一、集合系列问题

hash 系列问题

hash 为了解决什么出现的

通过某种特定的函数、算法将要检索的项与用来检索的索引关联起来，生成一种便于搜索的数据结构。也译为散列。

hash 的应用范围

Hash 算法在数据结构，加密，海量数据处理中有着广泛的应用。

Java 中很多数据结构，比如 HashTable, HashMap 等都应用了 hash 算法。还有如果自定义对象重写了 equals()方法则一定要重写 hashCode 方法，首先的一条是 equals 方法相等的对象一定要保证 hashCode 相等，然后就是 hashCode 的算法影响到键值的存放和查找，详细讨论可以参考: http://blog.csdn.net/michaellufhl/article/details/5833188

加密算法里也大量用到 hash 算法，比如 MD5 等，海量数据的查找也看好了 hash 的时间复杂度优点。

重写 equals 和 hashcode 可以参考：

http://zhangjunhd.blog.51cto.com/113473/71571/

http://www.cnblogs.com/happyPawpaw/p/3744971.html

参考资料：

http://blog.csdn.net/v_july_v/article/details/6256463

http://blog.csdn.net/mycomputerxiaomei/article/details/7641221

http://blog.csdn.net/lightty/article/details/11191971

为什么 hash 存取比较快呢

很简单啦,因为有种算法叫做哈希算法,哈希算法会根据你要存入的数据,先通过该算法,计算出一个地址值,这个地址值就是你需要存入到集合当中的数据的位置,而不会像数组那样一个个的进行挨个存储,挨个遍历一遍后面有空位就存这种情况了,而你查找的时候,也是根据这个哈希算法来的,将你的要查找的数据进行计算,得出一个地址,这个地址会印射到集合当中的位置,这样就能够直接到这个位置上去找了,而不需要像数组那样,一个个遍历,一个个

对比去寻找,这样自然增加了速度,提高了效率了.

用 hash 会出现什么错误

哈希冲突 和 哈希溢出

当关键字值域远大于哈希表的长度，而且事先并不知道关键字的具体取值时。冲突就难免会发生。另外，当关键字的实际取值大于哈希表的长度时，而且表中已装满了记录，如果插入一个新记录，不仅发生冲突，而且还会发生溢出。因此，处理冲突和溢出是哈希技术中的两个重要问题。

Object 的 hashcode()是怎么计算的？

hashCode 方法的主要作用是为了配合基于散列的集合一起正常运行，这样的散列集合包括 HashSet、HashMap 以及 HashTable。考虑一种情况，当向集合中插入对象时，如何判别在集合中是否已经存在该对象了？（注意：集合中不允许重复的元素存在）也许大多数人都会想到调用 equals 方法来逐个进行比较，这个方法确实可行。但是如果集合中已经存在一万条数据或者更多的数据，如果采用 equals 方法去逐一比较，效率必然是一个问题。此时 hashCode 方法的作用就体现出来了，当集合要添加新的对象时，先调用这个对象的 hashCode 方法，得到对应的 hashcode 值，实际上在 HashMap 的具体实现中会用一个 table 保存已经存进去的对象的 hashcode 值，如果 table 中没有该 hashcode 值，它就可以直接存进去，不用再进行任何比较了；如果存在该 hashcode 值， 就调用它的 equals 方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址，所以这里存在一个冲突解决的问题，这样一来实际调用 equals 方法的次数就大大降低了，说通俗一点：Java 中的 hashCode 方法就是根据一定的规则将与对象相关的信息（比如对象的存储地址，对象的字段等）映射成一个数值，这个数值称作为散列值。即在散列集合包括 HashSet、HashMap 以及 HashTable 里，对每一个存储的桶元素都有一个唯一的"块编号"，即它在集合里面的存储地址；当你调用 contains 方法的时候，它会根据 hashcode 找到块的存储地址从而定位到该桶元素。

设计 hashCode()时最重要的因素就是：无论何时，对同一个对象调用 hashCode()都应该产生同样的值。如果在讲一个对象用 put()添加进 HashMap 时产生一个 hashCdoe 值，而用 get()取出时却产生了另一个 hashCode 值，那么就无法获取该对象了。所以如果你的 hashCode 方法依赖于对象中易变的数据，用户就要当心了，因为此数据发生变化时，hashCode()方法

就会生成一个不同的散列码。

实现上：

Integer.hashCode()实现：

```
public int hashCode() {
       return value;
   }
```

String.hashCode()实现为其 ACII 字符码：

```
public int hashCode() {
        int h = hash;
        if (h == 0) {
            int off = offset;
            char val[] = value;
            int len = count;
            for (int i = 0; i < len; i++) {
                h = 31*h + val[off++];
            }
            hash = h;
        }
        return h;
   }
```

下面是 HotSpot JVM 中生成 hash 散列值的实现：

```
static inline intptr_t get_next_hash(Thread * Self, oop obj) {
  intptr_t value = 0 ;
  if (hashCode == 0) {
     // This form uses an unguarded global Park-Miller RNG,
     // so it's possible for two threads to race and generate the same RNG.
     // On MP system we'll have lots of RW access to a global, so the
```

```
   // mechanism induces lots of coherency traffic.
   value = os::random() ;
} else
if (hashCode == 1) {
   // This variation has the property of being stable (idempotent)
   // between STW operations.  This can be useful in some of the 1-0
   // synchronization schemes.
   intptr_t addrBits = intptr_t(obj) >> 3 ;
   value = addrBits ^ (addrBits >> 5) ^ GVars.stwRandom ;
} else
if (hashCode == 2) {
   value = 1 ;           // for sensitivity testing
} else
if (hashCode == 3) {
   value = ++GVars.hcSequence ;
} else
if (hashCode == 4) {
   value = intptr_t(obj) ;
} else {
   // Marsaglia's xor-shift scheme with thread-specific state
   // This is probably the best overall implementation -- we'll
   // likely make this the default in future releases.
   unsigned t = Self->_hashStateX ;
   t ^= (t << 11) ;
   Self->_hashStateX = Self->_hashStateY ;
   Self->_hashStateY = Self->_hashStateZ ;
   Self->_hashStateZ = Self->_hashStateW ;
   unsigned v = Self->_hashStateW ;
   v = (v ^ (v >> 19)) ^ (t ^ (t >> 8)) ;
   Self->_hashStateW = v ;
```

```
    value = v ;
  }

  value &= markOopDesc::hash_mask;
  if (value == 0) value = 0xBAD ;
  assert (value != markOopDesc::no_hash, "invariant") ;
  TEVENT (hashCode: GENERATE) ;
  return value;
}
```

若 hashcode 方法永远返回 1 会产生什么结果

首先，说一下这个“题目”中 hashCode()返回常量 1，重点在什么地方，重点在“常量”这两个字上，也就是说，每次使用 java 对象“.”hashCode()时，返回的都是相同的数值；

其次，说下 hashCode()的值并不一定是对象在内存地址或物理地址，但是初学者可以这么理解；

第三，判断 java 对象的值是否相同的是 equals 方法，判断对象基本类型是否相同是用的==，而 hashCode()这个方法也是比较对象是否相同的一个依据，

当 hashCode()返回常量时，所有对象都出现 hash 冲突，而 hashCode()本身的性能也会降级。做 hash 的 key 的时候效率会极度变低。

变量比较也会变慢

解决 hash 冲突的方法？

1、开放定址法

用开放定址法解决冲突的做法是：当冲突发生时，使用某种探查(亦称探测)技术在散列表中形成一个探查(测)序列。沿此序列逐个单元地查找，直到找到给定 的关键字，或者碰到一个开放的地址(即该地址单元为空)为止（若要插入，在探查到开放的地址，则可将待插入的新结点存人该地址单元）。查找时探查到开放的 地址则表明表中无待查的关键字，即查找失败。

注意：

①用开放定址法建立散列表时，建表前须将表中所有单元(更严格地说，是指单元中存储的关键字)置空。

②空单元的表示与具体的应用相关。

按照形成探查序列的方法不同，可将开放定址法区分为线性探查法、线性补偿探测法、随机探测等。

（1）线性探查法(Linear Probing)

该方法的基本思想是：

将散列表 T[0..m-1]看成是一个循环向量，若初始探查的地址为 d(即 h(key)=d)，则最长的探查序列为：

d，d+l，d+2，…，m-1，0，1，…，d-1

即:探查时从地址 d 开始，首先探查 T[d]，然后依次探查 T[d+1]，…，直到 T[m-1]，此后又循环到 T[0]，T[1]，…，直到探查到 T[d-1]为止。

探查过程终止于三种情况：

(1)若当前探查的单元为空，则表示查找失败（若是插入则将 key 写入其中）；

(2)若当前探查的单元中含有 key，则查找成功，但对于插入意味着失败；

(3)若探查到 T[d-1]时仍未发现空单元也未找到 key，则无论是查找还是插入均意味着失败(此时表满)。

利用开放地址法的一般形式，线性探查法的探查序列为：

hi=(h(key)+i)％m 0≤i≤m-1 //即 di=i

用线性探测法处理冲突，思路清晰，算法简单，但存在下列缺点：

① 处理溢出需另编程序。一般可另外设立一个溢出表，专门用来存放上述哈希表中放不下的记录。此溢出表最简单的结构是顺序表，查找方法可用顺序查找。

② 按上述算法建立起来的哈希表，删除工作非常困难。假如要从哈希表 HT 中删除一个记录，按理应将这个记录所在位置置为空，但我们不能这样做，而只能标上已被删除的标记，否则，将会影响以后的查找。

③ 线性探测法很容易产生堆聚现象。所谓堆聚现象，就是存入哈希表的记录在表中连成一片。按照线性探测法处理冲突，如果生成哈希地址的连续序列愈长 (即不同关键字值的哈希地址相邻在一起愈长) ，则当新的记录加入该表时，与这个序列发生冲突的可能性愈大。因此，哈希地址的较长连续序列比较短连续序列生长得快，这就意味着，一旦出现堆聚 (伴

随着冲突) ，就将引起进一步的堆聚。

（2）线性补偿探测法

线性补偿探测法的基本思想是：

将线性探测的步长从 1 改为 Q ，即将上述算法中的 j ＝ (j ＋ 1) % m 改为： j ＝ (j ＋ Q) % m ，而且要求 Q 与 m 是互质的，以便能探测到哈希表中的所有单元。

【例】 PDP-11 小型计算机中的汇编程序所用的符合表，就采用此方法来解决冲突，所用表长 m ＝ 1321 ，选用 Q ＝ 25 。

（3）随机探测

随机探测的基本思想是：

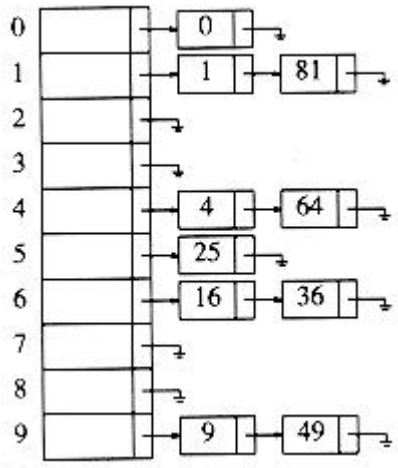
将 线性探测的步长从常数改为随机数，即令： j ＝ (j ＋ RN) % m ，其中 RN 是一个随机数。在实际程序中应预先用随机数发生器产生一个随机序列，将此序列作为依次探测的步长。这样就能使不同的关键字具有不同的探测次序，从而可以避 免或减少堆聚。基于与线性探测法相同的理由，在线性补偿探测法和随机探测法中，删除一个记录后也要打上删除标记。

2、拉链法

（1）拉链法解决冲突的方法

拉链法解决冲突的做法是：将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为 m，则可将散列表定义为一个由 m 个头指针组成的指针数组 T[0..m-1]。凡是散列地址为 i 的结点，均插入到以 T[i]为头指针的单链表中。T 中各分量的初值均应为空指针。在拉链法中，装填因子 α 可以大于 1，但一般均取 α ≤1。

按外链地址法所建立的哈希表如下图所示：



（2）拉链法的优点

这样就是前面说的结合了数组和链表的优点。与开放定址法相比，拉链法有如下几个优点：

①拉链法处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短；

②由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况；

③开放定址法为减少冲突，要求装填因子α较小，故当结点规模较大时会浪费很多空间。而拉链法中可取α≥1，且结点较大时，拉链法中增加的指针域可忽略不计，因此节省空间；

④ 在用拉链法构造的散列表中，删除结点的操作易于实现。只要简单地删去链表上相应的结点即可。而对开放地址法构造的散列表，删除结点不能简单地将被删结 点的空间置为空，否则将截断在它之后填人散列表的同义词结点的查找路径。这是因为各种开放地址法中，空地址单元(即开放地址)都是查找失败的条件。因此在 用开放地址法处理冲突的散列表上执行删除操作，只能在被删结点上做删除标记，而不能真正删除结点。

