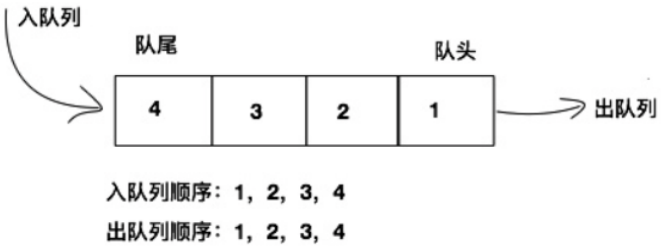
主讲老师：Fox老师

> 有道笔记链接：https://note.youdao.com/s/U6CaYXOB

1. 阻塞队列介绍

1.1 队列

- 是限定在一端进行插入，另一端进行删除的特殊线性表。
- 先进先出(FIFO)线性表。
- 允许出队的一端称为队头，允许入队的一端称为队尾。



Queue接口

```
public interface Queue<E> extends Collection<E> {
    //添加一个元素，添加成功返回true，如果队列满了，就会抛出异常
    boolean add(E e);
    //添加一个元素，添加成功返回true，如果队列满了，返回false
    boolean offer(E e);
    //返回并删除队首元素，队列为空则抛出异常
    E remove();
    //返回并删除队首元素，队列为空则返回null
    E poll();
    //返回队首元素，但不移除，队列为空则抛出异常
    E element();
    //获取队首元素，但不移除，队列为空则返回null
    E peek();
}
```

1.2 阻塞队列

阻塞队列 (BlockingQueue)是Java util.concurrent包下重要的数据结构，BlockingQueue提供了线程安全的队列访问方式：当阻塞队列插入数据时，如果队列已满，线程将会阻塞等待直到队列非满；从

阻塞队列取数据时，如果队列已空，线程将会阻塞等待直到队列非空。并发包下很多高级同步类的实现都是基于BlockingQueue实现的。

BlockingQueue接口

方法	抛出异常	返回特定值	阻塞	阻塞特定时间
入队	add(e)	offer(e)	put(e)	offer(e, time, unit)
出队	remove()	poll()	take()	poll(time, unit)
获取队首元素	element()	peek()	不支持	不支持

应用场景

阻塞队列在实际应用中有很多场景，以下是一些常见的应用场景：

1. 线程池

线程池中的任务队列通常是一个阻塞队列。当任务数超过线程池的容量时，新提交的任务将被放入任务队列中等待执行。线程池中的工作线程从任务队列中取出任务进行处理，如果队列为空，则工作线程会被阻塞，直到队列中有新的任务被提交。

2. 生产者-消费者模型

在生产者-消费者模型中，生产者向队列中添加元素，消费者从队列中取出元素进行处理。阻塞队列可以很好地解决生产者和消费者之间的并发问题，避免线程间的竞争和冲突。

3. 消息队列

消息队列使用阻塞队列来存储消息，生产者将消息放入队列中，消费者从队列中取出消息进行处理。消息队列可以实现异步通信，提高系统的吞吐量和响应性能，同时还可以将不同的组件解耦，提高系统的可维护性和可扩展性。

4. 缓存系统

缓存系统使用阻塞队列来存储缓存数据，当缓存数据被更新时，它会被放入队列中，其他线程可以从队列中取出最新的数据进行使用。使用阻塞队列可以避免并发更新缓存数据时的竞争和冲突。

5. 并发任务处理

在并发任务处理中，可以将待处理的任务放入阻塞队列中，多个工作线程可以从队列中取出任务进行处理。使用阻塞队列可以避免多个线程同时处理同一个任务的问题，并且可以将任务的提交和执行解耦，提高系统的可维护性和可扩展性。

总之，阻塞队列在实际应用中有很多场景，它可以帮助我们解决并发问题，提高程序的性能和可靠性。

1.3 JUC包下的阻塞队列

BlockingQueue 接口的实现类都被放在了 juc 包中，它们的区别主要体现在存储结构上或对元素操作上的不同，但是对于take与put操作的原理却是类似的。

