

# 超高性能无锁队列 Mpsc Queue

注：本文从对超高性能无锁队列，做了一个系统化、由浅入深的穿透式介绍，帮助大家彻底掌握这个高性能的算法。

另外

本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的 PDF 文件，请从下面的链接获取：[语雀](#) 或者 [码云](#)



在大规模流量的高并发系统中，需要一些比链表结构性能更好的，基于数组 + CAS 操作实现的无锁安全队列，在行业中被广受认可的王者队列，是以下两个高性能无锁队列：

- Disruptor 环形队列
- JCTools MSPC 队列

Disruptor 环形队列已经在老的文章中，进行过非常详细的介绍。接下来，从架构、源码的角度，介绍一下 JCTools MSPC 队列。

这里解释一下 MSPC 的含义，如下：

- M: Multiple, 多个的。
- S: Single, 单个的。
- P: Producer, 生产者。
- C: Consumer, 消费者。

因此 MpscQueue 适用于多生产者，单消费者的高性能无锁队列。多生产者单消费者的使用场景，最佳的案例有：

- Netty Reactor 线程中任务队列 taskQueue 必须满足多个生产者可以同时提交任务。
- Caffeine 中多个生产者业务线程，去进行缓存写入操作，而只有单个的维护线程，基于 W-TinyLRU 进行访问记录的维护

所以 JCTools 提供的 Mpsc Queue 非常适合：

- Netty 异步任务场景，
- Caffeine 的写入操作访问记录维护场景。

在介绍 Mpsc Queue 之前，我们先回顾下 JDK 原生队列的工作原理。

## JDK 内置并发队列

JDK 内置并发队列按照实现方式可以分为阻塞队列和非阻塞队列两种类型，阻塞队列是基于锁实现的，非阻塞队列是基于 CAS 操作实现的。

JDK 中包含多种阻塞和非阻塞的队列实现，如下图所示。

队列是一种 FIFO（先进先出）的数据结构，JDK 中定义了 `java.util.Queue` 的队列接口，与 `List`、`Set` 接口类似，`java.util.Queue` 也继承于 `Collection` 集合接口。

此外，JDK 还提供了一种双端队列接口 `java.util.Deque`，我们最常用的 `LinkedList` 就是实现了 `Deque` 接口。

下面我们简单说说上图中的每个队列的特点，并给出一些对比和总结。

## 阻塞队列

阻塞队列在队列为空或者队列满时，都会发生阻塞。阻塞队列自身是线程安全的，使用者无需关心线程安全问题，降低了多线程开发难度。

阻塞队列主要分为以下几种：

- **ArrayBlockingQueue:**

最基础且开发中最常用的阻塞队列，底层采用数组实现的有界队列，初始化需要指定队列的容量。`ArrayBlockingQueue` 是如何保证线程安全的呢？

它内部是使用了一个重入锁 `ReentrantLock`，并搭配 `notEmpty`、`notFull` 两个条件变量 `Condition` 来控制并发访问。

从队列读取数据时，如果队列为空，那么会阻塞等待，直到队列有数据了才会被唤醒。

如果队列已经满了，也同样会进入阻塞状态，直到队列有空闲才会被唤醒。

- **LinkedBlockingQueue:**

内部采用的数据结构是链表，队列的长度可以有界或者无界的，初始化不需要指定队列长度，默认是 `Integer.MAX_VALUE`。

`LinkedBlockingQueue` 内部使用了 `takeLock`、`putLock` 两个重入锁 `ReentrantLock`，以及 `notEmpty`、`notFull` 两个条件变量 `Condition` 来控制并发访问。

采用读锁和写锁的好处是可以避免读写时相互竞争锁的现象，所以相比于 `ArrayBlockingQueue`，`LinkedBlockingQueue` 的性能要更好。

- **PriorityBlockingQueue:**

采用最小堆实现的优先级队列，队列中的元素按照优先级进行排列，每次出队都是返回优先级最高的元素。`PriorityBlockingQueue` 内部是使用了一个 `ReentrantLock` 以及一个条件变量 `Condition` `notEmpty` 来控制并发访问，

