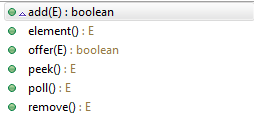
## Queue

**public** **interface** Queue<E> **extends** Collection<E>



**boolean** add(E e);

插入一个元素，如果容器大小允许，如果已满，抛出异常

**boolean** offer(E e);

插入一个元素，如果容器大小允许，如果已满，返回false

E remove();

获取队列的头元素然后删除他，如果队列为空，抛出异常

E poll();

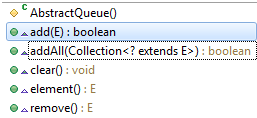
获取队列的头元素然后删除他，如果队列为空，返回空

E element();获取队列的头元素，但不删除，抛出异常如果队列为空

E peek();获取队列的头元素，但不删除，返回null如果队列为空

## AbstractQueue

**public** **abstract** **class** AbstractQueue<E> **extends** AbstractCollection<E> **implements** Queue<E>



提供了对add,element,remove的基本实现

**public** **boolean** add(E e) {

**if** (offer(e))

**return** **true**;

**else**

**throw** **new** IllegalStateException("Queue full");

}

**public** **boolean** addAll(Collection<? **extends** E> c) {

**if** (c == **null**)

**throw** **new** NullPointerException();

**if** (c == **this**)

**throw** **new** IllegalArgumentException();

**boolean** modified = **false**;

**for** (E e : c)

**if** (add(e))

modified = **true**;

**return** modified;

}

**public** **void** clear() {

**while** (poll() != **null**)

;

}

**public** E element() {

E x = peek();

**if** (x != **null**)

**return** x;

**else**

**throw** **new** NoSuchElementException();

}

**public** E remove() {

E x = poll();

**if** (x != **null**)

**return** x;

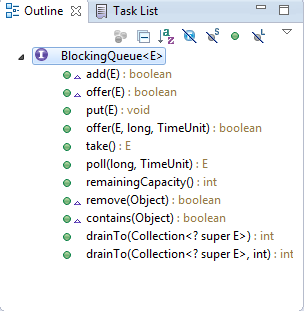
**else**

**throw** **new** NoSuchElementException();

}

## BlockingQueue

**public** **interface** BlockingQueue<E> **extends** Queue<E>



**void** put(E e) **throws** InterruptedException;

插入元素到队列，如果队列已满，则等待有空间插入，类似生产者

**boolean** offer(E e, **long** timeout, TimeUnit unit) **throws** InterruptedException;

插入元素，如果队列已满，等待指定时间插入，若在指定时间队列还是没有空间插入，则返回false

E take() **throws** InterruptedException;

获取队列的头元素然后删除，如果队列为空，则等待队列中有元素,类似消费者

E poll(**long** timeout, TimeUnit unit)

获取队列的头元素然后删除，如果队列为空，则等待指定时间，若在等待时间还是没有元素可用，则返回null

**int** remainingCapacity();

剩下的容量，可用于插入

**public** **boolean** contains(Object o);

**int** drainTo(Collection<? **super** E> c);

把队列当中所有的元素删除，转移到Collection c当中

**int** drainTo(Collection<? **super** E> c, **int** maxElements);

类似上面，限定了个数

## ArrayBlockingQueue

**public** **class** ArrayBlockingQueue<E> **extends** AbstractQueue<E> **implements** BlockingQueue<E>, java.io.Serializable

基于数组的有界的阻塞队列，支持先进先出，队列的头元素是呆在队列里最长的，队尾的元素是呆在队列里最短的，元素从头取出，从尾插入;

队列容量一旦设定不能更改，生产者消费者模式可以用这个，插入的时候队列满了要阻塞等待，取的时候队列空了也阻塞等待;

