并发容器

一、ConcurrentHashMap

在多线程情况下，使用HashMap扩容时会出现CPU接近100%的情况，因为HashMap不是线程安全的；对于HashTable该类基本上所有的方法都采用synchronized进行线程安全控制的，在高并发的情况下，每次只有一个线程能够获取对象监视器锁，这样的并发性能的确不令人满意。另外一种方式是通过Collections的Map<k,v> synchronizedMap(Map<k,v> m)将hashmap包装成一个线程安全的map，比如synchronizedMap的put方法的源码为：

```
public V put(K key, V value) {
    synchronized (mutex) {return m.put(key, value);}
}
```

实际上SynchronizedMap实现依然是采用synchronized独占式锁进行线程安全的并发控制的。同样，这种方案的性能也是令人不太满意的。相对于hashmap来说，ConcurrentHashMap就是线程安全的map，其中**利用了锁分段的思想提高了并发度**，首先将数据分成一段一段存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

JDK 1.8的ConcurrentHashMap就有了很大的变化，光是代码量就足足增加了很多。1.8版本舍弃了segment，并且大量使用了synchronized，以及CAS无锁操作以保证ConcurrentHashMap操作的线程安全性。至于为什么不用ReentrantLock而是Synchronzied呢？实际上，synchronzied做了很多的优化，包括偏向锁，轻量级锁，重量级锁，可以依次向上升级锁状态，但不能降级，因此，使用synchronized相较于ReentrantLock的性能会持平甚至在某些情况更优。

1、关键属性及类

（1）关键属性

- table： transient volatile Node<K,V>[] table;装载Node的数组，作为ConcurrentHashMap的数据容器，采用懒加载的方式，直到第一次插入数据的时候才会进行初始化操作，数组的大小总是为2的幂次方。
- nextTable：transient volatile Node<K,V>[] nextTable;扩容时使用，平时为null，只有在扩容的时候才为非null
- sizeCtl：transient volatile int sizeCtl;该属性用来控制table数组的大小，根据是否初始化和是否正在扩容有几种情况： **当值为负数时：**如果为-1表示正在初始化，如果为-N则表示当前正有N-1个线程进行扩容操作； **当值为正数时：**如果当前数组为null的话表示table在初始化过程中，sizeCtl表示为需要新建数组的长度； 若已经初始化了，表示当前数据容器（table数组）可用容量也可以理解成临界值（插入节点数超过了该临界值就需要扩容），具体指为数组的长度n 乘以 加载因子loadFactor； 当值为0时，即数组长度为默认初始值。
- U：sun.misc.Unsafe U；在ConcurrentHashMapde的实现中可以看到大量的U.compareAndSwapXXXX的方法去修改ConcurrentHashMap的一些属性。这些方法实际上是利用了CAS算法保证了线程安全性，这是一种乐观策略，假设每一次操作都不会产生冲突，当且仅当冲突发生的时候再去尝试。而CAS操作依赖于现代处理器指令集，通过底层CMPXCHG指令实现。CAS(V,O,N)核心思想为：若当前变量实际值V与期望的旧值O相同，则表明该变量没被其他线程进行修改，因此可以安全的将新值N赋值给变量；若当前变量实际值V与期望的旧值O不相同，则表明该变量已经被其他线程做了处理，此时将新值N赋给变量操作就是不安全的，在进行重试。而在大

量的同步组件和并发容器的实现中使用CAS是通过sun.misc.Unsafe类实现的，该类提供了一些可以直接操控内存和线程的底层操作，可以理解为java中的“指针”。该成员变量的获取是在静态代码块中：

```
static {
    try {
        U = sun.misc.Unsafe.getUnsafe();
        Class<?> k = TreeBin.class;
        LOCKSTATE = U.objectFieldOffset
            (k.getDeclaredField("lockState"));
    } catch (Exception e) {
        throw new Error(e);
    }
}
```

(2) 关键类

- Node：Node类实现了Map.Entry接口，主要存放key-value对，并且具有next域，另外可以看出很多属性都是用volatile进行修饰的，也就是为了保证内存可见性。

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
    ......
}
```

- TreeNode 树节点，继承于承载数据的Node类。而红黑树的操作是针对TreeBin类的，从该类的注释也可以看出，也就是TreeBin会将TreeNode进行再一次封装

```
/**
 * Nodes for use in TreeBins
 */
static final class TreeNode<K,V> extends Node<K,V> {
    TreeNode<K,V> parent;  // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;    // needed to unlink next upon deletion
    boolean red;
    ......
}
```

- TreeBin 这个类并不负责包装用户的key、value信息，而是包装的很多TreeNode节点。实际的ConcurrentHashMap“数组”中，存放的是TreeBin对象，而不是TreeNode对象。

```
static final class TreeBin<K,V> extends Node<K,V> {
    TreeNode<K,V> root;
    volatile TreeNode<K,V> first;
    volatile Thread waiter;
    volatile int lockState;
    // values for lockState
    static final int WRITER = 1; // set while holding write lock
    static final int WAITER = 2; // set when waiting for write lock
    static final int READER = 4; // increment value for setting read lock
    ......
}
```

- ForwardingNode 在扩容时才会出现的特殊节点，其key,value,hash全部为null。并拥有nextTable指针引用新的table数组。

```
static final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
    ......
}
```

(3) CAS关键操作

ConcurrentHashMap中会大量使用CAS修改它的属性和一些操作。因此，在理解ConcurrentHashMap的方法前我们需要了解下面几个常用的利用CAS算法来保障线程安全的操作。

- tabAt：该方法用来获取table数组中索引为i的Node元素

```
static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
    return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
}
```

- casTabAt：利用CAS操作设置table数组中索引为i的元素

```
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
                                    Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}
```

- setTabAt：该方法用来设置table数组中索引为i的元素

```
static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {
    U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);
}
```

2、重点方法

（1）实例构造器方法

```
// 1. 构造一个空的map，即table数组还未初始化，初始化放在第一次插入数据时，默认大小为16
ConcurrentHashMap()
// 2. 给定map的大小
ConcurrentHashMap(int initialCapacity)
// 3. 给定一个map
ConcurrentHashMap(Map<? extends K, ? extends V> m)
// 4. 给定map的大小以及加载因子
ConcurrentHashMap(int initialCapacity, float loadFactor)
// 5. 给定map大小，加载因子以及并发度（预计同时操作数据的线程）
ConcurrentHashMap(int initialCapacity,float loadFactor, int concurrencyLevel)
```

对于第二种，传入指定大小时的情况：

```
public ConcurrentHashMap(int initialCapacity) {
    // 小于0抛出异常
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
    // 判断是否超过了最大值
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap;
}
```

这段代码的逻辑请看注释，很容易理解，如果小于0就直接抛出异常，如果指定值大于了所允许的最大值的话就取最大值，否则，在对指定值做进一步处理。最后将cap赋值给sizeCtl,关于sizeCtl的说明请看上面的说明，当调用构造器方法之后，sizeCtl的大小应该就代表了ConcurrentHashMap的大小，即table数组长度。tableSizeFor做了哪些事情了？源码为：

```
private static final int tableSizeFor(int c) {
    int n = c - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

通过注释就很清楚了，该方法会将调用构造器方法时指定的大小转换成一个2的幂次方数，也就是说ConcurrentHashMap的大小一定是2的幂次方，比如，当指定大小为18时，为了满足2的幂次方特性，实际上concurrentHashMapd的大小为2的5次方（32）。另外，需要注意的是，**调用构造器方法的时候并未构造出table数组（可以理解为ConcurrentHashMap的数据容器），只是算出table数组的长度，当第一次向ConcurrentHashMap插入数据的时候才真正的完成初始化创建table数组的工作。**

（2）initTable方法

```
private final Node<K,V>[] initTable() {
```

```
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            // 1.保证只有一个线程在创建
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    // 2.得出数组的大小，默认为16
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    // 3.创建数组
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    // 4.计算出数组中的可用大小：实际大小*0.75（加载因子）
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

代码的逻辑请见注释，有可能存在一个情况是多个线程同时走到这个方法中，为了保证能够正确初始化，在第1步中会先通过if进行判断，若当前已经有一个线程正在初始化即sizeCtl值变为-1，这个时候其他线程在If判断为true从而调用Thread.yield()让出CPU时间片。正在进行初始化的线程会调用U.compareAndSwapInt方法将sizeCtl改为-1即正在初始化的状态。另外还需要注意的事情是，在第四步中会进一步计算数组中可用的大小即为数组实际大小n乘以加载因子0.75.可以看看这里乘以0.75是怎么算的，0.75为四分之三，这里n - (n >>> 2)是不是刚好是n-(1/4)n=(3/4)n，挺有意思的吧:)。如果选择是无参的构造器的话，这里在new Node数组的时候会使用默认大小为DEFAULT_CAPACITY（16），然后乘以加载因子0.75为12，也就是说数组的可用大小为12。

（3）put方法

构造哈希函数的方法有：

- 直接定址法：当关键字是整型数时，可以取关键字本身或它的线性函数作为它的哈希地址。即 $H(K)=K$或 $H(K)=aK+b(a,b$为常数$)$。但是该方法会造成空间的大量浪费。
- 除留余数法：选取一个合适的不大于哈希表长的正整数m，用m去除关键字K，所得的余数作为其哈希地址，即：$H(K)=Kmodm$，这种方法称为除留余数法，该方法的优劣取决于m值的选取。若m取某个偶数值，其结果是将奇数关键字的记录映射到奇数地址上，将偶数关键字的记录映射到偶数地址上，因此产生的哈希地址很可能不均匀分布。若m取关键字的基数的幂次值，那么产生的哈希地址是关键字的某几个低位值。这种方法是一种简单而且行之有效的构造哈希函数的方法。
- 数字分析法：关键字有d位数，选取其中若干位的值构造哈希地址的方法称为数字分析法。这种方法要事先知道所有关键字或大多数关键字的值，对这些关键字的各位值做分析，丢掉不均匀的位值，留下分布较均匀的位值构造器哈希函数。
- 平方取中法：取关键字平方后的中间若干位作为其哈希地址。即：$H(K)=$” K^2的中间几位 ”。因为关键字平方后使得它的中间几位和组成关键字的各位值均有关，从而使哈希地址的分布较均匀，减少了发生冲突的可能性。所取的位数取决于哈希表的表长。

- 折叠移位法：根据哈希表将关键字分成尽可能等长的若干段，然后将这几段的值相加，并将最高位的进位舍去，所得结果即为其哈希地址。相加时有两种方法：一种是顺折，即把每一段中的各位值对齐相加，称之为移位法；另一种是对折，像这纸条一样，把原来关键字中的数字按照划分的中界向中间段折叠，然后求和，诚挚为折叠法。

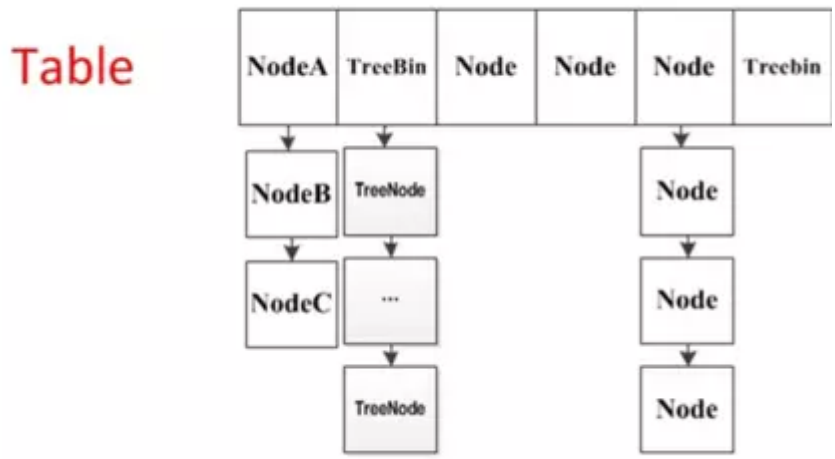
调用put方法时实际具体实现是putVal方法。

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    // 键和值都不能为null
    if (key == null || value == null) throw new NullPointerException();
    // 1.计算key的hash值
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // 2.如果当前table还没有初始化，则调用initTable方法进行初始化
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // 3.获得索引i = (n - 1) & hash，然后根据索引获取索引i对应位置的元素
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 3.1表示当前索引i位置没有元素，则直接使用CAS操作插入元素
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        // 4.表示索引i出有值了，且当前正在扩容，MOVED=-1
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {

            V oldVal = null;
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    // 5.当前为链表，在链表中插入新的键值对
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,
                                                          value, null);
                                break;
                            }
                        }
                    }
```

```
                    // 当前为红黑树，将新的键值对插入到红黑树中
                    else if (f instanceof TreeBin) {
                        Node<K,V> p;
                        binCount = 2;
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                       value)) != null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                }
            }
            // 7.插完键值对后再根据实际大小看是否需要转换成红黑树，TREEIFY_THRESHOLD=8
            if (binCount != 0) {
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    addCount(1L, binCount);
    return null;
}
```

