDeque，它们适用于多个线程并发使用一个队列的场合，都是基于链表实现的，都没有限制大小，是无界的，与ConcurrentSkipListMap类似，它们的size方法不是一个常量运算，不过这个方法在并发应用中用处也不大。

ConcurrentLinkedQueue实现了Queue接口，表示一个先进先出的队列，从尾部入队，从头部出队，内部是一个单向链表。ConcurrentLinkedDeque实现了Deque接口，表示一个双端队列，在两端都可以入队和出队，内部是一个双向链表。它们的用法类似于LinkedList，我们就不赘述了。

这两个类最基础的原理是循环CAS，ConcurrentLinkedQueue的算法基于一篇论文："Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" (https://www.research.ibm.com/people/m/michael/podc-1996.pdf)，ConcurrentLinkedDeque扩展了ConcurrentLinkedQueue的技术，但它们的具体实现都非常复杂，我们就不探讨了。

普通阻塞队列

除了刚介绍的两个队列，其他队列都是阻塞队列，都实现了接口BlockingQueue，在入队/出队时可能等待，主要方法有：

```
//入队，如果队列满，等待直到队列有空间
void put(E e) throws InterruptedException;
//出队，如果队列空，等待直到队列不为空，返回头部元素
E take() throws InterruptedException;
//入队，如果队列满，最多等待指定的时间，如果超时还是满，返回false
boolean offer(E e, long timeout, TimeUnit unit) throws InterruptedException;
```

```
//出队，如果队列空，最多等待指定的时间，如果超时还是空，返回null
E poll(long timeout, TimeUnit unit) throws InterruptedException;
```

普通阻塞队列是常用的队列，常用于生产者/消费者模式。

ArrayBlockingQueue和LinkedBlockingQueue都是实现了Queue接口，表示先进先出的队列，尾部进，头部出，而LinkedBlockingDeque实现了Deque接口，是一个双端队列。

ArrayBlockingQueue是基于循环数组实现的，有界，创建时需要指定大小，且在运行过程中不会改变，这与我们在容器类中介绍的ArrayDeque是不同的，ArrayDeque也是基于循环数组实现的，但是是无界的，会自动扩展。

LinkedBlockingQueue是基于单向链表实现的，在创建时可以指定最大长度，也可以不指定，默认是无限的，节点都是动态创建的。LinkedBlockingDeque与LinkedBlockingQueue一样，最大长度也是在创建时可选的，默认无限，不过，它是基于双向链表实现的。

内部，它们都是使用显式锁ReentrantLock和显式条件Condition实现的。

ArrayBlockingQueue的实现很直接，有一个数组存储元素，有两个索引表示头和尾，有一个变量表示当前元素个数，有一个锁保护所有访问，有两个条件，"不满"和"不空"用于协作，成员声明如下：

```
final Object[] items;
int takeIndex; // 头
int putIndex; //尾
int count; //元素个数
final ReentrantLock lock;
private final Condition notEmpty;
private final Condition notFull;
```

实现思路与我们在72节实现的类似，就不赘述了。

与ArrayBlockingQueue类似，LinkedBlockingDeque也是使用一个锁和两个条件，使用锁保护所有操作，使用"不满"和"不空"两个条件，LinkedBlockingQueue稍微不同，因为它使用链表，且只从头部出队、从尾部入队，它做了一些优化，使用了两个锁，一个保护头部，一个保护尾部，每个锁关联一个条件。

优先级阻塞队列

普通阻塞队列是先进先出的，而优先级队列是按优先级出队的，优先级高的先出，我们在容器类中介绍过优先级队列PriorityQueue及其背后的数据结构堆。

PriorityBlockingQueue是PriorityQueue的并发版本，与PriorityQueue一样，它没有大小限制，是无界的，内部的数组大小会动态扩展，要求元素要么实现Comparable接口，要么创建PriorityBlockingQueue时提供一个Comparator对象。

与PriorityQueue的区别是，PriorityBlockingQueue实现了BlockingQueue接口，在队列为空时，take方法会阻塞等待。

另外，PriorityBlockingQueue是线程安全的，它的基本实现原理与PriorityQueue是一样的，也是基于堆，但它使用了一个锁ReentrantLock保护所有访问，使用了一个条件协调阻塞等待。

延时阻塞队列

延时阻塞队列DelayQueue是一种特殊的优先级队列，它也是无界的，它要求每个元素都实现Delayed接口，该接口的声明为：

```
public interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
```

Delayed扩展了Comparable接口，也就是说，DelayQueue的每个元素都是可比较的，它有一个额外方法getDelay返回一个给定时间单位unit的整数，表示再延迟多长时间，如果小于等于0，表示不再延迟。