（3）拉链法的缺点

拉链法的缺点是：指针需要额外的空间，故当结点规模较小时，开放定址法较为节省空间

map 系列问题

hashmap 问题

hashmap 怎么实现的

HashMap 由数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前 entry 的 next 指向 null）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为 O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过 key 对象的 equals 方法逐一比对查找。所以，性能考虑，HashMap 中的链表出现越少，性能才会越好。

hashmap 底层原理，存取过程

Put 方法：

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
```

```
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
                }
        } if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

存的过程，就是 put 的过程，首先判断数组是否为空，然后在判断存储是否需要扩容。

如果这些都不需要，开始计算 hash 值，然后根据 hash 值存储到数组相应的位置上。如果这个位子上有链表节点了，直接将新增加的元素放在这个位置，然后让他的 next 指向原来的链结，如果由 hash 计算出的这个位置上没有其他的节点，那么直接将新的节点添加到这个位置上。

Get 方法：

```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
            if ((e = first.next) != null) {
                if (first instanceof TreeNode)
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do {
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

Get 方法根据当前的对象的 hash 值进行查找在数组当中的位置，如果在数组当中有多个值，那么在根据 key 值进行查找。

HashMap 数组的第一个元素存的是什么(我不知道然后他说 HashMap 可以存'呐亩'吗，我听成能不能存 enum，我说可以可以能存 Integer，String，enum，然后他说是 null，不是 enum...)

Node：HashMap 的内部类，Node 是 HashMap 存储的节点

--既然提到红黑树，就说说红黑树

巴拉巴拉------手写吧

hashmap 为什么不直接使用红黑树

因为红黑树需要进行左旋，右旋操作， 而单链表不需要，以下都是单链表与红黑树结构对比。

如果元素小于 8 个，查询成本高，新增成本低

如果元素大于 8 个，查询成本低，新增成本高

链表长度超过 8 后为什么转为红黑树

若桶中链表元素个数大于等于 8 时，链表转换成树结构；若桶中链表元素个数小于等于 6 时，树结构还原成链表。因为红黑树的平均查找长度是 log(n)，长度为 8 的时候，平均查找长度为 3，如果继续使用链表，平均查找长度为 8/2=4，这才有转换为树的必要。链表长度如果是小于等于 6，6/2=3，虽然速度也很快的，但是转化为树结构和生成树的时间并不会太短。还有选择 6 和 8，中间有个差值 7 可以有效防止链表和树频繁转换。假设一下，如果设计成链表个数超过 8 则链表转换成树结构，链表个数小于 8 则树结构转换成链表，如果一个 HashMap 不停的插入、删除元素，链表个数在 8 左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。

hanshmap 线程安全吗，线程不安全表现在哪

1.resize 死循环，出现 resize，rehash 的时候，在扩容的时候链表重新计算 hash 值进行分配，在多线程的情况下会出现链表成环的情况。

2.fail-fast，如果在使用迭代器的过程中有其他线程修改了 map，那么将抛出

ConcurrentModificationException，这就是所谓 fail-fast 策略。

详情可以看：http://www.importnew.com/22011.html

hashMap 的 put 方法说下，它是安全的嘛，举个例子，多线程同时 put 会有什么问题

多线程下 put 时，链表会成环。

使用 Iterator 遍历 HashMap，删除值为 a 的元素会发生什么

首先 HashMap 当中没有 Iterator 方法，它只能用 set 中的 Iterator 方法进行遍历。在遍历的时候进行删除 a 会出现 ConcurrentModificationException 异常。

怎么解决 hashmap 的线程安全问题

使用 Collections 中的 SynchronizedMap 对象，SynchronizedMap 源码如下：

```
// synchronizedMap 方法
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {
       return new SynchronizedMap<>(m);
   }
// SynchronizedMap 类
private static class SynchronizedMap<K,V>
       implements Map<K,V>, Serializable {
       private static final long serialVersionUID = 1978198479659022715L;

       private final Map<K,V> m;     // Backing Map
       final Object      mutex;        // Object on which to synchronize

       SynchronizedMap(Map<K,V> m) {
           this.m = Objects.requireNonNull(m);
           mutex = this;
       }

       SynchronizedMap(Map<K,V> m, Object mutex) {
           this.m = m;
           this.mutex = mutex;
       }
```

```
        public int size() {
            synchronized (mutex) {return m.size();}
        }
        public boolean isEmpty() {
            synchronized (mutex) {return m.isEmpty();}
        }
        public boolean containsKey(Object key) {
            synchronized (mutex) {return m.containsKey(key);}
        }
        public boolean containsValue(Object value) {
            synchronized (mutex) {return m.containsValue(value);}
        }
        public V get(Object key) {
            synchronized (mutex) {return m.get(key);}
        }

        public V put(K key, V value) {
            synchronized (mutex) {return m.put(key, value);}
        }
        public V remove(Object key) {
            synchronized (mutex) {return m.remove(key);}
        }
        // 省略其他方法
    }
```

hashtable 问题

hashtable 是线程安全吗？怎么实现的线程安全的？

put 和 get 方法是 synchronized 的所以可以保证其线程安全

Hashtable 在 null 的处理是怎样的

```
if (value == null) {
    throw new NullPointerException();
}
```

以上是 hashtable 的 put 中第一行代码

concurrentHashMap 问题

cuncurrenthashmap 和 hashtable 区别是什么（锁效率方面）？

它们都可以用于多线程的环境，但是当 Hashtable 的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。因为 ConcurrentHashMap 引入了分割(segmentation)，不论它变得多么大，仅仅需要锁定 map 的某个部分，而其它的线程不需要等到迭代完成才能访问 map。简而言之，在迭代的过程中，ConcurrentHashMap 仅仅锁定 map 的某个部分，而 Hashtable 则会锁定整个 map。

concurrentHashMap 加锁原理

最主要加锁的方法有 get，put，remove 这 3 个操作

ReentrantLock，它基本作用和 synchronized 相似，但提供了更多的操作方式，比如在获取锁时不必像 synchronized 那样只是傻等，可以设置定时，轮询，或者中断，这些方法使得它在获取多个锁的情况可以避免死锁操作）。实现了 AQS 接口的一个锁。

分拆锁(lock spliting)就是若原先的程序中多处逻辑都采用同一个锁，但各个逻辑之间又相互独立，就可以拆(Spliting)为使用多个锁，每个锁守护不同的逻辑。 分拆锁有时候可以被扩展，分成可大可小加锁块的集合，并且它们归属于相互独立的对象，这样的情况就是分离锁(lock striping)。

concurrentHashMap 的分段锁：

jdk1.5、jdk1.6 版本的所有方法都是放静态内部类 Segment 中进行封装的，将数据 table 也封装到了 Segment 类中，他会在每一个方法的开始时使用 lock()将整个方法锁上，然后再方法执行结束使用 unlock()方法进行解锁。Segment 继承了 ReentrantLock，表明每个 segment 都可以当做一个锁。（ReentrantLock 前文已经提到，不了解的话就把当做 synchronized 的替代者吧）这样对每个 segment 中的数据需要同步操作的话都是使用每个 segment 容器对象自身的锁来实现。只有对全局需要改变时锁定的是所有的 segment。

jdk1.7 版本的 put 方法放在了静态内部类 segment 中进行了封装。get 方法没有被封装，直接使用 Segment 对象锁进行加锁，这个实现有点像 synchronized 锁的代码块锁。

jdk1.8 以后的 concurrentHashMap 中静态内部类 segment 没有封装任何方法，取消 segments 字段，直接采用 transient volatile HashEntry<K,V> table 保存数据，采用 table 数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。

https://blog.csdn.net/wang35235966/article/details/54706414

说下 ConcurrentHashMap 你知道些什么？怎么设计的？为什么不同的 key 会放在相同的位置？

list 系列问题

数组和链表区别？

数组静态分配内存，链表动态分配内存；

数组在内存中连续，链表不一定连续；

数组元素在栈区，链表元素在堆区；　？

数组利用下标定位，时间复杂度为 O(1)，链表定位元素时间复杂度 O(n)；

数组插入或删除元素的时间复杂度 O(n)，链表的时间复杂度 O(1)。

ArrayList 是否会越界。

在多线程的情况下会出现越界的操作

```
public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        elementData[size++] = e;
```

```
        return true;
    }
```

两个线程进行 add 添加，假如现在的 array 的 lenght 为 10，size 为 8，第一个线程执行扩容函数发现并不需要扩容，然后执行下一行代码，下一行代码可分解为：elementData[size] = e; size++; 如果第一个线程执行完了第一步后，第二个线程开始执行，那么它执行到扩容函数的时候也不需要扩容，继续执行，这样就会根据线程一和线程二都进行了 size++操作，并且他们的数据同时添加到了 8 的位子上，size 这个时候已经变成了 10。然后就会发现已经越界了！

函数 grow()解释了基于数组的 ArrayList 是如何扩容的。数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是其原容量的 1.5 倍。

ArrayList 中删除值为 a 的元素怎么操作

```
    public boolean remove(Object o) {
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
                    fastRemove(index);
                    return true;
                }
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }
```

首先判断传进来的值是否为空，如果为空，将数组中第一个存储为 null 的节点删除。如果不为空，进行遍历查找这个元素，如果找到了，进行删除节点，返回 true，如果没有找到，返回 false。

list 中还有哪些是线程安全的。

直系下属：Vector

其他 list 的实现类：Stack（继承了 Vector）

线程不安全的有：

linkedList、ArrayList

ArrayList 的 sublist 修改是否影响 list 本身。

```
public List<E> subList(int fromIndex, int toIndex) {
    subListRangeCheck(fromIndex, toIndex, size);
    return new SubList(this, 0, fromIndex, toIndex);
}

static void subListRangeCheck(int fromIndex, int toIndex, int size) {
    if (fromIndex < 0)
        throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);
    if (toIndex > size)
        throw new IndexOutOfBoundsException("toIndex = " + toIndex);
    if (fromIndex > toIndex)
        throw new IllegalArgumentException("fromIndex(" + fromIndex +
                                           ") > toIndex(" + toIndex + ")");
}
```

不会，因为 subList 方法里返回的是另一个类的对象，而他的一系列操作也是在新的对象上进行操作的，与原对象无关。

stack 底层实现

Stack 继承了 Vector，说明 Stack 是线程安全的。

Stack 的底层依旧使用的是 Vector 的底层数据结构数组。

Queue 系列问题

需要知道两个主要接口：Deque 和 Queue

Queue 接口继承了 Collection 接口，它实际上限制了容器的操作，如果使用 Queue 去实例化一个类的话，这个类只能使用 Queue 接口中的方法。

Deque 接口继承了 Queue 接口，添加了更多的方法。Deque 接口继承自 Queue 接口，但 Deque 支持同时从两端添加或移除元素，因此又被成为双端队列。鉴于此，Deque 接口的实现可以被当作 FIFO 队列使用，也可以当作 LIFO 队列（栈）来使用。官方也是推荐使用 Deque 的实现来替代 Stack。

Queue 系列的问题存在 java 源码中的两个文件包中，有直接存在 util 文件包中的，也有存在 util.concurrent 文件包中的。

Util：

AbstractQueue<E>

ArrayDeque<E>

LinkedList<E>

PriorityQueue<E>

Util.concurrent:

ArrayBlockingQueue<E> ： 一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue<E> ： 一个由链表结构组成的有界阻塞队列。

PriorityBlockingQueue<E> ： 一个支持优先级排序的无界阻塞队列。

SynchronousQueue<E> ： 一个不存储元素的阻塞队列。

LinkedBlockingDeque<E> ： 一个由链表结构组成的双向阻塞队列。

DelayQueue<E> ： 一个使用优先级队列实现的无界阻塞队列。

ConcurrentLinkedQueue<E>

LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

Util 包下面的

AbstractQueue

它实际上是一个抽象类，继承了 AbstractCollection 抽象类，实现了 Queue 接口

LinkedList

LinkedList 是一个继承于 AbstractSequentialList 的双向链表。它也可以被当作堆栈、队列或双端队列进行操作。

LinkedList 实现 List 接口，能对它进行队列操作。

LinkedList 实现 Deque 接口，即能将 LinkedList 当作双端队列使用。

LinkedList 实现了 Cloneable 接口，即覆盖了函数 clone()，能克隆。

LinkedList 实现 java.io.Serializable 接口，这意味着 LinkedList 支持序列化，能通过序列化去传输。

LinkedList 是非同步的。

(01) LinkedList 实际上是通过双向链表去实现的。

它包含一个非常重要的内部类：Entry。Entry 是双向链表节点所对应的数据结构，它包括的属性有：当前节点所包含的值，上一个节点，下一个节点。

(02) 从 LinkedList 的实现方式中可以发现，它不存在 LinkedList 容量不足的问题。

(03) LinkedList 的克隆函数，即是将全部元素克隆到一个新的 LinkedList 对象中。

(04) LinkedList 实现 java.io.Serializable。当写入到输出流时，先写入“容量”，再依次写入“每一个节点保护的值”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”。

(05) 由于 LinkedList 实现了 Deque，而 Deque 接口定义了在双端队列两端访问元素的方法。提供插入、移除和检查元素的方法。每种方法都存在两种形式：一种形式在操作失败时抛出异常，另一种形式返回一个特殊值（null 或 false，具体取决于操作）。

总结起来如下表格：

	第一个元素（头部）	最后一个元素（尾部）

	抛出异常	特殊值	抛出异常	特殊值
插入	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
移除	removeFirst()	pollFirst()	removeLast()	pollLast()
检查	getFirst()	peekFirst()	getLast()	peekLast()

(06) LinkedList 可以作为 FIFO(先进先出)的队列，作为 FIFO 的队列时，下表的方法等价：

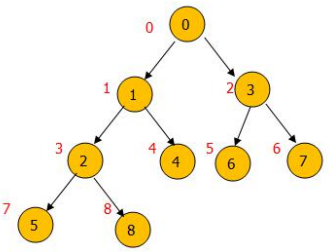
队列方法	等效方法
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

(07) LinkedList 可以作为 LIFO(后进先出)的栈，作为 LIFO 的栈时，下表的方法等价：

栈方法	等效方法
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()

PriorityQueue

其实在 Java 1.5 版本后就提供了一个具备了小根堆性质的数据结构也就是优先队列 PriorityQueue。下面详细了解一下 PriorityQueue 到底是如何实现小顶堆的，然后利用 PriorityQueue 实现大顶堆。



0	1	3	2	4	6	7	5	8

PriorityQueue 的逻辑结构是一棵完全二叉树，存储结构其实是一个数组。逻辑结构层次遍历的结果刚好是一个数组。

添加元素操作：

add(E e) 和 offer(E e) 方法

add(E e) 和 offer(E e) 方法都是向 PriorityQueue 中加入一个元素，其中 add（）其实调用了 offer（）方法如下：

