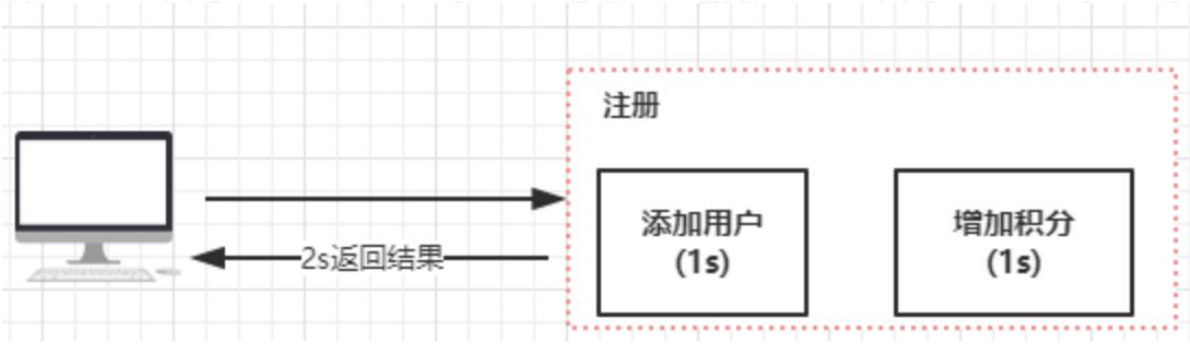
把多线程环境比作是分布式的话，那么线程与线程之间是不是也可以使用这种消息队列的方式进行数据通信和解耦呢?

一，阻塞队列的使用案例

1，注册成功后增加积分

假如模拟一个场景，就是用户注册的时候，在注册成功以后发放积分。这个场景在一般来说，会这么去实现：



但是实际上，我们需要考虑两个问题：

1，性能，在注册这个环节里面，假如添加用户需要花费 1 秒钟，增加积分需要花费 1 秒钟，那么整个注册结果的返回就可能需要大于 2 秒，虽然影响不是很大，但是在量比较大的时候，我们也需要做一些优化。

2，耦合，添加用户和增加积分，可以认为是两个领域，也就是说，增加积分并不是注册必须要具备的功能，但是一旦增加积分这个逻辑出现异常，就会导致注册失败。这种耦合在程序设计的时候是一定要规避的。

因此我们可以通过异步的方式来实现

2，改造之前的代码逻辑

```
public class UserService {

    public boolean register() {
        User user = new User();
        user.setName("Mic");
        addUser(user);
        sendPoints(user);
        return true;
    }

    public static void main(String[] args) {
        new UserService().register();
    }

    private void addUser(User user) {
        System.out.println(" 添加用户：" + user);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

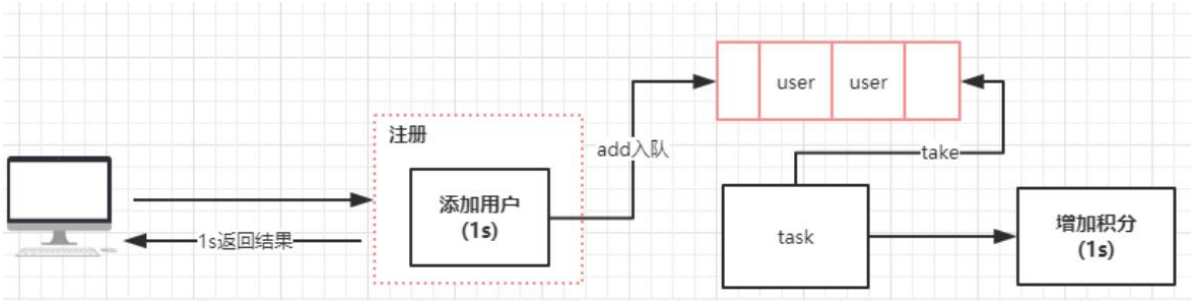
```
        }
    }

    private void sendPoints(User user) {
        System.out.println("  发 送 积 分 给 指 定 用 户:" + user);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

@Data
class User {
    private String name;
}
```

3，改造之后的逻辑

```
public class UserService {

    private final ExecutorService single = Executors.newSingleThreadExecutor();

    private volatile boolean isRunning = true;