DelayQueue也是优先级队列，它按元素的延时时间出队，它的特殊之处在于，只有当元素的延时过期之后才能被从队列中拿走，也就是说，take方法总是返回第一个过期的元素，如果没有，则阻塞等待。

DelayQueue可以用于实现定时任务，我们看段简单的示例代码：

```
public class DelayedQueueDemo {
    private static final AtomicLong taskSequencer = new AtomicLong(0);

    static class DelayedTask implements Delayed {
        private long runTime;
        private long sequence;
        private Runnable task;

        public DelayedTask(int delayedSeconds, Runnable task) {
            this.runTime = System.currentTimeMillis() + delayedSeconds * 1000;
            this.sequence = taskSequencer.getAndIncrement();
            this.task = task;
        }

        @Override
```

```
    public int compareTo(Delayed o) {
        if (o == this) {
            return 0;
        }
        if (o instanceof DelayedTask) {
            DelayedTask other = (DelayedTask) o;
            if (runTime < other.runTime) {
                return -1;
            } else if (runTime > other.runTime) {
                return 1;
            } else if (sequence < other.sequence) {
                return -1;
            } else {
                return 1;
            }
        }
        throw new IllegalArgumentException();
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return unit.convert(runTime - System.currentTimeMillis(),
                TimeUnit.MICROSECONDS);
    }

    public Runnable getTask() {
        return task;
    }
}

public static void main(String[] args) throws InterruptedException {
    DelayQueue<DelayedTask> tasks = new DelayQueue<>();
    tasks.put(new DelayedTask(2, new Runnable() {
        @Override
        public void run() {
            System.out.println("execute delayed task");
        }
    }));

    DelayedTask task = tasks.take();
    task.getTask().run();
```

```
    }
}
```

DelayedTask表示延时任务，只有延时过期后任务才会执行，任务按延时时间排序，延时一样的按照入队顺序排序。

内部，DelayQueue是基于PriorityQueue实现的，它使用一个锁ReentrantLock保护所有访问，使用一个条件available表示头部是否有元素，当头部元素的延时未到时，take操作会根据延时计算需睡眠的时间，然后睡眠，如果在此过程中有新的元素入队，且成为头部元素，则阻塞睡眠的线程会被提前唤醒然后重新检查。以上是基本思路，DelayQueue的实现有一些优化，以减少不必要的唤醒，具体我们就不探讨了。

其他阻塞队列

Java并发包中还有两个特殊的阻塞队列，SynchronousQueue和LinkedTransferQueue。

SynchronousQueue

SynchronousQueue与一般的队列不同，它不算一种真正的队列，它没有存储元素的空间，存储一个元素的空间都没有。它的入队操作要等待另一个线程的出队操作，反之亦然。如果没有其他线程在等待从队列中接收元素，put操作就会等待。take操作需要等待其他线程往队列中放元素，如果没有，也会等待。SynchronousQueue适用于两个线程之间直接传递信息、事件或任务。

LinkedTransferQueue

LinkedTransferQueue实现了TransferQueue接口，TransferQueue是BlockingQueue的子接口，但增加了一些额外功能，生产者在往队列中放元素时，可以等待消费者接收后再返回，适用于一些消息传递类型的应用中。TransferQueue的接口定义为：

```
public interface TransferQueue<E> extends BlockingQueue<E> {
    //如果有消费者在等待(执行take或限时的poll)，直接转给消费者，
    //返回true，否则返回false，不入队
    boolean tryTransfer(E e);
    //如果有消费者在等待，直接转给消费者，
    //否则入队，阻塞等待直到被消费者接收后再返回
    void transfer(E e) throws InterruptedException;
    //如果有消费者在等待，直接转给消费者，返回true
    //否则入队，阻塞等待限定的时间，如果最后被消费者接收，返回true
    boolean tryTransfer(E e, long timeout, TimeUnit unit)
        throws InterruptedException;
    //是否有消费者在等待
    boolean hasWaitingConsumer();
    //等待的消费者个数
    int getWaitingConsumerCount();
}
```

LinkedTransferQueue是基于链表实现的、无界的TransferQueue，具体实现比较复杂，我们就不探讨了。

小结

本节简要介绍了Java并发包中的各种队列，包括其基本概念和基本原理。

从73节到本节，我们介绍了Java并发包的各种容器，至此，就介绍完了，在实际开发中，应该尽量使用这些现成的容器，而非重新发明轮子。

Java并发包中还提供了一种方便的任务执行服务，使用它，可以将要执行的并发任务与线程的管理相分离，大大简化并发任务和线程的管理，让我们下一节来探讨。

(与其他章节一样，本节所有代码位于 https://github.com/swiftma/program-logic)

-- 长文连载，未完待续，敬请关注（长按下图二维码，或公众号搜索"老马说编程"）
