目录

22 ArrayBlockingQueue 源码解析

更新时间：2019-11-07 10:48:29

> 耐心和恒心总会得到报酬的。
>
> ——爱因斯坦

引导语

本小节我们来介绍本章最后一个队列：ArrayBlockingQueue。按照字面翻译，中文叫做数组阻塞队列，从名称上看，我们就比较清楚此阻塞队列底层使用的是数组。一说到数组，大家可能会想到 ArrayList 和 HashMap，举新增场景来说 ArrayList 通过 size ++ 找到新增的数组下标位置，HashMap 通过 hash 算法计算出下标位置，那么 ArrayBlockingQueue 是不是也是这两种方法呢？都不是，ArrayBlockingQueue 使用的是一种非常奇妙的方式，我们一起拭目以待。

全文为了方便说明，队头的说法就是数组头，队尾的说法就是数组尾。

1 整体架构

我们从类注释上可以得到一些有用的信息：

1.1 类注释

1. 有界的阻塞数组，容量一旦创建，后续大小无法修改；
2. 元素是有顺序的，按照先入先出进行排序，从队尾插入数据数据，从队头拿数据；
3. 队列满时，往队列中 put 数据会被阻塞，队列空时，往队列中拿数据也会被阻塞。

从类注释上可以看出 ArrayBlockingQueue 和一般的数组结构的类不太一样，是不能够动态扩容的，如果队列满了或者空时，take 和 put 都会被阻塞。

1.2 数据结构

```
final Object[] items;

// 下次拿数据的时候的索引位置
int takeIndex;

// 下次放数据的索引位置
int putIndex;

// 当前已有元素的大小
int count;

// 可重入的锁
final ReentrantLock lock;

// take的队列
private final Condition notEmpty;

// put的队列
private final Condition notFull;
```

以上代码有两个关键的字段，takeIndex 和 putIndex，分别表示下次拿数据和放数据的索引位置。所以说在新增数据和拿数据时，都无需计算，就能知道应该新增到什么位置，应该从什么位置拿数据。

2 初始化

初始化时，有两个重要的参数：数组的大小、是否是公平，源码如下：

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    // 队列不为空 Condition，在 put 成功时使用
    notEmpty = lock.newCondition();
    // 队列不满 Condition，在 take 成功时使用
    notFull =  lock.newCondition();
}
```

从源码中我们可以看出，第二个参数是否公平，主要用于读写锁是否公平，如果是公平锁，那么在锁竞争时，就会按照先来先到的顺序，如果是非公平锁，锁竞争时随机的。

对于锁公平和非公平，我们举个例子：比如说现在队列是满的，还有很多线程执行 put 操作，必然会有很多线程阻塞等待，当有其它线程执行 take 时，会唤醒等待的线程，如果是公平锁，会按照阻塞等待的先后顺序，依次唤醒阻塞的线程，如果是非公平锁，会随机唤醒沉睡的线程。

所以说队列满很多线程执行 put 操作时，如果是公平锁，数组元素新增的顺序就是阻塞线程被释放的先后顺序，是有顺序的，而非公平锁，由于阻塞线程被释放的顺序是随机的，所以元素插入到数组的顺序也就不会按照插入的顺序了。

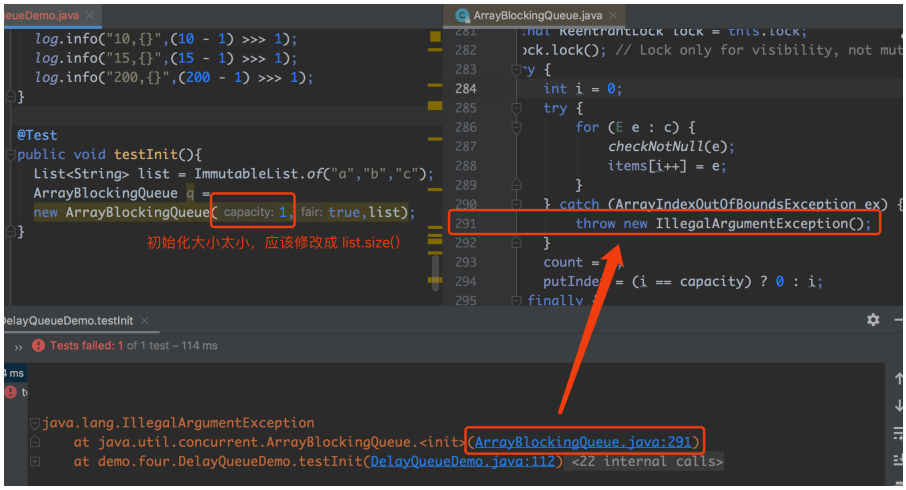
队列空时，也是一样的道理。

ArrayBlockingQueue 通过锁的公平和非公平，轻松实现了数组元素的插入顺序的问题。如果要实现这个功能，你会怎么做呢？会想到利用锁的功能么？其实这种思想我们在文中多次提到，

初始化时，如果给定了原始数据的话，一定要注意原始数据的大小一定要小于队列的容量，否则会抛异常，如下图所示：