    ArrayBlockingQueue arrayBlockingQueue = new ArrayBlockingQueue(10);

    {
        init();
    }

    public void init() {
        single.execute(() -> {
            while (isRunning) {
                try {
                    User user = (User) arrayBlockingQueue.take();// 阻塞的方式获取
队列中的数据
                    sendPoints(user);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }

    public boolean register() {
        User user = new User();
        user.setName("Mic");
        addUser(user);
        arrayBlockingQueue.add(user);// 添加到异步队列
        return true;
    }

    public static void main(String[] args) {
        new UserService().register();
    }
```

```
    private void addUser(User user) {
        System.out.println(" 添加用户: " + user);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void sendPoints(User user) {
        System.out.println("  发 送 积 分 给 指 定 用 户:" + user);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

@Data
class User {
    private String name;
}
```

优化以后，整个流程就变成了这样



我们使用了 ArrayBlockingQueue 基于数组的阻塞队列，来优化代码的执行逻辑。

4，阻塞队列的应用场景

阻塞队列这块的应用场景，比较多的仍然是对于生产者消费者场景的应用，但是由于分布式架构的普及，更多的关注在分布式消息队列上。所以其实如果把阻塞队列比作成分布式消息队列的话，那么所谓的生产者和消费者其实就是基于阻塞队列的解耦。

另外，阻塞队列是一个 fifo 的队列，所以对于希望在线程级别需要实现对目标服务的顺序访问的场景中，也可以使用。

二，J.U.C 中的阻塞队列

1，JUC中提供的阻塞队列

在 Java8 中，提供了 7 个阻塞队列。

ArrayBlockingQueue	**数组实现的有界阻塞队列，此队列按照先进先出（FIFO）的原则对元素进行排序。**
LinkedBlockingQueue	**链表实现的有界阻塞队列，此队列的默认和最大长度为 Integer.MAX_VALUE。此队列按照先进先出的原则对元素进行排序**
PriorityBlockingQueue	**支持优先级排序的无界阻塞队列，默认情况下元素采取自然顺序升序排列。也可以自定义类实现 compareTo()方法来指定元素排序规则，或者初始化 PriorityBlockingQueue 时，指定构造参数 Comparator 来对元素进行排序。**
DelayQueue	**优先级队列实现的无界阻塞队列**
SynchronousQueue	**不存储元素的阻塞队列，每一个 put 操作必须等待一个 take 操作，否则不能继续添加元素。**
LinkedTransferQueue	**链表实现的无界阻塞队列**
LinkedBlockingDeque	**链表实现的双向阻塞队列**

2，阻塞队列的操作方法

在阻塞队列中，提供了四种处理方式

1）插入操作

add(e)： 添加元素到队列中，如果队列满了，继续插入元素会报错，IllegalStateException。

offer(e) : 添加元素到队列，同时会返回元素是否插入成功的状态，如果成功则返回 true

put(e)： 当阻塞队列满了以后，生产者继续通过 put添加元素，队列会一直阻塞生产者线程，直到队列可用

offer(e,time,unit)： 当阻塞队列满了以后继续添加元素，生产者线程会被阻塞指定时间，如果超时，则线程直接退出

2）移除操作

remove()：当队列为空时，调用 remove 会返回 false，如果元素移除成功，则返回 true

poll(): 当队列中存在元素，则从队列中取出一个元素，如果队列为空，则直接返回 null

take()：基于阻塞的方式获取队列中的元素，如果队列为空，则 take 方法会一直阻塞，直到队列中有新的数据可以消费

poll(time,unit)：带超时机制的获取数据，如果队列为空，则会等待指定的时间再去获取元素返回

三，ArrayBlockingQueue 原理分析

1，构造方法

ArrayBlockingQueue 提供了三个构造方法，分别如下。

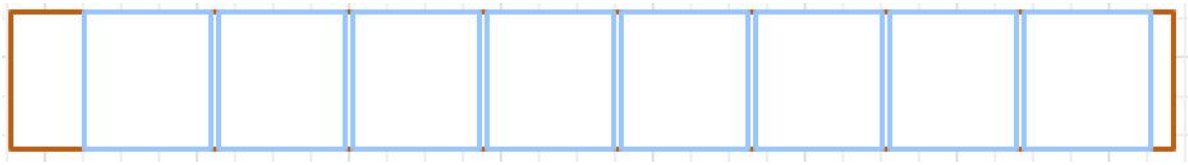
capacity： 表示数组的长度，也就是队列的长度。

fair：表示是否为公平的阻塞队列，默认情况下构造的是非公平的阻塞队列。