队列	描述
ArrayBlockingQueue	基于数组结构实现的一个有界阻塞队列
LinkedBlockingQueue	基于链表结构实现的一个无界阻塞队列，指定容量为有界阻塞队列
PriorityBlockingQueue	支持按优先级排序的无界阻塞队列
DelayQueue	基于优先级队列（PriorityBlockingQueue）实现的无界阻塞队列
SynchronousQueue	不存储元素的阻塞队列
LinkedTransferQueue	基于链表结构实现的一个无界阻塞队列
LinkedBlockingDeque	基于链表结构实现的一个双端阻塞队列

https://www.processon.com/view/link/618ce3941e0853689b0818e2

2. ArrayBlockingQueue

ArrayBlockingQueue是最典型的有界阻塞队列，其内部是用数组存储元素的，初始化时需要指定容量大小，利用 ReentrantLock 实现线程安全。ArrayBlockingQueue可以用于实现数据缓存、限流、生产者-消费者模式等各种应用。

在生产者-消费者模型中使用时，如果生产速度和消费速度基本匹配的情况下，使用ArrayBlockingQueue是个不错选择；当如果生产速度远远大于消费速度，则会导致队列填满，大量生产线程被阻塞。

2.1 ArrayBlockingQueue使用

```
BlockingQueue queue = new ArrayBlockingQueue(1024);
queue.put("1");    //向队列中添加元素
Object object = queue.take();    //从队列中取出元素
```

2.2 ArrayBlockingQueue的原理

ArrayBlockingQueue使用独占锁ReentrantLock实现线程安全，入队和出队操作使用同一个锁对象，也就是只能有一个线程可以进行入队或者出队操作；这也就意味着生产者和消费者无法并行操作，在高并发场景下会成为性能瓶颈。

数据结构

利用了Lock锁的Condition通知机制进行阻塞控制。

核心：一把锁，两个条件

```
//数据元素数组
final Object[] items;
//下一个待取出元素索引
int takeIndex;
//下一个待添加元素索引
int putIndex;
//元素个数
int count;
//内部锁
final ReentrantLock lock;
//消费者
private final Condition notEmpty;
//生产者
private final Condition notFull;

public ArrayBlockingQueue(int capacity) {
    this(capacity, false);
}
public ArrayBlockingQueue(int capacity, boolean fair) {
    ...
    lock = new ReentrantLock(fair); //公平，非公平
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}

```

入队put方法

```java
public void put(E e) throws InterruptedException {
 //检查是否为空
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    //加锁，如果线程中断抛出异常
    lock.lockInterruptibly();
    try {
       //阻塞队列已满，则将生产者挂起，等待消费者唤醒
       //设计注意点： 用while不用if是为了防止虚假唤醒
        while (count == items.length)
            notFull.await(); //队列满了，使用notFull等待（生产者阻塞）
        // 入队
        enqueue(e);
    } finally {
        lock.unlock(); // 唤醒消费者线程
    }
}

private void enqueue(E x) {
    final Object[] items = this.items;
    //入队   使用的putIndex
    items[putIndex] = x;
    if (++putIndex == items.length)
        putIndex = 0;  //设计的精髓： 环形数组，putIndex指针到数组尽头了，返回头部
    count++;
    //notEmpty条件队列转同步队列，准备唤醒消费者线程，因为入队了一个元素，肯定不为空了
    notEmpty.signal();
}

```

思考： 为什么ArrayBlockingQueue对数组操作要设计成双指针?

使用双指针的好处在于可以避免数组的复制操作。如果使用单指针，每次删除元素时需要将后面的元素全部向前移动，这样会导致时间复杂度为 O(n)。而使用双指针，我们可以直接将 takeIndex 指向下一个元素，而不需要将其前面的元素全部向前移动。同样地，插入新的元素时，我们可以直接将新元素插入到 putIndex 所指向的位置，而不需要将其后面的元素全部向后移动。这样可以使得插入和删除的时间复杂度都是 O(1) 级别，提高了队列的性能。