线程安全的类

/\*\* The queued items \*/

**final** Object[] items;

/\*\* items index for next take, poll, peek or remove \*/

**int** takeIndex;

/\*\* items index for next put, offer, or add \*/

**int** putIndex;

/\*\* Number of elements in the queue \*/

**int** count;

/\*\* Main lock guarding all access \*/

**final** ReentrantLock lock;

/\*\* Condition for waiting takes \*/

**private** **final** Condition notEmpty;

/\*\* Condition for waiting puts \*/

**private** **final** Condition notFull;

**private** **void** insert(E x) {

items[putIndex] = x; putIndex初始值为0

putIndex = inc(putIndex); 插入后,putIndex加1，但不支持线程安全,但这是私有方法，不会被外界调用

++count; 队列的元素个数

notEmpty.signal(); 队列元素非空信号，可以取了

}

**private** E extract() {

**final** Object[] items = **this**.items;

E x = **this**.<E>*cast*(items[takeIndex]); takeIndex初始值0

items[takeIndex] = **null**; 取出的元素的引用设置指向空，可以被回收

takeIndex = inc(takeIndex); takeIndex加1

--count; 队列元素减1

notFull.signal(); 队列元素不满信号，可以插入了

**return** x;

}

**void** removeAt(**int** i) {

**final** Object[] items = **this**.items;

// if removing front item, just advance

**if** (i == takeIndex) {

items[takeIndex] = **null**;

takeIndex = inc(takeIndex);

} **else** {

// slide over all others up through putIndex.

**for** (;;) {

**int** nexti = inc(i);

**if** (nexti != putIndex) {

items[i] = items[nexti];

i = nexti;

} **else** {

items[i] = **null**; 这里的i应该是最后一个元素的位置

putIndex = i; putIndex指向下一个插入元素的位置

**break**;

}

}

}

--count;

notFull.signal();

}

Deletes item at position i. 原理是items[i]=items[i+1]，item[i]指向下一个元素，这样就相当于把第i个元素删除了，然后把最后一个元素指向空

**public** **boolean** offer(E e) {

*checkNotNull*(e);

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**if** (count == items.length) 队列已满

**return** **false**;

**else** {

insert(e); insert有具体实现

**return** **true**;

}

} **finally** {

lock.unlock();

}

}

**public** **void** put(E e) **throws** InterruptedException {

*checkNotNull*(e);

**final** ReentrantLock lock = **this**.lock;

lock.lockInterruptibly();

**try** {

**while** (count == items.length) 队列已满

notFull.await(); 等待

insert(e); 队列不满，则插入

} **finally** {

lock.unlock();

}

}

**public** **boolean** offer(E e, **long** timeout, TimeUnit unit)

**throws** InterruptedException {

*checkNotNull*(e);

**long** nanos = unit.toNanos(timeout);

**final** ReentrantLock lock = **this**.lock;

lock.lockInterruptibly();

**try** {

**while** (count == items.length) {

**if** (nanos <= 0)

**return** **false**;

nanos = notFull.awaitNanos(nanos); 如果等待的时间全部用光了，则返回0或小于0,则返回false,不会执行下面的插入

}

insert(e);

**return** **true**;

} **finally** {

lock.unlock();

}

}

**public** E poll() {

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**return** (count == 0) ? **null** : extract(); extract() 有具体实现

} **finally** {

lock.unlock();

}

}

**public** E take() **throws** InterruptedException {

**final** ReentrantLock lock = **this**.lock;

lock.lockInterruptibly();

**try** {

**while** (count == 0) 队列为空，一直等待

notEmpty.await();

**return** extract(); 提取元素

} **finally** {

lock.unlock();

}

}

**public** E poll(**long** timeout, TimeUnit unit) **throws** InterruptedException {

**long** nanos = unit.toNanos(timeout);

**final** ReentrantLock lock = **this**.lock;

lock.lockInterruptibly();

**try** {

**while** (count == 0) { 同offer方法

**if** (nanos <= 0)

**return** **null**;

nanos = notEmpty.awaitNanos(nanos);

