# 阻塞队列API

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 抛异常 | 返回null/false | 无限阻塞 | 定时阻塞 |
| 入队 | add(e) | offer(e) | put(e) | offer(e, time, unit) |
| 出队 | remove() | poll() | take() | poll(time, unit) |
| 检查 | element() | peek() |  |  |

重点关注：无限阻塞、定时阻塞两列。

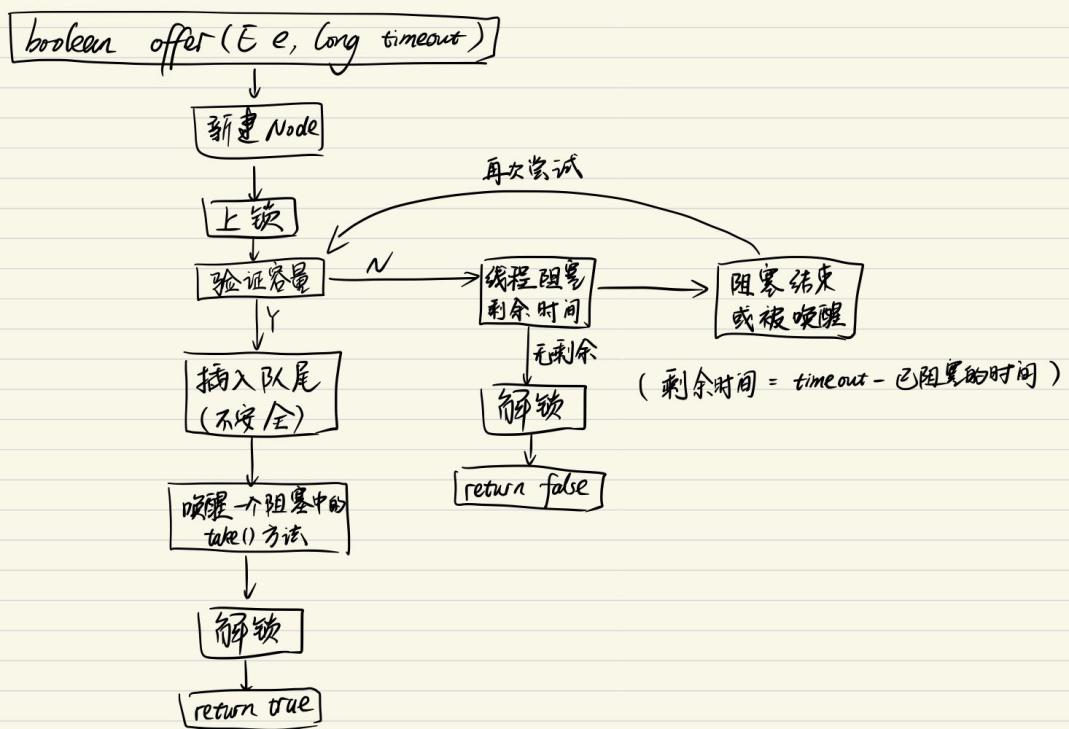
# LinkedBlockingDeque

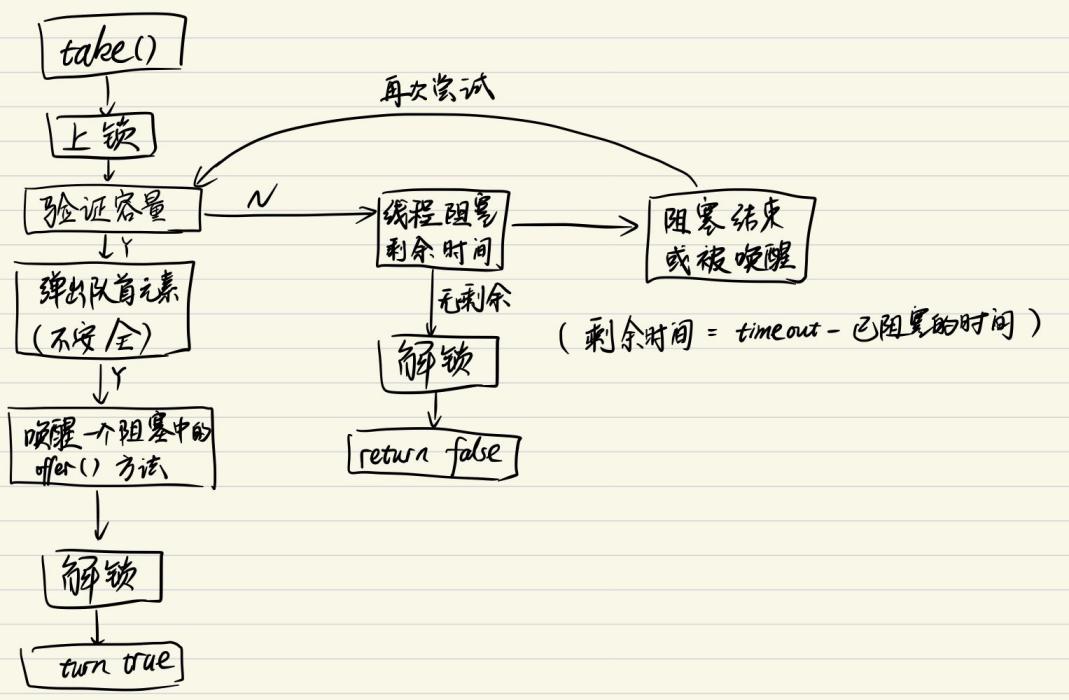
是一个双向链表实现的阻塞队列。

可以从普通的链表改进得到：

1. 一个普通的双向链表；
2. 为了实现**线程安全**，给入队、出队操作加锁。即，入队、出队无法同时执行；
3. 为了实现**限制容量**，每次入队前要检查一次容量；
4. 为了实现**定时阻塞**，使用Condition#await方法，可定时阻塞（实现原理：比较时间戳）；
5. 为了实现**互相唤醒**，一个锁对象分成“入队等待队列”和“出队等待队列”，入队结束后，唤醒一个出队等待队列中的线程；出队也一样。