出队take方法

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    //加锁，如果线程中断抛出异常
    lock.lockInterruptibly();
    try {
       //如果队列为空，则消费者挂起
        while (count == 0)
            notEmpty.await();
        //出队
        return dequeue();
    } finally {
        lock.unlock();// 唤醒生产者线程
    }
}
private E dequeue() {
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
    E x = (E) items[takeIndex]; //取出takeIndex位置的元素
    items[takeIndex] = null;
    if (++takeIndex == items.length)
        takeIndex = 0; //设计的精髓：  环形数组，takeIndex 指针到数组尽头了，返回头部
    count--;
    if (itrs != null)
        itrs.elementDequeued();
    //notFull条件队列转同步队列，准备唤醒生产者线程，此时队列有空位
    notFull.signal();
    return x;
}

```

3. LinkedBlockingQueue

LinkedBlockingQueue是一个基于链表实现的阻塞队列，默认情况下，该阻塞队列的大小为Integer.MAX_VALUE，由于这个数值特别大，所以 LinkedBlockingQueue 也被称作无界队列，代表它几乎没有界限，队列可以随着元素的添加而动态增长，但是如果没有剩余内存，则队列将抛出OOM错

误。所以为了避免队列过大造成机器负载或者内存爆满的情况出现，我们在使用的时候建议手动传一个队列的大小。

3.1 LinkedBlockingQueue使用

```
//指定队列的大小创建有界队列
BlockingQueue<Integer> boundedQueue = new LinkedBlockingQueue<>(100);
//无界队列
BlockingQueue<Integer> unboundedQueue = new LinkedBlockingQueue<>();

```

3.2 LinkedBlockingQueue原理

LinkedBlockingQueue内部由单链表实现，只能从head取元素，从tail添加元素。

LinkedBlockingQueue采用两把锁的锁分离技术实现入队出队互不阻塞，添加元素和获取元素都有独立的锁，也就是说LinkedBlockingQueue是读写分离的，读写操作可以并行执行。

数据结构

```
// 容量,指定容量就是有界队列
private final int capacity;
// 元素数量
private final AtomicInteger count = new AtomicInteger();
// 链表头  本身是不存储任何元素的，初始化时item指向null
transient Node<E> head;
// 链表尾
private transient Node<E> last;
// take锁   锁分离，提高效率
private final ReentrantLock takeLock = new ReentrantLock();
// notEmpty条件
// 当队列无元素时，take锁会阻塞在notEmpty条件上，等待其它线程唤醒
private final Condition notEmpty = takeLock.newCondition();
// put锁
private final ReentrantLock putLock = new ReentrantLock();
// notFull条件
// 当队列满了时，put锁会会阻塞在notFull上，等待其它线程唤醒
private final Condition notFull = putLock.newCondition();
```

```
19
20 //典型的单链表结构
21 static class Node<E> {
22     E item;  //存储元素
23     Node<E> next;  //后继节点     单链表结构
24     Node(E x) { item = x; }
25 }
26
```

构造器

```
1  public LinkedBlockingQueue() {
2      // 如果没传容量，就使用最大int值初始化其容量
3      this(Integer.MAX_VALUE);
4  }
5
6  public LinkedBlockingQueue(int capacity) {
7      if (capacity <= 0) throw new IllegalArgumentException();
8      this.capacity = capacity;
9      // 初始化head和last指针为空值节点
10     last = head = new Node<E>(null);
11 }
12
```

入队put方法

```
1  public void put(E e) throws InterruptedException {
2      // 不允许null元素
3      if (e == null) throw new NullPointerException();
4      int c = -1;
5      // 新建一个节点
6      Node<E> node = new Node<E>(e);
7      final ReentrantLock putLock = this.putLock;
8      final AtomicInteger count = this.count;
9      // 使用put锁加锁
10     putLock.lockInterruptibly();
11     try {
```