}

**return** extract();

} **finally** {

lock.unlock();

}

}

**public** E peek() {

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**return** (count == 0) ? **null** : itemAt(takeIndex);

} **finally** {

lock.unlock();

}

}

**public** **int** size() {

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**return** count;

} **finally** {

lock.unlock();

}

}

**public** **boolean** remove(Object o) {

**if** (o == **null**) **return** **false**;

**final** Object[] items = **this**.items;

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**for** (**int** i = takeIndex, k = count; k > 0; i = inc(i), k--) {

**if** (o.equals(items[i])) {

removeAt(i); removeAt会进行数组元素的位置移动，被删除对象o所在位置的后面元素会往前移动一位

**return** **true**;

}

}

**return** **false**;

} **finally** {

lock.unlock();

}

}

**public** **boolean** contains(Object o) {

**if** (o == **null**) **return** **false**;

**final** Object[] items = **this**.items;

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**for** (**int** i = takeIndex, k = count; k > 0; i = inc(i), k--)

**if** (o.equals(items[i]))

**return** **true**;

**return** **false**;

} **finally** {

lock.unlock();

}

}

@SuppressWarnings("unchecked")

**public** <T> T[] toArray(T[] a) {

**final** Object[] items = **this**.items;

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**final** **int** count = **this**.count;

**final** **int** len = a.length;

**if** (len < count) 形参数组a的长度小于队列元素的长度，重新声明一个数组，长度为count

a = (T[])java.lang.reflect.Array.*newInstance*(

a.getClass().getComponentType(), count);

**for** (**int** i = takeIndex, k = 0; k < count; i = inc(i), k++)

a[k] = (T) items[i]; 这里的复制完成后，两分数据就独立了，个性任何一份数据都不会影响到对方

**if** (len > count) 如果形参数组a的长度大于队列元素，表明数组太长了，count位置后面的元素不需要了，设置为空，提示GC回收

a[count] = **null**;

**return** a;

} **finally** {

lock.unlock();

}

}

**public** **void** clear() {

**final** Object[] items = **this**.items;

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**for** (**int** i = takeIndex, k = count; k > 0; i = inc(i), k--)

items[i] = **null**; 这里是关键

count = 0;

putIndex = 0;

takeIndex = 0;

notFull.signalAll();

} **finally** {

lock.unlock();

}

}

**public** Iterator<E> iterator() {

**return** **new** Itr();

}

**private** **class** Itr **implements** Iterator<E> {

**private** **int** remaining; // Number of elements yet to be returned

**private** **int** nextIndex; // Index of element to be returned by next

**private** E nextItem; // Element to be returned by next call to next

**private** E lastItem; // Element returned by last call to next

**private** **int** lastRet; // Index of last element returned, or -1 if none

Itr() {

**final** ReentrantLock lock = ArrayBlockingQueue.**this**.lock;

lock.lock();

**try** {

lastRet = -1;

**if** ((remaining = count) > 0)

nextItem = itemAt(nextIndex = takeIndex);

} **finally** {

lock.unlock();

}

}

**public** **boolean** hasNext() {

**return** remaining > 0;

}

**public** E next() {

**final** ReentrantLock lock = ArrayBlockingQueue.**this**.lock;

lock.lock();

**try** {

**if** (remaining <= 0)

**throw** **new** NoSuchElementException();

lastRet = nextIndex;

E x = itemAt(nextIndex); // check for fresher value

**if** (x == **null**) {

x = nextItem; // we are forced to report old value

lastItem = **null**; // but ensure remove fails

}

**else**

lastItem = x;

**while** (--remaining > 0 && // skip over nulls

(nextItem = itemAt(nextIndex = inc(nextIndex))) == **null**)

;

**return** x;

} **finally** {

lock.unlock();

}

}

**public** **void** remove() {

**final** ReentrantLock lock = ArrayBlockingQueue.**this**.lock;

lock.lock();

