什么是阻塞队列?

阻塞队列的作用:

阻塞队列，也就是 BlockingQueue，它是一个接口，如代码所示:

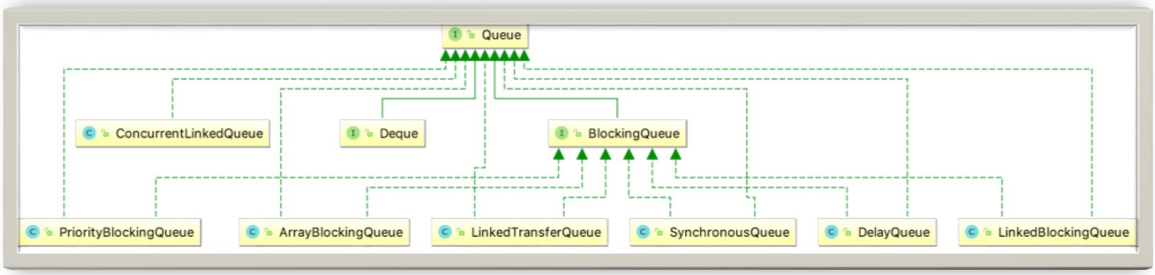
```
1 public interface BlockingQueue<E> extends Queue<E>{...}
```

BlockingQueue 继承了 Queue 接口，是队列的一种。Queue 和 BlockingQueue 都是在 Java 5 中加入的。

BlockingQueue是线程安全的，所以生产者和消费者都可以是多线程的，不会发生线程安全问题。

BlockingQueue实现了任务与执行任务类之间的解耦，任务被放在了阻塞队列中，而负责放任务的线程是无法直接访问到我们银行具体实现转账操作的对象的，实现了隔离，提高了安全性。

并发队列关系图:



上图展示了 Queue 最主要的实现类，可以看出 Java 提供的线程安全的队列（也称为并发队列）分为**阻塞队列**和**非阻塞队列**两大类。

阻塞队列的典型例子就是 BlockingQueue 接口的实现类，BlockingQueue 下面有 6 种最主要的实现，分别是 ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、DelayQueue、PriorityBlockingQueue 和 LinkedTransferQueue，它们各自有不同的特点，对于这些常见的阻塞队列的特点，我们会在第 36 课时中展开说明。

非阻塞并发队列的典型例子是 ConcurrentLinkedQueue，这个类不会让线程阻塞，利用 CAS 保证了线程安全。

我们可以根据需要自由选取阻塞队列或者非阻塞队列来满足业务需求。

还有一个和 Queue 关系紧密的 Deque 接口，它继承了 Queue，如代码所示：

```
1 public interface Deque<E> extends Queue<E> {//...}
```

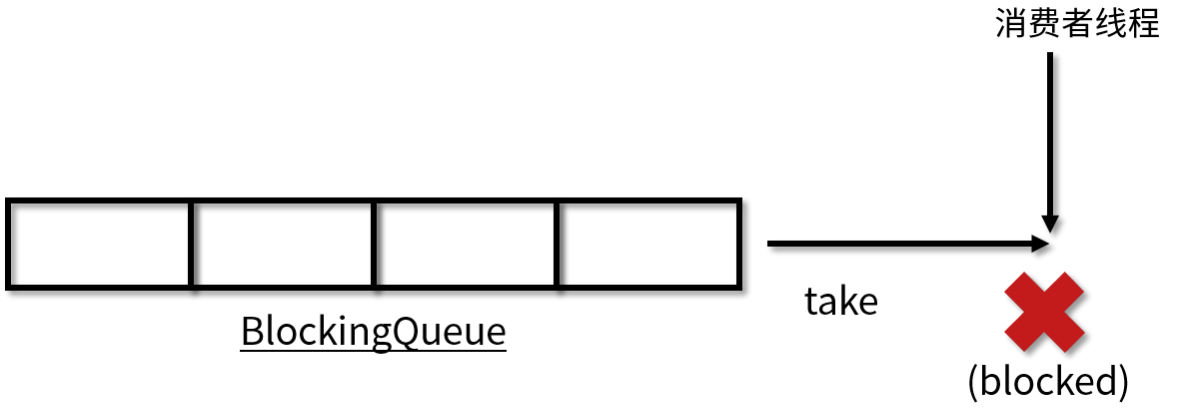
Deque 的意思是双端队列，音标是 [dek]，是 double-ended-queue 的缩写，它从头和尾都能添加和删除元素；而普通的 Queue 只能从一端进入，另一端出去。这是 Deque 和 Queue 的不同之处，Deque 其他方面的性质都和 Queue 类似。