```
    public ArrayBlockingQueue(int capacity) {
```

```
        this(capacity, false);
    }

    public ArrayBlockingQueue(int capacity, boolean fair) {
        if (capacity <= 0)
            throw new IllegalArgumentException();
        this.items = new Object[capacity];
        lock = new ReentrantLock(fair);//重入锁，出队和入队持有这一把锁
        notEmpty = lock.newCondition();//初始化非空等待队列
        notFull =  lock.newCondition();//初始化非满等待队列
    }

    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 {
            int i = 0;
            try {
                for (E e : c) {
                    checkNotNull(e);
                    items[i++] = e;
                }
            } catch (ArrayIndexOutOfBoundsException ex) {
                throw new IllegalArgumentException();
            }
            count = i;
            putIndex = (i == capacity) ? 0 : i;
        } finally {
            lock.unlock();
        }
    }
```

items 构造以后，大概是一个这样的数组结构



2，Add 方法

以 add 方法作为入口，在 add 方法中会调用父类的 add 方法，也就是 AbstractQueue.

```
public boolean add(E e) {
    if (offer(e))
        return true;
    else
        throw new IllegalStateException("Queue full");
}
```

从父类的 add 方法可以看到，这里做了一个队列是否满了的判断，如果队列满了直接抛出一个异常。

1）offer 方法

add 方法最终还是调用 offer 方法来添加数据，返回一个添加成功或者失败的布尔值反馈。

```
public boolean offer(E e) {
    checkNotNull(e); //判断添加的数据是否为空
    final ReentrantLock lock = this.lock; //添加重入锁
    lock.lock();
    try {
        if (count == items.length) //判断队列长度，如果队列长度等于数组长度，表示满了直接返回 false
            return false;
        else {
            enqueue(e); //直接调用 enqueue 将元素添加到队列中
            return true;
        }
    } finally {
        lock.unlock();
    }
}
```

2）enqueue