```
public boolean add(E e) {
    return offer(e);
}

public boolean offer(E e) {
        if (e == null)
            throw new NullPointerException();
      //如果压入的元素为 null 抛出异常
        int i = size;
        if (i >= queue.length)
            grow(i + 1);
            //如果数组的大小不够扩充
        size = i + 1;
        if (i == 0)
            queue[0] = e;
```

```
            //如果只有一个元素之间放在堆顶
        else
            siftUp(i, e);
            //否则调用 siftUp 函数从下往上调整堆。
        return true;
    }
```

对上面代码做几点说明：

①优先队列中不能存放空元素。

②压入元素后如果数组的大小不够会进行扩充，上面的 queue 其实就是一个默认初始值为 11 的数组（也可以赋初始值）。

③offer 元素的主要调整逻辑在 siftUp (i, e)函数中。下面看看 siftUp(i, e) 函数到底是怎样实现的。

```
    private void siftUpComparable(int k, E x) {
        Comparable<? super E> key = (Comparable<? super E>) x;
        while (k > 0) {
            int parent = (k - 1) >>> 1;
            Object e = queue[parent];
            if (key.compareTo((E) e) >= 0)
                break;
            queue[k] = e;
            k = parent;
        }
        queue[k] = key;
    }
```

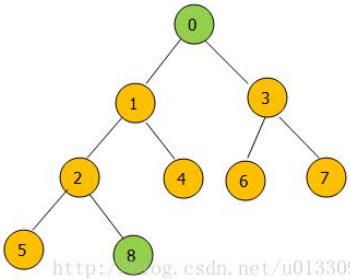
上面的代码还是比较简明的，就是当前元素与父节点不断比较如果比父节点小就交换然后继续向上比较，否则停止比较的过程。

移除元素操作：

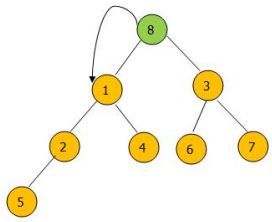
poll() 和 remove() 方法

poll 方法每次从 PriorityQueue 的头部删除一个节点，也就是从小顶堆的堆顶删除一个

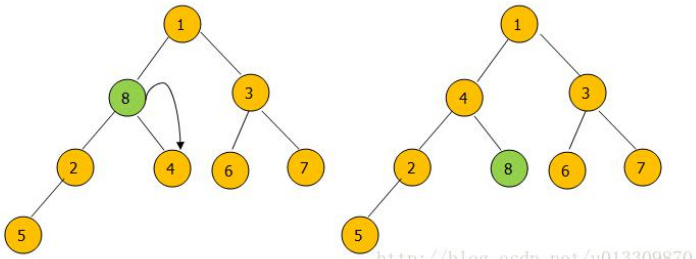
节点，而 remove（）不仅可以删除头节点而且还可以用 remove(Object o) 来删除堆中的与给定对象相同的最先出现的对象。先看看 poll（）方法。下面是 poll（）之后堆的操作删除元素后要对堆进行调整：



堆中每次删除只能删除头节点。也就是数组中的第一个节点。



将最后一个节点替代头节点然后进行调整。



Poll()方法的源码

```
public E poll() {
    if (size == 0)
        return null;
    //如果堆大小为0则返回null
    int s = --size;
    modCount++;
    E result = (E) queue[0];
    E x = (E) queue[s];
```

```
        queue[s] = null;
        //如果堆中只有一个元素直接删除
        if (s != 0)
            siftDown(0, x);
        //否则删除元素后对堆进行调整
        return result;
    }
```

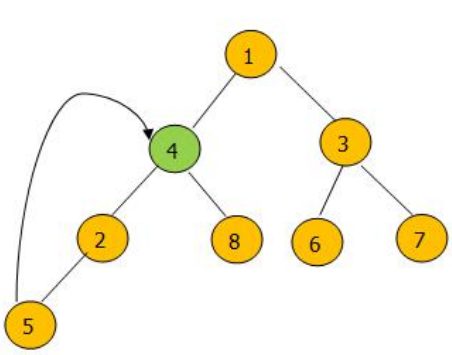
看看 siftDown(0, x) 方法的源码：

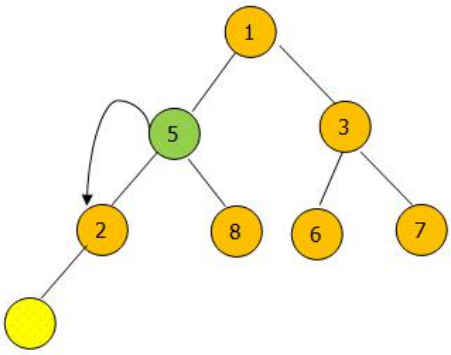
```
    private void siftDown(int k, E x) {
        if (comparator != null)
            siftDownUsingComparator(k, x);
        else
            siftDownComparable(k, x);
    }

    @SuppressWarnings("unchecked")
    private void siftDownComparable(int k, E x) {
        Comparable<? super E> key = (Comparable<? super E>)x;
        int half = size >>> 1;        // loop while a non-leaf
        while (k < half) {
            int child = (k << 1) + 1; // assume left child is least
            Object c = queue[child];
            int right = child + 1;
            if (right < size &&
                ((Comparable<? super E>) c).compareTo((E) queue[right]) > 0)
                c = queue[child = right];
            if (key.compareTo((E) c) <= 0)
                break;
            queue[k] = c;
            k = child;
```

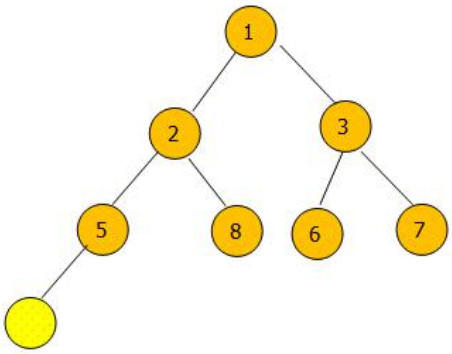
```
        }
        queue[k] = key;
    }
```

siftDown（）方法就是从堆的第一个元素往下比较，如果比左右孩子节点的最小值小则与最小值交换，交换后继续向下比较，否则停止比较。

remove（4）的过程图：







先用堆的最后一个元素 5 代替 4 然后从 5 开始向下调整堆。这个过程和 poll（）函数一样，只不过 poll（）函数每次都是从堆顶开始。

remove(Object o)的代码：

```
    public boolean remove(Object o) {
        int i = indexOf(o);
        if (i == -1)
            return false;
```

```
        else {
            removeAt(i);
            return true;
        }
    }
```

removeAt(int i)的代码：

```
    private E removeAt(int i) {
        // assert i >= 0 && i < size;
        modCount++;
        int s = --size;
        if (s == i) // removed last element
            queue[i] = null;
        else {
            E moved = (E) queue[s];
            queue[s] = null;
            siftDown(i, moved);
            if (queue[i] == moved) {
                siftUp(i, moved);
                if (queue[i] != moved)
                    return moved;
            }
        }
        return null;
    }
```

使用 PriorityQueue 实现大顶堆

```
private static final int DEFAULT_INITIAL_CAPACITY = 11;
PriorityQueue<Integer> maxHeap=
new PriorityQueue<Integer>(DEFAULT_INITIAL_CAPACITY, new Comparator<Integer>()
```

```
{
        @Override
        public int compare(Integer o1, Integer o2) {
            return o2-o1;
        }
    });
```

实现了一个初始大小为 11 的大顶堆。这里只是简单的传入一个自定义的 Comparator 函数，就可以实现大顶堆了。

ArrayDeque

实现了 Deque 接口，ArrayDeque 提供了三个构造方法，分别是默认容量，指定容量及依据给定的集合中的元素进行创建。默认容量为 16。

```
public ArrayDeque() {
    elements = new Object[16];
}

public ArrayDeque(int numElements) {
    allocateElements(numElements);
}

public ArrayDeque(Collection<? extends E> c) {
    allocateElements(c.size());
    addAll(c);
}
```

遍历：

ArrayDeque 提供了两个迭代器，一个是正向的迭代器，另一个是反向的迭代器。

```
    public Iterator<E> iterator() {
        return new DeqIterator();
```

```
    }

    public Iterator<E> descendingIterator() {
        return new DescendingIterator();
    }
```

ArrayDeque 对数组的大小(即队列的容量)有特殊的要求，必须是 2^n。通过 allocateElements 方法计算初始容量：

```
private void allocateElements(int numElements) {
    int initialCapacity = MIN_INITIAL_CAPACITY;
    // Find the best power of two to hold elements.
    // Tests "<=" because arrays aren't kept full.
    if (numElements >= initialCapacity) {
        initialCapacity = numElements;
        initialCapacity |= (initialCapacity >>>  1);
        initialCapacity |= (initialCapacity >>>  2);
        initialCapacity |= (initialCapacity >>>  4);
        initialCapacity |= (initialCapacity >>>  8);
        initialCapacity |= (initialCapacity >>> 16);
        initialCapacity++;

        if (initialCapacity < 0)   // Too many elements, must back off
            initialCapacity >>>= 1;// Good luck allocating 2 ^ 30 elements
    }
    elements = new Object[initialCapacity];
}
```

>>>是无符号右移操作，|是位或操作，经过五次右移和位或操作可以保证得到大小为 2^k-1 的数。

在 ArrayDeque 中数组是当作环形来使用的，索引 0 看作是紧挨着索引(length-1)之后的。

添加操作：

```
    public void addFirst(E e) {
        if (e == null)
            throw new NullPointerException();
        elements[head = (head - 1) & (elements.length - 1)] = e;
        if (head == tail)
            doubleCapacity();
    }
    public void addLast(E e) {
        if (e == null)
            throw new NullPointerException();
        elements[tail] = e;
        if ( (tail = (tail + 1) & (elements.length - 1)) == head)
            doubleCapacity();
    }
```

扩容操作：

在每次添加元素后，如果头索引和尾部索引相遇，则说明数组空间已满，需要进行扩容操作。

ArrayDeque 每次扩容都会在原有的容量上翻倍，这也是对容量必须是 2 的幂次方的保证。

```
private void doubleCapacity() {
    assert head == tail; //扩容时头部索引和尾部索引肯定相等
    int p = head;
    int n = elements.length;
    //头部索引到数组末端(length-1 处)共有多少元素
    int r = n - p; // number of elements to the right of p
    //容量翻倍
    int newCapacity = n << 1;
    //容量过大，溢出了
    if (newCapacity < 0)
        throw new IllegalStateException("Sorry, deque too big");
```