```
        // 如果队列满了，就阻塞在notFull上等待被其它线程唤醒（阻塞生产者线程）
        while (count.get() == capacity) {
            notFull.await();
        }
        // 队列不满，就入队
        enqueue(node);
        c = count.getAndIncrement();// 队列长度加1，返回原值
        // 如果现队列长度小于容量，notFull条件队列转同步队列，准备唤醒一个阻塞在notFull条件上的线程(可以继续入队)
        // 这里为啥要唤醒一下呢？
        // 因为可能有很多线程阻塞在notFull这个条件上,而取元素时只有取之前队列是满的才会唤醒notFull,此处不用等到取元素时才唤醒
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        putLock.unlock(); // 真正唤醒生产者线程
    }
    // 如果原队列长度为0，现在加了一个元素后立即唤醒阻塞在notEmpty上的线程
    if (c == 0)
        signalNotEmpty();
}
private void enqueue(Node<E> node) {
    // 直接加到last后面,last指向入队元素
    last = last.next = node;
}
private void signalNotEmpty() {
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();// 加take锁
    try {
        notEmpty.signal();// notEmpty条件队列转同步队列，准备唤醒阻塞在notEmpty上的线程
    } finally {
        takeLock.unlock();  // 真正唤醒消费者线程
    }
}
```

出队take方法

```
public E take() throws InterruptedException {
    E x;
```

```
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    // 使用takeLock加锁
    takeLock.lockInterruptibly();
    try {
        // 如果队列无元素，则阻塞在notEmpty条件上（消费者线程阻塞）
        while (count.get() == 0) {
            notEmpty.await();
        }
        // 否则，出队
        x = dequeue();
        c = count.getAndDecrement();//长度-1，返回原值
        if (c > 1)// 如果取之前队列长度大于1，notEmpty条件队列转同步队列，准备唤醒阻塞在notEmpty上的线程，原因与入队同理
            notEmpty.signal();
    } finally {
        takeLock.unlock(); // 真正唤醒消费者线程
    }
    // 为什么队列是满的才唤醒阻塞在notFull上的线程呢？
    // 因为唤醒是需要加putLock的，这是为了减少锁的次数,所以，这里索性在放完元素就检测一下，未满就唤醒其它notFull上的线程，
    // 这也是锁分离带来的代价
    // 如果取之前队列长度等于容量（已满），则唤醒阻塞在notFull的线程
    if (c == capacity)
        signalNotFull();
    return x;
}
private E dequeue() {
     // head节点本身是不存储任何元素的
    // 这里把head删除，并把head下一个节点作为新的值
    // 并把其值置空，返回原来的值
    Node<E> h = head;
    Node<E> first = h.next;
    h.next = h; // 方便GC
    head = first;
    E x = first.item;
    first.item = null;
    return x;
}
```

```
41 private void signalNotFull() {
42     final ReentrantLock putLock = this.putLock;
43     putLock.lock();
44     try {
45         notFull.signal();// notFull条件队列转同步队列，准备唤醒阻塞在notFull上的线程
46     } finally {
47         putLock.unlock(); // 解锁，这才会真正的唤醒生产者线程
48     }
49 }
```

3.3 LinkedBlockingQueue与ArrayBlockingQueue对比

LinkedBlockingQueue是一个阻塞队列，内部由两个ReentrantLock来实现出入队列的线程安全，由各自的Condition对象的await和signal来实现等待和唤醒功能。它和ArrayBlockingQueue的不同点在于：

- 队列大小有所不同，ArrayBlockingQueue是有界的初始化必须指定大小，而LinkedBlockingQueue可以是有界的也可以是无界的(Integer.MAX_VALUE)，对于后者而言，当添加速度大于移除速度时，在无界的情况下，可能会造成内存溢出等问题。
- 数据存储容器不同，ArrayBlockingQueue采用的是数组作为数据存储容器，而LinkedBlockingQueue采用的则是以Node节点作为连接对象的链表。
- 由于ArrayBlockingQueue采用的是数组的存储容器，因此在插入或删除元素时不会产生或销毁任何额外的对象实例，而LinkedBlockingQueue则会生成一个额外的Node对象。这可能在长时间内需要高效并发地处理大批量数据的时，对于GC可能存在较大影响。
- 两者的实现队列添加或移除的锁不一样，ArrayBlockingQueue实现的队列中的锁是没有分离的，即添加操作和移除操作采用的同一个ReenterLock锁，而LinkedBlockingQueue实现的队列中的锁是分离的，其添加采用的是putLock，移除采用的则是takeLock，这样能大大提高队列的吞吐量，也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

4. DelayQueue

DelayQueue 是一个支持延时获取元素的阻塞队列， 内部采用优先队列 PriorityQueue 存储元素，同时元素必须实现 Delayed 接口；在创建元素时可以指定多久才可以从队列中获取当前元素，只有在延迟期满时才能从队列中提取元素。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。

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> {
    //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。
    long getDelay(TimeUnit unit);
}

