java并发编程笔记— ArrayBlockingQueue

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | |  |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  |   Summary：摘要   |  | | --- | |  |   Reference：参考文献   |  | | --- | | [Collections Framework Overview](http://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html)  [不只是给面试加分 -- Java WeakReference的理解与使用](http://puretech.iteye.com/blog/2008663)  [理解 Java 的 GC 与 幽灵引用](http://www.iteye.com/topic/401478)  [无锁环形队列的一种高效实现](http://www.cnblogs.com/dodng/p/4367791.html)  [缓冲区设计--环形队列](http://blog.csdn.net/billow_zhang/article/details/4420789)  [浅析 Redis 复制](http://blog.jobbole.com/101678/)  [Java多线程系列--“JUC集合”07之 ArrayBlockingQueue](http://www.cnblogs.com/skywang12345/p/3498652.html) | |

目录

[1 描述 4](#_Toc462653468)

[1.1 BlockingQueue接口 4](#_Toc462653469)

[1.2 BlockingQueue实现类 4](#_Toc462653470)

[1.3 环形队列 5](#_Toc462653471)

[1.4 ArrayBlockingQueue 5](#_Toc462653472)

[2 类图 6](#_Toc462653473)

[3 主要实现 7](#_Toc462653474)

[3.1 基本属性定义 7](#_Toc462653475)

[3.2 基本方法定义 8](#_Toc462653476)

[3.3 构造器定义 9](#_Toc462653477)

[3.4 添加元素 10](#_Toc462653478)

[3.5 访问元素 11](#_Toc462653479)

[3.6 删除元素 14](#_Toc462653480)

[3.7 迭代器实现 16](#_Toc462653481)

[4 适用场景 28](#_Toc462653482)

[4.1 生产者消费者模型 28](#_Toc462653483)

[4.2 队列有界，不用担心内存溢出问题 28](#_Toc462653484)

[5 缺点&权衡点 28](#_Toc462653485)

[5.1 队列大小固定，使用前需要预估队列大小，以及生产者和消费者的数量； 28](#_Toc462653486)

[5.2 精确控制线程对元素的处理较为困难； 28](#_Toc462653487)

[5.3 吞吐量受线程互斥的限制； 28](#_Toc462653488)

[5.4 数据读写密切相关，难以划分模块； 28](#_Toc462653489)

[5.5 数据读写处理速率均匀，或者处理速率是否均匀非主要问题； 29](#_Toc462653490)

[5.6 需要同步处理数据； 29](#_Toc462653491)

[6 应用案例 29](#_Toc462653492)

[6.1 redis增量同步，使用环形队列存放增量数据 29](#_Toc462653493)

[7 相关知识点 29](#_Toc462653494)

[7.1 Guarded Suspension模型 29](#_Toc462653495)

[7.2 生产者-消费者模型 29](#_Toc462653496)

[7.3 弱引用 29](#_Toc462653497)

[7.4 虚假唤醒 30](#_Toc462653498)

[8 问题思考 30](#_Toc462653499)

# 描述

## BlockingQueue接口

1. 阻塞队列，作为队列的一种扩展实现，当队列为空时，阻塞取元素的方法；当队列满了时，阻塞添加元素的方法；
2. 阻塞队列提供四种类型的方法，如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Summary of BlockingQueue methods | | | | |
|  | *Throws exception* | *Special value* | *Blocks* | *Times out* |
| **Insert** | [add(e)](psi_element://java.util.concurrent.BlockingQueue#add(java.lang.Object)) | [offer(e)](psi_element://java.util.concurrent.BlockingQueue#offer(java.lang.Object)) | [put(e)](psi_element://java.util.concurrent.BlockingQueue#put(java.lang.Object)) | [offer(e, time, unit)](psi_element://java.util.concurrent.BlockingQueue#offer(java.lang.Object, long, java.util.concurrent.TimeUnit)) |
| **Remove** | [remove()](psi_element://java.util.concurrent.BlockingQueue#remove(java.lang.Object)) | [poll()](psi_element://java.util.concurrent.BlockingQueue#poll(long, java.util.concurrent.TimeUnit)) | [take()](psi_element://java.util.concurrent.BlockingQueue#take()) | [poll(time, unit)](psi_element://java.util.concurrent.BlockingQueue#poll(long, java.util.concurrent.TimeUnit)) |
| **Examine** | [element()](psi_element://java.util.Queue#element()) | [peek()](psi_element://java.util.Queue#peek()) | *not applicable* | *not applicable* |

1. 阻塞队列不接受null元素，如果add或者offer一个null元素，将会抛出NullPointerException；因为poll()使用null作为操作失败的标识；
2. 阻塞队列可以限制容量大小，如果元素数量到达容量，则添加元素的操作会被阻塞；默认大小为：Integer.MAX\_VALUE；
3. 阻塞队列主要被设计用来作为生产者-消费者模型的队列，同时队列实现了Collection接口，也可以进行其它的集合操作（比如：remove操作），但是效率一般都偏低，仅适合与偶尔进行此类操作；
4. 阻塞队列的实现类都是线程安全的，对于单个元素的操作都是原子的，但是对于批量操作，并不要求是原子操作；
5. Happens-Before规则：往队列中放入元素的操作happens-before于从队列中读取或者删除这个元素的操作；

