- NETTY

精尽 Netty 源码解析 —— Util 之 MpscUnboundedArrayQueue

笔者先把 Netty 主要的内容写完，所以关于 MpscUnboundedArrayQueue 的分享，先放在后续的计划里。

> 老芳芳：其实是因为，自己想去研究下 Service Mesh ，所以先简单收个小尾。

当然，良心如我，还是为对这块感兴趣的胖友，先准备好了一篇不错的文章：

- HMILYYLIMH 《原理剖析（第 012 篇）Netty之无锁队列MpscUnboundedArrayQueue原理分析》

为避免可能 《原理剖析（第 012 篇）Netty之无锁队列MpscUnboundedArrayQueue原理分析》 被作者删除，笔者这里先复制一份作为备份。

666. 备份

一、大致介绍

```
1、了解过netty原理的童鞋，其实应该知道工作线程组的每个子线程都维护了一个任务队列；
2、细心的童鞋会发现netty的队列是重写了队列的实现方法，覆盖了父类中的LinkedBlockingQueue队列，但是如今却换成了J
3、那么问题就来了，现在的netty要用新的队列呢？难道是新的队列确实很高效么？
4、那么本章节就来和大家分享分析一下Netty新采用的队列之一MpscUnboundedArrayQueue，分析Netty的源码版本为：netty
```

二、回顾预习

2.1 构造队列

```
1、源码：
    // NioEventLoop.java
    @Override
    protected Queue<Runnable> newTaskQueue(int maxPendingTasks) {
        // This event loop never calls takeTask()
        // 由于默认是没有配置io.netty.eventLoop.maxPendingTasks属性值的，所以maxPendingTasks默认值为Intege
        // 那么最后配备的任务队列的大小也就自然使用无参构造队列方法
        return maxPendingTasks == Integer.MAX_VALUE ? PlatformDependent.<Runnable>newMpscQueue()
                                                    : PlatformDependent.<Runnable>newMpscQueue(maxPend
    }
```

```
// PlatformDependent.java
/**
 * Create a new {@link Queue} which is safe to use for multiple producers (different threads) and
 * consumer (one thread!).
 * @return A MPSC queue which may be unbounded.
 */
public static <T> Queue<T> newMpscQueue() {
    return Mpsc.newMpscQueue();
}

// Mpsc.java
static <T> Queue<T> newMpscQueue() {
    return USE_MPSC_CHUNKED_ARRAY_QUEUE ? new MpscUnboundedArrayQueue<T>(MPSC_CHUNK_SIZE)
                                        : new MpscUnboundedAtomicArrayQueue<T>(MPSC_CHUNK_SIZE);
}
```

2、通过源码回顾，想必大家已经隐约回忆起之前分析过这段代码，我们在构建工作线程管理组的时候，还需要实例化子线程数组

3、这段代码其实就是为了实现一个无锁方式的线程安全队列，总之一句话，效率相当相当的高；

2.2 何为JCTools?

1、JCTools是服务虚拟机并发开发的工具，提供一些JDK没有的并发数据结构辅助开发。

2、是一个聚合四种 SPSC/MPSC/SPMC/MPMC 数据变量的并发队列：

- SPSC：单个生产者对单个消费者（无等待、有界和无界都有实现）
- MPSC：多个生产者对单个消费者（无锁、有界和无界都有实现）
- SPMC：单生产者对多个消费者（无锁 有界）
- MPMC：多生产者对多个消费者（无锁、有界）

3、SPSC/MPSC 提供了一个在性能，分配和分配规则之间的平衡的关联数组队列；

2.3 常用重要的成员属性及方法

```
1、private volatile long producerLimit;
   // 数据链表所分配或者扩展后的容量值

2、protected long producerIndex;
   // 生产者指针，每添加一个数据，指针加2

3、protected long consumerIndex;
   // 消费者指针，每移除一个数据，指针加2

4、private static final int RETRY = 1; // 重新尝试，有可能是因为并发原因，CAS操作指针失败，所以需要重新尝试添
   private static final int QUEUE_FULL = 2; // 队列已满，直接返回false操作
   private static final int QUEUE_RESIZE = 3; // 需要扩容处理，扩容的后的容量值producerLimit一般都是mask的
   // 添加数据时，根据offerSlowPath返回的状态值来做各种处理

5、protected E[] producerBuffer;
   // 数据缓冲区，需要添加的数据放在此

6、protected long producerMask;
```

```
   // 生产者扩充容量值，一般producerMask与consumerMask是一致的，而且需要扩容的数值一般和此值一样

7、public boolean offer(final E e)
   // 添加元素

8、public E poll()
   // 移除元素
```