```

4.1 DelayQueue使用

DelayQueue 实现延迟订单

在实现一个延迟订单的场景中，我们可以定义一个 Order 类，其中包含订单的基本信息，例如订单编号、订单金额、订单创建时间等。同时，我们可以让 Order 类实现 Delayed 接口，重写 getDelay 和 compareTo 方法。在 getDelay 方法中，我们可以计算订单的剩余延迟时间，而在 compareTo 方法中，我们可以根据订单的延迟时间进行比较。
下面是一个简单的示例代码，演示了如何使用 DelayQueue 来实现一个延迟订单的场景：

```
public class DelayQueueExample {

    public static void main(String[] args) throws InterruptedException {
        DelayQueue<Order> delayQueue = new DelayQueue<>();

        // 添加三个订单，分别延迟 5 秒、2 秒和 3 秒
        delayQueue.put(new Order("order1", System.currentTimeMillis(), 5000));
        delayQueue.put(new Order("order2", System.currentTimeMillis(), 2000));
        delayQueue.put(new Order("order3", System.currentTimeMillis(), 3000));

        // 循环取出订单，直到所有订单都被处理完毕
        while (!delayQueue.isEmpty()) {
            Order order = delayQueue.take();
            System.out.println("处理订单: " + order.getOrderId());
        }
    }

```

```
18      static class  Order implements Delayed{
19          private String orderId;
20          private long createTime;
21          private long delayTime;
22
23          public Order(String orderId, long createTime, long delayTime) {
24              this.orderId = orderId;
25              this.createTime = createTime;
26              this.delayTime = delayTime;
27          }
28
29          public String getOrderId() {
30              return orderId;
31          }
32
33          @Override
34          public long getDelay(TimeUnit unit) {
35              long diff = createTime + delayTime - System.currentTimeMillis();
36              return unit.convert(diff, TimeUnit.MILLISECONDS);
37          }
38
39          @Override
40          public int compareTo(Delayed o) {
41              long diff = this.getDelay(TimeUnit.MILLISECONDS) -
    o.getDelay(TimeUnit.MILLISECONDS);
42              return Long.compare(diff, 0);
43          }
44      }
45  }
```

由于每个订单都有不同的延迟时间，因此它们将会按照延迟时间的顺序被取出。当延迟时间到达时，对应的订单对象将会被从队列中取出，并被处理。

4.2 DelayQueue原理

数据结构

```java
//用于保证队列操作的线程安全
private final transient ReentrantLock lock = new ReentrantLock();
// 优先级队列,存储元素，用于保证延迟低的优先执行
private final PriorityQueue<E> q = new PriorityQueue<E>();
// 用于标记当前是否有线程在排队（仅用于取元素时） leader 指向的是第一个从队列获取元素阻塞的线程
private Thread leader = null;
// 条件，用于表示现在是否有可取的元素    当新元素到达，或新线程可能需要成为leader时被通知
private final Condition available = lock.newCondition();

public DelayQueue() {}
public DelayQueue(Collection<? extends E> c) {
    this.addAll(c);
}
```

入队put方法

```java
public void put(E e) {
    offer(e);
}
public boolean offer(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        // 入队
        q.offer(e);
        if (q.peek() == e) {
            // 若入队的元素位于队列头部，说明当前元素延迟最小
            // 将 leader 置空
            leader = null;
            // available条件队列转同步队列,准备唤醒阻塞在available上的线程
            available.signal();
        }
        return true;
    } finally {
        lock.unlock(); // 解锁，真正唤醒阻塞的线程
    }
}

