# [BlockingQueue](http://www.importnew.com/28053.html)

BlockingQueue 支持当获取队列元素但是队列为空时，会阻塞等待队列中有元素再返回；也支持添加元素时，如果队列已满，那么等到队列可以放入新元素时再放入。

其提供了4种类型的方法：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *Throws exception* | *Special value* | *Blocks* | *Times out* |
| Insert | add(e) | offer(e) | put(e) | offer(e, time, unit) |
| Remove | remove() | poll() | take() | poll(time, unit) |
| Examine | element() | peek() | *not applicable* | *not applicable* |

BlockingQueue不接受 null 元素。所有实现应当抛出 NullPointerException 在所有的 add,put以及offer方法上。null被用来标记poll失败。

在任意时刻，当有界BlockingQueue 队列元素放满之后，所有的元素都将在放入的时候阻塞。无界BlockingQueue 没有任何容量限制，容量大小始终是Integer.MAX\_VALUE。

BlockingQueue的实现是用于 生产者-消费者 的队列，同时也支持 Collection 接口。所以可通过remove(x)来移除队列里的一个元素。通常情况下，这样的操作效率不是很好，只在诸如队列消息被取消的情况下才会偶尔使用。

BlockingQueue 的实现都是线程安全的。所有 queue 的方法都需要通过内部锁机制或者其他形式来进行并发控制来实现其原子操作。然而，Collection 接口的方法，比如：**addAll, containsAll, retainAll 以及 removeAll 都没有必要进行原子操作**，除非实现类有特别说明。所以对于addAll(c)有可能在添加部分c元素后抛出异常。

BlockingQueue 本质上不支持任何的 close 或者 shutdown 操作，来表明不会有新的元素添加。如果需要这些特性，得实现类来支持。

## ArrayBlockingQueue

ArrayBlockingQueue 是底层由数组存储的有界队列。遵循FIFO，所以在队首的元素是在队列中等待时间最长的，而在队尾的则是最短时间的元素。新元素被插入到队尾，队列的取出 操作队首元素。

这是一个经典的有界缓存，由一个长度确定的数组持有所有由生产者插入、由消费者取出的元素。一旦创建，整个队列的容量将不会改变。尝试向一个已满的队列 put 将会导致调用被阻塞，同样的向一个空队列 take 也会阻塞。

该队列支持队等待的生产者和消费者实施可选的公平策略。默认情况下，是非公平策略。可以通过构造函数来指定是否进行公平策略。一般情况下公平策略会减小吞吐量，但是也会降低可变性以及防止饥饿效应。

### 实现

ArrayBlockingQueue 内部使用了 ReentrantLock 以及两个 Condition 来实现。

/\*\* Main lock guarding all access \*/  
final ReentrantLock lock;  
/\*\* Condition for waiting takes \*/  
private final Condition notEmpty;  
/\*\* Condition for waiting puts \*/  
private final Condition notFull;

PUT 方法也很简单，就是 Condition 的应用。

public void put(E e) throws InterruptedException {  
 checkNotNull(e);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 //队列已满，wait 在 condition 上  
 while (count == items.length)  
 notFull.await();  
 enqueue(e);  
 } finally {  
 lock.unlock();  
 }  
}

take 方法也同样的。

public E take() throws InterruptedException {  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 //队列为空，wait 在 condition 上  
 while (count == 0)  
 notEmpty.await();  
 return dequeue();  
 } finally {  
 lock.unlock();  
 }  
}