阻塞队列的特点：

阻塞功能使得生产者和消费者两端的能力得以平衡，当有任何一端速度过快时，阻塞队列便会把过快的速度给降下来。实现阻塞最重要的两个方法是 take 方法和 put 方法。

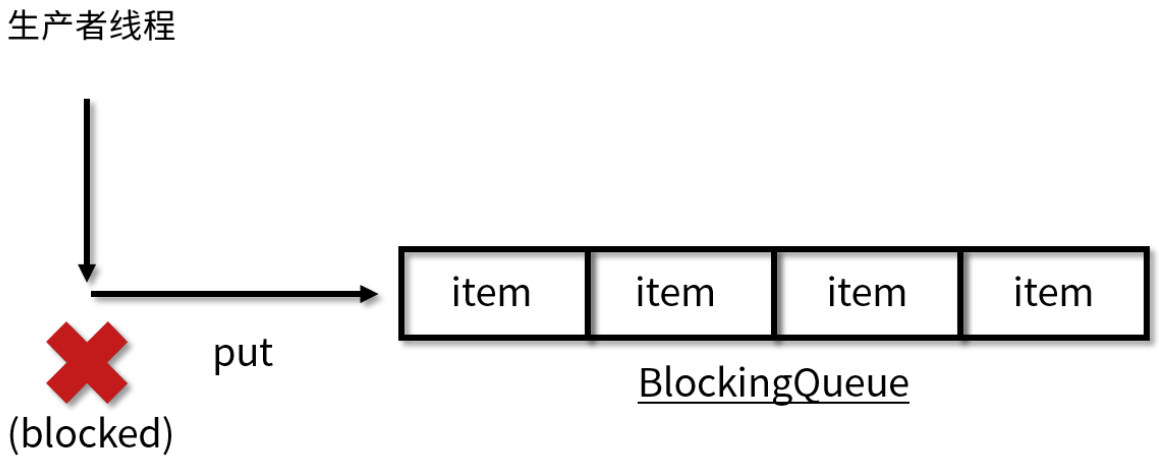
take 方法：

take 方法的功能是获取并移除队列的头结点，通常在队列里有数据的时候是可以正常移除的。可是一旦执行 take 方法的时候，队列里无数据，则阻塞，直到队列里有数据。一旦队列里有数据了，就会立刻解除阻塞状态，并且取到数据。



put 方法：

put 方法插入元素时，如果队列没有满，那就和普通的插入一样是正常的插入，但是如果队列已满，那么就无法继续插入，则阻塞，直到队列里有了空闲空间。如果后续队列有了空闲空间，比如消费者消费了一个元素，那么此时队列就会解除阻塞状态，并把需要添加的数据添加到队列中。过程如图所示：



以上过程中的阻塞和解除阻塞，都是 BlockingQueue 完成的，不需要我们自己处理。

是否有界（容量有多大）：

此外，阻塞队列还有一个非常重要的属性，那就是容量的大小，分为有界和无界两种

无界队列意味着里面可以容纳非常多的元素，例如 LinkedBlockingQueue 的上限是 Integer.MAX_VALUE，约为 2 的 31 次方，是非常大的一个数，可以近似认为是无限容量，因为我们几乎无法把这个容量装满。

但是有的阻塞队列是有界的，例如 ArrayBlockingQueue 如果容量满了，也不会扩容，所以一旦满了就无法再往里放数据了。

阻塞队列包含哪些常用的方法？add、offer、put 等方法的区别？

详见下图：

组别	方法	含义	特点
第1组	add	添加一个元素	如果队列满了，操作失败，抛出IllegalStateException
	remove	返回并删除队列的头元素	如果队列空，删除失败，抛出NoSuchElementException
	element	返回队列的头元素	如果队列空，操作失败，抛出NoSuchElementException
第2组	offer	添加一个元素	如果队列满了，返回 false。如果添加成功，返回 true
	poll	返回并删除队列的头元素	如果队列空，删除失败，返回 null
	peek	返回队列的头元素	如果队列空，操作失败，返回 null
第3组	put	添加一个元素	如果队列满了，就会阻塞
	take	返回并删除队列的头元素	如果队列是空的，就会阻塞

有哪几种常见的阻塞队列?