因为 `PriorityBlockingQueue` 是无界队列，所以不需要 `notFull`，每次 `put` 都不会发生阻塞。

`PriorityBlockingQueue` 底层的最小堆是采用数组实现的，当元素个数大于等于最大容量时会触发扩容，

在扩容时会先释放锁，保证其他元素可以正常出队，然后使用 `CAS` 操作确保只有一个线程可以执行扩容逻辑。

- **DelayQueue,**

一种支持延迟获取元素的阻塞队列，常用于缓存、定时任务调度等场景。

`DelayQueue` 内部是采用优先级队列 `PriorityQueue` 存储对象。

`DelayQueue` 中的每个对象都必须实现 `Delayed` 接口，并重写 `compareTo` 和 `getDelay` 方法。向队列中存放元素的时候必须指定延迟时间，只有延迟时间已满的元素才能从队列中取出。

- **SynchronizedQueue,**

又称无缓冲队列。

比较特别的是 `SynchronizedQueue` 内部不会存储元素。与 `ArrayBlockingQueue`、`LinkedBlockingQueue` 不同，`SynchronizedQueue` 直接使用 `CAS` 操作控制线程的安全访问。

其中 `put` 和 `take` 操作都是阻塞的，每一个 `put` 操作都必须阻塞等待一个 `take` 操作，反之亦然。

所以 `SynchronizedQueue` 可以理解为生产者和消费者配对的场景，双方必须互相等待，直至配对成功。

在 JDK 的线程池 `Executors.newCachedThreadPool` 中就存在 `SynchronousQueue` 的运用，对于新提交的任务，如果有空闲线程，将重复利用空闲线程处理任务，否则将新建线程进行处理。

- **LinkedTransferQueue**

一种特殊的无界阻塞队列，可以看作 `LinkedBlockingQueues`、`SynchronousQueue`（公平模式）、`ConcurrentLinkedQueue` 的合体。

与 `SynchronousQueue` 不同的是，`LinkedTransferQueue` 内部可以存储实际的数据，当执行 `put` 操作时，如果有等待线程，那么直接将数据交给对方，否则放入队列中。与 `LinkedBlockingQueues` 相比，`LinkedTransferQueue` 使用 CAS 无锁操作进一步提升了性能。

## 非阻塞队列

说完阻塞队列，我们再来看下非阻塞队列。

非阻塞队列不需要通过加锁的方式对线程阻塞，并发性能更好。

JDK 中常用的非阻塞队列有以下几种：

- **ConcurrentLinkedQueue,**

它是一个采用双向链表实现的无界并发非阻塞队列，它属于 `LinkedList` 的安全版本。`ConcurrentLinkedQueue` 内部采用 CAS 操作保证线程安全，这是非阻塞队列实现的基础，相比 `ArrayBlockingQueue`、`LinkedBlockingQueue` 具备较高的性能。