2.4 数据结构

1、如果chunkSize初始化大小为4，则最后显示的数据结构如下：
E1，E2，。。。，EN：表示具体的元素；
NBP：表示下一个缓冲区的指针，我采用的是英文的缩写(Next Buffer Pointer)；

而且你看着我是拆分开写的，其实每一个NBP指向的就是下面一组缓冲区；
Buffer1中的NBP其实就是Buffer2的指针引用；
Buffer2中的NBP其实就是Buffer3的指针引用；
以此类推。。。

```
+------+------+------+------+------+
|      |      |      |      |      |
|  E1  |  E2  |  E3  | JUMP |  NBP |    Buffer1
|      |      |      |      |      |
+------+------+------+------+------+

+------+------+------+------+------+
|      |      |      |      |      |
|  E5  |  E6  | JUMP |  E4  |  NBP |    Buffer2
|      |      |      |      |      |
+------+------+------+------+------+

+------+------+------+------+------+
|      |      |      |      |      |
|  E9  | JUMP |  E7  |  E8  |  NBP |    Buffer3
|      |      |      |      |      |
+------+------+------+------+------+

+------+------+------+------+------+
|      |      |      |      |      |
| JUMP |  E10 |  E11 |  E12 |  NBP |    Buffer4
|      |      |      |      |      |
+------+------+------+------+------+

+------+------+------+------+------+
|      |      |      |      |      |
|  E13 |  E14 |  E15 | JUMP |  NBP |    Buffer5
|      |      |      |      |      |
+------+------+------+------+------+
```

2、这个数据结构和我们通常所认知的链表是不是有点异样，其实大体还是雷同的，这种数据结构其实也是指针的单项指引罢了；

三、源码分析MpscUnboundedArrayQueue

3.1、MpscUnboundedArrayQueue(int)

1、源码：

```
// MpscUnboundedArrayQueue.java
public MpscUnboundedArrayQueue(int chunkSize)
{
    super(chunkSize); // 调用父类的含参构造方法
}

// BaseMpscLinkedArrayQueue.java
/**
 * @param initialCapacity the queue initial capacity. If chunk size is fixed this will be the chun
 *                        Must be 2 or more.
 */
public BaseMpscLinkedArrayQueue(final int initialCapacity)
{
    // 校验队列容量值，大小必须不小于2
    RangeUtil.checkGreaterThanOrEqual(initialCapacity, 2, "initialCapacity");

    // 通过传入的参数通过Pow2算法获取大于initialCapacity最近的一个2的n次方的值
    int p2capacity = Pow2.roundToPowerOfTwo(initialCapacity);
    // leave lower bit of mask clear
    long mask = (p2capacity - 1) << 1; // 通过p2capacity计算获得mask值，该值后续将用作扩容的值
    // need extra element to point at next array
    E[] buffer = allocate(p2capacity + 1); // 默认分配一个 p2capacity + 1 大小的数据缓冲区
    producerBuffer = buffer;
    producerMask = mask;
    consumerBuffer = buffer;
    consumerMask = mask;
    // 同时用mask作为初始化队列的Limit值，当生产者指针producerIndex超过该Limit值时就需要做扩容处理
    soProducerLimit(mask); // we know it's all empty to start with
}

// RangeUtil.java
public static int checkGreaterThanOrEqual(int n, int expected, String name)
{
    // 要求队列的容量值必须不小于 expected 值，这个 expected 值由上层决定，但是对 MpscUnboundedArrayQue
    // 那么就是说 MpscUnboundedArrayQueue 的值必须不小于 2；
    if (n < expected)
    {
        throw new IllegalArgumentException(name + ": " + n + " (expected: >= " + expected + ')');
    }

    return n;
}
```