put方法的代码量有点长，我们按照上面的分解的步骤一步步来看。**从整体而言，为了解决线程安全的问题，ConcurrentHashMap使用了synchronzied和CAS的方式。**在之前了解过HashMap以及1.8版本之前的ConcurrenHashMap都应该知道ConcurrentHashMap结构图，。



ConcurrentHashMap是一个哈希桶数组，如果不出现哈希冲突的时候，每个元素均匀的分布在哈希桶数组中。当出现哈希冲突的时候，是**标准的链地址的解决方式**，将hash值相同的节点构成链表的形式，称为“拉链法”，另外，在1.8版本中为了防止拉链过长，当链表的长度大于8的时候会将链表转换成红黑树。table数组中的每个元素实际上是单链表的头结点或者红黑树的根节点。当插入键值对时首先应该定位到要插入的桶，即插入table数组的索引i处。那么，怎样计算得出索引i呢？当然是根据key的hashCode值。

- **spread()**：重哈希，以减小哈希冲突。

```
static final int spread(int h) {
    return (h ^ (h >>> 16)) & HASH_BITS;
}
```

该方法主要是**将key的hashCode的低16位于高16位进行异或运算**，这样不仅能够使得hash值能够分散能够均匀减小hash冲突的概率，另外只用到了异或运算，在性能开销上也能兼顾，做到平衡的trade-off。

- **初始化table**：判断当前数组是否初始化了，如果没有初始化先初始化
- **能够将新值插入到table数组中**：如果插入值待插入的位置刚好所在的table数组为null的话就可以直接将值插入即可。那么怎样根据hash确定在table中待插入的索引i呢？很显然可以通过hash值与数组的长度取模操作，从而确定新值插入到数组的哪个位置。而之前我们提过ConcurrentHashMap的大小总是2的幂次方，(n - 1) & hash运算等价于对长度n取模，也就是hash%n，但是位运算比取模运算的效率要高很多，Doug lea大师在设计并发容器的时候也是将性能优化到了极致，令人钦佩。确定好数组的索引i后，就可以可以tabAt()方法（该方法在上面已经说明了，有疑问可以回过头去看看）获取该位置上的元素，如果当前Node f为null的话，就可以直接用casTabAt方法将新值插入即可。
- **当前数组是否正在扩容**：如果当前节点不为null，且该节点为特殊节点（forwardingNode）的话，就说明当前concurrentHashMap正在进行扩容操作，关于扩容操作，下面会作为一个具体的方法进行讲解。那么怎样确定当前的这个Node是不是特殊的节点了？是通过判断该节点的hash值是不是等于-1（MOVED）,代码为(fh = f.hash) == MOVED，对MOVED的解释在源码上也写的很清楚了。
- **当table[i]为链表的头结点，在链表中插入新值**：在table[i]不为null并且不为forwardingNode时，并且当前Node f的hash值大于0（fh >= 0）的话说明当前节点f为当前桶的所有的节点组成的链表的头结点。那么接下来，要想向ConcurrentHashMap插入新值的话就是向这个链表插入新值。通过synchronized (f)的方式进行加锁以实现线程安全性。找到key相同的会覆盖掉原来的旧value值。
- **当table[i]为红黑树的根节点，在红黑树中插入新值**：按照之前的数组+链表的设计方案，这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，甚至在极端情况下，查找一个节点会出现时间复杂度为O(n)的情况，则会严重影响ConcurrentHashMap的性能，于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高ConcurrentHashMap的性能，其中会用到红黑树的插入、删除、查找等算法。首先在if中通过"f instanceof TreeBin"判断当前table[i]是否是树节点，这下也正好验证了我们在最上面介绍时说的TreeBin会对TreeNode做进一步封装，对红黑树进行操作的时候针对的是TreeBin而不是TreeNode。
- **根据当前节点个数进行调整**：如果当前链表节点个数大于等于8（TREEIFY_THRESHOLD）的时候，就会调用treeifyBin方法将tabel[i]（第i个散列桶）拉链转换成红黑树。

整体流程：

1. 首先对于每一个放入的值，首先利用spread方法对key的hashcode进行一次hash计算，由此来确定这个值在table中的位置；
2. 如果当前table数组还未初始化，先将table数组进行初始化操作；
3. 如果这个位置是null的，那么使用CAS操作直接放入；
4. 如果这个位置存在结点，说明发生了hash碰撞，首先判断这个节点的类型。如果该节点fh==MOVED(代表forwardingNode,数组正在进行扩容)的话，说明正在进行扩容；
5. 如果是链表节点（fh>0）,则得到的结点就是hash值相同的节点组成的链表的头节点。需要依次向后遍历确定这个新加入的值所在位置。如果遇到key相同的节点，则只需要覆盖该结点的value值即可。否则依次向后遍历，直到链表尾插入这个结点；
6. 如果这个节点的类型是TreeBin的话，直接调用红黑树的插入方法进行插入新的节点；

7. 插入完节点之后再次检查链表长度，如果长度大于8，就把这个链表转换成红黑树；
8. 对当前容量大小进行检查，如果超过了临界值（实际大小*加载因子）就需要扩容。

（4）get方法

，首先先看当前的hash桶数组节点即table[i]是否为查找的节点，若是则直接返回；若不是，则继续再看当前是不是树节点？通过看节点的hash值是否为小于0，如果小于0则为树节点。如果是树节点在红黑树中查找节点；如果不是树节点，那就只剩下为链表的形式的一种可能性了，就向后遍历查找节点，若查找到则返回节点的value即可，若没有找到就返回null。

（5）transfer方法

当ConcurrentHashMap容量不足的时候，需要对table进行扩容。这个方法的基本思想跟HashMap是很像的，但是由于它是支持并发扩容的，所以要复杂的多。原因是它支持多线程进行扩容操作，而并没有加锁。我想这样做的目的不仅仅是为了满足concurrent的要求，而是希望利用并发处理去减少扩容带来的时间影响。

```
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    int n = tab.length, stride;
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE; // subdivide range
    if (nextTab == null) {            // initiating
        try {
            @SuppressWarnings("unchecked")
            // 1.新建Node数组，容量为之前的两倍
            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
            nextTab = nt;
        } catch (Throwable ex) {      // try to cope with OOME
            sizeCtl = Integer.MAX_VALUE;
            return;
        }
        nextTable = nextTab;
        transferIndex = n;
    }
    int nextn = nextTab.length;
    // 2.新建一个forwardingNode
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab
    for (int i = 0, bound = 0;;) {
        Node<K,V> f; int fh;
        // 3.确定遍历中的索引i
        while (advance) {
            int nextIndex, nextBound;
            if (--i >= bound || finishing)
                advance = false;
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1;
                advance = false;
            }
            else if (U.compareAndSwapInt
                     (this, TRANSFERINDEX, nextIndex,
                      nextBound = (nextIndex > stride ?
```

```
                                    nextIndex - stride : 0))) {
                bound = nextBound;
                i = nextIndex - 1;
                advance = false;
            }
        }
        // 4.将原数组中的元素复制到新数组中去
        // 4.5for循环退出，扩容结束修改sizeCtl属性
        if (i < 0 || i >= n || i + n >= nextn) {
            int sc;
            if (finishing) {
                nextTable = null;
                table = nextTab;
                sizeCtl = (n << 1) - (n >>> 1);
                return;
            }
            if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                    return;
                finishing = advance = true;
                i = n; // recheck before commit
            }
        }
        // 4.1 当前数组中第i个元素为null，用CAS设置成特殊节点forwardingNode(可以理解成占位符)
        else if ((f = tabAt(tab, i)) == null)
            advance = casTabAt(tab, i, null, fwd);
        // 4.2 如果遍历到ForwardingNode节点  说明这个点已经被处理过了 直接跳过  这里是控制并发扩容的核心
        else if ((fh = f.hash) == MOVED)
            advance = true; // already processed
        else {
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    Node<K,V> ln, hn;
                    if (fh >= 0) {
                        // 4.3 处理当前节点为链表的头结点的情况，构造两个链表，一个是原链表  另一个是原链表的反序排列
                        int runBit = fh & n;
                        Node<K,V> lastRun = f;
                        for (Node<K,V> p = f.next; p != null; p = p.next) {
                            int b = p.hash & n;
                            if (b != runBit) {
                                runBit = b;
                                lastRun = p;
                            }
                        }
                        if (runBit == 0) {
                            ln = lastRun;
                            hn = null;
                        }
                        else {
                            hn = lastRun;
                            ln = null;
```

```
                        }
                        for (Node<K,V> p = f; p != lastRun; p = p.next) {
                            int ph = p.hash; K pk = p.key; V pv = p.val;
                            if ((ph & n) == 0)
                                ln = new Node<K,V>(ph, pk, pv, ln);
                            else
                                hn = new Node<K,V>(ph, pk, pv, hn);
                        }
                        // 在nextTable的i位置上插入一个链表
                        setTabAt(nextTab, i, ln);
                        // 在nextTable的i+n的位置上插入另一个链表
                        setTabAt(nextTab, i + n, hn);
                        // 在table的i位置上插入forwardNode节点  表示已经处理过该节点
                        setTabAt(tab, i, fwd);
                        // 设置advance为true 返回到上面的while循环中 就可以执行i--操作
                        advance = true;
                    }
                    // 4.4 处理当前节点是TreeBin时的情况，操作和上面的类似
                    else if (f instanceof TreeBin) {
                        TreeBin<K,V> t = (TreeBin<K,V>)f;
                        TreeNode<K,V> lo = null, loTail = null;
                        TreeNode<K,V> hi = null, hiTail = null;
                        int lc = 0, hc = 0;
                        for (Node<K,V> e = t.first; e != null; e = e.next) {
                            int h = e.hash;
                            TreeNode<K,V> p = new TreeNode<K,V>
                                (h, e.key, e.val, null, null);
                            if ((h & n) == 0) {
                                if ((p.prev = loTail) == null)
                                    lo = p;
                                else
                                    loTail.next = p;
                                loTail = p;
                                ++lc;
                            }
                            else {
                                if ((p.prev = hiTail) == null)
                                    hi = p;
                                else
                                    hiTail.next = p;
                                hiTail = p;
                                ++hc;
                            }
                        }
                        ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
                            (hc != 0) ? new TreeBin<K,V>(lo) : t;
                        hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
                            (lc != 0) ? new TreeBin<K,V>(hi) : t;
                        setTabAt(nextTab, i, ln);
                        setTabAt(nextTab, i + n, hn);
                        setTabAt(tab, i, fwd);
                        advance = true;
                    }
```