```
    //分配新空间
    Object[] a = new Object[newCapacity];
    //复制头部索引到数组末端的元素到新数组的头部
    System.arraycopy(elements, p, a, 0, r);
    //复制其余元素
    System.arraycopy(elements, 0, a, r, p);
    elements = a;
    //重置头尾索引
    head = 0;
    tail = n;
}
```

Juc 包下面的

ArrayBlockingQueue

ArrayBlockingQueue 是数组实现的线程安全的有界的阻塞队列。线程安全是指，ArrayBlockingQueue 内部通过“互斥锁”保护竞争资源，实现了多线程对竞争资源的互斥访问。而有界，则是指 ArrayBlockingQueue 对应的数组是有界限的。 阻塞队列，是指多线程访问竞争资源时，当竞争资源已被某线程获取时，其它要获取该资源的线程需要阻塞等待；而且，ArrayBlockingQueue 是按 FIFO（先进先出）原则对元素进行排序，元素都是从尾部插入到队列，从头部开始返回。

注意：ArrayBlockingQueue 不同于 ConcurrentLinkedQueue，ArrayBlockingQueue 是数组实现的，并且是有界限的；而 ConcurrentLinkedQueue 是链表实现的，是无界限的。

1. ArrayBlockingQueue 继承于 AbstractQueue，并且它实现了 BlockingQueue 接口。

2. ArrayBlockingQueue 内部是通过 Object[]数组保存数据的，也就是说 ArrayBlockingQueue 本质上是通过数组实现的。ArrayBlockingQueue 的大小，即数组的容量是创建 ArrayBlockingQueue 时指定的。

3. ArrayBlockingQueue 与 ReentrantLock 是组合关系，ArrayBlockingQueue 中包含一个

ReentrantLock 对象(lock)。ReentrantLock 是可重入的互斥锁，ArrayBlockingQueue 就是根据该互斥锁实现“多线程对竞争资源的互斥访问”。而且，ReentrantLock 分为公平锁和非公平锁，关于具体使用公平锁还是非公平锁，在创建 ArrayBlockingQueue 时可以指定；而且，ArrayBlockingQueue 默认会使用非公平锁。

4. ArrayBlockingQueue 与 Condition 是组合关系，ArrayBlockingQueue 中包含两个 Condition 对象(notEmpty 和 notFull)。而且，Condition 又依赖于 ArrayBlockingQueue 而存在，通过 Condition 可以实现对 ArrayBlockingQueue 的更精确的访问 -- (01)若某线程(线程 A)要取数据时，数组正好为空，则该线程会执行 notEmpty.await()进行等待；当其它某个线程(线程 B)向数组中插入了数据之后，会调用 notEmpty.signal()唤醒“notEmpty 上的等待线程”。此时，线程 A 会被唤醒从而得以继续运行。(02)若某线程(线程 H)要插入数据时，数组已满，则该线程会它执行 notFull.await()进行等待；当其它某个线程(线程 I)取出数据之后，会调用 notFull.signal()唤醒“notFull 上的等待线程”。此时，线程 H 就会被唤醒从而得以继续运行。

变量声明

```
final Object[] items;     //队列数组
int takeIndex;            //添加队列时的下标
int putIndex;             //出队时的下标
int count;                //队列中元素个数
final ReentrantLock lock;                  //锁
private final Condition notEmpty;          //与取操作对应，在队列中没有元素的时候，
                                             notEmpty 会等待，当添加操作完成时，
                                             notEmpty 会被唤醒
private final Condition notFull;           //与添加操作对应，在队列中元素满了的时候，
                                             notFull 会等待，当取操作完成时，notFull 会
                                             被唤醒
transient Itrs itrs = null;                //迭代器
```

底层是一个 Object 数组，ArrayBlockingQueue 有三个构造函数：

构造方法

```
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);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}
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();
        }
    }
```

由构造函数可以看出，使用 ArrayBlockingQueue 使，至少要传递一个队列大小的值。

`public ArrayBlockingQueue(int capacity);`构造函数创建了一个长度为 capacity 长度的带有非公平锁的队列

`public ArrayBlockingQueue(int capacity, boolean fair);`构造函数创建了一个长度为 capacity 长度的可由使用者去设置公平与否的队列

`public ArrayBlockingQueue(int capacity,boolean fair,Collection<? extends E> c);`构造函数创建了一个长度为 capacity 长度的可由使用者去设置公平与否的队列，且这个队列可以传入一个容器对这个队列进行填充。

添加操作

```
/** 取出时的下标 */
int takeIndex;
/** 添加时的下标 */
int putIndex;
/** 队列中已有的个数 */
int count;

public boolean add(E e) {
    return super.add(e);
}
//super.add
public boolean add(E e) {
    if (offer(e))
```

```
        return true;
    else
        throw new IllegalStateException("Queue full");
}
public boolean offer(E e) {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        if (count == items.length)
            return false;
        else {
            enqueue(e);
            return true;
        }
    } finally {
        lock.unlock();
    }
}
private void enqueue(E x) {
    // assert lock.getHoldCount() == 1;
    // assert items[putIndex] == null;
    final Object[] items = this.items;
    items[putIndex] = x;
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
    notEmpty.signal();
}
```

添加操作为 add()，在 add 中仅有一行代码，那就是调用父类的 add 方法，父类的 add

方法最终调用了 offer 方法，这个方法也是在本类中进行实现的。Offer()方法首先进行判空操作，如果要添加的元素为空，就会抛出异常 NullPointerException。如果不为空继续往下运行，对本代码加锁操作，它是为这个类的原本的 lock 添加了一个引用，所以一直在用同一个锁。在锁中进行了一次判断，判断队列是否满了，如果队列已满返回 false，如果队列没有满，调用 enqueue 方法进行添加元素，然后将不为空的 Condition 进行唤醒，使得等待中的取出操作进行取出。

ArrayBlockingQueue 有两种添加方法：offer()和 put()方法

Offer()方法在数组满了后，在进行添加会直接返回失败，而 put()方法在数组满了时，在进行添加会执行自旋操作，直到数组取出了一个元素，它在进行添加操作。

取出操作

```
/** 取出时的下标 */
int takeIndex;
/** 添加时的下标 */
int putIndex;
/** 队列中已有的个数 */
int count;
```

ArrayBlockingQueue 的取出函数有两种：poll()和 take() 首先看一下这两个函数的源码：

```
public E poll() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return (count == 0) ? null : dequeue();
    } finally {
        lock.unlock();
    }
}
```

```
    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == 0)
                notEmpty.await();
            return dequeue();
        } finally {
            lock.unlock();
        }
    }
```

由代码可以看出一个是正常操作，另一个是有阻塞的操作，正常操作时如果队列的长度为 0，则会直接返回 null。有阻塞的操作时如果队列的长度为 0，不为空的 Condition 就会等待，直到有一个添加的操作。

接下来我们在看看 dequeue 这个方法如何实现的：

```
private E dequeue() {
    // assert lock.getHoldCount() == 1;
    // assert items[takeIndex] != null;
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
    E x = (E) items[takeIndex];
    items[takeIndex] = null;
    if (++takeIndex == items.length)
        takeIndex = 0;
    count--;
    if (itrs != null)
        itrs.elementDequeued();
    notFull.signal();
    return x;
}
```

在执行完 dequeue 函数后都会对没有满的操作 Condition 进行唤醒，使得等待中的添加操作可以进行添加。

遍历操作

因为 ArrayBlockingQueue 实现的是 BlockingQueue 接口，BlockingQueue 接口继承了 Queue 接口，Queue 接口继承了 Collection 接口，所以 ArrayBlockingQueue 中也会有一个实现迭代器的操作：

```
public Iterator<E> iterator() {
    return new Itr();
}
```

我们在看看 Itr()这个类如何实现的（只留下了有必要的函数）：

```
    public boolean hasNext() {
        // assert lock.getHoldCount() == 0;
        if (nextItem != null)
            return true;
        noNext();
        return false;
    }

    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;
        }
```

LinkedBlockingQueue

LinkedBlockingQueue 是一个单向链表实现的阻塞队列。该队列按 FIFO（先进先出）排序元素，新元素插入到队列的尾部，并且队列获取操作会获得位于队列头部的元素。链接队列的吞吐量通常要高于基于数组的队列，但是在大多数并发应用程序中，其可预知的性能要低。

此外，LinkedBlockingQueue 还是可选容量的(防止过度膨胀)，即可以指定队列的容量。如果不指定，默认容量大小等于 Integer.MAX_VALUE。

1. LinkedBlockingQueue 继承于 AbstractQueue，它本质上是一个 FIFO(先进先出)的队列。

2. LinkedBlockingQueue 实现了 BlockingQueue 接口，它支持多线程并发。当多线程竞争同一个资源时，某线程获取到该资源之后，其它线程需要阻塞等待。

3. LinkedBlockingQueue 是通过单链表实现的。

(01) head 是链表的表头。取出数据时，都是从表头 head 处插入。

(02) last 是链表的表尾。新增数据时，都是从表尾 last 处插入。

(03) count 是链表的实际大小，即当前链表中包含的节点个数。

(04) capacity 是列表的容量，它是在创建链表时指定的。

(05) putLock 是插入锁，takeLock 是取出锁；notEmpty 是“非空条件”，notFull 是“未满条件”。通过它们对链表进行并发控制。

LinkedBlockingQueue 在实现“多线程对竞争资源的互斥访问”时，对于“插入”和“取出(删除)”操作分别使用了不同的锁。对于插入操作，通过“插入锁 putLock”进行同步；对于取出操作，通过“取出锁 takeLock”进行同步。

此外，插入锁 putLock 和“非满条件 notFull”相关联，取出锁 takeLock 和“非空条件 notEmpty”相关联。通过 notFull 和 notEmpty 更细腻的控制锁。

变量声明

```
private final int capacity;
private final AtomicInteger count = new AtomicInteger();
transient Node<E> head;
private transient Node<E> last;
private final ReentrantLock takeLock = new ReentrantLock();
private final Condition notEmpty = takeLock.newCondition();
private final ReentrantLock putLock = new ReentrantLock();
private final Condition notFull = putLock.newCondition();
```

构造方法

```
public LinkedBlockingQueue() {
    this(Integer.MAX_VALUE);
}
```

```
public LinkedBlockingQueue(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
    last = head = new Node<E>(null);
}
public LinkedBlockingQueue(Collection<? extends E> c) {
    this(Integer.MAX_VALUE);
    final ReentrantLock putLock = this.putLock;
    putLock.lock(); // Never contended, but necessary for visibility
    try {
        int n = 0;
        for (E e : c) {
            if (e == null)
                throw new NullPointerException();
            if (n == capacity)
                throw new IllegalStateException("Queue full");
            enqueue(new Node<E>(e));
            ++n;
        }
        count.set(n);
    } finally {
        putLock.unlock();
    }
}
```

LinkedBlockingQueue()：构造函数，默认链表长度为 Integer.MAX_VALUE

public LinkedBlockingQueue(int capacity)：构造函数需要传入链表长度，初始化链表的长度为 capacity

public LinkedBlockingQueue(Collection<? extends E> c)：需要传入一个集合类，然后初始化的链表长度为 Integer.MAX_VALUE，并将集合中的元素添加到链表中。

添加操作

LinkedBlockingQueue 有两种添加方法：offer(E e)和 put(E e)

```
    public boolean offer(E e) {
        if (e == null) throw new NullPointerException();
        final AtomicInteger count = this.count;
        if (count.get() == capacity)
            return false;
        int c = -1;
        Node<E> node = new Node<E>(e);
        final ReentrantLock putLock = this.putLock;
        putLock.lock();
        try {
            if (count.get() < capacity) {
                enqueue(node);
                c = count.getAndIncrement();
                if (c + 1 < capacity)
                    notFull.signal();
            }
        } finally {
            putLock.unlock();
        }
        if (c == 0)
            signalNotEmpty();
        return c >= 0;
    }
    public void put(E e) throws InterruptedException {
        if (e == null) throw new NullPointerException();
        int c = -1;
        Node<E> node = new Node<E>(e);
```

```
        final ReentrantLock putLock = this.putLock;
        final AtomicInteger count = this.count;
        putLock.lockInterruptibly();
        try {
            while (count.get() == capacity) {
                notFull.await();
            }
            enqueue(node);
            c = count.getAndIncrement();
            if (c + 1 < capacity)
                notFull.signal();
        } finally {
            putLock.unlock();
        }
        if (c == 0)
            signalNotEmpty();
    }
```

根据上面的两种添加方法的源代码我们可以看出，offer 方法的添加如果遇到队列所拥有的长度大于 capacity 的长度后，会直接 false，put 方法的添加如果遇到队列所拥有的长度大于 capacity 的长度后，会自旋等待，直到有出队的操作后，它才会添加的。

下面我们看一下添加的操作：