```
private void enqueue(E x) {

    final Object[] items = this.items;
    items[putIndex] = x; //通过 putIndex 对数据赋值
    if (++putIndex == items.length) // 当putIndex 等于数组长度时，将 putIndex 重置为0
        putIndex = 0;
    count++; //记录队列元素的个数
    notEmpty.signal(); //唤醒处于等待状态下的线程，表示当前队列中的元素不为空,如果存在消费者线程阻塞，就可以开始取出元素
}
```

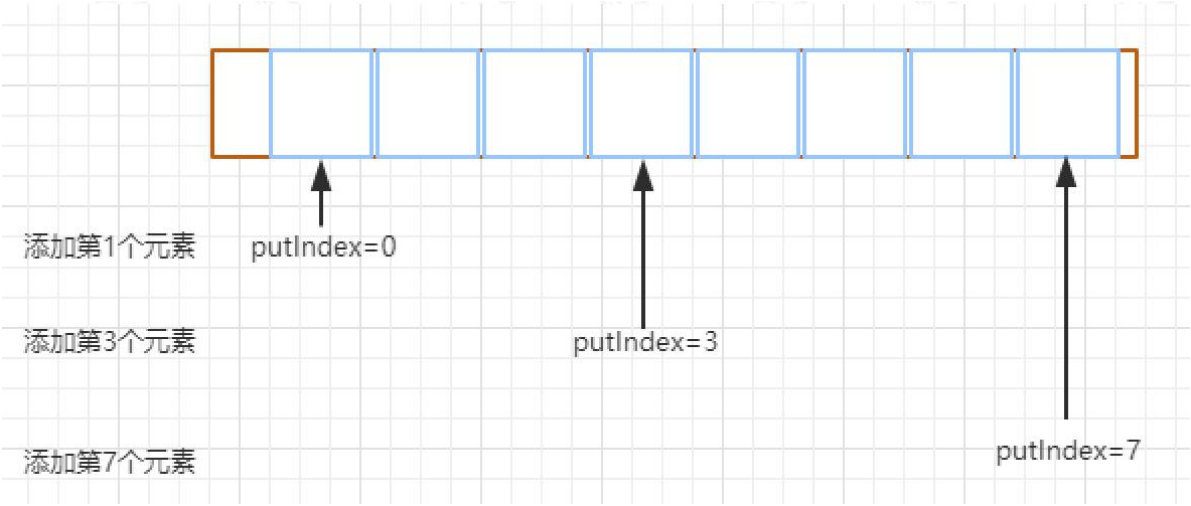
putIndex 为什么会在等于数组长度的时候重新设置为 0?

因为 ArrayBlockingQueue 是一个 FIFO 的队列，队列添加元素时，是从队尾获取 putIndex 来存储元素，当 putIndex等于数组长度时，下次就需要从数组头部开始添加了。

下面这个图模拟了添加到不同长度的元素时，putIndex 的变化，当 putIndex 等于数组长度时，不可能让 putIndex 继续累加，否则会超出数组初始化的容量大小。

1，当元素满了以后是无法继续添加的，因为会报错。

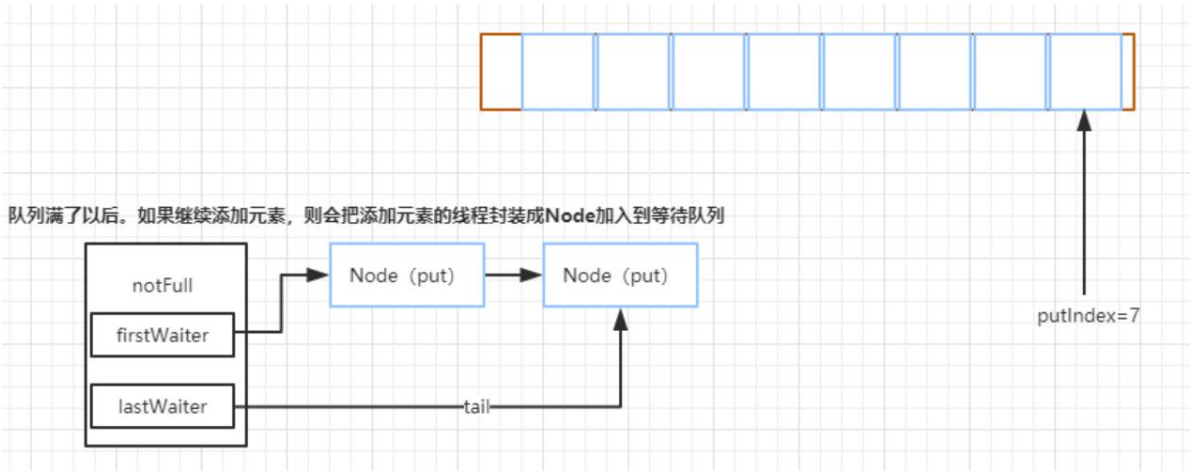
2，队列中的元素肯定会有一个消费者线程通过 take或者其他方法来获取数据，而获取数据的同时元素也会从队列中移除。



3，put 方法

put 方法和 add 方法功能一样，差异是 put 方法如果队列满了，会阻塞。这个在最开始的时候说过。接下来看一下它的实现逻辑