```
public ArrayBlockingQueue(int capacity, boolean fair,
                          Collection<? extends E> c) {
    this(capacity, fair);

    final ReentrantLock lock = this.lock;
    lock.lock(); // Lock only for visibility, not mutual exclusion
    try {
        // i 代表插入的位置
        int i = 0;
        try {
            // 此时需要注意的是，如果 c 的大小超过了数组的大小
            // 是会抛异常的
            for (E e : c) {
                checkNotNull(e);
                items[i++] = e;
            }
        } catch (ArrayIndexOutOfBoundsException ex) { 数组越界的异常
            throw new IllegalArgumentException();
        }
        count = i;
        // 如果插入的位置，正好是队尾了，下次需要从 0 开始插入
        putIndex = (i == capacity) ? 0 : i;
    } finally {
        lock.unlock();
    }
```

我们写了一个 demo，报错如下：



3 新增数据

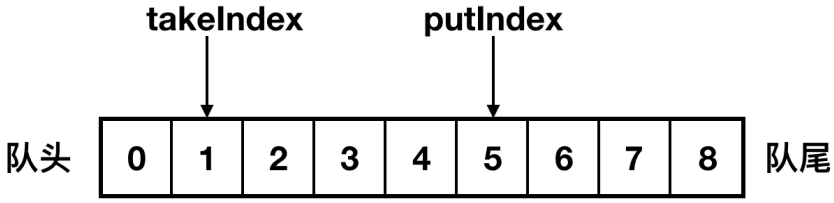
数据新增都会按照 putIndex 的位置进行新增，源码如下：

```
// 新增，如果队列满，无限阻塞
public void put(E e) throws InterruptedException {
    // 元素不能为空
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
```

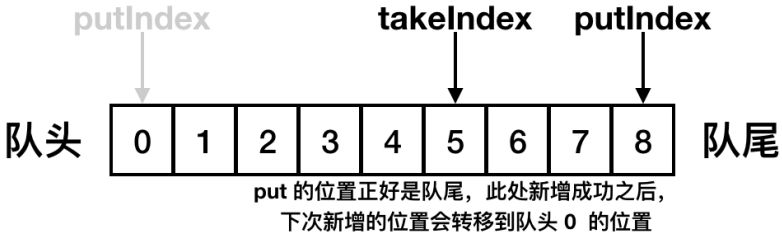
```
        while (count == items.length)
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}

private void enqueue(E x) {
    // assert lock.getHoldCount() == 1; 同一时刻只能一个线程进行操作此方法
    // assert items[putIndex] == null;
    final Object[] items = this.items;
    // putIndex 为本次插入的位置
    items[putIndex] = x;
    // ++ putIndex 计算下次插入的位置
    // 如果下次插入的位置，正好等于队尾，下次插入就从 0 开始
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
    // 唤醒因为队列空导致的等待线程
    notEmpty.signal();
}
```

从源码中，我们可以看出，其实新增就两种情况：

1. 本次新增的位置居中，直接新增，下图演示的是 putIndex 在数组下标为 5 的位置，还不到队尾，那么可以直接新增，计算下次新增的位置应该是 6；



2. 新增的位置到队尾了，那么下次新增时就要从头开始了，示意图如下：



上面这张图演示的就是这行代码： `if (++putIndex == items.length) putIndex = 0;`

可以看到当新增到队尾时，下次新增会重新从队头重新开始。

4 拿数据

拿数据都是从队头开始拿数据，源码如下：

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        // 如果队列为空，无限等待
```

```
        // 从队列中拿数据
        return dequeue();
    } finally {
        lock.unlock();
    }
}

private E dequeue() {
    final Object[] items = this.items;
    // takeIndex 代表本次拿数据的位置，是上一次拿数据时计算好的
    E x = (E) items[takeIndex];
    // 帮助 gc
    items[takeIndex] = null;
    // ++ takeIndex 计算下次拿数据的位置
    // 如果正好等于队尾的话，下次就从 0 开始拿数据
    if (++takeIndex == items.length)
        takeIndex = 0;
    // 队列实际大小减 1
    count--;
    if (itrs != null)
        itrs.elementDequeued();
    // 唤醒被队列满所阻塞的线程
    notFull.signal();
    return x;
}
```