## BlockingQueue实现类

|  |  |
| --- | --- |
| **实现类** | **描述** |
| ArrayBlockingQueue | 由数组实现的有界阻塞队列； |
| LinkedBlockingQueue | 由链表实现的有界阻塞队列，默认长度和最大长度为Integer.MAX\_VALUE； |
| PriorityBlockingQueue | 支持优先级排序的无界阻塞队列，默认元素自然顺序升序排列；可以通过实现元素的compareTo()方法来指定排序规则；不保证同优先级元素的顺序；元素必须实现Delayed接口； |
| DelayQueue | 使用PriorityQueue实现的无界阻塞队列；支持延迟获取元素； |
| SynchronousQueue | 不存放元素的阻塞队列，每个put操作必须等待一个take操作，否则无法添加元素；吞吐量高于ArrayBlockingQueue和LinkedBlockingQueue； |
| LinkedTransferQueue | 由链表实现的无界阻塞队列；增加了tryTransfer和transfer方法，会先尝试将put的元素直接传给正在等待take的线程，如果没有则将元素放入队尾；如果没有线程等待take，tryTransfer会返回false，transfer会阻塞等待； |
| LinkedBlockingDeque | 由链表实现的双向阻塞队列； |

## 环形队列

## ArrayBlockingQueue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0  元素5 | 1  元素6 | 2  元素7 | 3  putIndex | 4 |
| 15  元素4 |  |  |  | 5 |
| 14  元素3 |  |  |  | 6 |
| 13  元素2 |  |  |  | 7 |
| 12  元素1 | 11  元素0  takeIndex | 10 | 9 | 8 |

1. 通过数组实现的有界阻塞队列，长度固定，一旦创建则不能更改；
2. 底层通过名为items的Object[]数组存放元素，本质上是一个循环队列，通过putIndex和takeIndex标记队列的尾部和头部；如上图示意:
   1. 队列底层由长度为16的数组存放元素；
   2. 整个队列包含8个元素，第一个元素存放在items[11]，最后一个元素存放在items[2];
   3. takeIndex指向下一个要弹出队列的元素；putIndex指向下一个要放置元素的位置；
   4. 当takeIndex++等于items.length时，则重新指向items[0]，进入新一次循环；
   5. 当putIndex++等于items.length时，则重新指向items[0]，进入新一次循环；
3. ArrayBlockingQueue中的元素以FIFO方式排列，队列头中的元素是在队列中存在时间最长的元素，队列尾部元素是在队列中存在时间最短的元素；新元素从尾部插入，对元素的访问从头部操作；
4. 向满队列中put元素，或者从空队列中take元素，将会阻塞操作；
5. ArrayBlockingQueue支持以公平的方式让生产者或者消费者访问，默认不使用公平策略；使用公平策略，将按照FIFO的方式让线程访问队列，这回降低访问的吞吐量，但是能够防止饥饿；
6. ArrayBlockingQueue和其迭代器实现了Collection和Interator的所有可选方法；

# 类图

|  |
| --- |
|  |

主要角色：

1. Iterable接口：定义创建迭代器操作，赋予实现类创建迭代器的能力；
2. Collection接口：定义集合基本操作，所有集合类都需实现该接口；
3. Queue接口：定义队列基本操作，所有队列类都需要实现该接口；
4. Serializable接口：Serializable接口：标记接口，允许对象序列化；
5. BlockingQueue接口：阻塞队列定义接口
6. AbstractCollection类：实现Collection的基本框架；如果要实现不可修改的集合，需要继承该类并实现iterator()和size()方法；如果要实现可修改的集合，则还需要实现add()方法；
7. AbstractQueue类：实现Queue的基本框架；
8. Itr类：ArrayBlockingQueue的迭代器实现类；
9. Itrs类：用于维护ArrayBlockingQueue多个迭代器状态一致性的链表；
10. Node类：继承WeakReference<Itr>，作为Itr的弱引用链表节点，存放当前队列的迭代器的弱引用；

# 主要实现

## 基本属性定义

|  |
| --- |
| */\*\*  \* 存放底层元素的数组  \*/* **final** Object[] **items**;  */\*\*  \* 记录下一个消费者操作的索引,包括:take,poll,peek,remove  \*/* **int takeIndex**;  */\*\*  \* 记录下一个生产者操作的索引,包括:put,offer,add  \*/* **int putIndex**;  */\*\*  \* 记录当前队列中的元素素  \*/* **int count**;  */\*\*  \* 监视所有访问的锁  \*/* **final** ReentrantLock **lock**;  */\*\*  \* 用于阻塞take的条件  \*/* **private final** Condition **notEmpty**;  */\*\*  \* 用于阻塞put的条件  \*/* **private final** Condition **notFull**;  */\*\*  \* 维护迭代器状态的弱引用链表  \*/* **transient** Itrs **itrs** = **null**; |

1. **items属性：**存放ArrayBlockingQueue队列元素的底层数据结构；
2. **takeIndexs属性：**指向ArrayBlockingQueue的头索引，标识队列中下一个要弹出的元素（非空）；
3. **putIndexs属性：**指向ArrayBlockingQueue的尾索引，标识队列中下一个元素要存放的位置（为空）；
4. **count属性：**记录ArrayBlockingQueue存放的元素个数；
5. **lock属性：**ArrayBlockingQueue中的操作都是原子的、互斥的，通过lock保证每个操作的原子性；
6. **notEmpty属性：**用于阻塞take操作的条件；
7. **notFull属性：**用于阻塞put操作的条件；
8. **itrs属性：**维护迭代器状态的弱引用链表；

## 基本方法定义

|  |
| --- |
| */\*\*  \* 向队列中添加元素,需要在锁中执行,以保证原子性  \*/* **private void** enqueue(E x) { **final** Object[] items = **this**.**items**;   *//在下一个索引位置放入元素* items[**putIndex**] = x; **if** (++**putIndex** == items.**length**) {  **putIndex** = 0;  }   *//增加元素数,通知消费者线程* **count**++;  **notEmpty**.signal(); }  */\*\*  \* 从队列中取走元素,需要在锁中执行,以保证原子性  \*/* **private** E dequeue() {  **final** Object[] items = **this**.**items**;   *//取出元素,注意清除数组中相应的引用;* @SuppressWarnings(**"unchecked"**)  E x = (E) items[**takeIndex**];  items[**takeIndex**] = **null**; **if** (++**takeIndex** == items.**length**) {   **takeIndex** = 0;  }   *//减少元素数,通知迭代器和生产者线程* **count**--;  **if** (**itrs** != **null**) {  **itrs**.elementDequeued();  }  **notFull**.signal();  **return** x; } |