```
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();//这个也是获得锁，但是和 lock 的区别是，这个方法优先允许在等待时由其他线程调用等待线程的 interrupt 方法来中断等待直接返回。而 lock方法是尝试获得锁成功后才响应中断
    try {
        while (count == items.length)
            notFull.await();//队列满了的情况下，当前线程将会被 notFull 条件对象挂起加到等待队列中
        enqueue(e);
    } finally {
        lock.unlock();
    }
}
```

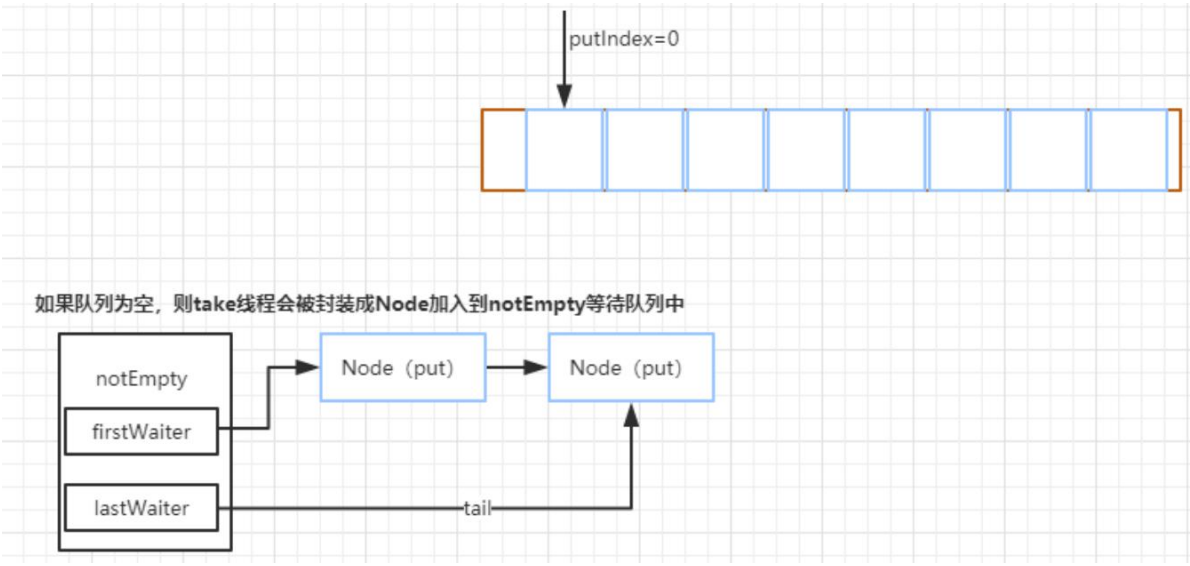


4，take 方法

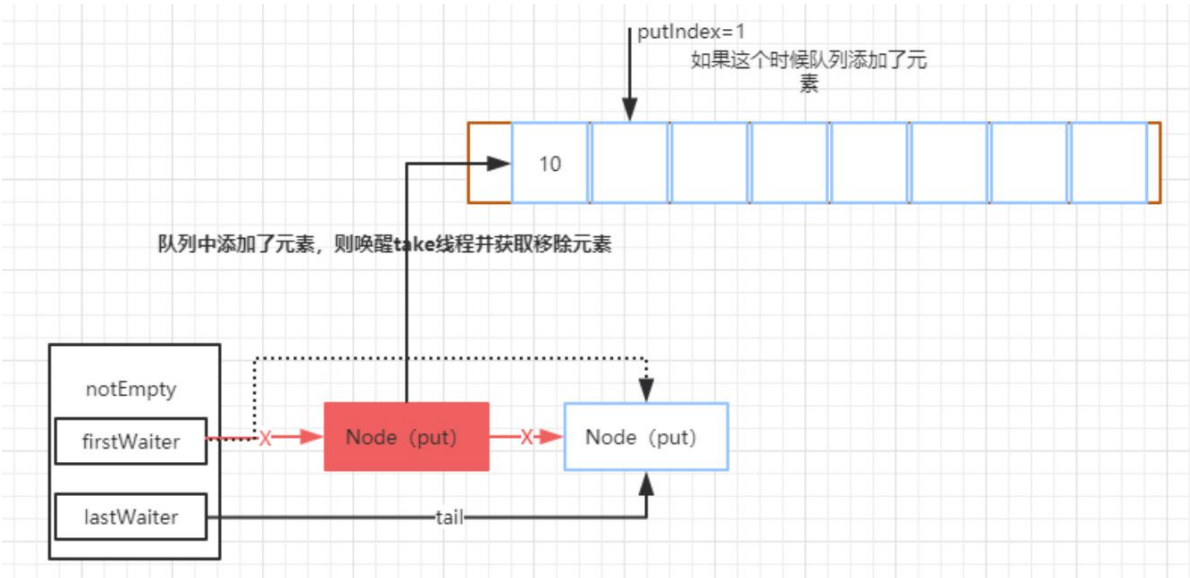
take 方法是一种阻塞获取队列中元素的方法。

它的实现原理很简单，有就删除没有就阻塞，注意这个阻塞是可以中断的，如果队列没有数据那么就加入 notEmpty条件队列等待(有数据就直接取走，方法结束)，如果有新的put 线程添加了数据，那么 put 操作将会唤醒 take 线程，执行 take 操作。

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();//如果队列为空的情况下，直接通过 await 方法阻塞
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```



如果队列中添加了元素，那么这个时候，会在 enqueue 中调用 notempty.signal 唤醒 take 线程来获得元素



1）dequeue 方法

这个是出队列的方法，主要是删除队列头部的元素并发返回给客户端。

takeIndex，是用来记录拿数据的索引值。