2、通过调用父类的构造方法，分配了一个数据缓冲区，初始化容量大小，并且容量值不小于2，差不多就这样队列的实例化操作

3.2、offer(E)

1、源码：

```
// BaseMpscLinkedArrayQueue.java
```

```
@Override
public boolean offer(final E e)
{
    if (null == e) // 待添加的元素e不允许为空，否则抛空指针异常
    {
        throw new NullPointerException();
    }

    long mask;
    E[] buffer;
    long pIndex;

    while (true)
    {
        long producerLimit = lvProducerLimit(); // 获取当前数据Limit的阈值
        pIndex = lvProducerIndex(); // 获取当前生产者指针位置
        // lower bit is indicative of resize, if we see it we spin until it's cleared
        if ((pIndex & 1) == 1)
        {
            continue;
        }
        // pIndex is even (lower bit is 0) -> actual index is (pIndex >> 1)

        // mask/buffer may get changed by resizing -> only use for array access after successful C
        mask = this.producerMask;
        buffer = this.producerBuffer;
        // a successful CAS ties the ordering, lv(pIndex) - [mask/buffer] -> cas(pIndex)

        // assumption behind this optimization is that queue is almost always empty or near empty
        if (producerLimit <= pIndex) // 当阈值小于等于生产者指针位置时，则需要扩容，否则直接通过CAS操作
        {
            // 通过offerSlowPath返回状态值，来查看怎么来处理这个待添加的元素
            int result = offerSlowPath(mask, pIndex, producerLimit);
            switch (result)
            {
                case CONTINUE_TO_P_INDEX_CAS:
                    break;
                case RETRY: // 可能由于并发原因导致CAS失败，那么则再次重新尝试添加元素
                    continue;
                case QUEUE_FULL: // 队列已满，直接返回false操作
                    return false;
                case QUEUE_RESIZE: // 队列需要扩容操作
                    resize(mask, buffer, pIndex, e); // 对队列进行直接扩容操作
                    return true;
            }
        }

        // 能走到这里，则说明当前的生产者指针位置还没有超过阈值，因此直接通过CAS操作做加2处理
        if (casProducerIndex(pIndex, pIndex + 2))
        {
            break;
        }
    }
    // INDEX visible before ELEMENT
```

```
        // 获取计算需要添加元素的位置
        final long offset = modifiedCalcElementOffset(pIndex, mask);
        // 在buffer的offset位置添加e元素
        soElement(buffer, offset, e); // release element e
        return true;
    }

    // BaseMpscLinkedArrayQueueProducerFields.java
    @Override
    public final long lvProducerIndex()
    {
        // 通过Unsafe对象调用native方法，获取生产者指针位置
        return UNSAFE.getLongVolatile(this, P_INDEX_OFFSET);
    }

    // UnsafeRefArrayAccess.java
    /**
     * An ordered store(store + StoreStore barrier) of an element to a given offset
     *
     * @param buffer this.buffer
     * @param offset computed via {@link UnsafeRefArrayAccess#calcElementOffset}
     * @param e      an orderly kitty
     */
    public static <E> void soElement(E[] buffer, long offset, E e)
    {
        // 通过Unsafe对象调用native方法，将元素e设置到buffer缓冲区的offset位置
        UNSAFE.putOrderedObject(buffer, offset, e);
    }
```

2、此方法为添加新的元素对象，当pIndex指针超过阈值producerLimit时则扩容处理，否则直接通过CAS操作添加记录pIndex

3.3、offerSlowPath(long, long, long)