```
    private void enqueue(Node<E> node) {
        // assert putLock.isHeldByCurrentThread();
        // assert last.next == null;
        last = last.next = node;
    }
```

添加的操作实际很简单，就是直接将队尾指针的 next 指向新的队列节点上，就完成了。

取出操作

LinkedBlockingQueue 的取出操作有两个函数：poll()和 take()

```
public E poll() {
    final AtomicInteger count = this.count;
    if (count.get() == 0)
        return null;
    E x = null;
    int c = -1;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        if (count.get() > 0) {
            x = dequeue();
            c = count.getAndDecrement();
            if (c > 1)
                notEmpty.signal();
        }
    } finally {
        takeLock.unlock();
    }
    if (c == capacity)
        signalNotFull();
    return x;
}
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        while (count.get() == 0) {
```

```
                notEmpty.await();
            }
            x = dequeue();
            c = count.getAndDecrement();
            if (c > 1)
                notEmpty.signal();
        } finally {
            takeLock.unlock();
        }
        if (c == capacity)
            signalNotFull();
        return x;
    }
```

根据上面两个添加方法可以看到，poll 方法会在队列数为 0 的时候直接返回 false，而 take()方法在队列为 0 时会自旋等待，直到队列的长度大于 0 后继续执行取出操作。

取出操作的源代码：

```
    private E dequeue() {
        // assert takeLock.isHeldByCurrentThread();
        // assert head.item == null;
        Node<E> h = head;
        Node<E> first = h.next;
        h.next = h; // help GC
        head = first;
        E x = first.item;
        first.item = null;
        return x;
    }
```

他的取出操作时将 head 节点取出然后使用 head 的下一个节点为头结点

遍历操作

因为 LinkedBlockingQueue 实现的是 BlockingQueue 接口，BlockingQueue 接口继承了 Queue 接口，Queue 接口继承了 Collection 接口，所以 LinkedBlockingQueue 中也会有一个实现迭代器的操作：

```
public Iterator<E> iterator() {
    return new Itr();
}
```

PriorityBlockingQueue

这是一个无界有序的阻塞队列，排序规则和之前介绍的 PriorityQueue 一致，只是增加了阻塞操作。同样的该队列不支持插入 null 元素，同时不支持插入非 comparable 的对象。它的迭代器并不保证队列保持任何特定的顺序，如果想要顺序遍历，考虑使用 Arrays.sort(pq.toArray())。该类不保证同等优先级的元素顺序，如果你想要强制顺序，就需要考虑自定义顺序或者是 Comparator 使用第二个比较属性。

变量声明

```
//如果创建 PriorityBlockingQueue 对象的时候没有创建有参的对象，数组的默认长度为 DEFAULT_INITIAL_CAPACITY。
private static final int DEFAULT_INITIAL_CAPACITY = 11;
//在扩容的时候，如果扩容的长度大于 MAX_ARRAY_SIZE 了，将长度扩容到这个值。
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
//队列的底层数组
private transient Object[] queue;
//队列中已有元素的长度
private transient int size;
//这个参数是让这个堆怎么进行比较的
private transient Comparator<? super E> comparator;
//添加和获取操作公用这一把锁
private final ReentrantLock lock;
```

```
private final Condition notEmpty;
private transient volatile int allocationSpinLock;
private PriorityQueue<E> q;
```

构造方法

```
public PriorityBlockingQueue() {
    this(DEFAULT_INITIAL_CAPACITY, null);
}
public PriorityBlockingQueue(int initialCapacity) {
    this(initialCapacity, null);
}
public PriorityBlockingQueue(int initialCapacity,
                             Comparator<? super E> comparator) {
    if (initialCapacity < 1)
        throw new IllegalArgumentException();
    this.lock = new ReentrantLock();
    this.notEmpty = lock.newCondition();
    this.comparator = comparator;
    this.queue = new Object[initialCapacity];
}
public PriorityBlockingQueue(Collection<? extends E> c) {
    this.lock = new ReentrantLock();
    this.notEmpty = lock.newCondition();
    boolean heapify = true; // true if not known to be in heap order
    boolean screen = true;  // true if must screen for nulls
    if (c instanceof SortedSet<?>) {
        SortedSet<? extends E> ss = (SortedSet<? extends E>) c;
        this.comparator = (Comparator<? super E>) ss.comparator();
        heapify = false;
```

```
        }
        else if (c instanceof PriorityBlockingQueue<?>) {
            PriorityBlockingQueue<? extends E> pq =
                (PriorityBlockingQueue<? extends E>) c;
            this.comparator = (Comparator<? super E>) pq.comparator();
            screen = false;
            if (pq.getClass() == PriorityBlockingQueue.class) // exact match
                heapify = false;
        }
        Object[] a = c.toArray();
        int n = a.length;
        // If c.toArray incorrectly doesn't return Object[], copy it.
        if (a.getClass() != Object[].class)
            a = Arrays.copyOf(a, n, Object[].class);
        if (screen && (n == 1 || this.comparator != null)) {
            for (int i = 0; i < n; ++i)
                if (a[i] == null)
                    throw new NullPointerException();
        }
        this.queue = a;
        this.size = n;
        if (heapify)
            heapify();
    }
```

public PriorityBlockingQueue(); 构造函数队列长度为默认长度DEFAULT_INITIAL_CAPACITY=11，比较方法不自定。

public PriorityBlockingQueue(int initialCapacity); 构造方法队列长度为initialCapacity，比较方法不自定。

public PriorityBlockingQueue(int initialCapacity,
Comparator<? super E> comparator); 构造函数的队列长

度为 initialCapacity，比较方法自定。

`public PriorityBlockingQueue(Collection<? extends E> c);` 构造函数的队列长度为 c 集合的长度。

添加操作

PriorityBlockingQueue 的添加操作虽说有 3 个：put(e)、add(e)和 offer(e)，但是前两个方法中都只有一个调用 offer 的语句。所以我们就亮出 offer 方法的源码

```
public boolean offer(E e) {
    if (e == null)
        throw new NullPointerException();
    final ReentrantLock lock = this.lock;
    lock.lock();
    int n, cap;
    Object[] array;
    while ((n = size) >= (cap = (array = queue).length))
        tryGrow(array, cap);
    try {
        Comparator<? super E> cmp = comparator;
        if (cmp == null)
            siftUpComparable(n, e, array);
        else
            siftUpUsingComparator(n, e, array, cmp);
        size = n + 1;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
    return true;
}
```

我们可以看到添加方法第一件事做的是加锁，所以此方法是线程安全的，第二件事是对其判断是否需要扩容操作。如果需要就去扩容，直到不需要扩容为止。然后会根据是否有比较方法进行选择调用哪个最小堆的调整方法。下面是两个添加最小堆的调整

```
private static <T> void siftUpComparable(int k, T x, Object[] array) {
    Comparable<? super T> key = (Comparable<? super T>) x;
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        Object e = array[parent];
        if (key.compareTo((T) e) >= 0)
            break;
        array[k] = e;
        k = parent;
    }
    array[k] = key;
}
private static <T> void siftUpUsingComparator(int k, T x, Object[] array,
                                   Comparator<? super T> cmp) {
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        Object e = array[parent];
        if (cmp.compare(x, (T) e) >= 0)
            break;
        array[k] = e;
        k = parent;
    }
    array[k] = x;
}
```

由这两个函数可以看到他们的调整都是由新添加进来的元素进行寻找父节点，与父节点比较，如果符合比较函数那么继续比较知道 k 小于 0 或不符合比较函数的时候。

下面介绍一下扩容函数：

```
private void tryGrow(Object[] array, int oldCap) {
    lock.unlock(); // must release and then re-acquire main lock
    Object[] newArray = null;
    if (allocationSpinLock == 0 &&
        UNSAFE.compareAndSwapInt(this, allocationSpinLockOffset,
                                 0, 1)) {
        try {
            int newCap = oldCap + ((oldCap < 64) ?
                                   (oldCap + 2) : // grow faster if small
                                   (oldCap >> 1));
            if (newCap - MAX_ARRAY_SIZE > 0) {    // possible overflow
                int minCap = oldCap + 1;
                if (minCap < 0 || minCap > MAX_ARRAY_SIZE)
                    throw new OutOfMemoryError();
                newCap = MAX_ARRAY_SIZE;
            }
            if (newCap > oldCap && queue == array)
                newArray = new Object[newCap];
        } finally {
            allocationSpinLock = 0;
        }
    }
    if (newArray == null) // back off if another thread is allocating
        Thread.yield();
    lock.lock();
    if (newArray != null && queue == array) {
        queue = newArray;
        System.arraycopy(array, 0, newArray, 0, oldCap);
    }
}
```

```
if (allocationSpinLock == 0 &&
        UNSAFE.compareAndSwapInt(this, allocationSpinLockOffset,0, 1)) {
```

这段代码是用于分配的自旋锁，通过 CAS 获取。这段代码可以防止扩容多次。如果原本长度小于 64 时，扩容机制会扩容到原来长度的 2 倍在加 2，如果原本长度大于 64 时，扩容机制会扩容到 1.5 倍。如果扩容后超过了 MAX_ARRAY_SIZE 的值且原数组长度不小于零（即没有越界）或不大于 MAX_ARRAY_SIZE（扩容后会越界）即可扩容，且扩容的大小为 MAX_ARRAY_SIZE。

取出操作

PriorityBlockingQueue 有两个出队列函数：poll()和 take()

```
    public E poll() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            return dequeue();
        } finally {
            lock.unlock();
        }
    }
    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        E result;
        try {
            while ( (result = dequeue()) == null)
                notEmpty.await();
        } finally {
            lock.unlock();
        }
```

```
        return result;
    }
```

根据以上操作可以看到 poll 方法直接将信息 dequeue 方法的信息返回回来，而 take 方法在数组队列为 0 的时候会出现等待操作。直到有添加的操作，使数组的长度大于 0，然后退出等待。下面我们看一下 dequeue 函数的源代码：

```
    private E dequeue() {
        int n = size - 1;
        if (n < 0)
            return null;
        else {
            Object[] array = queue;
            E result = (E) array[0];
            E x = (E) array[n];
            array[n] = null;
            Comparator<? super E> cmp = comparator;
            if (cmp == null)
                siftDownComparable(0, x, array, n);
            else
                siftDownUsingComparator(0, x, array, n, cmp);
            size = n;
            return result;
        }
    }
```

由这个函数可以看到首先会判断队列中是否还有节点。如果没有直接的返回 null。如果还有节点就会根据有没有比较函数去分别调用 siftDownComparable 函数和 siftDownUsing Comparator 函数。下面分析一下这两个函数的源码，查看是如何将节点删除的：

```
    private static <T> void siftDownComparable(int k, T x, Object[] array,
                                               int n) {
        if (n > 0) {
            Comparable<? super T> key = (Comparable<? super T>)x;
```

```
        int half = n >>> 1;           // loop while a non-leaf
        while (k < half) {
            int child = (k << 1) + 1; // assume left child is least
            Object c = array[child];
            int right = child + 1;
            if (right < n &&
                ((Comparable<? super T>) c).compareTo((T) array[right]) > 0)
                c = array[child = right];
            if (key.compareTo((T) c) <= 0)
                break;
            array[k] = c;
            k = child;
        }
        array[k] = key;
    }
}
private static <T> void siftDownUsingComparator(int k, T x, Object[] array,
                                                int n,
                                                Comparator<? super T> cmp) {
    if (n > 0) {
        int half = n >>> 1;
        while (k < half) {
            int child = (k << 1) + 1;
            Object c = array[child];
            int right = child + 1;
            if (right < n && cmp.compare((T) c, (T) array[right]) > 0)
                c = array[child = right];
            if (cmp.compare(x, (T) c) <= 0)
                break;
            array[k] = c;
```

```
                k = child;
            }
            array[k] = x;
        }
    }
```

无论是 siftDownComparable 函数还是 siftDownUsingComparator 函数都会出现去除这个节点的操作。这个就是去除堆顶然后对堆进行调整的操作，首先他会利用 result 变量将要去除的 0 位上的元素保存，然后 x 变量将堆中最后一个叶子节点记录。然后就会从堆顶进行往下遍历操作，循环操作为（比如比较方法中式选出最小的值）：获取当前节点的左孩子和右孩子，利用选出其中最小的的一个节点进行与 x（即 key 节点，当前已经赋值给可以 key）比较，如果两个孩子中最小的比本节点小，那么将这个最小的节点赋值到父节点 k 上，然后将父节点的 k 移到较小的孩子上，即这个时候这个最小的孩子又为新一轮的循环的父节点。继续循环，直到将这个 x 节点放在堆中

遍历操作

因为 PriorityBlockingQueue 实现的是 BlockingQueue 接口，BlockingQueue 接口继承了 Queue 接口，Queue 接口继承了 Collection 接口，所以 PriorityBlockingQueue 中也会有一个实现迭代器的操作：