```
private E dequeue() {
    // assert lock.getHoldCount() == 1;
    // assert items[takeIndex] != null;
```

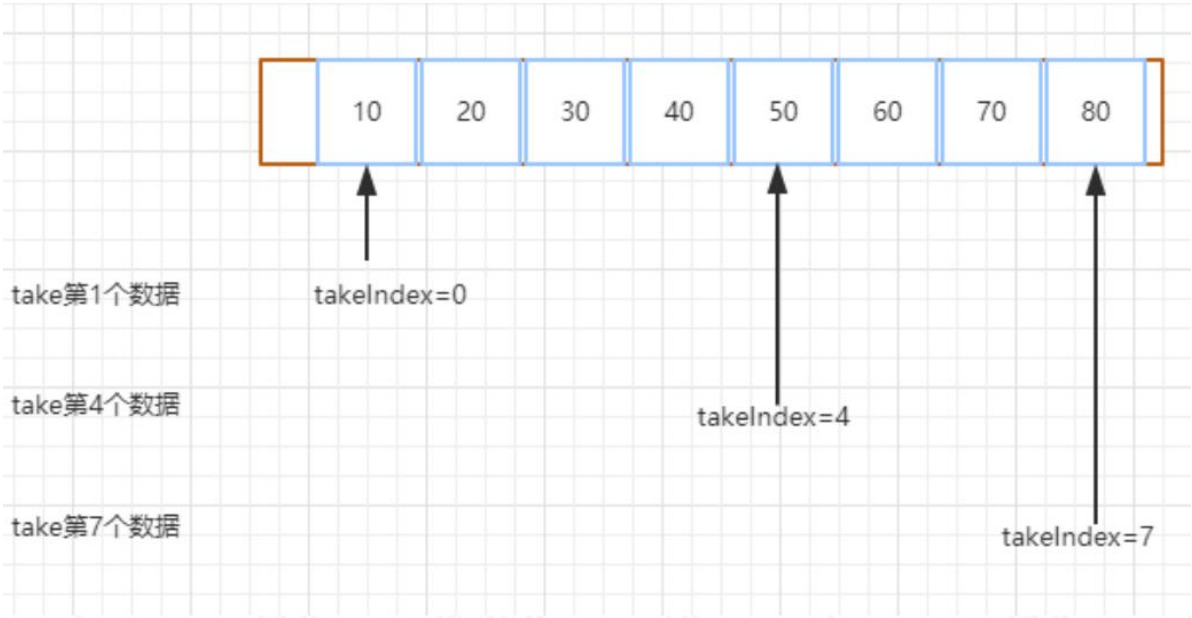
```
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
    E x = (E) items[takeIndex];//默认获取 0 位置的元素
    items[takeIndex] = null;//将该位置的元素设置为空
    if (++takeIndex == items.length)//这里的作用也是一样，如果拿到数组的最大值，那么重置为 0，继续从头部位置开始获取数据
        takeIndex = 0;
    count--;//记录 元素个数递减
    if (itrs != null)
        itrs.elementDequeued();//同时更新迭代器中的元素数据
    notFull.signal();//触发 因为队列满了以后导致的被阻塞的线程
    return x;
}
```

2）itrs.elementDequeued()

ArrayBlockingQueue 中，实现了迭代器的功能，也就是可以通过迭代器来遍历阻塞队列中的元素。

所以 itrs.elementDequeued() 是用来更新迭代器中的元素数据的。

takeIndex 的索引变化图如下，同时随着数据的移除，会唤醒处于 put 阻塞状态下的线程来继续添加数据。



5，remove 方法

remove 方法是移除一个指定元素。看看它的实现代码：