**try** {

**int** i = lastRet;

**if** (i == -1)

**throw** **new** IllegalStateException();

lastRet = -1;

E x = lastItem;

lastItem = **null**;

// only remove if item still at index

**if** (x != **null** && x == items[i]) {

**boolean** removingHead = (i == takeIndex);

removeAt(i);

**if** (!removingHead)

nextIndex = dec(nextIndex);

}

} **finally** {

lock.unlock();

}

}

}

## ConcurrentLinkedQueue

**public** **class** ConcurrentLinkedQueue<E> **extends** AbstractQueue<E>

**implements** Queue<E>, java.io.Serializable

基于列接节点的无边界的线程安全的队列;

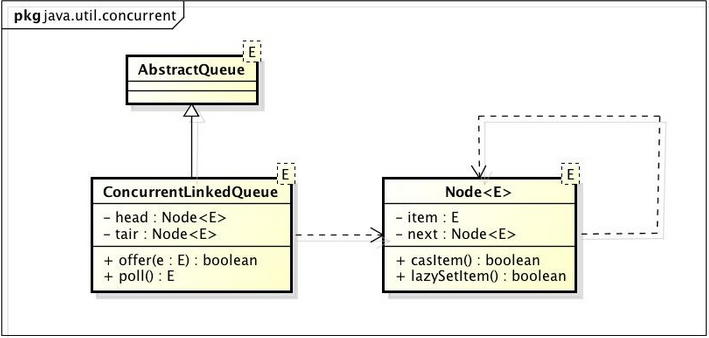
支持先进先出，队列的头元素是呆在队列里最长的，队尾的元素是呆在队列里最短的，元素从头取出，从尾插入;

ConcurrentLinkedQueue是一个好的选择当多个线程共同访问一个公共的集合;

不允许使用null元素;

Iterator是弱一致性的，不会扫出ConcurrentModificationException，返回的元素是某一时刻的状态;

Size()方法的执行时间不是常量的，调用该方法会去遍历该队列中的所有元素，结果不一定会准确，addAll(),removeAll(),retainAll(),containsAll(),equals(),toArray()都不保证是原子性操作的



**private** **static** **class** Node<E> {

**volatile** E item;

**volatile** Node<E> next;

/\*\*

\* Constructs a new node. Uses relaxed write because item can

\* only be seen after publication via casNext.

\*/

Node(E item) {

*UNSAFE*.putObject(**this**, *itemOffset*, item);

}

**boolean** casItem(E cmp, E val) {

**return** *UNSAFE*.compareAndSwapObject(**this**, *itemOffset*, cmp, val); Node偏移itemOffset的值如果是cmp的话(也即item变量的值为cmp)，就更新为val

}

**void** lazySetNext(Node<E> val) {

*UNSAFE*.putOrderedObject(**this**, *nextOffset*, val);

}

**boolean** casNext(Node<E> cmp, Node<E> val) {

**return** *UNSAFE*.compareAndSwapObject(**this**, *nextOffset*, cmp, val); Node偏移itemOffset的值如果是cmp的话(也即next变量的值为cmp)，就更新为val

}

// Unsafe mechanics

**private** **static** **final** sun.misc.Unsafe *UNSAFE*;

**private** **static** **final** **long** *itemOffset*;

**private** **static** **final** **long** *nextOffset*;

**static** {

**try** {

*UNSAFE* = sun.misc.Unsafe.*getUnsafe*();

Class k = Node.**class**;

*itemOffset* = *UNSAFE*.objectFieldOffset

(k.getDeclaredField("item"));

*nextOffset* = *UNSAFE*.objectFieldOffset

(k.getDeclaredField("next"));