- **ConcurrentLinkedDeque,**

也是一种采用双向链表结构的无界并发非阻塞队列。

与 `ConcurrentLinkedQueue` 不同的是，`ConcurrentLinkedDeque` 属于双端队列，

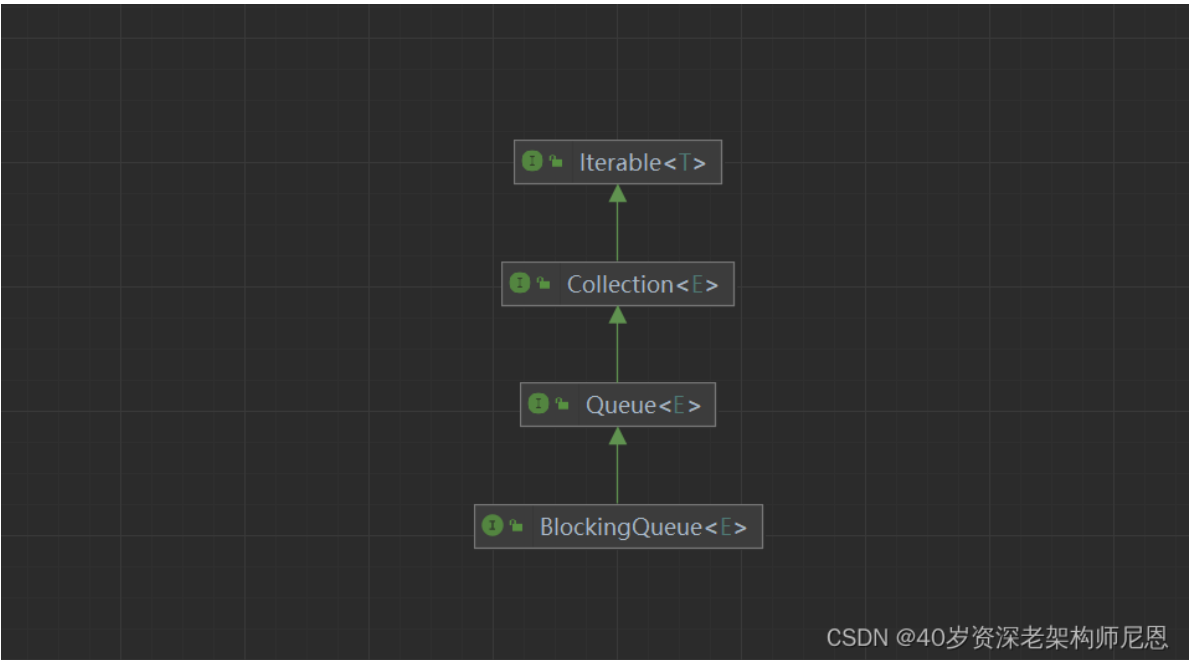
它同时支持 FIFO 和 FILO 两种模式，可以从队列的头部插入和删除数据，也可以从队列尾部插入和删除数据，适用于多生产者和多消费者的场景。

## BlockingQueue 阻塞队列超级接口

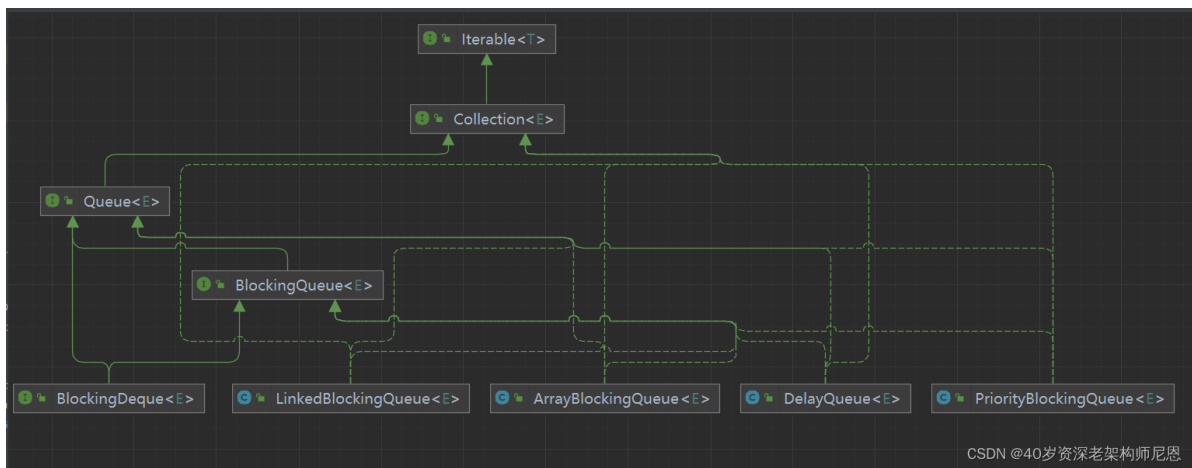
至此，常见的队列类型我们已经介绍完了。我们在平时开发中使用频率最高的是 `BlockingQueue`。