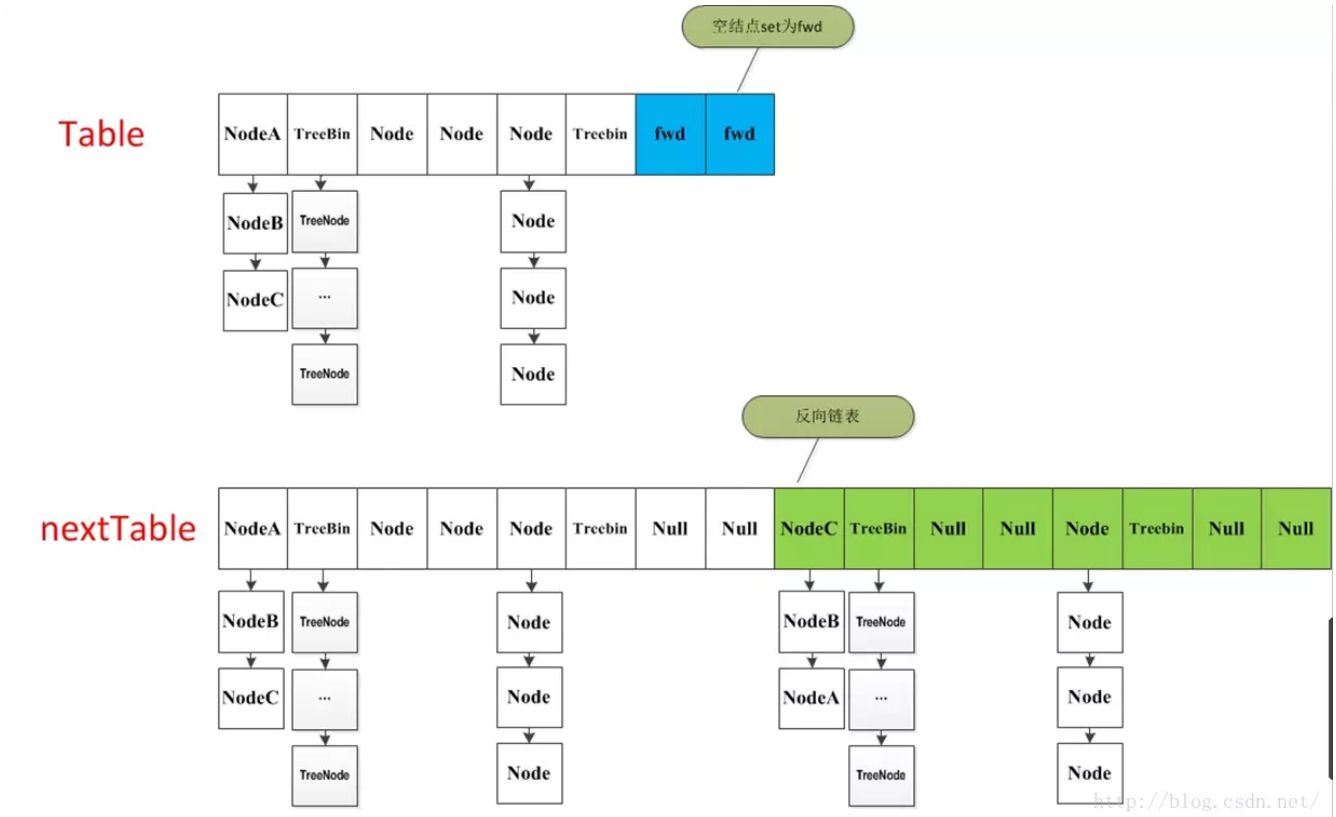
```
                }
            }
        }
    }
}
```

代码逻辑请看注释,整个扩容操作分为**两个部分**:

第一部分是构建一个nextTable,它的容量是原来的两倍，这个操作是单线程完成的。新建table数组的代码为:`Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1]`,在原容量大小的基础上右移一位。

第二个部分就是将原来table中的元素复制到nextTable中，主要是遍历复制的过程。 根据运算得到当前遍历的数组的位置i，然后利用tabAt方法获得i位置的元素再进行判断:

1. 如果这个位置为空，就在原table中的i位置放入forwardNode节点，这个也是触发并发扩容的关键点;
2. 如果这个位置是Node节点（fh>=0），如果它是一个链表的头节点，就构造一个反序链表，把他们分别放在nextTable的i和i+n的位置上
3. 如果这个位置是TreeBin节点（fh<0），也做一个反序处理，并且判断是否需要untreefi，把处理的结果分别放在nextTable的i和i+n的位置上
4. 遍历过所有的节点以后就完成了复制工作，这时让nextTable作为新的table，并且更新sizeCtl为新容量的0.75倍，完成扩容。设置为新容量的0.75倍代码为 `sizeCtl = (n << 1) - (n >>> 1)`，仔细体会下是不是很巧妙，n<<1相当于n右移一位表示n的两倍即2n,n>>>1左右一位相当于n除以2即0.5n,然后两者相减为2n-0.5n=1.5n,是不是刚好等于新容量的0.75倍即2n*0.75=1.5n。最后用一个示意图来进行总结（图片摘自网络）：



（6）与size相关的一些方法

对于ConcurrentHashMap来说，这个table里到底装了多少东西其实是个不确定的数量，因为**不可能在调用size()方法的时候像GC的“stop the world”一样让其他线程都停下来让你去统计，因此只能说这个数量是个估计值。对于这个估计值**，ConcurrentHashMap也是大费周章才计算出来的。

为了统计元素个数，ConcurrentHashMap定义了一些变量和一个内部类：

```
/**
 * A padded cell for distributing counts.  Adapted from LongAdder
 * and Striped64.  See their internal docs for explanation.
 */
@sun.misc.Contended static final class CounterCell {
    volatile long value;
    CounterCell(long x) { value = x; }
}

/*****************************************/

/**
 * 实际上保存的是hashmap中的元素个数  利用CAS锁进行更新
 但它并不用返回当前hashmap的元素个数

 */
private transient volatile long baseCount;
/**
 * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
 */
private transient volatile int cellsBusy;

/**
 * Table of counter cells. When non-null, size is a power of 2.
 */
private transient volatile CounterCell[] counterCells;
```

mappingCount与size方法：

mappingCount与**size**方法的类似 从给出的注释来看，应该使用mappingCount代替size方法 两个方法都没有直接返回basecount 而是统计一次这个值，而这个值其实也是一个大概的数值，因此可能在统计的时候有其他线程正在执行插入或删除操作。

```
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}
 /**
 * Returns the number of mappings. This method should be used
 * instead of {@link #size} because a ConcurrentHashMap may
 * contain more mappings than can be represented as an int. The
 * value returned is an estimate; the actual count may differ if
 * there are concurrent insertions or removals.
 *
 * @return the number of mappings
 * @since 1.8
 */
public long mappingCount() {
```

```
    long n = sumCount();
    return (n < 0L) ? 0L : n; // ignore transient negative values
}

 final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;//所有counter的值求和
        }
    }
    return sum;
}
```

addCount方法:

在put方法结尾处调用了addCount方法，把当前ConcurrentHashMap的元素个数+1这个方法一共做了两件事,更新baseCount的值，检测是否进行扩容。

```
private final void addCount(long x, int check) {
    CounterCell[] as; long b, s;
    //利用CAS方法更新baseCount的值
    if ((as = counterCells) != null ||
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            !(uncontended =
              U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        s = sumCount();
    }
    //如果check值大于等于0 则需要检验是否需要进行扩容操作
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            //
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                //如果已经有其他线程在执行扩容操作
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
```

```
                    transfer(tab, nt);
                }
                //当前线程是唯一的或是第一个发起扩容的线程  此时nextTable=null
                else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                             (rs << RESIZE_STAMP_SHIFT) + 2))
                    transfer(tab, null);
                s = sumCount();
            }
        }
    }
```

二、CopyOnWriteArrayList

- add方法中使用ReentrantLock进行加锁解锁
- 读写分离：读直接从数组中读取数据，写的时候拷贝原来数组中的元素，对新数组进行插入再赋值回去
- 每次新增一个元素，数组长度 + 1的方式添加元素
- volatile修饰存储数据的数组，保证可见性
- 只能保证数据最终的一致性，不能保证实时一致性

ArrayList不是线程安全的，Vector是线程安全的或者使用Collections的静态方法将ArrayList包装成一个线程安全的类，但是这些方法都是采用java关键字synchronzied对方法进行修饰，利用独占式锁来保证线程安全的。但是，由于独占式锁在同一时刻只有一个线程能够获取到对象监视器，很显然这种方式效率并不是太高。

对于读多写少的应用场景，Vector和Collections转换的这些方式是不合理的，因为尽管多个读线程从同一个数据容器中读取数据，但是读线程对数据容器的数据并不会发生发生修改。很自然而然的我们会联想到ReenTrantReadWriteLock，通过**读写分离**的思想，使得读读之间不会阻塞，无疑如果一个list能够做到被多个读线程读取的话，性能会大大提升不少。但是，如果仅仅是将list通过读写锁（ReentrantReadWriteLock）进行再一次封装的话，由于读写锁的特性，当写锁被写线程获取后，读写线程都会被阻塞。如果仅仅使用读写锁对list进行封装的话，这里仍然存在读线程在读数据的时候被阻塞的情况，如果想list的读效率更高的话，这里就是我们的突破口，如果我们保证读线程无论什么时候都不被阻塞，效率岂不是会更高？**CopyOnWriteArrayList容器可以保证线程安全，保证读读之间在任何时候都不会被阻塞**，CopyOnWriteArrayList也被广泛应用于很多业务场景之中。

1、COW的设计思想

如果简单的使用读写锁的话，在写锁被获取之后，读写线程被阻塞，只有当写锁被释放后读线程才有机会获取到锁从而读到最新的数据，站在**读线程的角度来看，即读线程任何时候都是获取到最新的数据，满足数据实时性。**既然我们说到要进行优化，必然有trade-off,我们就可以**牺牲数据实时性满足数据的最终一致性即可。**而CopyOnWriteArrayList就是通过Copy-On-Write(COW)，即写时复制的思想来通过延时更新的策略来实现数据的最终一致性，并且能够保证读线程间不阻塞。

COW通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。对CopyOnWrite容器进行并发的读的时候，不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，延时更新的策略是通过在写的时候针对的是不同的数据容器来实现的，放弃数据实时性达到数据的最终一致性。