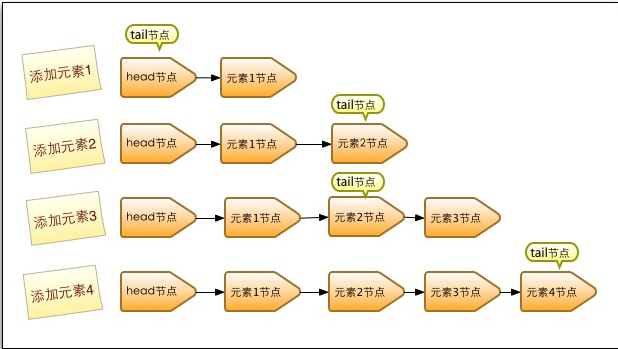
} **catch** (Exception e) {

**throw** **new** Error(e);

}

}

}



入队列主要做两件事

1. 将入队节点设置成当前队列尾节点的下一个节点;
2. 第二是更新尾节点，如果尾节点的下一个节点不为空，则将入队节点设置成尾节点，如果尾节点的下一个节点为空，则将入队节点设置成尾节点的下一个节点;

具体步骤

1. 一开始，head和tail都指向一个null节点
2. 增加节点元素1后

Head节点指向新生成的节点，节点的item为节点1的值，next节点为空;

Tail节点还是指向原来的节点，item为空，next为空;

1. 增加节点元素2后

Head节点的next节点指向节点2，Tail节点指向节点2, item为节点2的值，next节点为空

1. 增加节点元素3后

Head节点保持不变，Tail节点唯一的变化是next节点指向新生成的节点

1. 增加节点元素4后

Head节点保持不变，但Tail节点指向了新生成的节点，tail节点的next节点为空

出队列:

1. 如果head节点为空(从上面的入队情况来看，不太可能，除非有多线程操作)，弹出节点1，然后更新head节点指向节点2，因为这时节点在最前面;
2. 如果head节点不为空，直接弹出head节点，然后head节点指向下一个节点，再出队，再弹出一个节点，如果是最后一个节点，则head和tail都指向null节点;
3. 队列中不止一个元素，Remove(object)，如果object是最后一个节点，head保持不变，tail指向null;
4. 没有其他特殊情况了

## LinkedBlockingDeque

**public** **class** LinkedBlockingDeque<E> **extends** AbstractQueue<E> **implements** BlockingDeque<E>, java.io.Serializable

基于连接节点的可选 边界的阻塞双边队列；

可选边界是指你可以指定容量，默认是Integer.MAX；

双边队列是指头尾都可以入队列出队列；

很多操作运行都是常量时间，除了remove, removeFirstOccurance, removeLastOccurance, contains, iterator.remove, bulk operation, 他们是线性的

因为是阻塞的，所以入队列满了会等待，出队列空了会等待；takeFirst(), takeLast(),putFirst(), putLast()

线程安全的，每个操作都会先获取锁；

## LinkedBlockingQueue

跟LinkedBlockingDeque相似一个是双边一个是单边

## LinkedTransferQueue

## SynchronousQueue

**public** **class** SynchronousQueue<E> **extends** AbstractQueue<E> **implements** BlockingQueue<E>, java.io.Serializable