1. **enqueue方法：**入队列的基本操作，供其它操作调用，需要在锁中执行，执行完操作需要唤醒一个阻塞的消费者线程；
2. **dequeue方法：**出队列的基本操作，供其它操作调用，需要在锁中执行，执行完操作需要唤醒一个阻塞的生产者线程；

## 构造器定义

|  |
| --- |
| */\*\*  \* 根据容量初始化队列  \*/* **public** ArrayBlockingQueue(**int** capacity) {  **this**(capacity, **false**); }  */\*\*  \* 容量  \* 线程访问是否公平  \*/* **public** ArrayBlockingQueue(**int** capacity, **boolean** fair) {  *//容量必须大于0* **if** (capacity <= 0) {  **throw new** IllegalArgumentException();  }  *//初始化底层数组* **this**.**items** = **new** Object[capacity];  *//初始化锁* **lock** = **new** ReentrantLock(fair);  **notEmpty** = **lock**.newCondition();  **notFull** = **lock**.newCondition(); }  */\*\*  \* 容量  \* 线程访问是否公平  \* 队列中要包含的元素  \*/* **public** ArrayBlockingQueue(**int** capacity, **boolean** fair,  Collection<? **extends** E> c) {  *//执行初始化* **this**(capacity, fair);   *//执行拷贝操作,* **final** ReentrantLock lock = **this**.**lock**;   *//****TODO 通过加锁保持可见性 why?*** lock.lock(); *// Lock only for visibility, not mutual exclusion* **try** {  **int** i = 0;  **try** {  **for** (E e : c) {  *//不可以添加null  checkNotNull*(e);  **items**[i++] = e;  }  } **catch** (ArrayIndexOutOfBoundsException ex) {  **throw new** IllegalArgumentException();  }  **count** = i;  *//如果队列满了,则设置put索引位置指向队列尾* **putIndex** = (i == capacity) ? 0 : i;  } **finally** {  lock.unlock();  } } |

1. 因为ArrayBlockingQueue的容量是固定的，所以在初始化时需要指定容量；
2. 可以指定ArrayBlockingQueue是否允许每个线程公平访问，公平访问降低吞吐量，但是可以防止饥饿，底层实现实际是通过在创建时定义内部使用的lock是否为公平锁；
3. 可以拷贝集合参数中的元素创建ReentrantLock；

## 添加元素

|  |
| --- |
| */\*\*  \* 向队列尾部添加元素,如果事失败,抛出异常  \*/* **public boolean** add(E e) {  *//调用offer方法,因为ArrayBlockingQueue的offer是原子操作,故add方法也是原子的;* **return super**.add(e); }  */\*\*  \* 向队列尾部添加元素,如果失败,返回false;  \*/* **public boolean** offer(E e) {  *//参数检查  checkNotNull*(e);   *//添加元素过程加锁* **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  *//如果队列已满,则直接返回false* **if** (**count** == **items**.**length**) {  **return false**;  }  *//队列未满,执行入队操作* **else** {  enqueue(e);  **return true**;  }  } **finally** {  lock.unlock();  } }  */\*\*  \* 添加元素,如果队列满,则阻塞;  \*/* **public void** put(E e) **throws** InterruptedException {  *checkNotNull*(e);  **final** ReentrantLock lock = **this**.**lock**;  lock.lockInterruptibly();  **try** {  *//如果队列满,则阻塞等待;使用while循环而不是if,防止线程被过早唤醒* **while** (**count** == **items**.**length**) {  **notFull**.await();  }   *//唤醒后执行添加元素操作* enqueue(e);  } **finally** {  lock.unlock();  } }  */\*\*  \* 添加元素时队列已满,则等待一定时间,如果仍然满,则返回false  \*/* **public boolean** offer(E e, **long** timeout, TimeUnit unit)  **throws** InterruptedException {  *//参数检查  checkNotNull*(e);  **long** nanos = unit.toNanos(timeout);  **final** ReentrantLock lock = **this**.**lock**;  lock.lockInterruptibly();  **try** {  *//等待指定时长* **while** (**count** == **items**.**length**) {  **if** (nanos <= 0) {  **return false**;  }  nanos = **notFull**.awaitNanos(nanos);  }  enqueue(e);  **return true**;  } **finally** {  lock.unlock();  } } |