2、CopyOnWriteArrayList的实现原理

CopyOnWriteArrayList内部维护的就是一个数组

```
/** The array, accessed only via getArray/setArray. */
private transient volatile Object[] array;
```

并且该数组引用是被volatile修饰，注意这里**仅仅是修饰的是数组引用**，其中另有玄机，稍后揭晓。关于volatile很重要的一条性质是它能够够保证可见性。

（1）get方法实现原理

```
public E get(int index) {
    return get(getArray(), index);
}
/**
 * Gets the array.  Non-private so as to also be accessible
 * from CopyOnWriteArraySet class.
 */
final Object[] getArray() {
    return array;
}
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

可以看出来get方法实现非常简单，几乎就是一个“单线程”程序，没有对多线程添加任何的线程安全控制，也没有加锁也没有CAS操作等等，原因是，所有的读线程只是会读取数据容器中的数据，并不会进行修改。

（2）add方法实现原理

```
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    //1. 使用Lock,保证写线程在同一时刻只有一个
    lock.lock();
    try {
        //2. 获取旧数组引用
        Object[] elements = getArray();
        int len = elements.length;
        //3. 创建新的数组，并将旧数组的数据复制到新数组中，每次新增一个元素，数组长度+1
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        //4. 往新数组中添加新的数据
        newElements[len] = e;
        //5. 将旧数组引用指向新的数组
        setArray(newElements);
        return true;
    } finally {
        lock.unlock();
    }
}
```

add方法的逻辑也比较容易理解，请看上面的注释。需要注意这么几点：

1. 采用ReentrantLock，保证同一时刻只有一个写线程正在进行数组的复制，否则的话内存中会有多份被复制的数据；

2. 前面说过数组引用是volatile修饰的，因此将旧的数组引用指向新的数组，根据volatile的happens-before规则，写线程对数组引用的修改对读线程是可见的。
3. 由于在写数据的时候，是在新的数组中插入数据的，从而保证读写实在两个不同的数据容器中进行操作。

3、总结

COW和读写锁：

相同点：两数都是通过读写分离的思想实现的；读线程间是互不阻塞的。

不同点：对读线程而言，为了实现数据实时性，在写锁被获取后，读线程会等待或者当读锁被获取后，写线程会等待，从而解决“脏读”等问题。也就是说如果使用读写锁依然会出现读线程阻塞等待的情况。而COW则完全放开了牺牲数据实时性而保证数据最终一致性，即读线程对数据的更新是延时感知的，因此读线程不会存在等待的情况。

如果将array 数组设定为volitile的， 对volatile变量写happens-before读，读线程不是能够感知到volatile变量的变化。原因是，这里volatile的修饰的**仅仅**只是**数组引用**，**数组中的元素的修改是不能保证可见性的。**因此COW采用的是新旧两个数据容器。这也是为什么concurrentHashMap只具有弱一致性的原因。

COW的缺点：内存占用问题和数据一致性问题。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

三、ConcurrentLinkedQueue

- 内部使用单向链表实现
- offer中使用CAS操作设置队尾

1、Node

```
private static class Node<E> {
    volatile E item;
    volatile Node<E> next;
    ......
}
```

Node节点主要包含了两个域：一个是数据域item，另一个是next指针，用于指向下一个节点从而构成链式队列。并且都是用volatile进行修饰的，以保证内存可见性。另外ConcurrentLinkedQueue含有这样两个成员变量：

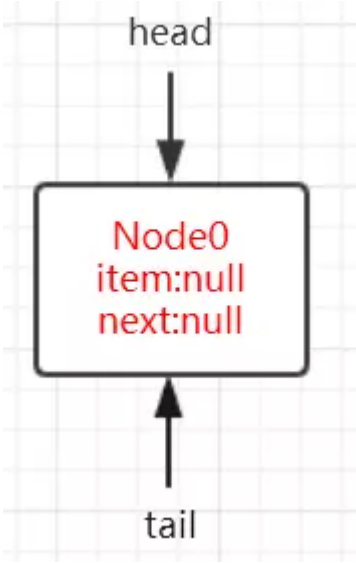
```
private transient volatile Node<E> head;
private transient volatile Node<E> tail;
```

说明ConcurrentLinkedQueue通过持有头尾指针进行管理队列。当我们调用无参构造器时，其源码为：

```
public ConcurrentLinkedQueue() {
    head = tail = new Node<E>(null);
}
```

head和tail指针会指向一个item域为null的节点,此时ConcurrentLinkedQueue状态如下图所示：

如图，head和tail指向同一个节点Node0，该节点item域为null,next域为null。



2、操作Node的几个CAS操作

在队列进行出队入队的时候免不了对节点需要进行操作，在多线程就很容易出现线程安全的问题。可以看出在处理器指令集能够支持**CMPXCHG**指令后，在java源码中涉及到并发处理都会使用CAS操作，那么在ConcurrentLinkedQueue对Node的CAS操作有这样几个：

```
//更改Node中的数据域item
boolean casItem(E cmp, E val) {
    return UNSAFE.compareAndSwapObject(this, itemOffset, cmp, val);
}
//更改Node中的指针域next
void lazySetNext(Node<E> val) {
    UNSAFE.putOrderedObject(this, nextOffset, val);
}
//更改Node中的指针域next
boolean casNext(Node<E> cmp, Node<E> val) {
    return UNSAFE.compareAndSwapObject(this, nextOffset, cmp, val);
}
```

可以看出这些方法实际上是通过调用UNSAFE实例的方法，UNSAFE为**sun.misc.Unsafe**类，该类是hotspot底层方法。

3、常用方法

可以有以下几种情况：

- 单个线程offer
- 多个线程offer
- 部分线程offer，部分线程poll
- offer的速度快于poll：即向队列尾部插入节点的速度比从队列中删除头部节点的速度快，此时，offer和poll线程不会出现“交集”，两类线程之间不会相互影响。
- offer的速度慢于poll：即向队列尾部插入节点的速度比从队列中删除头部节点的速度慢，此时offer和poll线程会出现“交集”，在出现“交集”的那一刻就可以称之为offer线程和poll线程同时操作的节点为临界点，且在该节点offer线程和poll线程必定相互影响。根据在临界点时offer和poll发生的相对顺序又可从两个角度去思考：1. 执

行顺序为offer-->poll-->offer，即表现为当offer线程在Node1后插入Node2时，此时poll线程已经将Node1删除，这种情况很显然需要在offer方法中考虑； 2.执行顺序可能为：poll-->offer-->poll，即表现为当poll线程准备删除的节点为null时（队列为空队列），此时offer线程插入一个节点使得队列变为非空队列。

（1）offer方法（CAS操作设置队尾）

```
ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();
queue.offer(1);
queue.offer(2);
```

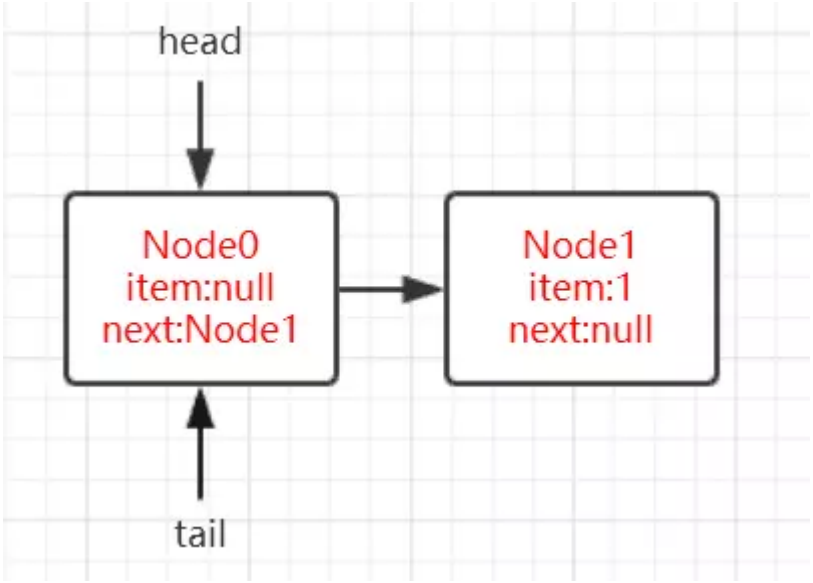
队列的offer方法是向队列尾部添加一个元素。方法源码为：