SynchronousQueue是这样一种阻塞队列，其中每个 put 必须等待一个 take，反之亦然。同步队列没有任何内部容量，甚至连一个队列的容量都没有。   
     不能在同步队列上进行 peek，因为仅在试图要取得元素时，该元素才存在；   
     除非另一个线程试图移除某个元素，否则也不能（使用任何方法）添加元素;也不能迭代队列，因为其中没有元素可用于迭代。队列的头是尝试添加到队列中的首个已排队线程元素； 如果没有已排队线程，则不添加元素并且头为 null。   
     对于其他 Collection 方法（例如 contains），SynchronousQueue 作为一个空集合。此队列不允许 null 元素。  
    它非常适合于传递性设计，在这种设计中，在一个线程中运行的对象要将某些信息、   
事件或任务传递给在另一个线程中运行的对象，它就必须与该对象同步。   
    对于正在等待的生产者和使用者线程而言，此类支持可选的公平排序策略。默认情况下不保证这种排序。   
    但是，使用公平设置为 true 所构造的队列可保证线程以 FIFO 的顺序进行访问。 公平通常会降低吞吐量，但是可以减小可变性并避免得不到服务。   
    注意1：它一种阻塞队列，其中每个 put 必须等待一个 take，反之亦然。同步队列没有任何内部容量，甚至连一个队列的容量都没有。   
    注意2：它是线程安全的，是阻塞的。   
    注意3:不允许使用 null 元素。   
    注意4：公平排序策略是指调用put的线程之间，或take的线程之间。公平排序策略可以查考ArrayBlockingQueue中的公平策略。   
    注意5:SynchronousQueue的以下方法：   
    \* iterator() 永远返回空，因为里面没东西。   
    \* peek() 永远返回null。   
    \* put() 往queue放进去一个element以后就一直wait直到有其他thread进来把这个element取走。   
    \* offer() 往queue里放一个element后立即返回，如果碰巧这个element被另一个thread取走了，offer方法返回true，认为offer成功；否则返回false。   
    \* offer(2000, TimeUnit.SECONDS) 往queue里放一个element但是等待指定的时间后才返回，返回的逻辑和offer()方法一样。   
    \* take() 取出并且remove掉queue里的element（认为是在queue里的。。。），取不到东西他会一直等。   
    \* poll() 取出并且remove掉queue里的element（认为是在queue里的。。。），只有到碰巧另外一个线程正在往queue里offer数据或者put数据的时候，该方法才会取到东西。否则立即返回null。   
    \* poll(2000, TimeUnit.SECONDS) 等待指定的时间然后取出并且remove掉queue里的element,其实就是再等其他的thread来往里塞。   
    \* isEmpty()永远是true。   
    \* remainingCapacity() 永远是0。   
    \* remove()和removeAll() 永远是false。

这是一个很有意思的阻塞队列，其中每个插入操作必须等待另一个线程的移除操作，同样任何一个移除操作都等待另一个线程的插入操作。因此此队列内部其 实没有任何一个元素，或者说容量是0，严格说并不是一种容器。由于队列没有容量，因此不能调用peek操作，因为只有移除元素时才有元素。

一个没有容量的并发队列有什么用了？或者说存在的意义是什么？

SynchronousQueue 的实现非常复杂，当然了如果真要去分析还是能够得到一些经验的，但是前面分析了过多的结构后，发现越来越陷于数据结构与算法里面了。我的初衷是通过研究并 发实现的原理来更好的利用并发来最大限度的利用可用资源。所以在后面的章节中尽可能的少研究数据结构和算法，但是为了弄清楚里面的原理，必不可免的会涉及 到一些这方面的知识，希望后面能够适可而止。

再回到话题。SynchronousQueue 内部没有容量，但是由于一个插入操作总是对应一个移除操作，反过来同样需要满足。那么一个元素就不会再SynchronousQueue 里面长时间停留，一旦有了插入线程和移除线程，元素很快就从插入线程移交给移除线程。也就是说这更像是一种信道（管道），资源从一个方向快速传递到另一方 向。

需要特别说明的是，尽管元素在SynchronousQueue 内部不会“停留”，但是并不意味之SynchronousQueue 内部没有队列。实际上SynchronousQueue 维护者线程队列，也就是插入线程或者移除线程在不同时存在的时候就会有线程队列。既然有队列，同样就有公平性和非公平性特性，公平性保证正在等待的插入线 程或者移除线程以FIFO的顺序传递资源。