1、源码：

```
    // BaseMpscLinkedArrayQueue.java
    /**
     * We do not inline resize into this method because we do not resize on fill.
     */
    private int offerSlowPath(long mask, long pIndex, long producerLimit)
    {
        // 获取消费者指针
        final long cIndex = lvConsumerIndex();
        // 获取当前缓冲区的容量值，getCurrentBufferCapacity方法由子类MpscUnboundedArrayQueue实现，默认返回r
        long bufferCapacity = getCurrentBufferCapacity(mask);

        // 如果消费指针加上容量值如果超过了生产指针，那么则会尝试进行扩容处理
        if (cIndex + bufferCapacity > pIndex)
        {
            if (!casProducerLimit(producerLimit, cIndex + bufferCapacity))
            {
                // retry from top
                return RETRY;
            }
```

```
            else
            {
                // continue to pIndex CAS
                return CONTINUE_TO_P_INDEX_CAS;
            }
        }
        // full and cannot grow 子类MpscUnboundedArrayQueue默认返回Integer.MAX_VALUE值，所以不会进入此分支
        else if (availableInQueue(pIndex, cIndex) <= 0)
        {
            // offer should return false;
            return QUEUE_FULL;
        }
        // grab index for resize -> set lower bit 尝试扩容队列
        else if (casProducerIndex(pIndex, pIndex + 1))
        {
            // trigger a resize
            return QUEUE_RESIZE;
        }
        else
        {
            // failed resize attempt, retry from top
            return RETRY;
        }
    }

    // MpscUnboundedArrayQueue.java
    @Override
    protected long getCurrentBufferCapacity(long mask)
    {
        // 获取当前缓冲区的容量值
        return mask;
    }

    // BaseMpscLinkedArrayQueue.java
    final boolean casProducerLimit(long expect, long newValue)
    {
        // 通过CAS尝试对阈值进行修改扩容处理
        return UNSAFE.compareAndSwapLong(this, P_LIMIT_OFFSET, expect, newValue);
    }

    // MpscUnboundedArrayQueue.java
    @Override
    protected long availableInQueue(long pIndex, long cIndex)
    {
        // 获取可用容量值
        return Integer.MAX_VALUE;
    }

    // BaseMpscLinkedArrayQueueProducerFields.java
    final boolean casProducerIndex(long expect, long newValue)
    {
        // 通过CAS操作更新生产者指针
        return UNSAFE.compareAndSwapLong(this, P_INDEX_OFFSET, expect, newValue);
    }
```

2、该方法主要通过一系列的if...else判断，并结合子类MpscUnboundedArrayQueue的一些重写方法来判断针对该新添加的元

3.4、resize(long, E[], long, E)

1、源码：

```
// BaseMpscLinkedArrayQueue.java
private void resize(long oldMask, E[] oldBuffer, long pIndex, E e)
{
    // 获取oldBuffer的长度值
    int newBufferLength = getNextBufferSize(oldBuffer);
    // 重新创建新的缓冲区
    final E[] newBuffer = allocate(newBufferLength);

    producerBuffer = newBuffer; // 将新创建的缓冲区赋值到生产者缓冲区对象上
    final int newMask = (newBufferLength - 2) << 1;
    producerMask = newMask;

    // 根据oldMask获取偏移位置值
    final long offsetInOld = modifiedCalcElementOffset(pIndex, oldMask);
    // 根据newMask获取偏移位置值
    final long offsetInNew = modifiedCalcElementOffset(pIndex, newMask);

    // 将元素e设置到新的缓冲区newBuffer的offsetInNew位置处
    soElement(newBuffer, offsetInNew, e);// element in new array

    // 通过nextArrayOffset(oldMask)计算新的缓冲区将要放置旧的缓冲区的哪个位置
    // 将新的缓冲区newBuffer设置到旧的缓冲区oldBuffer的nextArrayOffset(oldMask)位置处
    // 主要是将oldBuffer中最后一个元素的位置指向新的缓冲区newBuffer
    // 这样就构成了一个单向链表指向的关系
    soElement(oldBuffer, nextArrayOffset(oldMask), newBuffer);// buffer linked

    // ASSERT code
    final long cIndex = lvConsumerIndex();
    final long availableInQueue = availableInQueue(pIndex, cIndex);
    RangeUtil.checkPositive(availableInQueue, "availableInQueue");

    // Invalidate racing CASs
    // We never set the limit beyond the bounds of a buffer
    // 重新扩容阈值，因为availableInQueue反正都是Integer.MAX_VALUE值，所以自然就取mask值啦
    // 因此针对MpscUnboundedArrayQueue来说，扩容的值其实就是mask的值的大小
    soProducerLimit(pIndex + Math.min(newMask, availableInQueue));

    // make resize visible to the other producers
    // 设置生产者指针加2处理
    soProducerIndex(pIndex + 2);

    // INDEX visible before ELEMENT, consistent with consumer expectation

    // make resize visible to consumer
    // 用一个空对象来衔接新老缓冲区，凡是在缓冲区中碰到JUMP对象的话，那么就得琢磨着准备着获取下一个缓冲区的
    soElement(oldBuffer, offsetInOld, JUMP);
}
```

```
// MpscUnboundedArrayQueue.java
@Override
protected int getNextBufferSize(E[] buffer)
{
    // 获取buffer缓冲区的长度
    return length(buffer);
}

// LinkedArrayQueueUtil.java
static int length(Object[] buf)
{
    // 直接通过length属性来获取数组的长度
    return buf.length;
}

// CircularArrayOffsetCalculator.java
@SuppressWarnings("unchecked")
public static <E> E[] allocate(int capacity)
{
    // 根据容量值创建数组
    return (E[]) new Object[capacity];
}
```