```
public boolean offer(E e) {
1.    checkNotNull(e);
2.    final Node<E> newNode = new Node<E>(e);
3.    for (Node<E> t = tail, p = t;;) {
4.        Node<E> q = p.next;
5.        if (q == null) {
6.            // p is last node
7.            if (p.casNext(null, newNode)) {
                  // Successful CAS is the linearization point
                  // for e to become an element of this queue,
                  // and for newNode to become "live".
8.                if (p != t) // hop two nodes at a time
9.                    casTail(t, newNode);  // Failure is OK.
10.                return true;
                  }
              // Lost CAS race to another thread; re-read next
            }
11.         else if (p == q)
             // We have fallen off list.  If tail is unchanged, it
             // will also be off-list, in which case we need to
             // jump to head, from which all live nodes are always
             // reachable.  Else the new tail is a better bet.
             // 此处的看视频，https://www.bilibili.com/video/av35040927?from=search&seid=4510582629928004591，有很大差别！局部变量和操作栈
12.            p = (t != (t = tail)) ? t : head;
        else
            // Check for tail updates after two hops.
13.            p = (p != t && t != (t = tail)) ? t : q;
    }
}
```

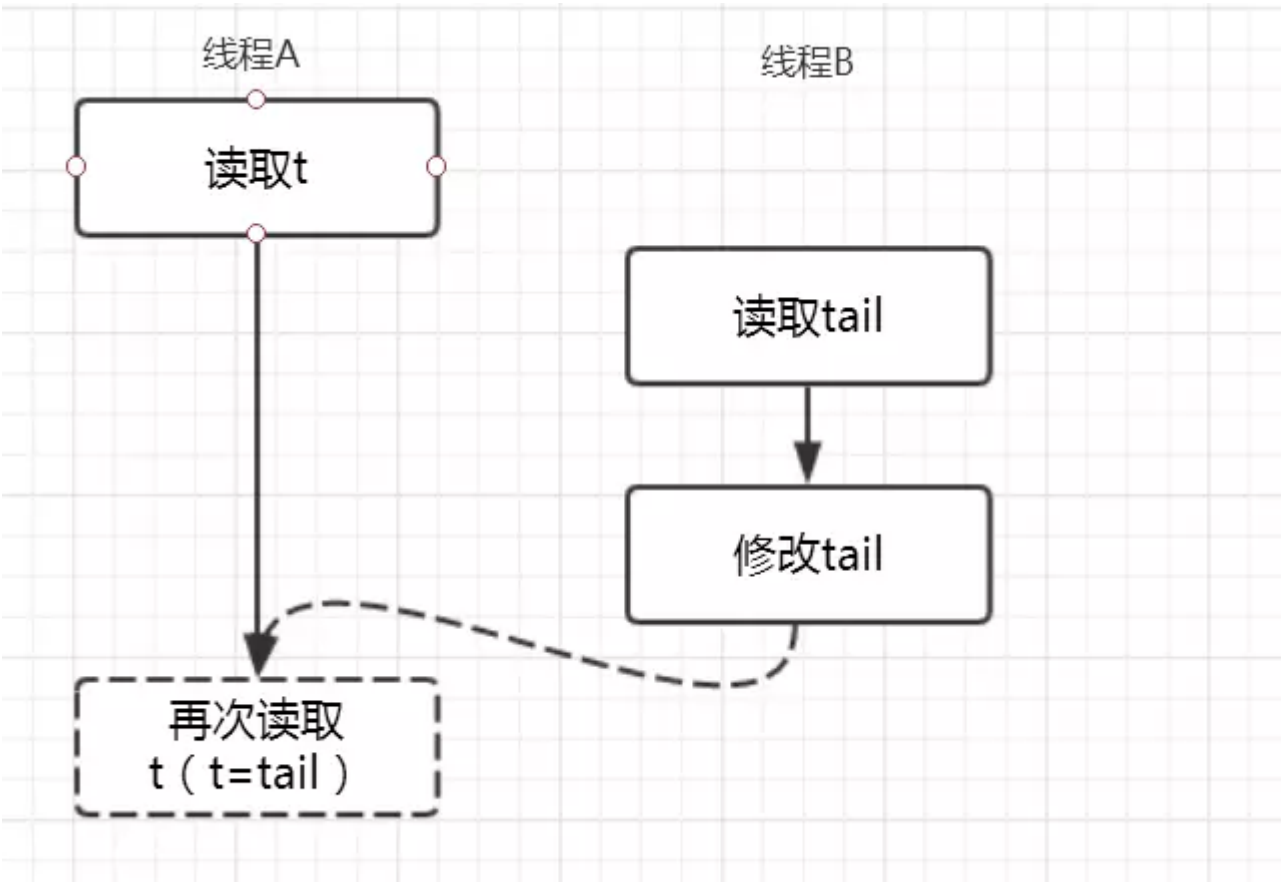
从单线程执行角度分析：

先从单线程执行的角度看起，分析offer 1的过程。第1行代码会对是否为null进行判断，为null的话就直接抛出空指针异常，第2行代码将e包装成一个Node类，第3行为for循环，只有初始化条件没有循环结束条件，这很符合CAS的“套路”，在循环体CAS操作成功会直接return返回，如果CAS操作失败的话就在for循环中不断重试直至成功。这里实例变量t被初始化为tail，p被初始化为t即tail。为了方便下面的理解，p被认为队列真正的尾节点，tail不一定指向对象真正的尾节点，因为在ConcurrentLinkedQueue中tail是被延迟更新的，具体原因我们慢慢来看。代码走到第3行的时候，t和p都分别指向初始化时创建的item域为null，next域为null的Node0。第4行变量q被赋值为null，第5行if判

断为true，在第7行使用casNext将插入的Node设置成当前队列尾节点p的next节点，如果CAS操作失败，此次循环结束在下次循环中进行重试。CAS操作成功走到第8行，此时p==t，if判断为false,直接return true返回。如果成功插入1的话，此时ConcurrentLinkedQueue的状态如下图所示：



多线程（多个线程offer）：



如图，假设线程A此时读取了变量t，线程B刚好在这个时候offer一个Node后，此时会修改tail指针,那么这个时候线程A再次执行t=tail时t会指向另外一个节点，很显然线程A前后两次读取的变量t指向的节点不相同，即`t != (t = tail)`为true,并且由于t指向节点的变化`p != t`也为true，此时该行代码的执行结果为p和t最新的t指针指向了同一个节点，并且此时t也是队列真正的对尾节点。那么，现在已经定位到队列真正的队尾节点，就可以执行offer操作了。

poll->offer->poll：

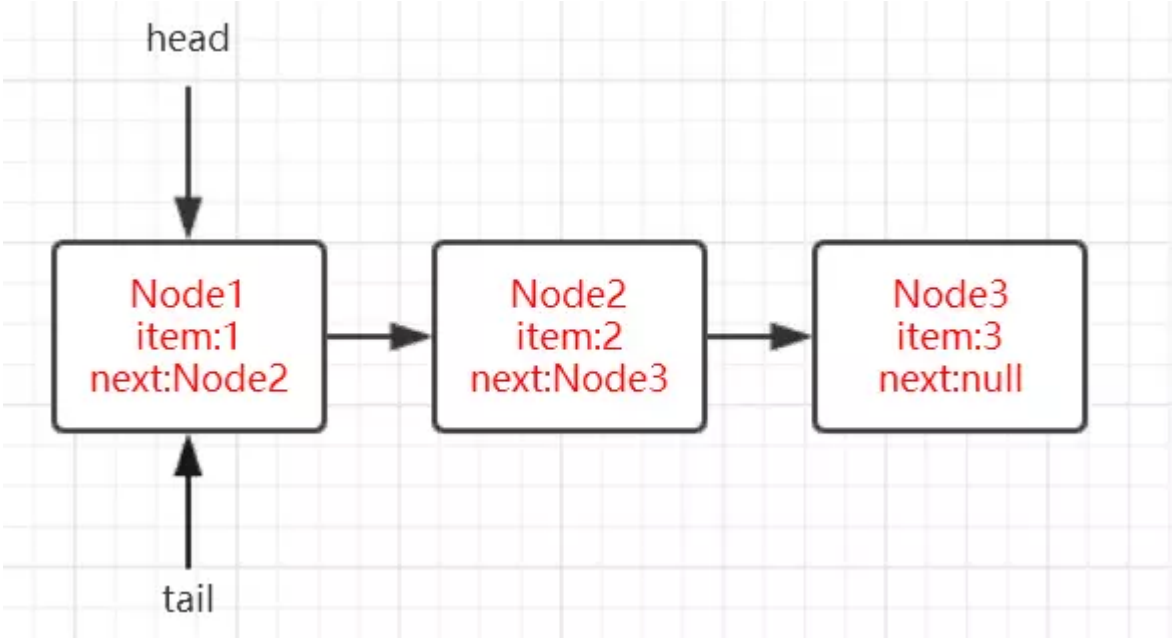
那么还剩下第11行的代码我们没有分析，大致可以猜想到应该就是回答一部分线程offer，一部分poll的这种情况。当if (p == q)为true时，说明p指向的节点的next也指向它自己，这种节点称之为哨兵节点，这种节点在队列中存在的价值不大，一般表示为要删除的节点或者是空节点。为了能够很好的理解这种情况，我们先看看poll方法的执行过程后，再回过头来看，总之这是一个很有意思的事情 :)。

（2）poll方法

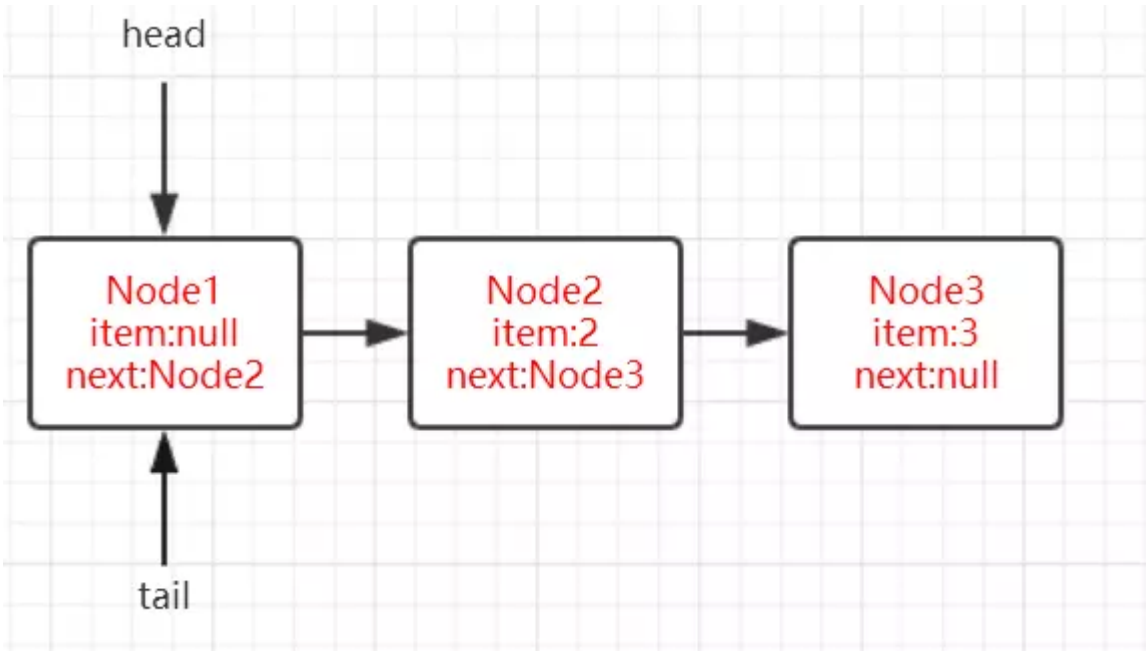
队列的poll方法是获取队列头部元素，并删除该元素。方法源码为：