```
public Iterator<E> iterator() {
    return new Itr(toArray());
}
```

可使用的入队和出队函数：

入队：put()

出队：take()、poll()

SynchronousQueue

在了解 SynchronousQueue 队列的时候我们先了解一下 CAS 的原理吧，因为

SynchronousQueue 使用了太多的 CAS 操作了：

CAS 机制中使用了 3 个基本操作数：内存地址 V，旧的预期值 A，要修改的新值 B。

更新一个变量的时候，只有当变量的预期值 A 和内存地址 V 当中的实际值相同时，才会将内存地址 V 对应的值修改为 B，而在匹配内存地址 V 和旧的预期值 A 的时候会出现自旋操作，直到等到 V 和 A 相等的时候，才进行修改，将 A 的值改为 B。

因为本文是介绍 SynchronousQueue 的，所以 CAS 只能给大家介绍这么多了！下面是开启 Synchronous 源码的时候了。

SynchronousQueue 与其他的 BlockingQueue 不同，它是没有容量的，每一个 put 都会等待一个 take，否则不会添加元素的，SynchronousQueue 可分为公平和不公平的，这里有点像 ReentrantLock 一样可以选择公平和不公平，但是底层实现却不是一样的，SynchronousQueue 的公平是由一个队列实现的，不公平是由一个栈实现的，这块的队列和栈会在下面进行详细讲解。

首先我们看一下 SynchronousQueue 的变量：

```
static final int NCPUS = Runtime.getRuntime().availableProcessors();
static final int maxTimedSpins = (NCPUS < 2) ? 0 : 32;
static final int maxUntimedSpins = maxTimedSpins * 16;
static final long spinForTimeoutThreshold = 1000L;
private ReentrantLock qlock;
private WaitQueue waitingProducers;
private WaitQueue waitingConsumers;
```

//Transferer 在 SynchronousQueue 中是一个很重要的变量，入队、出队都会对整个变量操作，而他只是一个抽象类，他的实现类有两个 TransferQueue<E>和 TransferStack<E>，可以看到 TransferQueue 类是公平的，TransferStack 类是不公平的（可由以下说明知道）

```
private transient volatile Transferer<E> transferer;
```

在看一下 SynchronousQueue 的初始化方法：

```
public SynchronousQueue() {
    this(false);
```

```
    }
    public SynchronousQueue(boolean fair) {
        transferer = fair ? new TransferQueue<E>() : new TransferStack<E>();
    }
```

根据两个构造函数可以看到实际执行的都是第二个构造函数，如果构造不带参数的构造函数会创建一个不公平的 SynchronousQueue，直接调用带参的构造函数，会由调用这决定使用公平的还是不公平的 SynchronousQueue。

SynchronousQueued 的入队函数有 offer(E)和 put(E)

```
    public boolean offer(E e) {
        if (e == null) throw new NullPointerException();
        return transferer.transfer(e, true, 0) != null;
    }
    public void put(E e) throws InterruptedException {
        if (e == null) throw new NullPointerException();
        if (transferer.transfer(e, false, 0) == null) {
            Thread.interrupted();
            throw new InterruptedException();
        }
    }
```

可以看到这两个方法最后都调用了 transferer 的 transfer 方法了，所以具体怎么操作的入队等待讲解 transferer 两个子类的时候进行说明。

SynchronousQueue 的出队函数有 poll()和 take()方法

```
    public E poll() {
        return transferer.transfer(null, true, 0);
    }
    public E take() throws InterruptedException {
        E e = transferer.transfer(null, false, 0);
        if (e != null)
```

```
            return e;
        Thread.interrupted();
        throw new InterruptedException();
    }
```

可以看到两个出队的方法也是调用了 transferer 的 transfer 方法。

Synchronous 的操作入队出队操作都是使用了 transferer 的 transfer 方法。所以接下来我会从 transterer 的 transfer 当做入口继续进行分析：

```
    abstract static class Transferer<E> {
        abstract E transfer(E e, boolean timed, long nanos);
    }
```

Transferer 类是一个抽象类，里面仅有一个方法 transfer，也就是说明在 Synchronous 中实例化的 Transferer<E> transferer 变量无论 Transferer 子类的方法中有多少方法，transferer 这个变量都只能调用 transfer 这个方法，这就更能鉴定我以 transfer 函数为切入点是正确的了，好，那么接下来我们分别分析 TransferQueue 类的 transfer 方法和 TransferStack 类的 transfer 方法。

TransferQueue 类：

首先看 TransferQueue 中队列节点中都含有什么元素：

```
static final class QNode {
    volatile QNode next;          // next node in queue
    volatile Object item;         // CAS'ed to or from null
    volatile Thread waiter;       // to control park/unpark
    final boolean isData;
    QNode(Object item, boolean isData) {
         this.item = item;
         this.isData = isData;
     }
}
```

TransferQueue 类的构造函数：

```
TransferQueue() {
        QNode h = new QNode(null, false); // initialize to dummy node.
        head = h;
        tail = h;
 }
```

可以看到 TransferQueue 这个队列在初始化的时候先会创建一个虚拟节点 dummy node，这也就说明使用 TransferQueue 的时候无论链表中有没有元素，TransferQueue 中都会有一个虚拟节点存在。

公平性调用 TransferQueue 的 transfer 方法：

```
E transfer(E e, boolean timed, long nanos) {
    QNode s = null;
    // 当前节点模式
    boolean isData = (e != null);

    for (;;) {
        QNode t = tail;
        QNode h = head;
        // 头、尾节点 为 null，没有初始化
        if (t == null || h == null)
            continue;

        // 头尾节点相等（队列为 null） 或者当前节点和队列节点模式一样
        if (h == t || t.isData == isData) {
            // tn = t.next
            QNode tn = t.next;
            // t != tail 表示已有其他线程操作了，修改了 tail，重新再来
            if (t != tail)
```

```
                continue;
            // tn != null，表示已经有其他线程添加了节点，tn 推进，重新处理
            if (tn != null) {
                // 当前线程帮忙推进尾节点，就是尝试将 tn 设置为尾节点
                advanceTail(t, tn);
                continue;
            }
            //  调用的方法的 wait 类型的，并且 超时了，直接返回 null
            // timed 在 take 操作阐述
            if (timed && nanos <= 0)
                return null;

            // s == null，构建一个新节点 Node
            if (s == null)
                s = new QNode(e, isData);

            // 将新建的节点加入到队列中，如果不成功，继续处理
            if (!t.casNext(null, s))
                continue;

            // 替换尾节点
            advanceTail(t, s);

            // 调用 awaitFulfill，若节点是 head.next，则进行自旋
            // 若不是的话，直接 block，直到有其他线程 与之匹配，或它自己进行线
程的中断
            Object x = awaitFulfill(s, e, timed, nanos);
            // 若返回的 x == s 表示，当前线程已经超时或者中断，不然的话 s == null
或者是匹配的节点
            if (x == s) {
```

```
            // 清理节点 S
            clean(t, s);
            return null;
        }
        // isOffList：用于判断节点是否已经从队列中离开了
        if (!s.isOffList()) {
            // 尝试将 S 节点设置为 head，移出 t
            advanceHead(t, s);
            if (x != null)
                s.item = s;
            // 释放线程 ref
            s.waiter = null;
        }
        // 返回
        return (x != null) ? (E)x : e;
    }

    // 这里是从 head.next 开始，因为 TransferQueue 总是会存在一个 dummy node
节点
    else {
        // 节点
        QNode m = h.next;

        // 不一致读，重新开始
        // 有其他线程更改了线程结构
        if (t != tail || m == null || h != head)
            continue;
        /**
         * 生产者 producer 和消费者 consumer 匹配操作
         */
```

```
                Object x = m.item;
                // isData == (x != null)：判断 isData 与 x 的模式是否相同，相同表
示已经匹配了
                // x == m ： m 节点被取消了
                // !m.casItem(x, e)：如果尝试将数据 e 设置到 m 上失败
                if (isData == (x != null) ||  x  == m || !m.casItem(x, e)) {
                    // 将 m 设置为头结点，h 出列，然后重试
                    advanceHead(h, m);
                    continue;
                }

                // 成功匹配了，m 设置为头结点 h 出列，向前推进
                advanceHead(h, m);
                // 唤醒 m 上的等待线程
                LockSupport.unpark(m.waiter);
                return (x != null) ? (E)x : e;
            }
        }
    }
```

整个 transfer 的算法如下：

1. 如果队列为 null 或者尾节点模式与当前节点模式一致，则尝试将节点加入到等待队列中（采用自旋的方式），直到被匹配或、超时或者取消。匹配成功的话要么返回 null（producer 返回的）要么返回真正传递的值（consumer 返回的），如果返回的是 node 节点本身则表示当前线程超时或者取消了。

2. 如果队列不为 null，且队列的节点是当前节点匹配的节点，则进行数据的传递匹配并返回匹配节点的数据

3. 在整个过程中都会检测并帮助其他线程推进

当队列为空时，节点入列然后通过调用 awaitFulfill()方法自旋，该方法主要用于自旋/阻塞节点，直到节点被匹配返回或者取消、中断。

```
    Object awaitFulfill(QNode s, E e, boolean timed, long nanos) {
```

```
        // 超时控制
        final long deadline = timed ? System.nanoTime() + nanos : 0L;
        Thread w = Thread.currentThread();
        // 自旋次数
        // 如果节点 Node 恰好是 head 节点，则自旋一段时间，这里主要是为了效率问题，如果里面阻塞，会存在唤醒、线程上下文切换的问题
        // 如果生产者、消费者者里面到来的话，就避免了这个阻塞的过程
        int spins = ((head.next == s) ?
                (timed ? maxTimedSpins : maxUntimedSpins) : 0);
        // 自旋
        for (;;) {
            // 线程中断了，剔除当前节点
            if (w.isInterrupted())
                s.tryCancel(e);

            // 如果线程进行了阻塞 -> 唤醒或者中断了，那么 x != e 肯定成立，直接返回当前节点即可
            Object x = s.item;
            if (x != e)
                return x;
            // 超时判断
            if (timed) {
                nanos = deadline - System.nanoTime();
                // 如果超时了，取消节点,continue，在 if(x != e)肯定会成立，直接返回 x
                if (nanos <= 0L) {
                    s.tryCancel(e);
                    continue;
                }
            }
```

```
            // 自旋- 1
            if (spins > 0)
                --spins;

            // 等待线程
            else if (s.waiter == null)
                s.waiter = w;

            // 进行没有超时的 park
            else if (!timed)
                LockSupport.park(this);

            // 自旋次数过了，直接 + timeout 方式 park
            else if (nanos > spinForTimeoutThreshold)
                LockSupport.parkNanos(this, nanos);
        }
    }
```

在自旋/阻塞过程中做了一点优化，就是判断当前节点是否为对头元素，如果是的则先自旋，如果自旋次数过了，则才阻塞，这样做的主要目的就在如果生产者、消费者立马来匹配了则不需要阻塞，因为阻塞、唤醒会消耗资源。在整个自旋的过程中会不断判断是否超时或者中断了，如果中断或者超时了则调用 tryCancel()取消该节点。

tryCancel

```
    void tryCancel(Object cmp) {
        UNSAFE.compareAndSwapObject(this, itemOffset, cmp, this);
    }
```

取消过程就是将节点的 item 设置为自身（itemOffset 是 item 的偏移量）。所以在调用 awaitFulfill()方法时，如果当前线程被取消、中断、超时了那么返回的值肯定时 S，否则返回的则是匹配的节点。如果返回值是节点 S，那么 if(x == s)必定成立，如下：

```
Object x = awaitFulfill(s, e, timed, nanos);
```

```
if (x == s) {                   // wait was cancelled
    clean(t, s);
    return null;
}
```

如果返回的 x == s 成立，则调用 clean()方法清理节点 S：

```
    void clean(QNode pred, QNode s) {
        //
        s.waiter = null;

        while (pred.next == s) {
            QNode h = head;
            QNode hn = h.next;
            // hn 节点被取消了，向前推进
            if (hn != null && hn.isCancelled()) {
                advanceHead(h, hn);
                continue;
            }

            // 队列为空，直接 return null
            QNode t = tail;
            if (t == h)
                return;

            QNode tn = t.next;
            // 不一致，说明有其他线程改变了 tail 节点，重新开始
            if (t != tail)
                continue;

            // tn != null 推进 tail 节点，重新开始
            if (tn != null) {
```

```
                advanceTail(t, tn);
                continue;
            }

            // s 不是尾节点 移出
            if (s != t) {
                QNode sn = s.next;
                // 如果 s 已经被移除退出循环，否则尝试断开 s
                if (sn == s || pred.casNext(s, sn))
                    return;
            }