2、该方法主要完成新的元素的放置，同时也完成了扩容操作，采用单向链表指针关系，将原缓冲区和新创建的缓冲区衔接起来：

3.5、poll()

1、源码：

```
// BaseMpscLinkedArrayQueue.java
/**
 * {@inheritDoc}
 * <p>
 * This implementation is correct for single consumer thread use only.
 */
@SuppressWarnings("unchecked")
@Override
public E poll()
{
    final E[] buffer = consumerBuffer; // 获取缓冲区的数据
    final long index = consumerIndex;
    final long mask = consumerMask;

    // 根据消费指针与mask来获取当前需要从哪个位置开始来移除元素
    final long offset = modifiedCalcElementOffset(index, mask);
    // 从buffer缓冲区的offset位置获取元素内容
    Object e = lvElement(buffer, offset);// LoadLoad
    if (e == null) // 如果元素为null的话
    {
        // 则再探讨看看消费指针是不是和生产指针是不是相同
        if (index != lvProducerIndex())
        {
            // poll() == null iff queue is empty, null element is not strong enough indicator, so
```

```
            // check the producer index. If the queue is indeed not empty we spin until element is
            // visible.
            // 若不相同的话，则先尝试从buffer缓冲区的offset位置获取元素先，若获取元素为null则结束while处
            do
            {
                e = lvElement(buffer, offset);
            }
            while (e == null);
        }
        // 说明消费指针是不是和生产指针是相等的，那么则缓冲区的数据已经被消费完了，直接返回null即可
        else
        {
            return null;
        }
    }

    // 如果元素为JUMP空对象的话，那么意味着我们就得获取下一缓冲区进行读取数据了
    if (e == JUMP)
    {
        //
        final E[] nextBuffer = getNextBuffer(buffer, mask);
        //
        return newBufferPoll(nextBuffer, index);
    }

    // 能执行到这里，说明需要移除的元素既不是空的，也不是JUMP空对象，那么则就按照正常处理置空即可
    // 移除元素时，则将buffer缓冲区的offset位置的元素置为空即可
    soElement(buffer, offset, null); // release element null
    // 同时也通过CAS操作增加消费指针的关系，加2操作
    soConsumerIndex(index + 2); // release cIndex
    return (E) e;
}

// BaseMpscLinkedArrayQueueProducerFields.java
@Override
public final long lvProducerIndex()
{
    // 通过Unsafe对象调用native方法，获取当前生产者指针值
    return UNSAFE.getLongVolatile(this, P_INDEX_OFFSET);
}

// UnsafeRefArrayAccess.java
/**
 * A volatile load (load + LoadLoad barrier) of an element from a given offset.
 *
 * @param buffer this.buffer
 * @param offset computed via {@link UnsafeRefArrayAccess#calcElementOffset(long)}
 * @return the element at the offset
 */
@SuppressWarnings("unchecked")
public static <E> E lvElement(E[] buffer, long offset)
{
    // 通过Unsafe对象调用native方法，获取buffer缓冲区offset位置的元素
    return (E) UNSAFE.getObjectVolatile(buffer, offset);
```

```
    }

    // BaseMpscLinkedArrayQueue.java
    @SuppressWarnings("unchecked")
    private E[] getNextBuffer(final E[] buffer, final long mask)
    {
        // 获取下一个缓冲区的偏移位置值
        final long offset = nextArrayOffset(mask);
        // 从buffer缓冲区的offset位置获取下一个缓冲区数组
        final E[] nextBuffer = (E[]) lvElement(buffer, offset);
        // 获取出来后，同时将buffer缓冲区的offset位置置为空，代表指针已经被取出，原来位置没用了，清空即可
        soElement(buffer, offset, null);
        return nextBuffer;
    }

    // BaseMpscLinkedArrayQueue.java
    private E newBufferPoll(E[] nextBuffer, long index)
    {
        // 从下一个新的缓冲区中找到需要移除数据的指针位置
        final long offset = newBufferAndOffset(nextBuffer, index);
        // 从newBuffer新的缓冲区中offset位置取出元素
        final E n = lvElement(nextBuffer, offset);// LoadLoad
        if (n == null) // 若取出的元素为空，则直接抛出异常
        {
            throw new IllegalStateException("new buffer must have at least one element");
        }
        // 如果取出的元素不为空，那么先将这个元素原先的位置内容先清空掉
        soElement(nextBuffer, offset, null);// StoreStore
        // 然后通过Unsafe对象调用native方法，修改消费指针的数值偏移加2处理
        soConsumerIndex(index + 2);
        return n;
    }
```