```
public boolean remove(Object o) {
    if (o == null) return false;
    final Object[] items = this.items;
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        if (count > 0) {
            final int putIndex = this.putIndex;
            int i = takeIndex;
            do {
                if (o.equals(items[i])) {
                    removeAt(i);
                    return true;
```

```
                }
                if (++i == items.length)
                    i = 0;
            } while (i != putIndex);
        }
        return false;
    } finally {
        lock.unlock();
    }
}
```

四，原子操作类

所谓的原子性表示一个或者多个操作，要么全部执行完，要么一个也不执行。不能出现成功一部分失败一部分的情况。

在多线程中，如果多个线程同时更新一个共享变量，可能会得到一个意料之外的值。比如 i=1 。A 线程更新 i+1 、B 线程也更新 i+1。

通过两个线程并行操作之后可能 i 的值不等于 3。而可能等于 2。因为 A 和 B 在更新变量 i 的时候拿到的 i 可能都是 1这就是一个典型的原子性问题。

从 JDK1.5 开始，在 J.U.C 包中提供了 Atomic 包，提供了对于常用数据结构的原子操作。它提供了简单、高效、以及线程安全的更新一个变量的方式。

1，JUC中的原子操作类

由于变量类型的关系，在 J.U.C 中提供了 12 个原子操作的类。这 12 个类可以分为四大类：

1，原子更新基本类型

AtomicBoolean、AtomicInteger、AtomicLong

2，原子更新数组

AtomicIntegerArray 、 AtomicLongArray 、AtomicReferenceArray

3，原子更新引用

AtomicReference 、 AtomicReferenceFieldUpdater 、AtomicMarkableReference（更新带有标记位的引用类型）

4，原子更新字段

AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicStampedReference

2，Atomicr Integer 原理分析

接下来，我们来剖析一下 AtomicInteger 的实现原理，仍然是基于我们刚刚在前面的案例中使用到的方法作为突破口

1）getAndIncrement

getAndIncrement 实际上是调用 unsafe 这个类里面提供的方法，这个类相当于是一个后门，使得 Java 可以像 C 语言的指针一样直接操作内存空间。当然也会带来一些弊端，就是指针的问题。

实际上这个类在很多方面都有使用，除了 J.U.C 这个包以外，还有 Netty、kafka 等等，这个类提供了很多功能，包括多线程同步(monitorEnter)、CAS 操 作 (compareAndSwap) 、 线 程 的 挂 起 和 恢 复 (park/unpark)、内存屏(loadFence/storeFence)内存管理（内存分配、释放内存、获取内存地址等.）

```
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
```

valueOffset，也比较熟了。通过 unsafe.objectFieldOffset()获取当前 Value 这个变量在内存中的偏移量，后续会基于这个偏移量从内存中得到value的值来和当前的值做比较，实现乐观锁。

```
private static final long valueOffset;
    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }
```

2）getAndAddInt

通过 do/while 循环，基于 CAS 乐观锁来做原子递增。实际上前面的 valueOffset 的作用就是从主内存中获得当前value 的值和预期值做一个比较，如果相等，对 value 做递增并结束循环。

```
public final int getAndAdd(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta);
}

    public final int getAndAddInt(Object var1, long var2, int var4) {
        int var5;
        do {
            var5 = this.getIntVolatile(var1, var2);
        } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

        return var5;
    }
```

3）get 方法

get 方法只需要直接返回 value 的值就行，这里的 value 是通过 Volatile 修饰的，用来保证可见性。

```
public final int get() {
    return value;
}
```

3，其他方法

它提供了 compareAndSet， 允许客户端基于AtomicInteger 来实现乐观锁的操作。

```
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}
```