## 访问元素

|  |
| --- |
| */\*\*  \* 从队列中弹出一个元素,如果队列为空,返回null;  \*/* **public** E poll() {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  **return** (**count** == 0) ? **null** : dequeue();  } **finally** {  lock.unlock();  } }  */\*\*  \* 从队列中弹出一个元素,如果队列为空,返回阻塞等待;  \*/* **public** E take() **throws** InterruptedException {  **final** ReentrantLock lock = **this**.**lock**;   *//****TODO 锁为可中断锁 why?*** lock.lockInterruptibly();  **try** {  *//阻塞等待* **while** (**count** == 0) {  **notEmpty**.await();  }  *//弹出元素* **return** dequeue();  } **finally** {  lock.unlock();  } }  */\*\*  \* 从队列中弹出一个元素,如果队列为空,返回阻塞等待指定时间;  \*/* **public** E poll(**long** timeout, TimeUnit unit) **throws** InterruptedException {  **long** nanos = unit.toNanos(timeout);  **final** ReentrantLock lock = **this**.**lock**;  lock.lockInterruptibly();  **try** {  *//阻塞等待一定时间* **while** (**count** == 0) {  **if** (nanos <= 0) {  **return null**;  }  nanos = **notEmpty**.awaitNanos(nanos);  }  *//弹出元素* **return** dequeue();  } **finally** {  lock.unlock();  } }  */\*\*  \* 查看队列顶端元素;  \*/* **public** E peek() {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  **return** itemAt(**takeIndex**); *// null when queue is empty* } **finally** {  lock.unlock();  } }  */\*\*  \* 返回队列中的元素数  \*/* **public int** size() {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  **return count**;  } **finally** {  lock.unlock();  } }  */\*\*  \* 返回队列剩余容量  \*/* **public int** remainingCapacity() {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  **return items**.**length** - **count**;  } **finally** {  lock.unlock();  } }  */\*\*  \* 将队列中元素导出到结合中  \*/* **public int** drainTo(Collection<? **super** E> c) {  **return** drainTo(c, Integer.***MAX\_VALUE***); }  */\*\*  \* 将队列中元素导出到结合中  \*/* **public int** drainTo(Collection<? **super** E> c, **int** maxElements) {  *checkNotNull*(c);  **if** (c == **this**) {  **throw new** IllegalArgumentException();  }  **if** (maxElements <= 0) {  **return** 0;  }  **final** Object[] items = **this**.**items**;  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  *//如果给出索引大于count,则设置索引为count* **int** n = Math.*min*(maxElements, **count**);  **int** take = **takeIndex**;  **int** i = 0;  **try** {  **while** (i < n) {  @SuppressWarnings(**"unchecked"**)  E x = (E) items[take];  c.add(x);  items[take] = **null**;  **if** (++take == items.**length**) {  take = 0;  }  i++;  }  **return** n;  } **finally** {  *// Restore invariants even if c.add() threw* **if** (i > 0) {  **count** -= i;  **takeIndex** = take;  **if** (**itrs** != **null**) {  **if** (**count** == 0) {  **itrs**.queueIsEmpty();  } **else if** (i > take) {  **itrs**.takeIndexWrapped();  }  }  *//通知被阻塞的生产者线程* **for** (; i > 0 && lock.hasWaiters(**notFull**); i--) {  **notFull**.signal();  }  }  }  } **finally** {  lock.unlock();  } }  */\*\*  \* 判断是否包含指定元素  \*/* **public boolean** contains(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])) {  **return true**;  }  **if** (++i == items.**length**) {  i = 0;  }  } **while** (i != putIndex);  }  **return false**;  } **finally** {  lock.unlock();  } } |

## 删除元素

|  |
| --- |
| */\*\*  \* 根据索引删除元素,需要在锁中执行,以保证原子性  \*/* **void** removeAt(**final int** removeIndex) {   **final** Object[] items = **this**.**items**;   *//如果删除索引在队列头部,则直接删除元素* **if** (removeIndex == **takeIndex**) {  *// removing front item; just advance* items[**takeIndex**] = **null**;  **if** (++**takeIndex** == items.**length**) {  **takeIndex** = 0;  }  **count**--;  **if** (**itrs** != **null**) {  **itrs**.elementDequeued();  }  }  *//* **else** {  *// an "interior" remove   // slide over all others up through putIndex.* **final int** putIndex = **this**.**putIndex**;  **for** (**int** i = removeIndex; ; ) {  **int** next = i + 1;  *//如果下一个索引指向队列尾部,则重新指向头部* **if** (next == items.**length**) {  next = 0;  }   *//如果下一个索引不等于* **if** (next != putIndex) {  items[i] = items[next];  i = next;  } **else** {  items[i] = **null**;  **this**.**putIndex** = i;  **break**;  }  }  **count**--;  **if** (**itrs** != **null**) {  **itrs**.removedAt(removeIndex);  }  }  **notFull**.signal(); }  */\*\*  \* 在队列中删除指定元素,该操作性能较差,不推荐在正常环境下频繁使用;  \*/* **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();  } }  */\*\*  \* 清空队列元素  \*/* **public void** clear() {  **final** Object[] items = **this**.**items**;  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  **int** k = **count**;  **if** (k > 0) {  **final int** putIndex = **this**.**putIndex**;  **int** i = **takeIndex**;  **do** {  items[i] = **null**;  **if** (++i == items.**length**) {  i = 0;  }  } **while** (i != putIndex);  **takeIndex** = putIndex;  **count** = 0;  **if** (**itrs** != **null**) {  **itrs**.queueIsEmpty();  }  **for** (; k > 0 && lock.hasWaiters(**notFull**); k--) {  **notFull**.signal();  }  }  } **finally** {  lock.unlock();  } } |