2、该方法主要阐述了该队列是如何的移除数据的；取出的数据如果为JUMP空对象的话，那么则准备从下一个缓冲区获取数据元素

3.6、size()

1、源码：

```
    // BaseMpscLinkedArrayQueue.java
    @Override
    public final int size()
    {
        // NOTE: because indices are on even numbers we cannot use the size util.

        /*
         * It is possible for a thread to be interrupted or reschedule between the read of the produce
         * consumer indices, therefore protection is required to ensure size is within valid range. In
         * event of concurrent polls/offers to this method the size is OVER estimated as we read consu
         * index BEFORE the producer index.
         */
        long after = lvConsumerIndex(); // 获取消费指针
        long size;
        while (true) // 为了防止在获取大小的时候指针发生变化，那么则死循环自旋方式获取大小数值
```

```
        {
            final long before = after;
            final long currentProducerIndex = lvProducerIndex(); // 获取生产者指针
            after = lvConsumerIndex(); // 获取消费指针

            // 如果后获取的消费指针after和之前获取的消费指针before相等的话，那么说明此刻还没有指针变化
            if (before == after)
            {
                // 那么则直接通过生产指针直接减去消费指针，然后向偏移一位，即除以2，得出最后size大小
                size = ((currentProducerIndex - after) >> 1);

                // 计算完了之后则直接break中断处理
                break;
            }

            // 若消费指针前后不一致，那么可以说是由于并发原因导致了指针发生了变化；
            // 那么则进行下一次循环继续获取最新的指针值再次进行判断
        }
        // Long overflow is impossible, so size is always positive. Integer overflow is possible for t
        // indexed queues.
        if (size > Integer.MAX_VALUE)
        {
            return Integer.MAX_VALUE;
        }
        else
        {
            return (int) size;
        }
    }
```

2、获取缓冲区数据大小其实很简单，就是拿着生产指针减去消费指针，但是为了防止并发操作计算错，才用了死循环的方式计算

3.7、isEmpty()

1、源码：

```
    // BaseMpscLinkedArrayQueue.java
    @Override
    public final boolean isEmpty()
    {
        // Order matters!
        // Loading consumer before producer allows for producer increments after consumer index is rea
        // This ensures this method is conservative in it's estimate. Note that as this is an MPMC the
        // nothing we can do to make this an exact method.
        // 这个就简单了，直接判断消费指针和生产指针是不是相等就知道了
        return (this.lvConsumerIndex() == this.lvProducerIndex());
    }
```