ArrayBlockingQueue：

ArrayBlockingQueue 是最典型的**有界队列**，内部使用数组存储元素，利用ReentrantLock （可重入锁）实现线程安全，指定容量后不再扩容，可以指定是否公平，在构造函数指定，如下代码：

```
ArrayBlockingQueue(int capacity, boolean fair)//capacity为容量，fair为是否公平
```

LinkedBlockingQueue：

内部用链表实现的 BlockingQueue，如果我们不指定它的初始容量，那么它容量默认就为整型的最大值 Integer.MAX_VALUE，由于这个数非常大，我们通常不可能放入这么多的数据，所以 LinkedBlockingQueue 也被称作无界队列，代表它几乎没有界限。

SynchronousQueue：

它的容量为 0，所以没有一个地方来暂存元素，导致每次取数据都要先阻塞，直到有数据被放入；同理，每次放数据的时候也会阻塞，直到有消费者来取（有点类似于懒加载的意思，需要拿的时候再去放）。

扩展：

SynchronousQueue 的 peek 方法永远返回 null，代码如下：

```
public E peek() {
    return null;
}
```

因为 peek 方法的含义是取出头结点，但是 SynchronousQueue 的容量是 0，所以连头结点都没有，peek 方法也就没有意义，所以始终返回 null。同理，element 始终会抛出 NoSuchElementException 异常。

而 SynchronousQueue 的 size 方法始终返回 0，因为它内部并没有容量，代码如下：

```
public int size() {
    return 0;
}
```

直接 return 0，同理，isEmpty 方法始终返回 true：

```
public boolean isEmpty() {
    return true;
}
```

因为它始终都是空的。

PriorityBlockingQueue：

ArrayBlockingQueue 和 LinkedBlockingQueue 都是采用先进先出的顺序进行排序，可是如果有的时候我们需要自定义排序怎么办呢？这时就需要使用 PriorityBlockingQueue。

PriorityBlockingQueue 是一个支持优先级的无界阻塞队列，可以通过自定义类实现 compareTo() 方法来指定元素排序规则，或者初始化时通过构造器参数 Comparator 来指定排序规则。同时，插入队列的对象必须是可比较大小的，也就是 Comparable 的，否则会抛出 ClassCastException 异常。

它的 take 方法在队列为空的时候会阻塞，但是正因为它是无界队列，而且会自动扩容，所以它的队列永远不会满，所以它的 put 方法永远不会阻塞，添加操作始终都会成功，也正因为如此，它的成员变量里只有一个 Condition：

```
private final Condition notEmpty;
```

这和之前的 ArrayBlockingQueue 拥有两个 Condition（分别是 notEmpty 和 notFull）形成了鲜明的对比，我们的 PriorityBlockingQueue 不需要 notFull，因为它永远都不会满，真是“有空间就可以任性”。

DelayQueue：

DelayQueue 这个队列比较特殊，具有“延迟”的功能。我们可以设定让队列中的任务延迟多久之后执行，比如 10 秒钟之后执行，这在例如“30 分钟后未付款自动取消订单”等需要延迟执行的场景中被大量使用。

它是无界队列，放入的元素必须实现 Delayed 接口，而 Delayed 接口又继承了 Comparable 接口，所以自然就拥有了比较和排序的能力，代码如下：

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

可以看出这个 Delayed 接口继承自 Comparable，里面有一个需要实现的方法，就是 getDelay。这里的 getDelay 方法返回的是“还剩下多长的延迟时间才会被执行”，如果返回 0 或者负数则代表任务已过期。

元素会根据延迟时间的长短被放到队列的不同位置，越靠近队列头代表越早过期。

DelayQueue 内部使用了 PriorityQueue 的能力来进行排序，而不是自己从头编写，我们在工作中可以学习这种思想，对已有的功能进行复用，不但可以减少开发量，同时避免了“重复造轮子”，更重要的是，对学到的知识进行合理的运用，让知识变得更灵活，做到触类旁通。

DelayQueue：就是判断该时间段该任务是否执行，超过时间执行为已过期,过期会返回0或负数，会根据延迟时间长短排序任务，排序调用PriorityQueue 进行排序。

阻塞和非阻塞队列的并发安全原理是什么？

阻塞队列：

以 ArrayBlockingQueue为例：

put（）：

```
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)//items:用于存放元素的数组
        notFull.await(); //Condition notFull
        enqueue(e);
    } finally {
        lock.unlock();
    }
}
```