```

出队take方法

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            E first = q.peek();// 取出堆顶元素( 最早过期的元素，但是不弹出对象)
            if (first == null)// 如果堆顶元素为空，说明队列中还没有元素，直接阻塞等待
                available.await();//当前线程无限期等待，直到被唤醒，并且释放锁。
            else {
                long delay = first.getDelay(NANOSECONDS);// 堆顶元素的到期时间

                if (delay <= 0)// 如果小于0说明已到期，直接调用poll()方法弹出堆顶元素
                    return q.poll();

                // 如果delay大于0 ，则下面要阻塞了
                // 将first置为空方便gc
                first = null;
                // 如果有线程争抢的Leader线程，则进行无限期等待。
                if (leader != null)
                    available.await();
                else {
                    // 如果leader为null，把当前线程赋值给它
                    Thread thisThread = Thread.currentThread();
                    leader = thisThread;
                    try {
                        // 等待剩余等待时间
                        available.awaitNanos(delay);
                    } finally {
                        // 如果leader还是当前线程就把它置为空，让其它线程有机会获取元素
                        if (leader == thisThread)
                            leader = null;
                    }
                }
            }
        }
    } finally {
```

```
36          // 成功出队后，如果leader为空且堆顶还有元素，就唤醒下一个等待的线程
37          if (leader == null && q.peek() != null)
38              // available条件队列转同步队列,准备唤醒阻塞在available上的线程
39              available.signal();
40          // 解锁，真正唤醒阻塞的线程
41          lock.unlock();
42      }
43  }
```

1. 当获取元素时，先获取到锁对象。
2. 获取最早过期的元素，但是并不从队列中弹出元素。
3. 最早过期元素是否为空，如果为空则直接让当前线程无限期等待状态，并且让出当前锁对象。
4. 如果最早过期的元素不为空
5. 获取最早过期元素的剩余过期时间，如果已经过期则直接返回当前元素
6. 如果没有过期，也就是说剩余时间还存在，则先获取Leader对象，如果Leader已经有线程在处理，则当前线程进行无限期等待，如果Leader为空，则首先将Leader设置为当前线程，并且让当前线程等待剩余时间。
7. 最后将Leader线程设置为空
8. 如果Leader已经为空，并且队列有内容则唤醒一个等待的队列。

5. 如何选择适合的阻塞队列

5.1 选择策略

通常我们可以从以下 5 个角度考虑，来选择合适的阻塞队列：

功能

第 1 个需要考虑的就是功能层面，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

第 2 个需要考虑的是容量，或者说是否有存储的要求，还是只需要“直接传递”。在考虑这一点的时候，我们知道前面介绍的那几种阻塞队列，有的是容量固定的，如 ArrayBlockingQueue；有的默认是容量无限的，如 LinkedBlockingQueue；而有的里面没有任何容量，如 SynchronousQueue；而对于 DelayQueue 而言，它的容量固定就是 Integer.MAX_VALUE。所以不同阻塞队列的容量是千差万别的，我们需要根据任务数量来推算出合适的容量，从而去选取合适的 BlockingQueue。

能否扩容

第 3 个需要考虑的是能否扩容。因为有时我们并不能在初始的时候很好的准确估计队列的大小，因为业务可能有高峰期、低谷期。如果一开始就固定一个容量，可能无法应对所有的情况，也是不合

适的，有可能需要动态扩容。如果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。所以我们可以根据是否需要扩容来选取合适的队列。

内存结构

第 4 个需要考虑的点就是内存结构。我们分析过 ArrayBlockingQueue 的源码，看到了它的内部结构是“数组”的形式。和它不同的是，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。另外，SynchronousQueue 性能往往优于其他实现，因为它只需要“直接传递”，而不需要存储的过程。如果我们的场景需要直接传递的话，可以优先考虑 SynchronousQueue。

5.2 线程池对于阻塞队列的选择

线程池有很多种，不同种类的线程池会根据自己的特点，来选择适合自己的阻塞队列。

- FixedThreadPool（SingleThreadExecutor 同理）选取的是 LinkedBlockingQueue
- CachedThreadPool 选取的是 SynchronousQueue
- ScheduledThreadPool（SingleThreadScheduledExecutor同理）选取的是延迟队列