2、通过前面我们已经知道了，添加数据的话生产指针在不停的累加操作，而做移除数据的时候消费指针也在不停的累加操作；

3、那么这种指针总会有一天会碰面的吧，碰面的那个时候则是数据已经空空如也的时刻；

四、性能测试

1、测试Demo：

```
/**
 * 比较队列的消耗情况。
 *
 * @author hmilyylimh
 * ^_^
 * @version 0.0.1
 * ^_^
 * @date 2018/3/30
 */
public class CompareQueueCosts {

    /** 生产者数量 */
    private static int COUNT_OF_PRODUCER = 2;

    /** 消费者数量 */
    private static final int COUNT_OF_CONSUMER = 1;

    /** 准备添加的任务数量值 */
    private static final int COUNT_OF_TASK = 1 << 20;

    /** 线程池对象 */
    private static ExecutorService executorService;

    public static void main(String[] args) throws Exception {

        for (int j = 1; j < 7; j++) {
            COUNT_OF_PRODUCER = (int) Math.pow(2, j);
            executorService = Executors.newFixedThreadPool(COUNT_OF_PRODUCER * 2);

            int basePow = 8;
            int capacity = 0;
            for (int i = 1; i <= 3; i++) {
                capacity = 1 << (basePow + i);
                System.out.print("Producers: " + COUNT_OF_PRODUCER + "\t\t");
                System.out.print("Consumers: " + COUNT_OF_CONSUMER + "\t\t");
                System.out.print("Capacity: " + capacity + "\t\t");
                System.out.print("LinkedBlockingQueue: " + testQueue(new LinkedBlockingQueue<Integer>(
                // System.out.print("ArrayList: " + testQueue(new ArrayList<Integer>(capacity), COUNT_(
                // System.out.print("LinkedList: " + testQueue(new LinkedList<Integer>(), COUNT_OF_TAS
                System.out.print("MpscUnboundedArrayQueue: " + testQueue(new MpscUnboundedArrayQueue<I
                System.out.print("MpscChunkedArrayQueue: " + testQueue(new MpscChunkedArrayQueue<Integ
                System.out.println();
            }
            System.out.println();

            executorService.shutdown();
        }
    }

    private static Double testQueue(final Collection<Integer> queue, final int taskCount) throws Excep
        CompletionService<Long> completionService = new ExecutorCompletionService<Long>(executorServic
```

```
        long start = System.currentTimeMillis();
        for (int i = 0; i < COUNT_OF_PRODUCER; i++) {
            executorService.submit(new Producer(queue, taskCount / COUNT_OF_PRODUCER));
        }
        for (int i = 0; i < COUNT_OF_CONSUMER; i++) {
            completionService.submit((new Consumer(queue, taskCount / COUNT_OF_CONSUMER)));
        }

        for (int i = 0; i < COUNT_OF_CONSUMER; i++) {
            completionService.take().get();
        }

        long end = System.currentTimeMillis();
        return Double.parseDouble("" + (end - start)) / 1000;
    }

    private static class Producer implements Runnable {
        private Collection<Integer> queue;
        private int taskCount;

        public Producer(Collection<Integer> queue, int taskCount) {
            this.queue = queue;
            this.taskCount = taskCount;
        }

        @Override
        public void run() {
            // Queue队列
            if (this.queue instanceof Queue) {
                Queue<Integer> tempQueue = (Queue<Integer>) this.queue;
                while (this.taskCount > 0) {
                    if (tempQueue.offer(this.taskCount)) {
                        this.taskCount--;
                    } else {
                        // System.out.println("Producer offer failed.");
                    }
                }
            }
            // List列表
            else if (this.queue instanceof List) {
                List<Integer> tempList = (List<Integer>) this.queue;
                while (this.taskCount > 0) {
                    if (tempList.add(this.taskCount)) {
                        this.taskCount--;
                    } else {
                        // System.out.println("Producer offer failed.");
                    }
                }
            }
        }
    }

    private static class Consumer implements Callable<Long> {
```

```
        private Collection<Integer> queue;
        private int taskCount;

        public Consumer(Collection<Integer> queue, int taskCount) {
            this.queue = queue;
            this.taskCount = taskCount;
        }

        @Override
        public Long call() {
            // Queue队列
            if (this.queue instanceof Queue) {
                Queue<Integer> tempQueue = (Queue<Integer>) this.queue;
                while (this.taskCount > 0) {
                    if ((tempQueue.poll()) != null) {
                        this.taskCount--;
                    }
                }
            }
            // List列表
            else if (this.queue instanceof List) {
                List<Integer> tempList = (List<Integer>) this.queue;
                while (this.taskCount > 0) {
                    if (!tempList.isEmpty() && (tempList.remove(0)) != null) {
                        this.taskCount--;
                    }
                }
            }
            return 0L;
        }
    }
}
```