显然这是一种快速传递元素的方式，也就是说在这种情况下元素总是以最快的方式从插入着（生产者）传递给移除着（消费者），这在多任务队列中是最快处理任务的方式

## DelayQueue

**public** **class** DelayQueue<E **extends** Delayed> **extends** AbstractQueue<E> **implements** BlockingQueue<E>

无边界的阻塞队列，元素到期了才能被获取,出队列，入队列的时候没有特殊，不允许空

队列的元素必须继承了Delayed接口,take()方法和poll()延时的方法比较特殊

DelayedQueue中处理的是实现Delayed接口的任务，DelayedQueue使用lock来实现线程同步，使用PriorityQueue来管理任务;

Delayed接口有两个重要的方法compareTo和getDelay，第一个是比较两个任务的延迟时间进行排序，第二个方法用来获取延迟时间。

priorityQueue是一种优先级队列，这里优先级就是延迟时间，也就是说进入队列的任务安照优先级进行排序，延迟时间最短的在队列前面，先被处理，也就是说，每次从队列中取出的任务都将是最近要到期的任务

**public** E poll() {

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

E first = q.peek();

如果first为空，或者还没有到期，返回null

**if** (first == **null** || first.getDelay(TimeUnit.*NANOSECONDS*) > 0)

**return** **null**;

**else**

**return** q.poll();

} **finally** {

lock.unlock();

}

}

**public** E take() **throws** InterruptedException {

**final** ReentrantLock lock = **this**.lock;

lock.lockInterruptibly();

**try** {

**for** (;;) {

E first = q.peek(); 通过PriorityQueue取出队列

**if** (first == **null**)

available.await(); 为空的话等待

**else** {

**long** delay = first.getDelay(TimeUnit.*NANOSECONDS*);

**if** (delay <= 0) 如果到期了，则取出

**return** q.poll();

**else** **if** (leader != **null**) 如果延时还没到期，判断leader线程是否为空，不为空说明其他线程是leader,等待被其他线程唤醒，只有leader线程才会等待到第一个元素过期，并尝试返回这个元素，

available.await();

**else** { 如果当前没有leader线程，当前线程成为leader线程，等待元素过期，

Thread thisThread = Thread.*currentThread*();

leader = thisThread;

**try** {

available.awaitNanos(delay);等待，过期之后会被唤醒

} **finally** {

**if** (leader == thisThread)

leader = **null**; 元素过期之后，leader线程置空

}

}

}

}

} **finally** {

**if** (leader == **null** && q.peek() != **null**) 如果leader为空而且有元素可以出队列，则激活所有等待的线程

available.signal();

lock.unlock();

}

}

**public** E poll(**long** timeout, TimeUnit unit) **throws** InterruptedException {

**long** nanos = unit.toNanos(timeout);

**final** ReentrantLock lock = **this**.lock;

lock.lockInterruptibly();

**try** {

**for** (;;) {

E first = q.peek();

**if** (first == **null**) {

**if** (nanos <= 0)

**return** **null**;

**else**

nanos = available.awaitNanos(nanos);

} **else** {

**long** delay = first.getDelay(TimeUnit.*NANOSECONDS*);

**if** (delay <= 0)

**return** q.poll();

**if** (nanos <= 0)

**return** **null**;

**if** (nanos < delay || leader != **null**)

nanos = available.awaitNanos(nanos);

**else** {

Thread thisThread = Thread.*currentThread*();

leader = thisThread;

**try** {

**long** timeLeft = available.awaitNanos(delay);

nanos -= delay - timeLeft;

} **finally** {

**if** (leader == thisThread)

leader = **null**;

}

}

}

}

} **finally** {

**if** (leader == **null** && q.peek() != **null**)

available.signal();

lock.unlock();

}

}

## PriorityQueue

**public** **class** PriorityQueue<E> **extends** AbstractQueue<E> **implements** java.io.Serializable

一个无边界的基于优先级的队列，优先级是根据排序决定的，不允许null元素，不允许插入不能排序的元素;

如果不提供Comparator的话，优先队列元素中默认按自然顺序排列，数字默认是小的在前，字符串则按字典序排列;

优先级队列是无边界的，但内部有一个属性capacity管理着存放元素的数组大小，capacity大小不应该小于队列大小；

迭代不能保证顺序，所以直接打印该队列的时候顺序是不固定的，要保证顺序，使用Arrays.sort(pq.toArray())

该队列是不同步的，多个线程不应该并发访问，要并发访问，使用PriorityBlockingQueue

O(log(n))时间：offer(), poll(), remove(), add();

Liner time: remove(object), contains(object),

Constant time: peek(), element(), size()

**private** **transient** Object[] queue;

/\*\*

\* The number of elements in the priority queue.