从源码中可以看出，每次拿数据的位置就是 takeIndex 的位置，在找到本次该拿的数据之后，会把 takeIndex 加 1，计算下次拿数据时的索引位置，有个特殊情况是，如果本次拿数据的位置已经是队尾了，那么下次拿数据的位置就要从头开始，就是从 0 开始了。

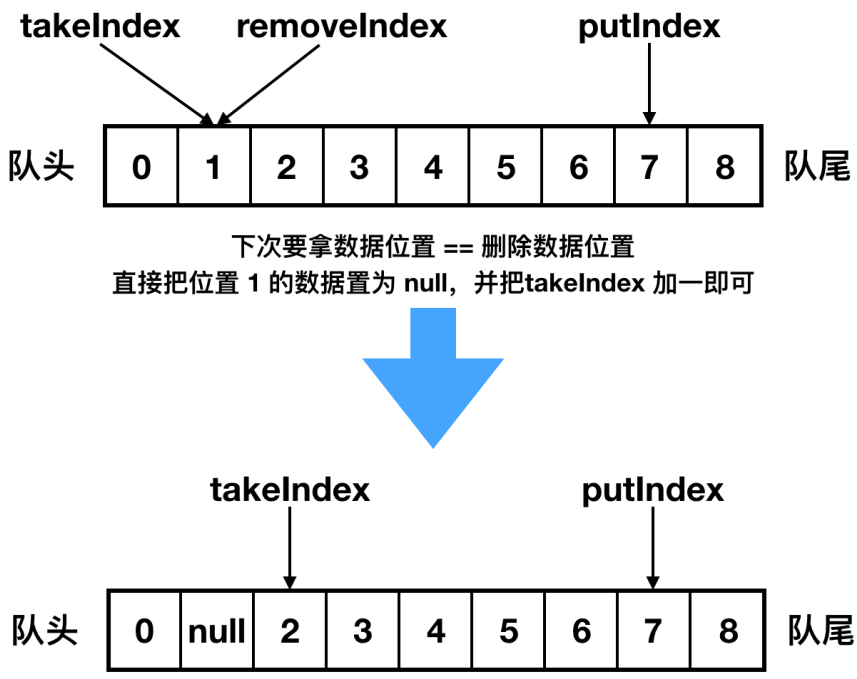
5 删除数据

删除数据很有意思，我们一起来看下核心源码：

```
// 一共有两种情况：
// 1：删除位置和 takeIndex 的关系：删除位置和 takeIndex 一样，比如 takeIndex 是 2， 而要删除
// 2：找到要删除元素的下一个，计算删除元素和 putIndex 的关系
// 如果下一个元素不是 putIndex，就把下一个元素往前移动一位
// 如果下一个元素是 putIndex，把 putIndex 的值修改成删除的位置
void removeAt(final int removeIndex) {
    final Object[] items = this.items;
    // 情况1 如果删除位置正好等于下次要拿数据的位置
    if (removeIndex == takeIndex) {
        // 下次要拿数据的位置直接置空
        items[takeIndex] = null;
        // 要拿数据的位置往后移动一位
        if (++takeIndex == items.length)
            takeIndex = 0;
        // 当前数组的大小减一
        count--;
        if (itrs != null)
            itrs.elementDequeued();
    // 情况 2
    } else {
        final int putIndex = this.putIndex;
        for (int i = removeIndex;;) {
            // 找到要删除元素的下一个
            int next = i + 1;
            if (next == items.length)
```

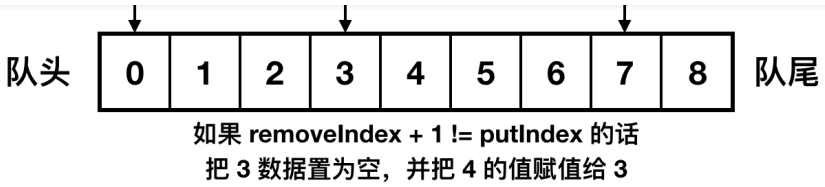
```
                // 下一个元素往前移动一位
                items[i] = items[next];
                i = next;
            // 下一个元素是 putIndex
            } else {
                // 删除元素
                items[i] = null;
                // 下次放元素时，应该从本次删除的元素放
                this.putIndex = i;
                break;
            }
        }
        count--;
        if (itrs != null)
            itrs.removedAt(removeIndex);
    }
    notFull.signal();
}
```