            // s 是尾节点，则有可能会有其他线程在添加新节点，则 cleanMe 出场
            QNode dp = cleanMe;
            // 如果 dp 不为 null，说明是前一个被取消节点，将其移除
            if (dp != null) {
                QNode d = dp.next;
                QNode dn;
                if (d == null ||               // 节点 d 已经删除
                    d == dp ||                 // 原来的节点 cleanMe 已经通过 advanceHead 进行删除
                    !d.isCancelled() ||        // 原来的节点 s 已经删除
                    (d != t &&                 // d 不是 tail 节点
                     (dn = d.next) != null &&  //
                     dn != d &&                //   that is on list
                     dp.casNext(d, dn)))       // d unspliced
                    // 清除 cleanMe 节点，这里的 dp == pred 若成立，说明清除节点 s，成功，直接 return，不然的话要再次循环
                    casCleanMe(dp, null);
                if (dp == pred)
```

```
                return;
            } else if (casCleanMe(null, pred))  // 原来的 cleanMe 是 null，则将
pred 标记为 cleamMe 为下次 清除 s 节点做标识
                return;
        }
    }
```

这个 clean() 方法感觉有点儿难度，我也看得不是很懂。这里是引用 http://www.jianshu.com/p/95cb570c8187

删除的节点不是 queue 尾节点，这时 直接 pred.casNext(s, s.next) 方式来进行删除(和 ConcurrentLikedQueue 中差不多)

删除的节点是队尾节点

此时 cleanMe == null，则 前继节点 pred 标记为 cleanMe，为下次删除做准备

此时 cleanMe != null，先删除上次需要删除的节点，然后将 cleanMe 至 null，让后再将 pred 赋值给 cleanMe

TransferStack 类：

非公平模式 transfer 方法如下：

```
E transfer(E e, boolean timed, long nanos) {
    SNode s = null; // constructed/reused as needed
    int mode = (e == null) ? REQUEST : DATA;

    for (;;) {
        SNode h = head;
        // 栈为空或者当前节点模式与头节点模式一样，将节点压入栈内，等待匹配
        if (h == null || h.mode == mode) {
            // 超时
            if (timed && nanos <= 0) {
                // 节点被取消了，向前推进
                if (h != null && h.isCancelled())
```

```
                    //  重新设置头结点（弹出之前的头结点）
                    casHead(h, h.next);
                else
                    return null;
            }
            // 不超时
            // 生成一个 SNode 节点，并尝试替换掉头节点 head (head -> s)
            else if (casHead(h, s = snode(s, e, h, mode))) {
                // 自旋，等待线程匹配
                SNode m = awaitFulfill(s, timed, nanos);
                // 返回的 m == s 表示该节点被取消了或者超时、中断了
                if (m == s) {
                    // 清理节点 S，return null
                    clean(s);
                    return null;
                }

                // 因为通过前面一步将 S 替换成了 head，如果 h.next == s ，则表示有其他节点插入到 S 前面了,变成了 head
                // 且该节点就是与节点 S 匹配的节点
                if ((h = head) != null && h.next == s)
                    // 将 s.next 节点设置为 head，相当于取消节点 h、s
                    casHead(h, s.next);

                // 如果是请求则返回匹配的域，否则返回节点 S 的域
                return (E) ((mode == REQUEST) ? m.item : s.item);
            }
        }

        // 如果栈不为 null，且两者模式不匹配（h != null && h.mode != mode）
```

```
// 说明他们是一队对等匹配的节点，尝试用当前节点 s 来满足 h 节点
else if (!isFulfilling(h.mode)) {
    // head 节点已经取消了，向前推进
    if (h.isCancelled())
        casHead(h, h.next);

    // 尝试将当前节点打上"正在匹配"的标记，并设置为 head
    else if (casHead(h, s=snode(s, e, h, FULFILLING|mode))) {
        // 循环 loop
        for (;;) {
            // s 为当前节点，m 是 s 的 next 节点，
            // m 节点是 s 节点的匹配节点
            SNode m = s.next;
            // m == null，其他节点把 m 节点匹配走了
            if (m == null) {
                // 将 s 弹出
                casHead(s, null);
                // 将 s 置空，下轮循环的时候还会新建
                s = null;
                // 退出该循环，继续主循环
                break;
            }
            // 获取 m 的 next 节点
            SNode mn = m.next;
            // 尝试匹配
            if (m.tryMatch(s)) {
                // 匹配成功，将 s 、 m 弹出
                casHead(s, mn);     // pop both s and m
                return (E) ((mode == REQUEST) ? m.item : s.item);
            } else
```

```
                        // 如果没有匹配成功，说明有其他线程已经匹配了，把 m 移出
                        s.casNext(m, mn);
                    }
                }
            }
            // 到这最后一步说明节点正在匹配阶段
            else {
                // head 的 next 的节点，是正在匹配的节点，m 和 h 配对
                SNode m = h.next;

                // m == null 其他线程把 m 节点抢走了，弹出 h 节点
                if (m == null)
                    casHead(h, null);
                else {
                    SNode mn = m.next;
                    if (m.tryMatch(h))
                        casHead(h, mn);
                    else
                        h.casNext(m, mn);
                }
            }
        }
    }
```

整个处理过程分为三种情况，具体如下：

1. 如果当前栈为空获取节点模式与栈顶模式一样，则尝试将节点加入栈内，同时通过自旋方式等待节点匹配，最后返回匹配的节点或者 null（被取消）

2. 如果栈不为空且节点的模式与首节点模式匹配，则尝试将该节点打上 FULFILLING 标记，然后加入栈中，与相应的节点匹配，成功后将这两个节点弹出栈并返回匹配节点的数据

3. 如果有节点在匹配，那么帮助这个节点完成匹配和出栈操作，然后在主循环中继续执行

当节点加入栈内后，通过调用 awaitFulfill()方法自旋等待节点匹配：

```
SNode awaitFulfill(SNode s, boolean timed, long nanos) {
    // 超时
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    // 当前线程
    Thread w = Thread.currentThread();

    // 自旋次数
    // shouldSpin 用于检测当前节点是否需要自旋
    // 如果栈为空、该节点是首节点或者该节点是匹配节点，则先采用自旋，否则阻塞
    int spins = (shouldSpin(s) ?
            (timed ? maxTimedSpins : maxUntimedSpins) : 0);
    for (;;) {
        // 线程中断了，取消该节点
        if (w.isInterrupted())
            s.tryCancel();

        // 匹配节点
        SNode m = s.match;

        // 如果匹配节点 m 不为空，则表示匹配成功，直接返回
        if (m != null)
            return m;
        // 超时
        if (timed) {
            nanos = deadline - System.nanoTime();
            // 节点超时，取消
            if (nanos <= 0L) {
                s.tryCancel();
                continue;
            }
```

```
            }

            // 自旋;每次自旋的时候都需要检查自身是否满足自旋条件，满足就 - 1，否则为0
            if (spins > 0)
                spins = shouldSpin(s) ? (spins-1) : 0;

            // 第一次阻塞时，会将当前线程设置到 s 上
            else if (s.waiter == null)
                s.waiter = w;

            // 阻塞 当前线程
            else if (!timed)
                LockSupport.park(this);
            // 超时
            else if (nanos > spinForTimeoutThreshold)
                LockSupport.parkNanos(this, nanos);
        }
    }
```

awaitFulfill()方法会一直自旋/阻塞直到匹配节点。在 S 节点阻塞之前会先调用 shouldSpin()方法判断是否采用自旋方式，为的就是如果有生产者或者消费者马上到来，就不需要阻塞了，在多核条件下这种优化是有必要的。同时在调用 park()阻塞之前会将当前线程设置到 S 节点的 waiter 上。匹配成功，返回匹配节点 m。

shouldSpin()方法如下：

```
        boolean shouldSpin(SNode s) {
            SNode h = head;
            return (h == s || h == null || isFulfilling(h.mode));
        }
```

同时在阻塞过程中会一直检测当前线程是否中断了，如果中断了，则调用 tryCancel()方法取消该节点，取消过程就是将当前节点的 math 设置为当前节点。所以如果线程中断了，那么

在返回 m 时一定是 S 节点自身。

```
            void tryCancel() {
                UNSAFE.compareAndSwapObject(this, matchOffset, null, this);
            }
```

awaitFullfill()方法如果返回的 m == s，则表示当前节点已经中断取消了，则需要调用 clean()方法，清理节点 S：

```
    void clean(SNode s) {

        // 清理 item 域
        s.item = null;
        // 清理 waiter 域
        s.waiter = null;

        // past 节点
        SNode past = s.next;
        if (past != null && past.isCancelled())
            past = past.next;

        // 从栈顶 head 节点，取消从栈顶 head 到 past 节点之间所有已经取消的节点
        // 注意：这里如果遇到一个节点没有取消，则会退出 while
        SNode p;
        while ((p = head) != null && p != past && p.isCancelled())
            casHead(p, p.next);     // 如果 p 节点已经取消了，则剔除该节点

        // 如果经历上面 while p 节点还没有取消，则再次循环取消掉所有 p 到 past 之间的取消节点
        while (p != null && p != past) {
            SNode n = p.next;
            if (n != null && n.isCancelled())
                p.casNext(n, n.next);
```

```
            else
                p = n;
        }
    }
```

LinkedBlockingDeque

LinkedBlockingDeque 是双向链表实现的双向并发阻塞队列。该阻塞队列同时支持 FIFO 和 FILO 两种操作方式，即可以从队列的头和尾同时操作(插入/删除)；并且，该阻塞队列是支持线程安全。

1. LinkedBlockingDeque 继承于 AbstractQueue，它本质上是一个支持 FIFO 和 FILO 的双向的队列。

2. LinkedBlockingDeque 实现了 BlockingDeque 接口，它支持多线程并发。当多线程竞争同一个资源时，某线程获取到该资源之后，其它线程需要阻塞等待。

3. LinkedBlockingDeque 是通过双向链表实现的。

3.1 first 是双向链表的表头。

3.2 last 是双向链表的表尾。

3.3 count 是 LinkedBlockingDeque 的实际大小，即双向链表中当前节点个数。

3.4 capacity 是 LinkedBlockingDeque 的容量，它是在创建 LinkedBlockingDeque 时指定的。

3.5 lock 是控制对 LinkedBlockingDeque 的互斥锁，当多个线程竞争同时访问 LinkedBlockingDeque 时，某线程获取到了互斥锁 lock，其它线程则需要阻塞等待，直到该线程释放 lock，其它线程才有机会获取 lock 从而获取 cpu 执行权。

3.6 notEmpty 和 notFull 分别是“非空条件”和“未满条件”。通过它们能够更加细腻进行并发控制。

变量声明

```
//双向队列头结点
transient Node<E> first;
```

```
//双向队列尾节点
transient Node<E> last;
//当前队列存在元素的个数
private transient int count;
//队列初始化长度
private final int capacity;
//为此队列加的锁
final ReentrantLock lock = new ReentrantLock();
private final Condition notEmpty = lock.newCondition();
private final Condition notFull = lock.newCondition();
```

构造方法

```
public LinkedBlockingDeque() {
    this(Integer.MAX_VALUE);
}
public LinkedBlockingDeque(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
}
public LinkedBlockingDeque(Collection<? extends E> c) {
    this(Integer.MAX_VALUE);
    final ReentrantLock lock = this.lock;
    lock.lock(); // Never contended, but necessary for visibility
    try {
        for (E e : c) {
            if (e == null)
                throw new NullPointerException();
            if (!linkLast(new Node<E>(e)))
                throw new IllegalStateException("Deque full");
```

```
            }
        } finally {
            lock.unlock();
        }
    }
```

public LinkedBlockingDeque(); 此够着函数可构造出一个长度为 Integer.MAX_VALUE 的队列。

public LinkedBlockingDeque(int capacity); 此构造方法可构造出一个长度为 capacity 的队列。且传入的 capacity 数值不能小于 0。

public LinkedBlockingDeque(Collection<? extends E> c); 此构造方法可构造出一个长度为 Integer.MAX_VALUE 的队列，并且会将集合 c 中的元素都添加到队列中。

添加操作

在 LinkedLockingDeque 类中添加函数统一有三种形式，即 add()，offer()，put()，而 add 形式中的所有方法都是调用的 offer()方法中的。所以我们需要研究 offer 和 put 函数就行了。首先看这两个方法的源码有什么不同之处之前，先看看这两个函数都会调用的添加操作函数怎么实现的吧：

```
    private boolean linkLast(Node<E> node) {
        // assert lock.isHeldByCurrentThread();
        if (count >= capacity)
            return false;
        Node<E> l = last;
        node.prev = l;
        last = node;
        if (first == null)
            first = node;
        else
            l.next = node;
        ++count;
```

```
        notEmpty.signal();
        return true;
    }
```

由源码可以看到当当前队列中存在元素的个数达到队列最大承受限度时，会直接返回 false。如果没有达到最大限度说明可以添加，那么就可以直接添加了。添加完成后会唤醒非空条件，然后返回 true；

接下来在看看 offer 函数和 put 函数的源码，存在什么异同