注：ArrayBlockingQueue的remove方法性能不高，故不要在正常环境下频繁使用；删除元素后，需要通知被阻塞的生产者线程；

## 迭代器实现

|  |
| --- |
| */\*\*  \* 创建迭代器  \*/* **public** Iterator<E> iterator() {  **return new** Itr(); }  */\*\*  \* 用于维护队列对应的多个迭代器的状态的一致性;  \* <p>  \* 维护一致性的方式如下:  \* (1) 使用cycles记录takeIndex循环到0的次数;  \* (2) 当删除元素时,通过回调函数通知所有的迭代器;  \* <p>  \* Itrs使用弱引用迭代器的链表维护迭代器,并进行状态检测和修正,方式如下:  \* (1) 初始化迭代器状态:创建迭代器时,检测迭代器状态和队列是否一致;  \* (2) 防止迭代器在takeIndex循环好几圈以后,仍未使用:takeIndex每一次循环到0时,都检测迭代器状态和队列是否一致;  \* (3) 当队列为空时,通知所有迭代器,废弃现有数据;  \* (4) 引入shutdown和takeIndexWrapped用于移除数据不一致的迭代器;  \*/* **class** Itrs {   */\*\*  \* 定义链表中的节点,存放定义的迭代器的引用,本身为弱引用,故在没有强引用时,会被回收;  \*/* **private class** Node **extends** WeakReference<Itr> {   */\*\*  \* 指向链表的下一个节点  \*/* Node **next**;   */\*\*  \** ***@param iterator*** *节点存放迭代器的引用  \** ***@param next*** *节点的下一级节点  \*/* Node(Itr iterator,  Node next) {  **super**(iterator);  **this**.**next** = next;  }  }   */\*\*  \* 循环次数,takeIndex每次循环到0时,加1  \*/* **int cycles** = 0;   */\*\*  \* 存放弱引用迭代器的链表  \*/* **private** Node **head**;   */\*\*  \* 用来清理存放旧数据的弱引用迭代器  \*/* **private** Node **sweeper** = **null**;   **private static final int *SHORT\_SWEEP\_PROBES*** = 4;  **private static final int *LONG\_SWEEP\_PROBES*** = 16;   */\*\*  \* 初始化时,需添加一个迭代器  \*/* Itrs(Itr initial) {  register(initial);  }   */\*\*  \* 扫描迭代器,查找旧数据迭代器;如果找到至少1个,则可以指定努力寻找更多的旧数据迭代器;  \* 在迭代线程中调用;  \*  \** ***@param tryHarder*** *指定是否努力寻找更多旧数据迭代器  \*/* **void** doSomeSweeping(**boolean** tryHarder) {  *// assert lock.getHoldCount() == 1;  // assert head != null;* **int** probes = tryHarder ? ***LONG\_SWEEP\_PROBES*** : ***SHORT\_SWEEP\_PROBES***;  Node o, p;  **final** Node sweeper = **this**.**sweeper**;   *//标记是否进行一起全量清理* **boolean** passedGo; *// to limit search to one full sweep   //如果目前没有标记旧数据迭代器,则从头开始查找;* **if** (sweeper == **null**) {  o = **null**;  p = **head**;  passedGo = **true**;  }  *//如果之前已经扫描过,并且有旧数据迭代器,则从下一个节点开始* **else** {  o = sweeper;  p = o.**next**;  passedGo = **false**;  }   *//扫描指定次数* **for** (; probes > 0; probes--) {   *//如果扫描到尾节点,则从头开始* **if** (p == **null**) {  **if** (passedGo) {  **break**;  }  o = **null**;  p = **head**;  passedGo = **true**;  }   **final** Itr it = p.get();  **final** Node next = p.**next**;   *//如果迭代器处于detached状态,则表示找到了旧数据迭代器* **if** (it == **null** || it.isDetached()) {  *// 找到了旧数据迭代器,则增加扫描次数* probes = ***LONG\_SWEEP\_PROBES***; *// "try harder"  // unlink p* p.clear();  p.**next** = **null**;  **if** (o == **null**) {  **head** = next;  **if** (next == **null**) {  *// We've run out of iterators to track; retire* **itrs** = **null**;  **return**;  }  } **else** {  o.**next** = next;  }  } **else** {  o = p;  }  p = next;  }   **this**.**sweeper** = (p == **null**) ? **null** : o;  }   */\*\*  \* 向Itrs链表中添加迭代器  \*/* **void** register(Itr itr) {  **head** = **new** Node(itr, **head**);  }   */\*\*  \* 每次takeIndex循环到0时,通知所有迭代器,并清理旧数据迭代器  \*/* **void** takeIndexWrapped() {  *// assert lock.getHoldCount() == 1;* **cycles**++;  **for** (Node o = **null**, p = **head**; p != **null**; ) {  **final** Itr it = p.get();  **final** Node next = p.**next**;  **if** (it == **null** || it.takeIndexWrapped()) {  *// unlink p  // assert it == null || it.isDetached();* p.clear();  p.**next** = **null**;  **if** (o == **null**) {  **head** = next;  } **else** {  o.**next** = next;  }  } **else** {  o = p;  }  p = next;  }  **if** (**head** == **null**) *// no more iterators to track* {  **itrs** = **null**;  }  }   */\*\*  \* Called whenever an interior remove (not at takeIndex) occurred.  \* <p>  \* Notifies all iterators, and expunges any that are now stale.  \*/* **void** removedAt(**int** removedIndex) {  **for** (Node o = **null**, p = **head**; p != **null**; ) {  **final** Itr it = p.get();  **final** Node next = p.**next**;  **if** (it == **null** || it.removedAt(removedIndex)) {  *// unlink p  // assert it == null || it.isDetached();* p.clear();  p.**next** = **null**;  **if** (o == **null**) {  **head** = next;  } **else** {  o.**next** = next;  }  } **else** {  o = p;  }  p = next;  }  **if** (**head** == **null**) *// no more iterators to track* {  **itrs** = **null**;  }  }   */\*\*  \* Called whenever the queue becomes empty.  \* <p>  \* Notifies all active iterators that the queue is empty,  \* clears all weak refs, and unlinks the itrs datastructure.  \*/* **void** queueIsEmpty() {  *// assert lock.getHoldCount() == 1;* **for** (Node p = **head**; p != **null**; p = p.**next**) {  Itr it = p.get();  **if** (it != **null**) {  p.clear();  it.shutdown();  }  }  **head** = **null**;  **itrs** = **null**;  }   */\*\*  \* Called whenever an element has been dequeued (at takeIndex).  \*/* **void** elementDequeued() {  *// assert lock.getHoldCount() == 1;* **if** (**count** == 0) {  queueIsEmpty();  } **else if** (**takeIndex** == 0) {  takeIndexWrapped();  }  } }  */\*\*  \* ArrayBlockingQueue的迭代器  \* 为了保证take和put的弱一致性,防止hasNext方法和next的结果不一致,迭代器会预先读取一个位置;  \*/* **private class** Itr **implements** Iterator<E> {  */\*\*  \* 存放下一个元素的位置索引,如果为NONE,则表示已经迭代完;  \*/* **private int cursor**;   */\*\*  \* 存放next()将要返回的元素;如果没有,则为null  \*/* **private** E **nextItem**;   */\*\*  \* 下一个元素的索引: NONE,没有下一个元素; REMOVED,在其它地方删除了;  \*/* **private int nextIndex**;   */\*\*  \* 存放上一次返回的元素,null if none or not detached.  \*/* **private** E **lastItem**;   */\*\*  \* 存放上一次返回的元素索引: NONE,没有上一个元素; REMOVED,在其它地方删除了;  \*/* **private int lastRet**;   */\*\*  \* takeIndex的上一个值,当迭代器处于detached状态时,值为DETACHED  \*/* **private int prevTakeIndex**;   */\*\*  \* iters.cycles的上一个值,用于结合prevTakeIndex判断迭代器的数据状态和队列数据状态是否一致;  \* 因为队列为循环队列,故单凭prevTakeIndex无法标记队列数据状态是否发生过变化;  \*/* **private int prevCycles**;   */\*\*  \* 未定义/不可用/迭代完  \*/* **private static final int *NONE*** = -1;   */\*\*  \* 标记元素被this.remove()以外的方法删除了  \*/* **private static final int *REMOVED*** = -2;   */\*\*  \* 标识迭代器处于detached状态  \*/* **private static final int *DETACHED*** = -3;   Itr() {  *// assert lock.getHoldCount() == 0;  //初始化迭代器为不可用状态* **lastRet** = ***NONE***;   **final** ReentrantLock lock = ArrayBlockingQueue.**this**.**lock**;  lock.lock();  **try** {  *//如果队列为空,则初始化迭代器状态* **if** (**count** == 0) {  *// assert itrs == null;* **cursor** = ***NONE***;  **nextIndex** = ***NONE***;  **prevTakeIndex** = ***DETACHED***;  }  *//如果队列不空,* **else** {  **final int** takeIndex = ArrayBlockingQueue.**this**.**takeIndex**;  **prevTakeIndex** = takeIndex;   *//获取下一个要读取的元素* **nextItem** = itemAt(**nextIndex** = takeIndex);  *//校正迭代器索引* **cursor** = incCursor(takeIndex);   *//如果是队列的第一个迭代器,则初始化Itrs链表中* **if** (**itrs** == **null**) {  **itrs** = **new** Itrs(**this**);  }  *//如果不是第一个迭代器,则添加迭代器到Itrs链表中* **else** {  **itrs**.register(**this**); *// in this order  //清理旧数据迭代器* **itrs**.doSomeSweeping(**false**);  }   *//记录迭代器当前的循环次数* **prevCycles** = **itrs**.**cycles**;  *// assert takeIndex >= 0;  // assert prevTakeIndex == takeIndex;  // assert nextIndex >= 0;  // assert nextItem != null;* }  } **finally** {  lock.unlock();  }  }   */\*\*  \* 判断迭代器是否处于DETACHED状态  \*/* **boolean** isDetached() {  *// assert lock.getHoldCount() == 1;* **return prevTakeIndex** < 0;  }   */\*\*  \* 校准迭代器的index  \*/* **private int** incCursor(**int** index) {  *// assert lock.getHoldCount() == 1;  //如果到了队列尾,则从头开始* **if** (++index == **items**.**length**) {  index = 0;  }   *//如果索引已经到达下一次要放的元素位置,则标示迭代器完成迭代* **if** (index == **putIndex**) {  index = ***NONE***;  }  **return** index;  }   */\*\*  \* 标记给定索引是否有效;  \* 如果索引所在位置已经被新的元素覆盖过,则视作无效;  \*/* **private boolean** invalidated(**int** index, **int** prevTakeIndex,  **long** dequeues, **int** length) {  **if** (index < 0) {  **return false**;  }   *//计算给定索引到队列头的距离* **int** distance = index - prevTakeIndex;  **if** (distance < 0) {  distance += length;  }   **return** dequeues > distance;  }   */\*\*  \* 标记所有不一致的出队列操作,只在迭代线程中运行  \*/* **private void** incorporateDequeues() {  *// assert lock.getHoldCount() == 1;  // assert itrs != null;  // assert !isDetached();  // assert count > 0;* **final int** cycles = **itrs**.**cycles**;  **final int** takeIndex = ArrayBlockingQueue.**this**.**takeIndex**;  **final int** prevCycles = **this**.**prevCycles**;  **final int** prevTakeIndex = **this**.**prevTakeIndex**;   *//如果迭代器中的出队状况和队列不一致* **if** (cycles != prevCycles || takeIndex != prevTakeIndex) {  **final int** len = **items**.**length**;   *//计算出队列元素的数量* **long** dequeues = (cycles - prevCycles) \* len  + (takeIndex - prevTakeIndex);   *// 检测上一个元素索引的有效性* **if** (invalidated(**lastRet**, prevTakeIndex, dequeues, len)) {  **lastRet** = ***REMOVED***;  }  *// 检测下一个元素索引的有效性* **if** (invalidated(**nextIndex**, prevTakeIndex, dequeues, len)) {  **nextIndex** = ***REMOVED***;  }  *// 检测当前索引的有效性* **if** (invalidated(**cursor**, prevTakeIndex, dequeues, len)) {  **cursor** = takeIndex;  }   *//如果所有标记索引都失效,则表明迭代器数据不一致* **if** (**cursor** < 0 && **nextIndex** < 0 && **lastRet** < 0) {  detach();  } **else** {  **this**.**prevCycles** = cycles;  **this**.**prevTakeIndex** = takeIndex;  }  }  }   */\*\*  \* 标记迭代器为detached装填,itrs将不会追踪调用该方法的迭代器的状态;只在迭代器线程中调用;  \*/* **private void** detach() {  *// Switch to detached mode  // assert lock.getHoldCount() == 1;  // assert cursor == NONE;  // assert nextIndex < 0;  // assert lastRet < 0 || nextItem == null;  // assert lastRet < 0 ^ lastItem != null;* **if** (**prevTakeIndex** >= 0) {  *// assert itrs != null;* **prevTakeIndex** = ***DETACHED***;  *// try to unlink from itrs (but not too hard)* **itrs**.doSomeSweeping(**true**);  }  }   **public boolean** hasNext() {  *// assert lock.getHoldCount() == 0;* **if** (**nextItem** != **null**) {  **return true**;  }   *//如果没有元素,则设置当前迭代器为detached状态* noNext();  **return false**;  }   **private void** noNext() {  **final** ReentrantLock lock = ArrayBlockingQueue.**this**.**lock**;  lock.lock();  **try** {  *// assert cursor == NONE;  // assert nextIndex == NONE;* **if** (!isDetached()) {  *// assert lastRet >= 0;* incorporateDequeues(); *// might update lastRet* **if** (**lastRet** >= 0) {  **lastItem** = itemAt(**lastRet**);  *// assert lastItem != null;* detach();  }  }  *// assert isDetached();  // assert lastRet < 0 ^ lastItem != null;* } **finally** {  lock.unlock();  }  }   **public** E next() {  *// assert lock.getHoldCount() == 0;* **final** E x = **nextItem**;  **if** (x == **null**) {  **throw new** NoSuchElementException();  }  **final** ReentrantLock lock = ArrayBlockingQueue.**this**.