\*/

**private** **int** size = 0;

/\*\*

\* The comparator, or null if priority queue uses elements'

\* natural ordering.

\*/

**private** **final** Comparator<? **super** E> comparator;

/\*\*

\* The number of times this priority queue has been

\* <i>structurally modified</i>. See AbstractList for gory details.

\*/

**private** **transient** **int** modCount = 0;

**public** **boolean** add(E e) {

**return** offer(e);

}

**public** **boolean** offer(E e) {

**if** (e == **null**)

**throw** **new** NullPointerException();

modCount++; 修改次数加1

**int** i = size;

**if** (i >= queue.length) 扩容

grow(i + 1);

size = i + 1;

**if** (i == 0)

queue[0] = e;

**else**

siftUp(i, e);把元素根据顺序插入指定位置，涉及位置移动

**return** **true**;

}

**public** E peek() {

**if** (size == 0)

**return** **null**;

**return** (E) queue[0];

}

**private** **int** indexOf(Object o) {

**if** (o != **null**) {

**for** (**int** i = 0; i < size; i++)

**if** (o.equals(queue[i]))

**return** i;

}

**return** -1;

}

**public** **boolean** remove(Object o) {

**int** i = indexOf(o);

**if** (i == -1) 没找到，返回false

**return** **false**;

**else** {

removeAt(i);需要移动数组元素，所以比较耗时

**return** **true**;

}

}

**private** E removeAt(**int** i) {

**assert** i >= 0 && i < size;

modCount++; 修改加1

**int** s = --size;

**if** (s == i) // removed last element

queue[i] = **null**;

**else** {

E moved = (E) queue[s];

queue[s] = **null**; 不管删除的是哪个元素，最后一个元素都指向null

siftDown(i, moved);下面进行查找以及移动数组元素

**if** (queue[i] == moved) {

siftUp(i, moved);

**if** (queue[i] != moved)

**return** moved;

}

}

**return** **null**;

}

**boolean** removeEq(Object o) {

**for** (**int** i = 0; i < size; i++) {

**if** (o == queue[i]) { 比较是不是同一个对象，而不是比较内容，注意与上面方法 的区别

removeAt(i);

**return** **true**;

}

}

**return** **false**;

}

**public** **boolean** contains(Object o) {

**return** indexOf(o) != -1;

}

返回的结果是安全的无序的

**public** Object[] toArray() {

**return** Arrays.*copyOf*(queue, size);

}

**public** **int** size() {

**return** size;

}

**public** **void** clear() {

modCount++;

**for** (**int** i = 0; i < size; i++)

queue[i] = **null**;

size = 0;

}

**public** E poll() {

**if** (size == 0)

**return** **null**;

**int** s = --size;

modCount++;

E result = (E) queue[0]; 从头出队列，也要涉及到位置的移动

E x = (E) queue[s];

queue[s] = **null**;

**if** (s != 0)

siftDown(0, x);

**return** result;

}

## PriorityBlockingQueue

**public** **class** PriorityBlockingQueue<E> **extends** AbstractQueue<E> **implements** BlockingQueue<E>, java.io.Serializable

参考PriorityQueue,因为是无界的，所以入队列不会被阻塞，但出队列方法take()如果没有元素还是会阻塞等待，offer(),peek都有加锁,如果队列需要根据某种顺序来的话，可以考虑这个