```
    public void put(E e) throws InterruptedException {
        putLast(e);
    }
    public void putLast(E e) throws InterruptedException {
        if (e == null) throw new NullPointerException();
        Node<E> node = new Node<E>(e);
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            while (!linkLast(node))
                notFull.await();
        } finally {
            lock.unlock();
        }
    }

    public boolean offer(E e) {
        return offerLast(e);
    }
    public boolean offerLast(E e) {
        if (e == null) throw new NullPointerException();
        Node<E> node = new Node<E>(e);
```

```
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            return linkLast(node);
        } finally {
            lock.unlock();
        }
    }
```

根据这两个的尾部添加可以看出 offer 方法在添加时，如果遇到队列满的时候则会直接返回 false，而 put 会一直在那等待。直到出队操作执行，将非满条件唤醒便可以正常执行了。

取出操作

在 LinkedLockingDeque 类中取出函数一共有三种，即 remove()，poll()，take()，而 remove 形式中的所有方法都是调用的 poll()方法中的。所以我们需要研究 poll 和 take 函数就行了。首先看这两个方法的源码有什么不同之处之前，先看看这两个函数都会调用的取出操作函数

```
    private E unlinkFirst() {
        // assert lock.isHeldByCurrentThread();
        Node<E> f = first;
        if (f == null)
            return null;
        Node<E> n = f.next;
        E item = f.item;
        f.item = null;
        f.next = f; // help GC
        first = n;
        if (n == null)
            last = null;
        else
            n.prev = null;
```

```
        --count;
        notFull.signal();
        return item;
    }
```

由源码可以看到当当前队列中存在元素为空时，会直接返回 false。如果不为空，那么就可以正常取出。取出完成后会唤醒非满条件，然后返回 true；

接下来在看看 poll 函数和 take 函数的异同

```
    public E poll() {
        return pollFirst();
    }
    public E pollFirst() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            return unlinkFirst();
        } finally {
            lock.unlock();
        }
    }

    public E take() throws InterruptedException {
        return takeFirst();
    }
    public E takeFirst() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            E x;
            while ( (x = unlinkFirst()) == null)
```

```
            notEmpty.await();
        return x;
    } finally {
        lock.unlock();
    }
}
```

这两个函数对队头取出的操作是有不同的。Poll 方法如果遇到队列为空的时候，会直接返回 false，take 方法如果遇到队列为空会一直在那等待。

遍历操作

因为 LinkedBlockingDeque 实现的是 BlockingQueue 接口，BlockingQueue 接口继承了 Queue 接口，Queue 接口继承了 Collection 接口，所以 LinkedBlockingDeque 中实现了两个迭代器：

返回在此双端队列元素上以恰当顺序进行迭代的迭代器。

```
public Iterator<E> iterator() {
    return new Itr();
}
```

返回在此双端队列的元素上以逆向连续顺序进行迭代的迭代器。

```
public Iterator<E> descendingIterator() {
    return new DescendingItr();
}
```

DelayQueue

DelayQueue 内部通过组合 PriorityQueue 来实现存储和维护元素顺序的。

DelayQueue 存储元素必须实现 Delayed 接口，通过实现 Delayed 接口，可以获取到元素延迟时间，以及可以比较元素大小（Delayed 继承 Comparable）

DelayQueue 通过一个可重入锁来控制元素的入队出队行为

DelayQueue 中 leader 标识 用于减少线程的竞争，表示当前有其它线程正在获取队头元素。

PriorityQueue 只是负责存储数据以及维护元素的顺序，对于延迟时间取数据则是在 DelayQueue 中进行判断控制的。

DelayQueue 没有实现序列化接口

--ConcurrentLinkedQueue

--LinkedTransferQueue

set 系列问题

ArrayList 和 hashset 有何区别。

ArrayList 是数组存储的方式

HashSet 存储会先进行 HashCode 值得比较(hashcode 和 equals 方法)，若相同就不会再存储

HashCode 和 HashSet 类

Hashset 就是采用哈希算法存取对象的集合

对象用完之后没有回收就是内存泄漏

一个对象一旦 hashCode 生成之后，再对属性值修改后

其 Hashcode 值就会发生改变

再通过 hashSet 删除就删除不掉了

```
Collection collections = new HashSet();
ReflectPoint pt1 = new ReflectPoint(3,3);
collections.add(pt1);
pt1.y =7;
collections.remove(pt1);//删除不了
```

ArrayList 底层是一个数组，hashset 底层是 hashMap

hashset 存的数是有序的么。

无序的，只不过在存储的时候按照 hashCode 函数进行存储，有些类别的会出现排好序的状态

HashSet 有什么特性

HashSet 实现了 Set 接口，它不允许集合中有重复的值。当我们提到 HashSet 时，第一件事情就是在将对象存储在 HashSet 之前，要先确保对象重写 equals()和 hashCode()方法，这样才能比较对象的值是否相等，以确保 set 中没有储存相等的对象。

public boolean add(Object o)方法用来在 Set 中添加元素，当元素值重复时则会立即返回 false，如果成功添加的话会返回 true。

hashset 怎么保证唯一性

因为 hashset 的底层是 hashmap，所以在 hashmap 当中会有一个判断机制(k = e.key) == key || (key != null && key.equals(k))，这个 key 就是 hashmap 当中的 hash()方法求出来的。Hash 算法中需要 hashCode 方法去求 hash 值。根据 hashCode 方法和 equals 方法就可以排除重复

的元素了。

使用 Set 集合是要去除重复元素，如果在存储的时候逐 equals()比较，效率太低，哈希算法提高了去重复的效率，减少了使用 equals()方法的次数，当 HashSet 对象调用 add()方法存储对象时，会调用对象的 HashCode()方法得到一个哈希值，然后在集合中查找是否有哈希值相同的对象，如果用，则调用 equals()方法比较，如果没有则直接存入集合。

因此，如果自定义类对象存入集合去重复，需要重写 equals()方法和 HashCode()方法。

HashSet 方法里面的 hashcode 存在哪(我说类似 HashMap 存在 Node 里面，他还是问了我好久，没看过源码很虚)

因为 hashset 的底层是 hashmap，所以 hashcode 放在的地方就和 hashmap 放在的位置一样，放在的 Node 里。

TreeSet 实现原理

TreeSer 的底层是 Treemap，使用元素的自然顺序对元素进行排序，或者根据创建 set 时提供的 Comparator 进行排序，具体取决于使用的构造方法。 无参数是自然排序。

具有的特点：

唯一：根据 Comparable 接口的 compareTo(Student o)方法返回值是否为 0 来决定。如果返回值是 0，说明元素重复，就不在添加元素了。

排序：根据 Comparable 接口的 compareTo(Student o)方法返回值决定

正：说明数据大，往后放。

0：说明重复，不添加

负：说明数据小，往前放

TreeSet 底层是红黑树结构

TreeSet 集合，采用无参构造方法的时候，要让集合实现元素所属类的 Comparable 接口。然后再重写该接口中的方法的时候，去按照需求来实现代码比较即可。

实现 TreeSet 排序的两种方式：

1.在每一个子类当中实现 Comparable 接口

2.在创建 TreeSet 对象的时候传入一个 Comparable 类进行比较

TreeSet 实现了哪些接口

NavigableSet<E>, Cloneable, java.io.Serializable

其他集合问题

collections 类中的方法？

1.将所有指定元素添加到指定 collection 中。

static <T> boolean addAll(Collection<? super T> c, T... elements)

2.以后进先出 (Lifo) Queue 的形式返回某个 Deque 的视图。

static <T> Queue<T> asLifoQueue(Deque<T> deque)

3.使用二分搜索法搜索指定列表，以获得指定对象。

static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)

4.使用二分搜索法搜索指定列表，以获得指定对象。

static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)

5.返回指定 collection 的一个动态类型安全视图。

static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> type)

6.返回指定列表的一个动态类型安全视图。

static <E> List<E> checkedList(List<E> list, Class<E> type)

7.返回指定映射的一个动态类型安全视图。

static <K,V> Map<K,V> checkedMap(Map<K,V> m, Class<K> keyType, Class<V> valueType)

8.返回指定 set 的一个动态类型安全视图。

static <E> Set<E> checkedSet(Set<E> s, Class<E> type)

9.返回指定有序映射的一个动态类型安全视图。

static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K,V> m, Class<K> keyType,

Class<V> valueType)

10.返回指定有序 set 的一个动态类型安全视图。

static <E> SortedSet<E> checkedSortedSet(SortedSet<E> s, Class<E> type)

11.将所有元素从一个列表复制到另一个列表。

static <T> void copy(List<? super T> dest, List<? extends T> src)

12.如果两个指定 collection 中没有相同的元素，则返回 true。

static boolean disjoint(Collection<?> c1, Collection<?> c2)

//13.返回空的列表（不可变的）。

static <T> List<T> emptyList()

14.返回空的映射（不可变的）。

static <K,V> Map<K,V> emptyMap()

15.返回空的 set（不可变的）。

static <T> Set<T> emptySet()

16.返回一个指定 collection 上的枚举。

static <T> Enumeration<T> enumeration(Collection<T> c)

17.使用指定元素替换指定列表中的所有元素。

static <T> void fill(List<? super T> list, T obj)

18.返回指定 collection 中等于指定对象的元素数。

static int frequency(Collection<?> c, Object o)

19.返回指定源列表中第一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回 -1。

static int indexOfSubList(List<?> source, List<?> target)

20.返回指定源列表中最后一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回 -1。

static int lastIndexOfSubList(List<?> source, List<?> target)

21.返回一个数组列表，它按返回顺序包含指定枚举返回的元素。

static <T> ArrayList<T> list(Enumeration<T> e)

22.根据元素的自然顺序，返回给定 collection 的最大元素。

static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)

23.根据指定比较器产生的顺序，返回给定 collection 的最大元素。

static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)

24.根据元素的自然顺序 返回给定 collection 的最小元素。

static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)

25.根据指定比较器产生的顺序，返回给定 collection 的最小元素。

static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)

26.返回由指定对象的 n 个副本组成的不可变列表。

static <T> List<T> nCopies(int n, T o)

27.返回指定映射支持的 set。

static <E> Set<E> newSetFromMap(Map<E,Boolean> map)

28.使用另一个值替换列表中出现的所有某一指定值。

static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)

29.反转指定列表中元素的顺序。

static void reverse(List<?> list)

30.返回一个比较器，它强行逆转实现了 Comparable 接口的对象 collection 的自然顺序。

static <T> Comparator<T> reverseOrder()

31.返回一个比较器，它强行逆转指定比较器的顺序。

static <T> Comparator<T> reverseOrder(Comparator<T> cmp)

32.根据指定的距离轮换指定列表中的元素。

static void rotate(List<?> list, int distance)

33.使用默认随机源对指定列表进行置换。

static void shuffle(List<?> list)

34.使用指定的随机源对指定列表进行置换。

static void shuffle(List<?> list, Random rnd)

35.返回一个只包含指定对象的不可变 set。

static <T> Set<T> singleton(T o)

36.返回一个只包含指定对象的不可变列表。

static <T> List<T> singletonList(T o)

37.返回一个不可变的映射，它只将指定键映射到指定值。

static <K,V> Map<K,V> singletonMap(K key, V value)

38.根据元素的自然顺序 对指定列表按升序进行排序。

static <T extends Comparable<? super T>> void sort(List<T> list)

39.根据指定比较器产生的顺序对指定列表进行排序。

static <T> void sort(List<T> list, Comparator<? super T> c)

40.在指定列表的指定位置处交换元素。

static void swap(List<?> list, int i, int j)

41.返回指定 collection 支持的同步（线程安全的）collection。

static <T> Collection<T> synchronizedCollection(Collection<T> c)

42.返回指定列表支持的同步（线程安全的）列表。

static <T> List<T> synchronizedList(List<T> list)

43.返回由指定映射支持的同步（线程安全的）映射。

static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)

44.返回指定 set 支持的同步（线程安全的）set。

static <T> Set<T> synchronizedSet(Set<T> s)

45.返回指定有序映射支持的同步（线程安全的）有序映射。

static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)

46.返回指定有序 set 支持的同步（线程安全的）有序 set。

static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)

47.返回指定 collection 的不可修改视图。

static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c)

48.返回指定列表的不可修改视图。

static <T> List<T> unmodifiableList(List<? extends T> list)

49.返回指定映射的不可修改视图。

static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)

50.返回指定 set 的不可修改视图。

static <T> Set<T> unmodifiableSet(Set<? extends T> s)

51.返回指定有序映射的不可修改视图。

static <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,? extends V> m)

52.返回指定有序 set 的不可修改视图。

static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)