**lock**;  lock.lock();  **try** {  **if** (!isDetached()) {  incorporateDequeues();  }  *// assert nextIndex != NONE;  // assert lastItem == null;* **lastRet** = **nextIndex**;  **final int** cursor = **this**.**cursor**;  **if** (cursor >= 0) {  **nextItem** = itemAt(**nextIndex** = cursor);  *// assert nextItem != null;* **this**.**cursor** = incCursor(cursor);  } **else** {  **nextIndex** = ***NONE***;  **nextItem** = **null**;  }  } **finally** {  lock.unlock();  }  **return** x;  }   **public void** remove() {  *// assert lock.getHoldCount() == 0;* **final** ReentrantLock lock = ArrayBlockingQueue.**this**.**lock**;  lock.lock();  **try** {  **if** (!isDetached()) {  incorporateDequeues(); *// might update lastRet or detach* }   **final int** lastRet = **this**.**lastRet**;  **this**.**lastRet** = ***NONE***;  **if** (lastRet >= 0) {  **if** (!isDetached()) {  removeAt(lastRet);  } **else** {  **final** E lastItem = **this**.**lastItem**;  *// assert lastItem != null;* **this**.**lastItem** = **null**;  **if** (itemAt(lastRet) == lastItem) {  removeAt(lastRet);  }  }  } **else if** (lastRet == ***NONE***) {  **throw new** IllegalStateException();  }  *// else lastRet == REMOVED and the last returned element was  // previously asynchronously removed via an operation other  // than this.remove(), so nothing to do.* **if** (**cursor** < 0 && **nextIndex** < 0) {  detach();  }  } **finally** {  lock.unlock();  *// assert lastRet == NONE;  // assert lastItem == null;* }  }   */\*\*  \* Called to notify the iterator that the queue is empty, or that it  \* has fallen hopelessly behind, so that it should abandon any  \* further iteration, except possibly to return one more element  \* from next(), as promised by returning true from hasNext().  \*/* **void** shutdown() {  *// assert lock.getHoldCount() == 1;* **cursor** = ***NONE***;  **if** (**nextIndex** >= 0) {  **nextIndex** = ***REMOVED***;  }  **if** (**lastRet** >= 0) {  **lastRet** = ***REMOVED***;  **lastItem** = **null**;  }  **prevTakeIndex** = ***DETACHED***;  *// Don't set nextItem to null because we must continue to be  // able to return it on next().  //  // Caller will unlink from itrs when convenient.* }   **private int** distance(**int** index, **int** prevTakeIndex, **int** length) {  **int** distance = index - prevTakeIndex;  **if** (distance < 0) {  distance += length;  }  **return** distance;  }   */\*\*  \* Called whenever an interior remove (not at takeIndex) occurred.  \*  \** ***@return*** *true if this iterator should be unlinked from itrs  \*/* **boolean** removedAt(**int** removedIndex) {  *// assert lock.getHoldCount() == 1;* **if** (isDetached()) {  **return true**;  }   **final int** cycles = **itrs**.**cycles**;  **final int** takeIndex = ArrayBlockingQueue.**this**.**takeIndex**;  **final int** prevCycles = **this**.**prevCycles**;  **final int** prevTakeIndex = **this**.**prevTakeIndex**;  **final int** len = **items**.**length**;  **int** cycleDiff = cycles - prevCycles;  **if** (removedIndex < takeIndex) {  cycleDiff++;  }  **final int** removedDistance =  (cycleDiff \* len) + (removedIndex - prevTakeIndex);  *// assert removedDistance >= 0;* **int** cursor = **this**.**cursor**;  **if** (cursor >= 0) {  **int** x = distance(cursor, prevTakeIndex, len);  **if** (x == removedDistance) {  **if** (cursor == **putIndex**) {  **this**.**cursor** = cursor = ***NONE***;  }  } **else if** (x > removedDistance) {  *// assert cursor != prevTakeIndex;* **this**.**cursor** = cursor = dec(cursor);  }  }  **int** lastRet = **this**.**lastRet**;  **if** (lastRet >= 0) {  **int** x = distance(lastRet, prevTakeIndex, len);  **if** (x == removedDistance) {  **this**.**lastRet** = lastRet = ***REMOVED***;  } **else if** (x > removedDistance) {  **this**.**lastRet** = lastRet = dec(lastRet);  }  }  **int** nextIndex = **this**.**nextIndex**;  **if** (nextIndex >= 0) {  **int** x = distance(nextIndex, prevTakeIndex, len);  **if** (x == removedDistance) {  **this**.**nextIndex** = nextIndex = ***REMOVED***;  } **else if** (x > removedDistance) {  **this**.**nextIndex** = nextIndex = dec(nextIndex);  }  } **else if** (cursor < 0 && nextIndex < 0 && lastRet < 0) {  **this**.**prevTakeIndex** = ***DETACHED***;  **return true**;  }  **return false**;  }   */\*\*  \* Called whenever takeIndex wraps around to zero.  \*  \** ***@return*** *true if this iterator should be unlinked from itrs  \*/* **boolean** takeIndexWrapped() {  *// assert lock.getHoldCount() == 1;* **if** (isDetached()) {  **return true**;  }  **if** (**itrs**.**cycles** - **prevCycles** > 1) {  *// All the elements that existed at the time of the last  // operation are gone, so abandon further iteration.* shutdown();  **return true**;  }  **return false**;  }   *// /\*\* Uncomment for debugging. \*/  // public String toString() {  // return ("cursor=" + cursor + " " +  // "nextIndex=" + nextIndex + " " +  // "lastRet=" + lastRet + " " +  // "nextItem=" + nextItem + " " +  // "lastItem=" + lastItem + " " +  // "prevCycles=" + prevCycles + " " +  // "prevTakeIndex=" + prevTakeIndex + " " +  // "size()=" + size() + " " +  // "remainingCapacity()=" + remainingCapacity());  // }* } |