首先用 checkNotNull 方法去检查插入的元素是不是 null。如果不是 null，我们会用 ReentrantLock 上锁，并且上锁方法是 lock.lockInterruptibly()，**锁方法是 lock.lockInterruptibly()，该方法可以响应中断的。**

由上代码可知，BlockingQueue 内部通过ReentrantLock、 Condition来保证线程安全。

非阻塞队列：

以ConcurrentLinkedQueue为例：

offer （）：

```
public boolean offer(E e) {
    checkNotNull(e);
    final Node<E> newNode = new Node<E>(e);
    for (Node<E> t = tail, p = t;;) {
        Node<E> q = p.next;
        if (q == null) {
            if (p.casNext(null, newNode)) {
                if (p != t)
                    casTail(t, newNode);
                return true;
            }
        }
        else if (p == q)
            p = (t != (t = tail)) ? t : head;
        else
            p = (p != t && t != (t = tail)) ? t : q;
    }
}
```

由上面代码可以看出，非阻塞队列 ConcurrentLinkedQueue 使用 CAS 非阻塞算法 + 不停重试，来实现线程安全，适合用在不需要阻塞功能，且并发不是特别剧烈的场景。

如何选择适合自己的阻塞队列?

线程池对于阻塞队列的选择:

FixedThreadPool	LinkedBlockingQueue
SingleThreadExecutor	LinkedBlockingQueue
CachedThreadPool	SynchronousQueue
ScheduledThreadPool	DelayedWorkQueue
SingleThreadScheduledExecutor	DelayedWorkQueue

你可以看到 5 种线程池只对应了 3 种阻塞队列:

- **FixedThreadPool（SingleThreadExecutor 同理）选取的是 LinkedBlockingQueue:**

  因为 LinkedBlockingQueue不同于 ArrayBlockingQueue， ArrayBlockingQueue 的容量是有限的，而 LinkedBlockingQueue 是链表长度默认是可以无限延长的。

  FixedThreadPool 的线程数是固定的，所以在任务激增的时候需要像 LinkedBlockingQueue 这样没有容量上限的 Queue 来存储那些还没处理的 Task，从而不会拒绝新任务的提交，也不会丢失数据。

- **CachedThreadPool 选取的是 SynchronousQueue:**

  对于可创建线程数无限大的CachedThreadPool ，选择SynchronousQueue 直接把任务交给线程，而不需要另外保存它们，效率更高，所以 CachedThreadPool 使用的 Queue 是 SynchronousQueue。

- **ScheduledThreadPool（SingleThreadScheduledExecutor同理）选取的是延迟队列:**

  延迟队列的特点是：不是先进先出，而是会按照延迟时间的长短来排序，下一个即将执行的任务会排到队列的最前面。

  我们选择使用延迟队列的原因是，ScheduledThreadPool 处理的是基于时间而执行的 Task，而延迟队列有能力把 Task 按照执行时间的**先后进行排序**，这正是我们所需要的功能。

选择合适的阻塞队列，可以从以下 5 个角度考虑:

- **功能:**

  比如是否需要阻塞队列帮我们排序，如优先级排序、延迟执行等。如果有这个需要，我们就必须选择类似于 PriorityBlockingQueue 之类的有排序能力的阻塞队列。

- **容量:**

  有的是容量固定的，有的默认是容量无限的，而有的里面没有任何容量（DelayQueue 它的容量固定就是 Integer.MAX_VALUE）。

- **能否扩容:**

  有可能需要动态扩容，我们需要动态扩容的话，那么就不能选择 ArrayBlockingQueue，因为它的容量在创建时就确定了，无法扩容。相反，PriorityBlockingQueue 即使在指定了初始容量之后，后续如果有需要，也可以自动扩容。 。

- **内存结构:**

LinkedBlockingQueue 的内部是用链表实现的，所以这里就需要我们考虑到，ArrayBlockingQueue 没有链表所需要的“节点”，空间利用率更高。所以如果我们对性能有要求可以从内存的结构角度去考虑这个问题。

- **性能：**

  第 5 点就是从性能的角度去考虑。比如 LinkedBlockingQueue 由于拥有两把锁，它的操作粒度更细，**在并发程度高的时候**，相对于只有一把锁的 ArrayBlockingQueue 性能会更好。