```
1  public E poll() {
2      restartFromHead:
3      1. for (;;) {
4      2.     for (Node<E> h = head, p = h, q;;) {
5      3.         E item = p.item;
6      4.         if (item != null && p.casItem(item, null)) {
7                   // Successful CAS is the linearization point
8                   // for item to be removed from this queue.
9      5.             if (p != h) // hop two nodes at a time
10     6.                 updateHead(h, ((q = p.next) != null) ? q : p);
11     7.             return item;
12             }
13     8.         else if ((q = p.next) == null) {
14     9.             updateHead(h, p);
15     10.             return null;
16             }
17     11.         else if (p == q)
18     12.             continue restartFromHead;
19             else
20     13.             p = q;
21         }
22     }
23 }
```

从单线程角度：



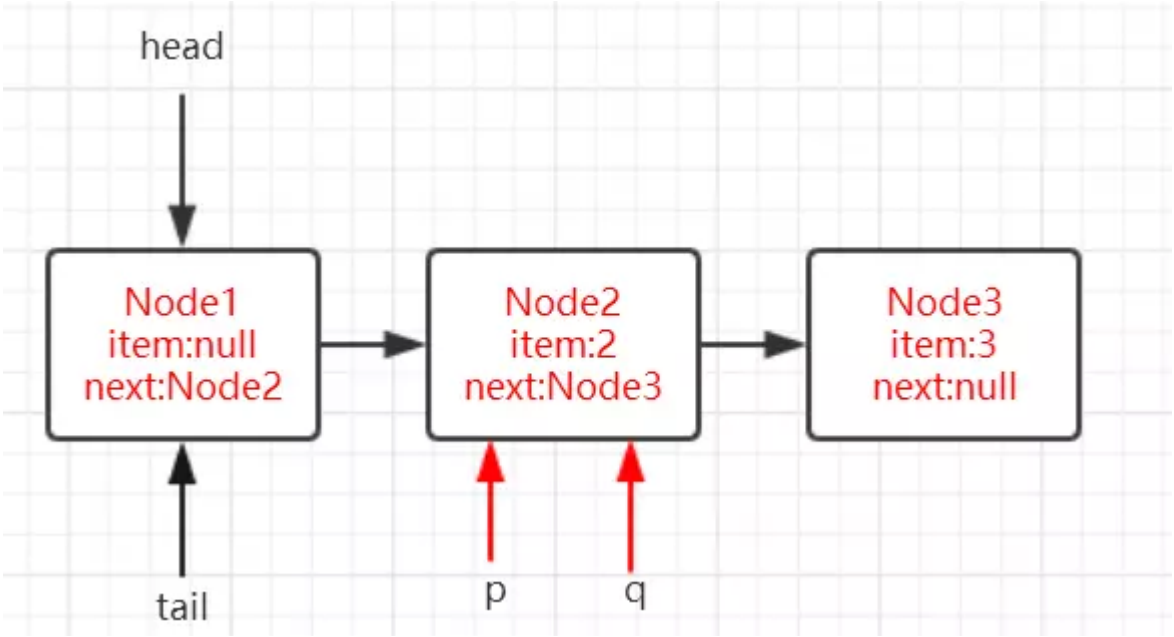
我们还是先将变量p作为队列要删除真正的队头节点，h（head）指向的节点并不一定是队列的队头节点。先来看poll出Node1时的情况，由于p=h=head，参照上图，很显然此时p指向的Node1的数据域不为null,在第4行代码中item!=null判断为true后接下来通过casItem将Node1的数据域设置为null。如果CAS设置失败则此次循环结束等待下一次循环进行重试。若第4行执行成功进入到第5行代码，此时p和h都指向Node1,第5行if判断为false,然后直接到第7行return回Node1的数据域1，方法运行结束，此时的队列状态如下图。



下面继续从队列中poll，很显然当前h和p指向的Node1的数据域为null，那么第一件事就是要**定位准备删除的队头节点(找到数据域不为null的节点)。**

定位删除的队头节点：

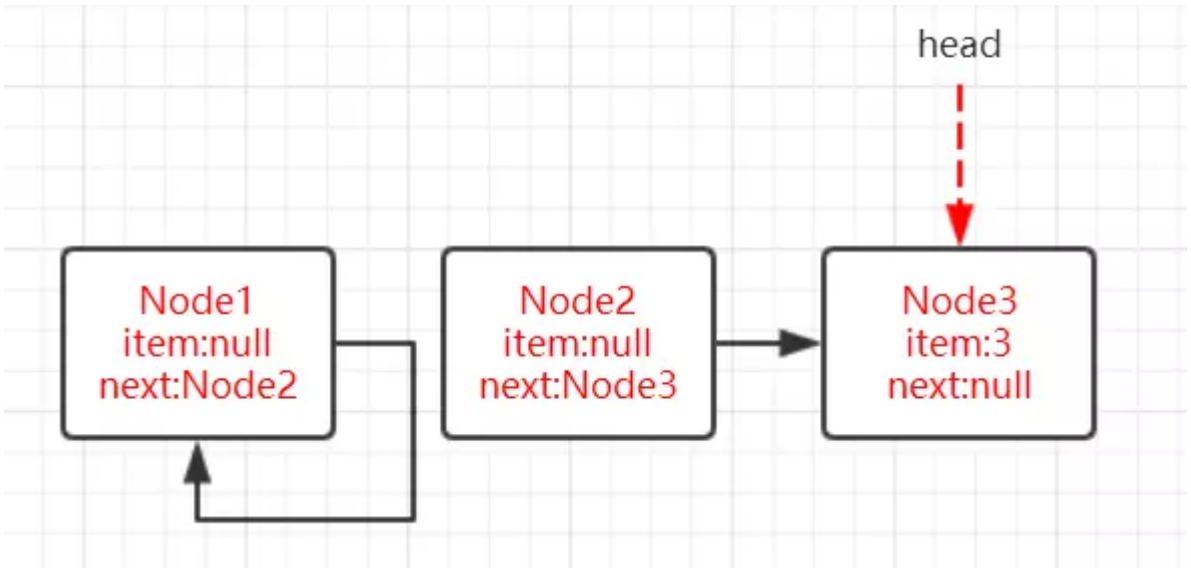
继续看，第三行代码item为null,第4行代码if判断为false,走到第8行代码（q = p.next）if也为false，由于q指向了Node2,在第11行的if判断也为false，因此代码走到了第13行，这个时候p和q共同指向了Node2,也就找到了要删除的真正的队头节点。可以总结出，定位待删除的队头节点的过程为：如果当前节点的数据域为null，很显然该节点不是待删除的节点，就用当前节点的下一个节点去试探。在经过第一次循环后，此时状态图为下图：



进行下一次循环，第4行的操作同上述，当前假设第4行中casItem设置成功，由于p已经指向了Node2,而h还依旧指向Node1,此时第5行的if判断为true，然后执行updateHead(h, ((q = p.next) != null) ? q : p)，此时q指向的Node3，所有传入updateHead方法的分别是指向Node1的h引用和指向Node3的q引用。updateHead方法的源码为：

```
1  final void updateHead(Node<E> h, Node<E> p) {
2      if (h != p && casHead(h, p))
3          h.lazySetNext(h);
4  }
```

该方法主要是通过casHead将队列的head指向Node3,并且通过 h.lazySetNext将Node1的next域指向它自己。最后在第7行代码中返回Node2的值。此时队列的状态如下图所示：



Node1的next域指向它自己，head指向了Node3。如果队列为空队列的话，就会执行到代码的第8行(q = p.next) == null，if判断为true,因此在第10行中直接返回null。以上的分析是从单线程执行的角度去看，也可以让我们了解poll的整体思路，现在来做一个总结：

- 如果当前head,h和p指向的节点的Item不为null的话，说明该节点即为真正的队头节点（待删除节点），只需要通过casItem方法将item域设置为null,然后将原来的item直接返回即可。

- 如果当前head,h和p指向的节点的item为null的话，则说明该节点不是真正的待删除节点，那么应该做的就是寻找item不为null的节点。通过让q指向p的下一个节点（q = p.next）进行试探，若找到则通过updateHead方法更新head指向的节点以及构造哨兵节点（通过updateHead方法的h.lazySetNext(h)）。

从多线程角度（多个线程poll）：

```
    11.         else if (p == q)
    12.             continue restartFromHead;
```

这一部分就是处理多个线程poll的情况，q = p.next也就是说q永远指向的是p的下一个节点，那么什么情况下会使得p,q指向同一个节点呢？根据上面我们的分析，只有p指向的节点在poll的时候转变成了哨兵节点（通过updateHead方法中的h.lazySetNext）。当线程A在判断p==q时，线程B已经将执行完poll方法将p指向的节点转换为哨兵节点并且head指向的节点已经发生了改变，所以就需要从restartFromHead处执行，保证用到的是最新的head。

poll->offer->poll:

如果当前队列为空队列，线程A进行poll操作，同时线程B执行offer，然后线程A在执行poll，那么此时线程A返回的是null还是线程B刚插入的最新的那个节点呢？有一个demo：