# 适用场景

## 生产者消费者模型

使用ArrayBlockingQueue，能够解耦元素的生产者和元素的消费者。生产者和消费者之间可以不用在意彼此的实现逻辑，从而更加专注于业务实现，由ArrayBlockingQueue作为消息传递的桥梁。

## 队列有界，不用担心内存溢出问题

因为ArrayBlockingQueue是有界的，故不用担心积压过多导致内存溢出的问题。

# 缺点&权衡点

## 队列大小固定，使用前需要预估队列大小，以及生产者和消费者的数量；

有利就有弊，虽然队列有界能够很好的控制内存的使用，但是增加了实现的复杂性。使用时需要根据实际业务需要调整队列的大小。同时，需要合理的划分生产者和消费者，保持合理的线程数量，防止生产过快或者消费过快造成线程长时间等待的情况出现。这个过程可能需要多次实践，才能得出经验值。

## 精确控制线程对元素的处理较为困难；

无论从生产者的角度还是消费者的角度去看，数据处理线程总是对等的，如果存在特定需求的差异，则不容易控制。比如：指定某几个元素数据只能由某一线程生产，某一线程消费，则无法控制。对于这种需求的处理，只能通过扩展整个生产者或者消费者线程的功能，以便兼容新需求，违背单一职责原则。

## 吞吐量受线程互斥的限制；

ArrayBlockingQueue的操作都是互斥的，同一时间只有一个线程可以获得锁，故会影响数据处理的吞吐量；故如果非必要，可以考虑无锁实现，即将数据提前分片，不同的分片交由不同的线程处理，线程之间彼此没有同步操作。

## 数据读写密切相关，难以划分模块；

如果数据读写处理逻辑相关性很强，无法进一步细分，则不适合使用ArrayBlockingQueue。

## 数据读写处理速率均匀，或者处理速率是否均匀非主要问题；

工具或者技术的引入总是为了解决某一个类问题，如果数据处理速率均匀，或者数据处理速率是否均匀并非主要问题，则可以不考虑使用ArrayBlockingQueue。

## 需要同步处理数据；

数据生产者只是将信息push到队列中，并不能得到消费者的处理响应信息，只能适合于异步的信息传递的情况。

# 应用案例

## redis增量同步，使用环形队列存放增量数据

1. redis增量同步时，使用积压队列存放增量数据。积压队列本身就是一个环形队列；
2. 积压队列在本质上是一个固定长度的先进先出队列，默认大小为1MB；可以通过repl-backlog-size配置来调整；
3. 积压队列仅在第一次使用时初始化，积压队列越大，允许的主从断连时间越长；故需要根据主从之间的网络状态设置合适的大小；
4. 因为积压队列存储的内容是命令本身，所以估算积压队列的大小只需要估算主从断连期间，执行命令的大小即可；
5. repl-backlog-ttl配置设置当所有slave都与master断连后，经过多长时间可以释放master的backlog空间；默认为1h，如果设置为0,则表示永远不释放；

# 相关知识点

## Guarded Suspension模型

阻塞队列的实现体现了Guarded Suspension的思路，当队列为空时，挂起执行出队操作的线程，直到队列中有新的元素放入，再唤醒线程；当队列满了时，挂起执行入队操作的线程，指导队列中有空位置时，再唤醒线程；

## 生产者-消费者模型

ArrayBlockingQueue可以用作生产者-消费者模型的阻塞队列；

## 弱引用

Itrs中使用弱引用存放迭代器引用，便于当迭代器不再使用时，能够被GC回收；

## 虚假唤醒

因为入队、出队操作涉及到线程的等待和唤醒，故需要考虑使用防范虚假唤醒问题；使用while循环而非if语句进行条件判断。

# 问题思考

1. 为什么put、offer（带超时）、take、poll（带超时）方法中，判断队列空/满的逻辑要放到while循环中而非if语句中？

|  |
| --- |
|  |