所以，改进后的流程如下：





# LinkedBlockingQueue

## 原理

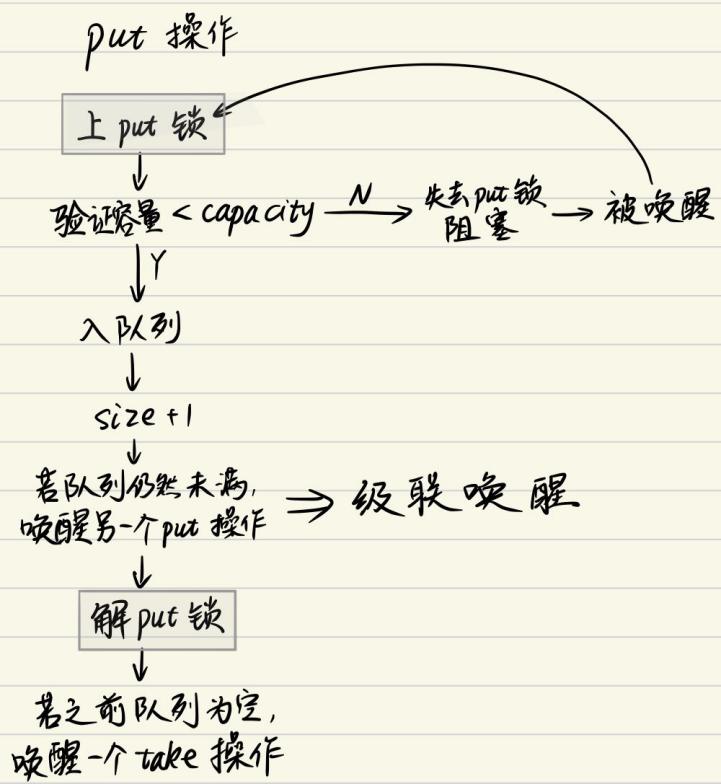
此结构是单向链表（上一个是双向链表），支持同时入队、出队，因此性能更高。

同样由普通链表改进得到：

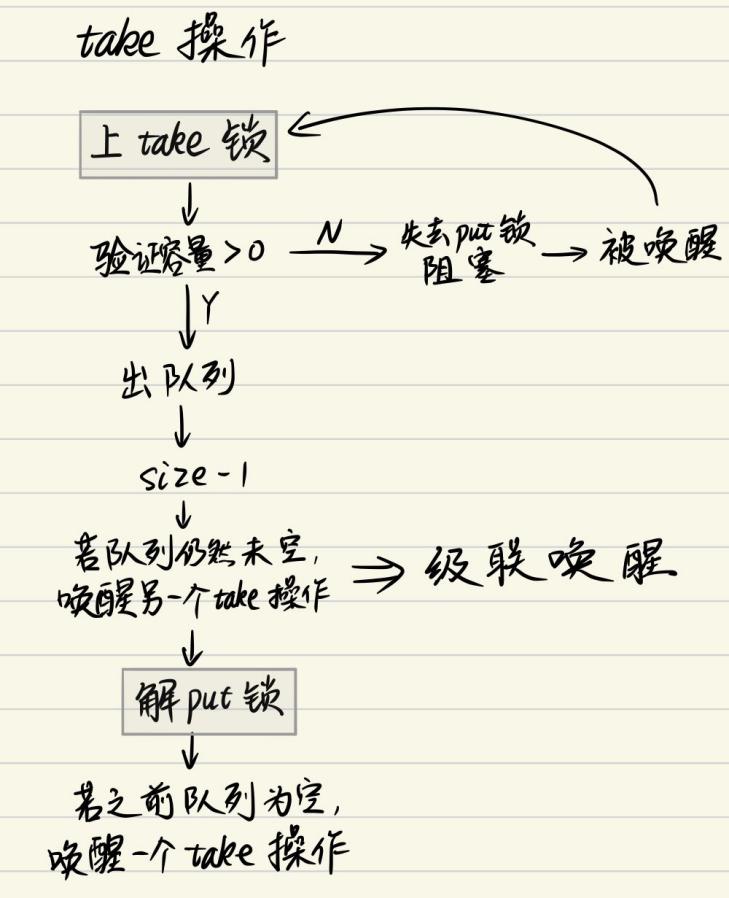
1. 一个普通的单向链表；
2. 为了实现**限制容量**，每次入队前要检查一次容量；
3. 为了**锁住同时入队或同时出队，但不锁住同时入队+出队**（最多同时1个入队+1个出队线程），使用双锁队列算法：队头锁+队尾锁；
4. 为了实现**定时阻塞**，队头锁、队尾锁分别有一个等待队列，并互相唤醒。

流程如下：

put操作：



Take操作：



## 思考

1. 为什么LinkedBlockingDeque中的元素个数使用普通int，而这里却使用AtomicInteger？