```
public static void main(String[] args) {
    Thread thread1 = new Thread(() -> {
        Integer value = queue.poll();
        System.out.println(Thread.currentThread().getName() + " poll 的值为: " +
value);
        System.out.println("queue当前是否为空队列: " + queue.isEmpty());
    });
    thread1.start();
    Thread thread2 = new Thread(() -> {
        queue.offer(1);
    });
    thread2.start();
}
```

输出的结果为：

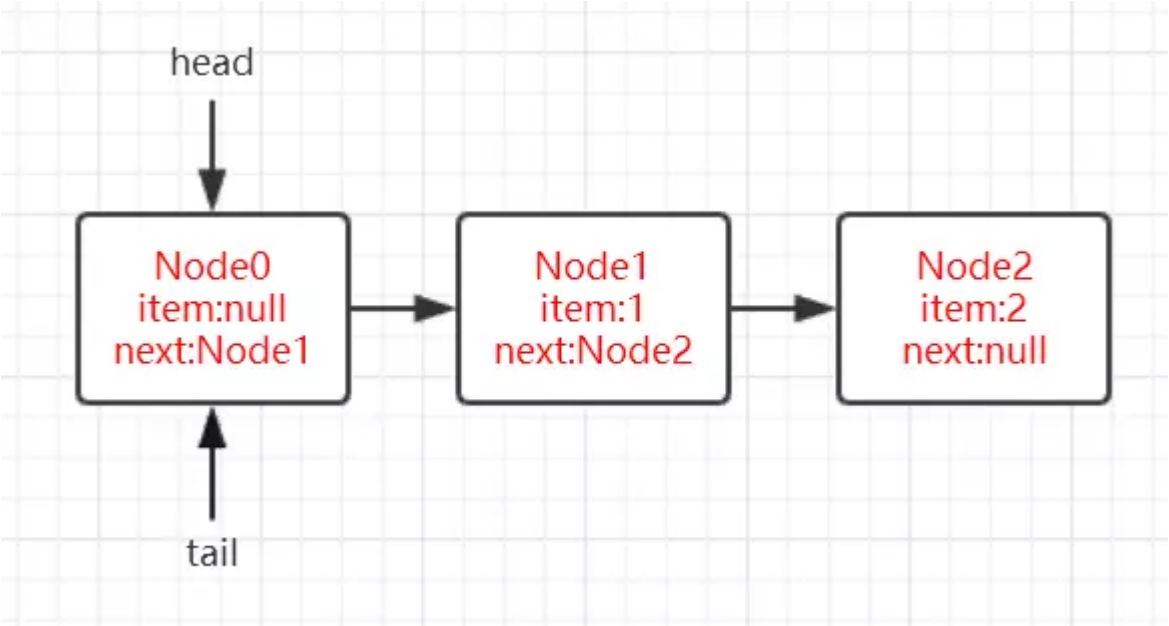
```
Thread-0 poll 的值为: null queue当前是否为空队列: false
```

通过debug控制线程thread1和线程thread2的执行顺序，thread1先执行到第8行代码if ((q = p.next) == null)，由于此时队列为空队列if判断为true，进入if块，此时先让thread1暂停，然后thread2进行offer插入值为1的节点后，thread2执行结束。再让thread1执行，这时thread1并没有进行重试，而是代码继续往下走，返回null，尽管此时队列由于thread2已经插入了值为1的新的节点。所以输出结果为thread0 poll的为null,然队列不为空队列。因此，在判断队列是否为空队列的时候是不能通过线程在poll的时候返回为null进行判断的，可以通过isEmpty方法进行判断。

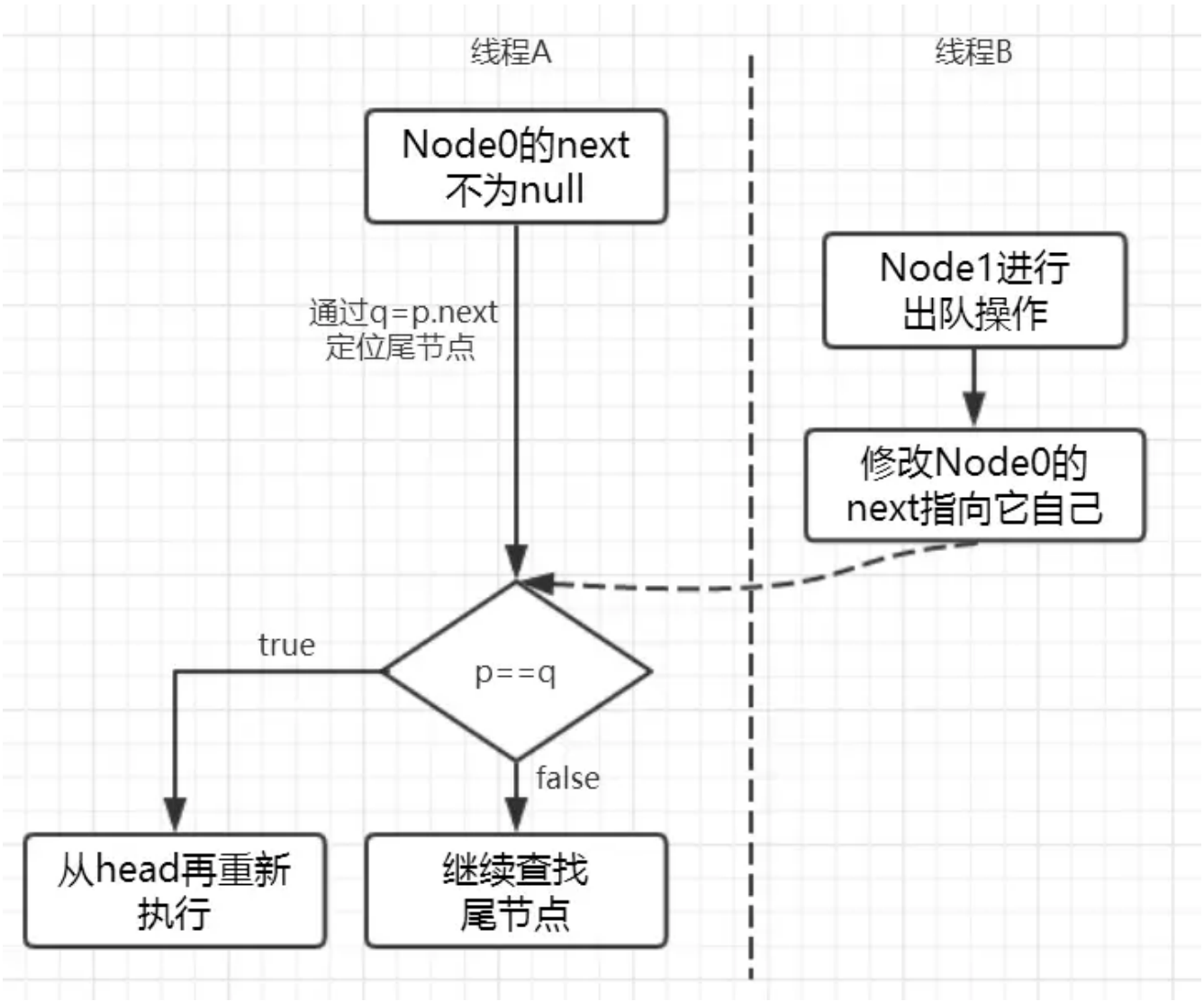
（3）offer方法中部分线程offer部分线程poll

offer->poll->offer

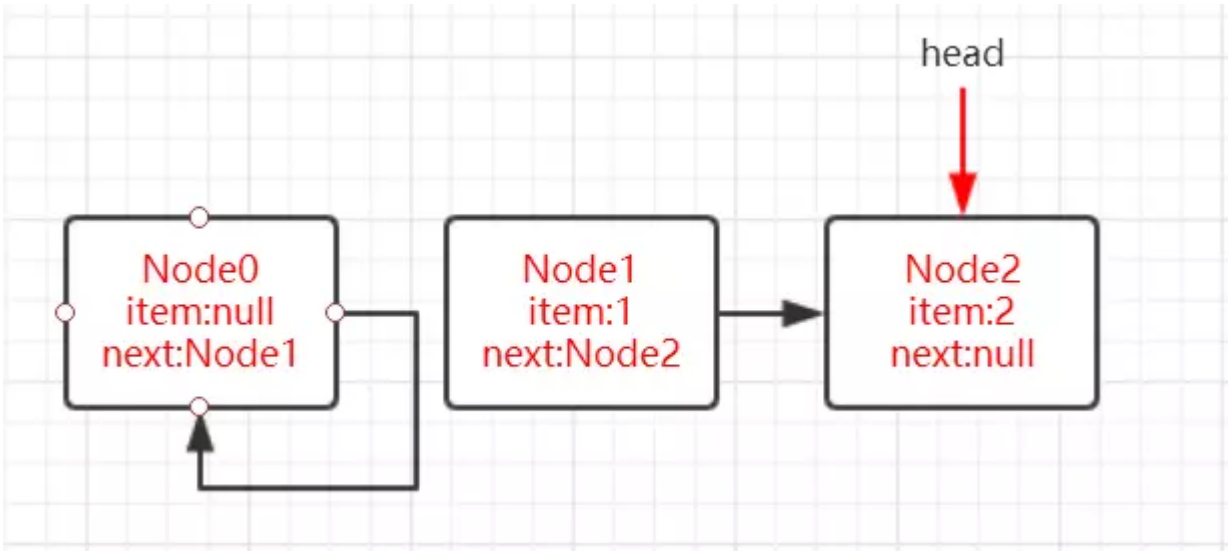
在offer方法的第11行代码`if (p == q)`，能够让if判断为true的情况为p指向的节点为**哨兵节点**，而什么时候会构造哨兵节点呢？在对poll方法的讨论中，我们已经找到了答案，即**当head指向的节点的item域为null时会寻找真正的队头节点，等到待插入的节点插入之后，会更新head，并且将原来head指向的节点设置为哨兵节点。**假设队列初始状态如下图所示：



因此在线程A执行offer时，线程B执行poll就会存在如下一种情况：



如图，线程A的tail节点存在next节点Node1,因此会通过引用q往前寻找队列真正的队尾节点，当执行到判断`if (p == q)`时，此时线程B执行poll操作，在对线程B来说，head和p指向Node0,由于Node0的item域为null,同样会往前递进找到队列真正的队头节点Node1,在线程B执行完poll之后，Node0就会转换为**哨兵节点**，也就意味着队列的head发生了改变，此时队列状态为下图。



此时线程A在执行判断`if (p == q)`时就为true,会继续执行`p = (t != (t = tail)) ? t : head;`，由于tail指针没有发生改变所以p被赋值为head,重新从head开始完成插入操作。

4、HOPS设计

通过上面对offer和poll方法的分析，我们发现tail和head是延迟更新的，两者更新触发时机为：

tail更新触发时机：当tail指向的节点的下一个节点不为null的时候，会执行定位队列真正的队尾节点的操作，找到队尾节点后完成插入之后才会通过casTail进行tail更新；当tail指向的节点的下一个节点为null的时候，只插入节点不更新tail。

head更新触发时机：当head指向的节点的item域为null的时候，会执行定位队列真正的队头节点的操作，找到队头节点后完成删除之后才会通过updateHead进行head更新；当head指向的节点的item域不为null的时候，只删除节点不更新head。

并且在更新操作时，源码中会有注释为：hop two nodes at a time。所以这种延迟更新的策略就被叫做HOPS的大概原因是这个（猜的 :)），从上面更新时的状态图可以看出，head和tail的更新是"跳着的"即中间总是间隔了一个。那么这样设计的意图是什么呢?

如果让tail永远作为队列的队尾节点，实现的代码量会更少，而且逻辑更易懂。但是，这样做有一个缺点，**如果大量的入队操作，每次都要执行CAS进行tail的更新，汇总起来对性能也会是大大的损耗。如果能减少CAS更新的操作，无疑可以大大提升入队的操作效率，所以doug lea大师每间隔1次（tail和队尾节点的距离为1）进行才利用CAS更新tail。**对head的更新也是同样的道理，虽然，这样设计会多出在循环中定位队尾节点，但总体来说读的操作效率要远远高于写的性能，因此，多出来的在循环中定位尾节点的操作的性能损耗相对而言是很小的。

四、阻塞队列（BlockingQueue）

最常用的"**生产者-消费者**"问题中，队列通常被视作线程间操作的数据容器，这样，可以对各个模块的业务功能进行解耦，生产者将“生产”出来的数据放置在数据容器中，而消费者仅仅只需要在“数据容器”中进行获取数据即可，这样生产者线程和消费者线程就能够进行解耦，只专注于自己的业务功能即可。阻塞队列（BlockingQueue）被广泛使用在“生产者-消费者”问题中，其原因是BlockingQueue提供了可阻塞的插入和移除的方法。

当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。

1、基本操作

BlockingQueue继承于Queue接口，因此，对数据元素的基本操作有：

（1）插入元素（队列已满add方法会抛出异常，offer会返回false）

add(E e)：往队列插入数据，当队列满时，插入元素时会抛出IllegalStateException异常；

offer(E e)：当往队列插入数据时，插入成功返回`true`，否则则返回`false`。当队列满时不会抛出异常；

（2）删除元素

remove(Object o)：从队列中删除数据，成功则返回true，否则为false

poll：删除数据，当队列为空时，返回null；

（3）查看元素

element：获取队头元素，如果队列为空则抛出NoSuchElementException异常；

peek： 获取队头元素，如果队列为空则抛出NoSuchElementException异常

（4）BlockingQueue具有的特殊操作（put和take方法会阻塞）

插入数据：

- put：当阻塞队列容量已经满时，往阻塞队列插入数据的线程会被**阻塞**，直至阻塞队列已经有空余的容量可供使用；
- offer(E e, long timeout, TimeUnit unit)：若阻塞队列已经满时，同样会阻塞插入数据的线程，直至阻塞队列已经有空余的地方，与put方法不同的是，该方法会有一个超时时间，若超过当前给定的超时时间，插入数据的线程会退出

删除数据：

- take()：当阻塞队列为空时，获取队头数据的线程会被**阻塞**；
- poll(long timeout, TimeUnit unit)：当阻塞队列为空时，获取数据的线程会被阻塞，另外，如果被阻塞的线程超过了给定的时长，该线程会退出

2、常用的BlockingQueue

实现BlockingQueue接口的常见阻塞队列：

阻塞队列	特点
ArrayBlockingQueue	有界阻塞队列，循环队列； 当队列已满，put放入元素会阻塞 当队列为空，take取元素会阻塞
LinkedBlockingQueue	有界阻塞队列； 比ArrayBlockingQueue具有更高的吞吐量
PriorityBlockingQueue	支持优先级的无界阻塞队列
SynchronousQueue	不存储任何元素
LinkedTransferQueue	无界阻塞队列
LinkedBlockingDeque	基于链表结构的有界阻塞双端队列
DelayQueue	实现Delay接口的无界队列

(1) ArrayBlockingQueue

- 由数组实现的有界阻塞队列，因此可以作为“有界缓冲区”
- 队列一旦创建，容量不能改变
- 在put(E e)的时候会检查e是否为null，若为null，则会抛出空指针异常
- 使用的是循环队列，就不用移动元素
- 使用表示 是否已满的Condition 和 是否已空的Condition 以及 lock.lockInterruptibly() 来实现

ArrayBlockingQueue是由数组实现的**有界阻塞队列**。该队列命令元素FIFO（先进先出）。因此，队头元素时队列中存在时间最长的数据元素，而队尾数据则是当前队列最新的数据元素。ArrayBlockingQueue可作为“有界数据缓冲区”，生产者插入数据到队列容器中，并由消费者提取。ArrayBlockingQueue一旦创建，容量不能改变。 当队列容量满时，**尝试将元素放入队列将导致操作阻塞;尝试从一个空队列中取一个元素也会同样阻塞。** ArrayBlockingQueue默认情况下不能保证线程访问队列的公平性，所谓公平性是指严格按照线程等待的绝对时间顺序，即最先等待的线程能够最先访问到ArrayBlockingQueue。而非公平性则是指访问ArrayBlockingQueue的顺序不是遵守严格的时间顺序，有可能存在，一旦ArrayBlockingQueue可以被访问时，长时间阻塞的线程依然无法访问到ArrayBlockingQueue。如果保证公平性，通常会降低吞吐量。如果需要获得公平性的ArrayBlockingQueue，可采用如下代码：