2、指标结果：

```
Producers: 2        Consumers: 1        Capacity: 512       LinkedBlockingQueue: 1.399s     MpscUnboun
Producers: 2        Consumers: 1        Capacity: 1024      LinkedBlockingQueue: 1.462s     MpscUnboun
Producers: 2        Consumers: 1        Capacity: 2048      LinkedBlockingQueue: 0.281s     MpscUnboun

Producers: 4        Consumers: 1        Capacity: 512       LinkedBlockingQueue: 0.681s     MpscUnboun
Producers: 4        Consumers: 1        Capacity: 1024      LinkedBlockingQueue: 0.405s     MpscUnboun
Producers: 4        Consumers: 1        Capacity: 2048      LinkedBlockingQueue: 0.248s     MpscUnboun

Producers: 8        Consumers: 1        Capacity: 512       LinkedBlockingQueue: 1.523s     MpscUnboun
Producers: 8        Consumers: 1        Capacity: 1024      LinkedBlockingQueue: 0.668s     MpscUnboun
Producers: 8        Consumers: 1        Capacity: 2048      LinkedBlockingQueue: 0.555s     MpscUnboun

Producers: 16       Consumers: 1        Capacity: 512       LinkedBlockingQueue: 2.676s     MpscUnboun
Producers: 16       Consumers: 1        Capacity: 1024      LinkedBlockingQueue: 2.135s     MpscUnboun
Producers: 16       Consumers: 1        Capacity: 2048      LinkedBlockingQueue: 0.944s     MpscUnboun

Producers: 32       Consumers: 1        Capacity: 512       LinkedBlockingQueue: 6.647s     MpscUnboun
Producers: 32       Consumers: 1        Capacity: 1024      LinkedBlockingQueue: 3.893s     MpscUnboun
Producers: 32       Consumers: 1        Capacity: 2048      LinkedBlockingQueue: 2.019s     MpscUnboun
```

```
Producers: 64       Consumers: 1       Capacity: 512       LinkedBlockingQueue: 26.59s     MpscUnboun
Producers: 64       Consumers: 1       Capacity: 1024      LinkedBlockingQueue: 22.566s    MpscUnboun
Producers: 64       Consumers: 1       Capacity: 2048      LinkedBlockingQueue: 1.719s     MpscUnboun
```

3、结果分析(一)：
通过结果打印耗时可以明显看到MpscUnboundedArrayQueue耗时几乎大多数都是不超过0.1s的，这添加、删除的操作效率不是

4、结果分析(二)：
CompareQueueCosts代码里面我将ArrayList、LinkedList注释掉了，那是因为队列数量太大，List的操作太慢，效率低下，

五、总结

1、通过底层无锁的Unsafe操作方式实现了多生产者同时访问队列的线程安全模型；

2、由于使用锁会造成的线程切换，特别消耗资源，因此使用无锁而是采用CAS的操作方式，虽然会在一定程度上造成CPU使用率

3、队列的数据结构是一种单向链表式的结构，通过生产、消费指针来标识添加、移除元素的指针位置，缓冲区与缓冲区之间通过