答：前者计数器没有线程安全问题，因为对计数器的操作都是在获取唯一锁后进行的。而这里入队、出队可能同时进行，即可能同时操作计数器，会有线程安全问题。

1. 级联唤醒（cascading notify）的意义在哪里？

答：不知道 TODO

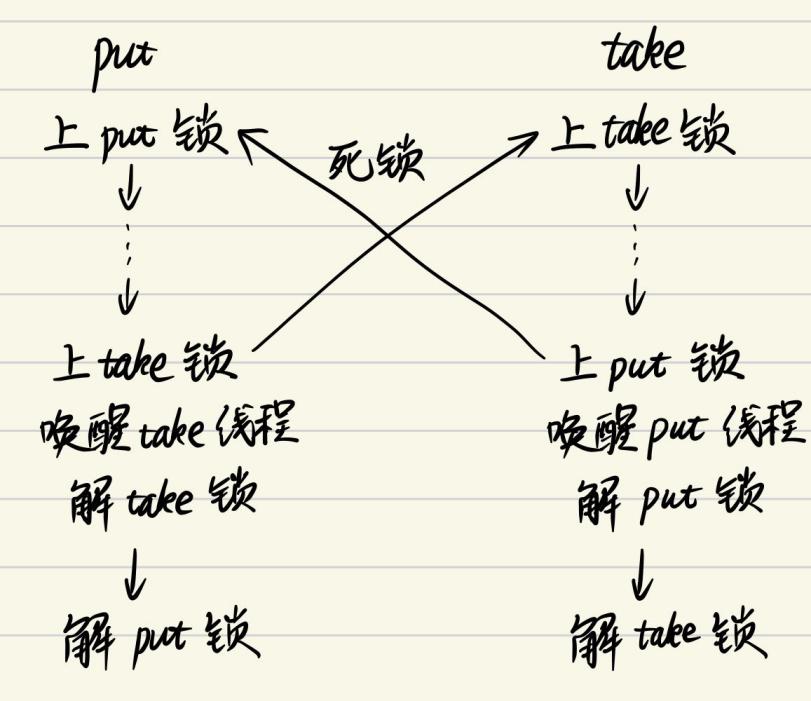
1. 容量为1的话，同时入队、出队也不会产生线程安全问题吗？

答：是的，若容量为1，无论当前元素个数是0还是1，入队、出队总要阻塞一个。所以此时不会同时入队、出队，也就没有线程安全问题。

1. 为什么LinkedBlockingDeque唤醒操作放在解锁前面，而这里却放在解锁后？

答：避免死锁。因为唤醒操作首先要获取到这个锁对象，才能调用唤醒方法，因此可能产生put线程拿着put锁等待take锁，take线程拿着take锁等待put锁的尴尬情况。