```
1 private static ArrayBlockingQueue<Integer> blockingQueue = new
  ArrayBlockingQueue<Integer>(10,true);
```

① 添加（put方法会阻塞，队列已满add方法异常，offer返回false）

add方法和offer方法的区别 使用add如果队列已满，则抛出异常throw new IllegalStateException("Queue full")，内部由offer方法实现。 使用offer如果队列已满，则返回false

② 删除（take方法会阻塞，队列为空poll返回null）

poll方法和take方法的区别 使用poll如果队列中没有元素，则返回null，否则返回队列中的第一个元素 使用take如果队列中没有元素，则当前方法处于阻塞状态，直到队列中有元素

```
1 public E poll() {
2     final ReentrantLock lock = this.lock;
```

```
    lock.lock();
    try {
        return (count == 0) ? null : dequeue();
    } finally {
        lock.unlock();
    }
}

private final Condition notFull;
/*
 成功获取锁 -> 如果队列已满，则notFull.await()阻塞，直到notFull.signal() -> 插入队列
*/
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}

/** Condition for waiting takes */
private final Condition notEmpty;
/*
 成功获取锁 -> 如果队列为空，notEmpty.await()直到有新元素插入则notEmpty.signal() -> 移除元素
*/
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}

/** items index for next take, poll, peek or remove */
int takeIndex;
/*
 循环队列
*/
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)
        // 表示已经到达数组尾部了，则回到数组头部
        takeIndex = 0;
    count--;
    if (itrs != null)
        itrs.elementDequeued();
    notFull.signal();
    return x;
}
```

(2) LinkedBlockingQueue

- 有界阻塞队列，满足FIFO的特性
- 使用单向链表实现，通过AtomicInteger count来约束队列中插入的元素的个数。
- 使用putLock、notFull和takeLock、notEmpty，分别用于put和take函数中进行加锁
- put元素不允许为null

LinkedBlockingQueue是用链表实现的**有界阻塞队列**，同样满足FIFO的特性，与ArrayBlockingQueue相比起来具有更高的吞吐量，为了防止LinkedBlockingQueue容量迅速增，损耗大量内存。通常在创建LinkedBlockingQueue对象时，会指定其大小，如果未指定，容量等于Integer.MAX_VALUE。

① 添加（队列已满，put方法会阻塞）

add和offer方法的区别 使用add当队列已满，则抛出异常throw new IllegalStateException("Queue full")，其内部由offer实现 使用offer当队列已满，则返回false，否则入队

② 删除（队列为空，take方法阻塞）

poll和take方法的区别 使用poll方法，当队列为空，返回null 使用take方法，当队列为空，则take方法阻塞

```
/*
 putLock加锁 -> 如果队列已满，则notFull.await()阻塞线程 -> 等待notFull.signal()
 -> 加入队列 -> notFull.signal()
 和ArrayBlockingQueue不同的是notFull的 await 和 signal 在putLock中调用。
*/
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    // Note: convention in all put/take/etc is to preset local var
    // holding count negative to indicate failure unless set.
    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();
}
/*
 takeLock加锁 -> 如果队列为空，则notEmpty.await()阻塞线程 -> 等待notEmpty.signal()
 -> 其他线程从队列中取元素，如果队列中还有元素就notEmpty.signal()
*/
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;
}
```

(3) PriorityBlockingQueue

- 支持优先级的无界阻塞队列，默认情况下元素采用自然顺序进行排序，也可以通过compareTo()方法
- 内部使用动态数组实现。当原始容量小于64时，扩容方式为2*oldCap+2，否则以1.5倍方式扩容
- 使用ReentrantLock和notEmpty（Condition）配合实现

PriorityBlockingQueue是一个支持优先级的**无界阻塞队列**。默认情况下元素采用自然顺序进行排序，也可以通过自定义类实现compareTo()方法来指定元素排序规则，或者初始化时通过构造器参数Comparator来指定排序规则。使用offer方法时不阻塞。

① 添加（非阻塞lock，队列不会满）

add和offer方法的区别 add和offer方法相同，当容量达到数组上限时，增加数组长度，完成offer

② 删除（阻塞lock，队列为空时take会阻塞）

poll和take方法的区别 使用poll当队列中没有元素的时候，返回null 使用take当队列中没有元素的时候，线程阻塞

```
public void put(E e) {
    offer(e); // never need to block
}
```

```
/*
 非中断获取锁 -> 检查是否需要扩容 -> 元素加入队列 -> notEmpty.signal()
*/
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;
}
/*
 可中断式获取锁 -> 若队列为空，则notEmpty.await() -> notEmpty.signal()后获取元素并返回
*/
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;
}
```

(4) SynchronousQueue

SynchronousQueue每个插入操作必须等待另一个线程进行相应的删除操作，因此，SynchronousQueue实际上没有存储任何数据元素，因为只有线程在删除数据时，其他线程才能插入数据，同样的，如果当前有线程在插入数据时，线程才能删除数据。SynchronousQueue也可以通过构造器参数来为其指定公平性。

(5) LinkedTransferQueue

- 链表结构构成的无界阻塞队列

LinkedTransferQueue是一个由链表数据结构构成的**无界阻塞队列**，由于该队列实现了TransferQueue接口，与其他阻塞队列相比主要有以下不同的方法：

transfer(E e)：如果当前有线程（消费者）正在调用take()方法或者可延时的poll()方法进行消费数据时，生产者线程可以调用transfer方法将数据传递给消费者线程。如果当前没有消费者线程的话，生产者线程就会将数据插入到队尾，直到有消费者能够进行消费才能退出；

tryTransfer(E e)：tryTransfer方法如果当前有消费者线程（调用take方法或者具有超时特性的poll方法）正在消费数据的话，该方法可以将数据立即传送给消费者线程，如果当前没有消费者线程消费数据的话，就立即返回false。因此，与transfer方法相比，transfer方法是必须等到有消费者线程消费数据时，生产者线程才能够返回。而tryTransfer方法能够立即返回结果退出。

tryTransfer(E e,long timeout,imeUnit unit)：与transfer基本功能一样，只是增加了超时特性，如果数据才规定的超时时间内没有消费者进行消费的话，就返回false。

(6) LinkedBlockingDeque

LinkedBlockingDeque是基于链表数据结构的**有界阻塞双端队列**，如果在创建对象时为指定大小时，其默认大小为Integer.MAX_VALUE。与LinkedBlockingQueue相比，主要的不同点在于，LinkedBlockingDeque具有双端队列的特性。LinkedBlockingDeque基本操作如下图所示（来源于java文档）

Summary of BlockingDeque methods

First Element (Head)				
	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`addFirst(e)`	`offerFirst(e)`	`putFirst(e)`	`offerFirst(e, time, unit)`
Remove	`removeFirst()`	`pollFirst()`	`takeFirst()`	`pollFirst(time, unit)`
Examine	`getFirst()`	`peekFirst()`	*not applicable*	*not applicable*
Last Element (Tail)				
	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`addLast(e)`	`offerLast(e)`	`putLast(e)`	`offerLast(e, time, unit)`
Remove	`removeLast()`	`pollLast()`	`takeLast()`	`pollLast(time, unit)`
Examine	`getLast()`	`peekLast()`	*not applicable*	*not applicable*

如上图所示，LinkedBlockingDeque的基本操作可以分为四种类型：1.特殊情况，抛出异常；2.特殊情况，返回特殊值如null或者false；3.当线程不满足操作条件时，线程会被阻塞直至条件满足；4. 操作具有超时特性。 另外，LinkedBlockingDeque实现了BlockingDueue接口而LinkedBlockingQueue实现的是BlockingQueue，这两个接口的主要区别如下图所示（来源于java文档）：

Comparison of BlockingQueue and BlockingDeque methods

BlockingQueue Method	Equivalent BlockingDeque Method
Insert	
add(e)	addLast(e)
offer(e)	offerLast(e)
put(e)	putLast(e)
offer(e, time, unit)	offerLast(e, time, unit)
Remove	
remove()	removeFirst()
poll()	pollFirst()
take()	takeFirst()
poll(time, unit)	pollFirst(time, unit)
Examine	
element()	getFirst()
peek()	peekFirst()

从上图可以看出，两个接口的功能是可以等价使用的，比如BlockingQueue的add方法和BlockingDeque的addLast方法的功能是一样的。

（7）DelayQueue

DelayQueue是一个存放实现Delayed接口的数据的无界阻塞队列，只有当数据对象的延时时间达到时才能插入到队列进行存储。如果当前所有的数据都还没有达到创建时所指定的延时期，则队列没有队头，并且线程通过poll等方法获取数据元素则返回null。所谓数据延时期满时，则是通过Delayed接口的getDelay(TimeUnit.NANOSECONDS)来进行判定，如果该方法返回的是小于等于0则说明该数据元素的延时期已满。

（8）结果展示

```
public static void main(String[] args) throws InterruptedException {
    BlockingQueue<Integer> abq = new ArrayBlockingQueue<>(2);
    BlockingQueue<Integer> lbq = new LinkedBlockingDeque<>(2);
    BlockingQueue<Integer> pbq = new PriorityBlockingQueue<>(2);
    BlockingQueue<Integer> ltbq = new LinkedTransferQueue<>();
    BlockingQueue<Integer> sbq = new SynchronousQueue<>();
    queueTest(abq);
    queueTest(lbq);
    queueTest(pbq);
    queueTest(ltbq);
    queueTest(sbq);
```

```
}

public static void queueTest(BlockingQueue<Integer> bq) throws InterruptedException{
    System.out.printf("队列类型为: %s\n",bq.getClass());
    System.out.printf("队列offer(1): %s\n",bq.offer(1));
    System.out.printf("队列offer(2): %s\n",bq.offer(2));
    System.out.printf("队列offer(3): %s\n",bq.offer(3));
    System.out.printf("队列的size为: %s\n",bq.size());
    System.out.printf("队列take(): %s\n",bq.take());
    System.out.printf("队列的size为: %s\n",bq.size());
}
```

上面示例输出的结果为:

```
队列类型为: class java.util.concurrent.ArrayBlockingQueue
队列offer(1): true
队列offer(2): true
队列offer(3): false
队列的size为: 2
队列take(): 1
队列的size为: 1
队列类型为: class java.util.concurrent.LinkedBlockingDeque
队列offer(1): true
队列offer(2): true
队列offer(3): false
队列的size为: 2
队列take(): 1
队列的size为: 1
队列类型为: class java.util.concurrent.PriorityBlockingQueue
队列offer(1): true
队列offer(2): true
队列offer(3): true
队列的size为: 3
队列take(): 1
队列的size为: 2
队列类型为: class java.util.concurrent.LinkedTransferQueue
队列offer(1): true
队列offer(2): true
队列offer(3): true
队列的size为: 3
队列take(): 1
队列的size为: 2
队列类型为: class java.util.concurrent.SynchronousQueue
队列offer(1): false
队列offer(2): false
队列offer(3): false
队列的size为: 0
// 在SynchronousQueue进行take操作时阻塞了
```