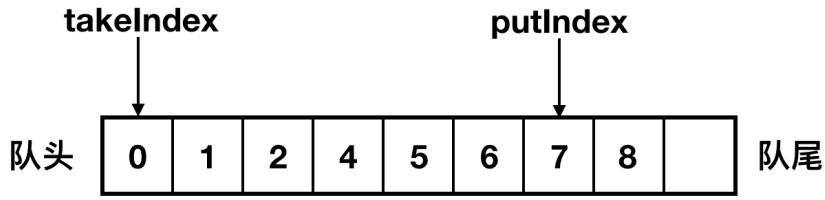
删除数据的情况比较复杂，一共有两种情况，第一种情况是 takeIndex == removeIndex，我们画个示意图来看下处理方式：



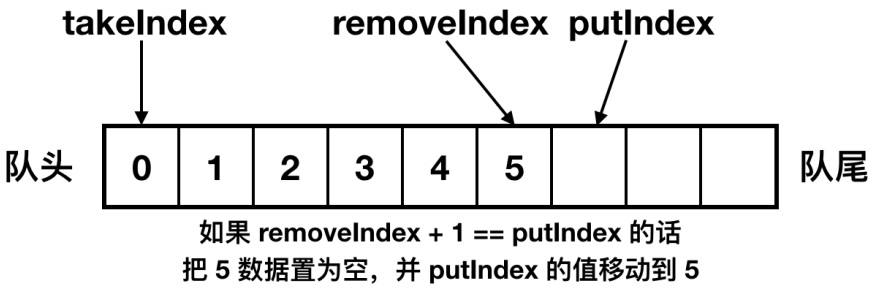
第二种情况又分两种：

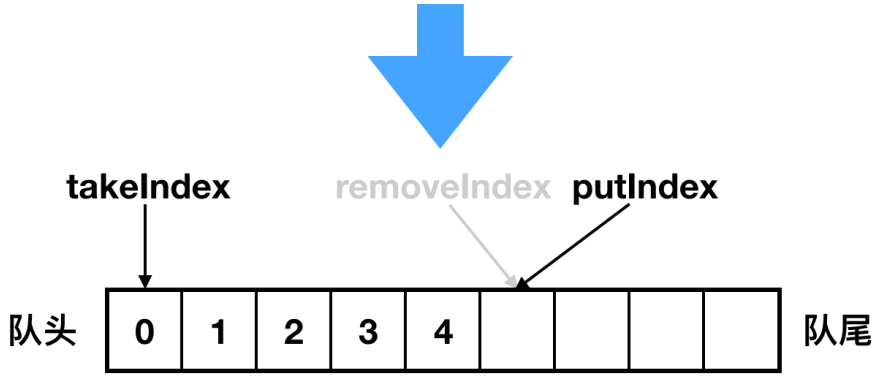
1. 如果 removeIndex + 1 != putIndex 的话，就把下一个元素往前移动一位，示意图如下：





2. 如果 removeIndex + 1 == putIndex 的话，就把 putIndex 的值修改成删除的位置，示意图如下：





ArrayBlockingQueue 的删除方法其实还蛮复杂的，需要考虑到很多特殊的场景。

6 总结

ArrayBlockingQueue 底层是有界的数组，整体来说，和其它队列差别不多，需要注意的是，当 takeIndex、putIndex 到队尾的时候，都会重新从 0 开始循环，这点是比较特殊的，在我们学习源码时，需要特别注意。

← 21 DelayQueue 源码解析 23 队列在源码方面的面试题 →

精选留言 3

目录

Sicimike

老师，我两个疑问。 第一个是构造方法ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c)中有一行注释“Lock only for visibility, not mutual exclusion”（第282行）是说ReentrantLock可以保证可见性吗？但是happens-before原则中的lock原则不是仅仅指synchronized吗？ 第二个是removeAt方法为什么不用加锁？

0 回复 2019-12-09

北京_鲁班七号

老师你好，这可不可以理解为ArrayBlockingQueue是一种循环队列，通过维护队首、队尾的指针，来优化插入、删除，从而使时间复杂度为O(1)

0 回复 2019-11-29

文贺 回复 **北京_鲁班七号**

是的，主要是因为有 takeIndex，putIndex，removeIndex 三个变量在维护位置

回复 2019-11-30 12:54:05

慕粉3445147

新增数据章节.."唤醒因为队列空,所以等待的线程"错别字hhh

0 回复 2019-11-06

文贺 回复 **慕粉3445147**

谢谢，已收到，语句可能不好理解，已经更改描述了

回复 2019-11-17 10:56:02

千学不如一看，千看不如一练