产生死锁的流程图：



参考文章：<https://www.cnblogs.com/zaizhoumo/p/7786793.html>

# DelayQueue

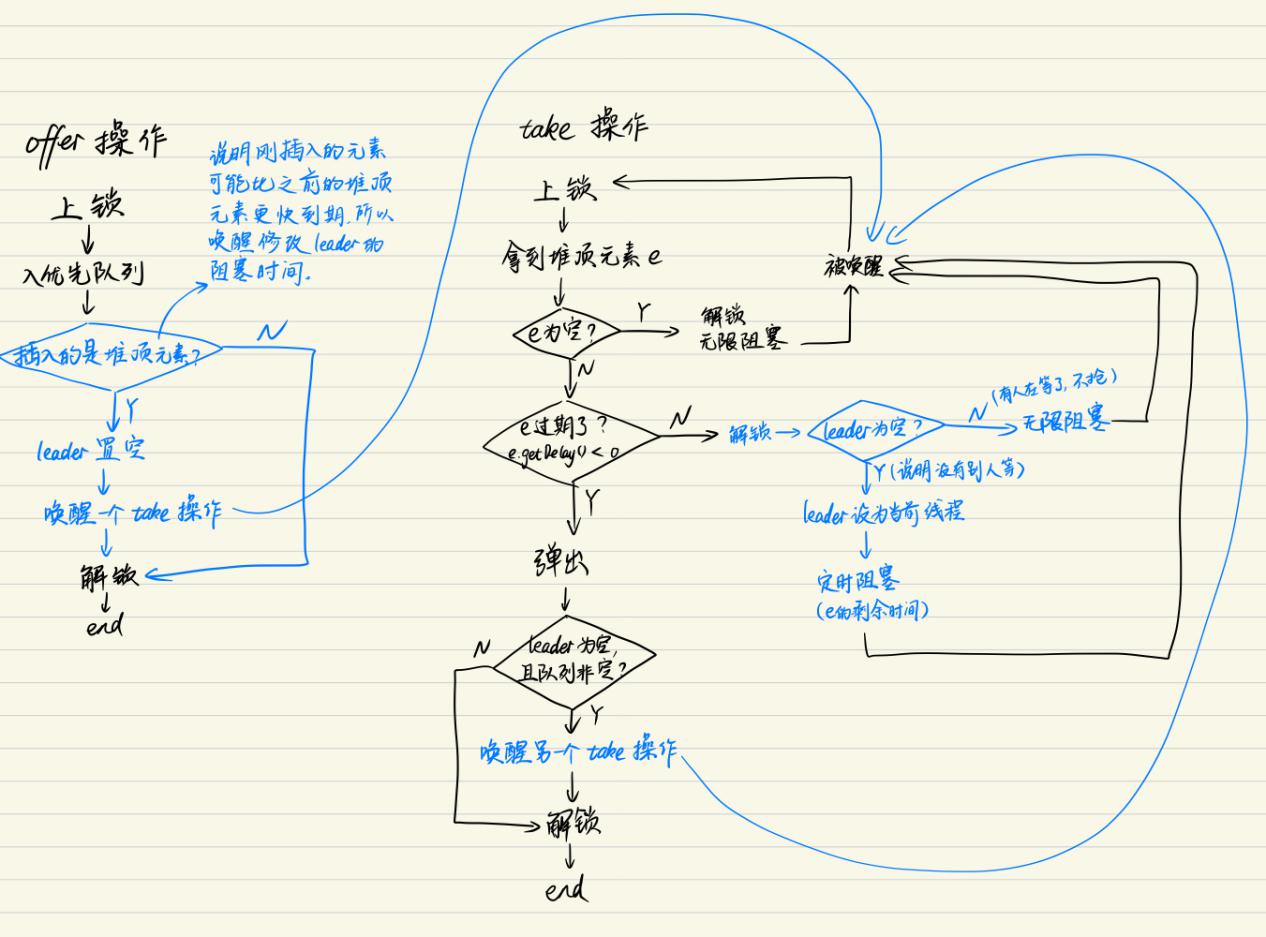
## 原理

一个使用优先队列实现的无界阻塞队列。

原理：

1. 一个普通的优先队列；
2. 为了保证**线程安全**，在所有操作前都上锁；
3. 为了实现**判断是否过期**，所有存入的对象必须提供“获取剩余过期时间”的方法（getDelayed()）；
4. 为了**防止多线程无意义的抢夺资源**，保存了一个等待中的线程对象leader（Leader-Follower线程模型），leader线程定时等待，其他线程均无限阻塞，由leader线程唤醒。

Offer和take操作流程如下：



注：offer和take的阻塞、相互唤醒逻辑，需要理清。

## 应用场景

任务的定时开始与关闭。

任务状态：未开始 -> 运行中 -> 已结束。

使用一个延迟队列，存储所有未开始的任务。用一个死循环去取将要运行的任务，取出后，放进运行中的延迟队列中。

对于从运行中转为结束状态，也一样。