实现一个阻塞队列需要具备哪些基本功能呢？

下面看 `BlockingQueue` 的接口继承关系，如下图所示。
















































下面看 `BlockingQueue` 的接口的各种实现子类，如下图所示。



`BlockingQueue` 实现子类太多，图里放不下，还请大家通过 IDEA 工具，自行查看。

下面看 `BlockingQueue` 的接口的 抽象方法，如下图所示。

## BlockingDeque<E>

  remove(Object)	boolean
  push(E)	void
  size()	int
  offer(E)	boolean
  offerLast(E, long, TimeUnit)	boolean
  addLast(E)	void
  pollFirst(long, TimeUnit)	E?
  removeLastOccurrence (Object)	boolean
  offer(E, long, TimeUnit)	boolean
  offerFirst(E)	boolean
  putFirst(E)	void
  add(E)	boolean
  element()	E
  iterator()	Iterator <E>
  put(E)	void
  removeFirstOccurrence (Object)	boolean
  peek()	E
  pollLast(long, TimeUnit)	E?
  takeLast()	E
  addFirst(E)	void
  takeFirst()	E
  remove()	E
  contains(Object)	boolean
  putLast(E)	void
  take()	E
  offerLast(E)	boolean
  poll(long, TimeUnit)	E?
  poll()	E
  offerFirst(E, long, TimeUnit)	boolean

我们可以通过下面一张表格，对上述 BlockingQueue 接口的具体行为进行归类。

以下表格，来自于《Java 高并发核心编程 卷 2 加强版》

3. 获取元素类方法

- 1) `element()`: 获取但不移除此队列的队首元素，没有元素则抛出异常。
- 2) `peek()`: 获取但不移除此队列的队首元素；若队列为空，则返回`null`。

阻塞队列对元素的增删查操作主要就是上述的三类方法，这里对这三类方法的特征进行总结，具体如表7-1所示。

表 7-1 阻塞队列三类方法的特征

	抛出异常	特 殊 值	阻 塞	限时阻塞
添 加	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e, time, unit)</code>
删 除	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time, unit)</code>
获取元素	<code>element()</code>	<code>peek()</code>	不可用	不可用

对表7-1中的4个特征说明如下：

CSDN @40岁资深老架构师尼恩

# Java 内置队列的问题

我们先来看一看常用的线程安全的内置队列有什么问题。Java 的内置队列如下表所示。

队列	有界性	锁	数据结构
<code>ArrayBlockingQueue</code>	<code>bounded</code>	加锁	<code>arraylist</code>
<code>LinkedBlockingQueue</code>	<code>optionally-bounded</code>	加锁	<code>linkedList</code>
<code>ConcurrentLinkedQueue</code>	<code>unbounded</code>	无锁	<code>linkedList</code>
<code>LinkedTransferQueue</code>	<code>unbounded</code>	无锁	<code>linkedList</code>
<code>PriorityBlockingQueue</code>	<code>unbounded</code>	加锁	<code>heap</code>
<code>DelayQueue</code>	<code>unbounded</code>	加锁	<code>heap</code>

队列的底层一般分成三种：数组、链表和堆。

其中，堆一般情况下是为了实现带有优先级特性的队列，暂时不做介绍，后面结合堆的结构，做专题介绍，这个也是一个  $O(\log n)$  的性能比较高的结构。

从数组和链表两种数据结构来看，两类结构如下：

- 基于数组线程安全的队列，比较典型的是 `ArrayBlockingQueue`，它主要通过加锁的方式来保证线程安全；
- 基于链表的线程安全队列分成 `LinkedBlockingQueue` 和 `ConcurrentLinkedQueue` 两大类，前者也通过锁的方式来实现线程安全，而后者是无锁结构，通过原子变量 `compare and swap`（以下简称“CAS”）这种无锁方式来实现的。

`LinkedTransferQueue` 和 `ConcurrentLinkedQueue` 一样，也是无锁结构，通过原子变量 `compare and swap`（以下简称“CAS”）这种不加锁的方式来实现的，性能还是比较高的。

那么 `LinkedTransferQueue` 和 `ConcurrentLinkedQueue` 的问题是啥呢？

链表结构的核心的问题是：

链接的节点，在高并发的场景下，存在者大量的节点创建 / GC 回收的压力，这会导致 STW 的产生，容易出现业务卡顿。

另外，LinkedTransferQueue 对和 ConcurrentLinkedQueue 对 volatile 类型的变量进行 CAS 操作，存在伪共享问题，

总之，在高并发的场景下，链表结构没有数组结构，性能那么优越。

而 JDK 内置的基于数据结构的 ArrayBlockingQueue 队列，有一个核心问题：

**有锁结构，高并发场景，性能没有无锁结构，那么高**

另外，最好是基于数组结构的环形队列模式，能最大的限度的复用内存空间，减少内存分配和 GC 的压力。

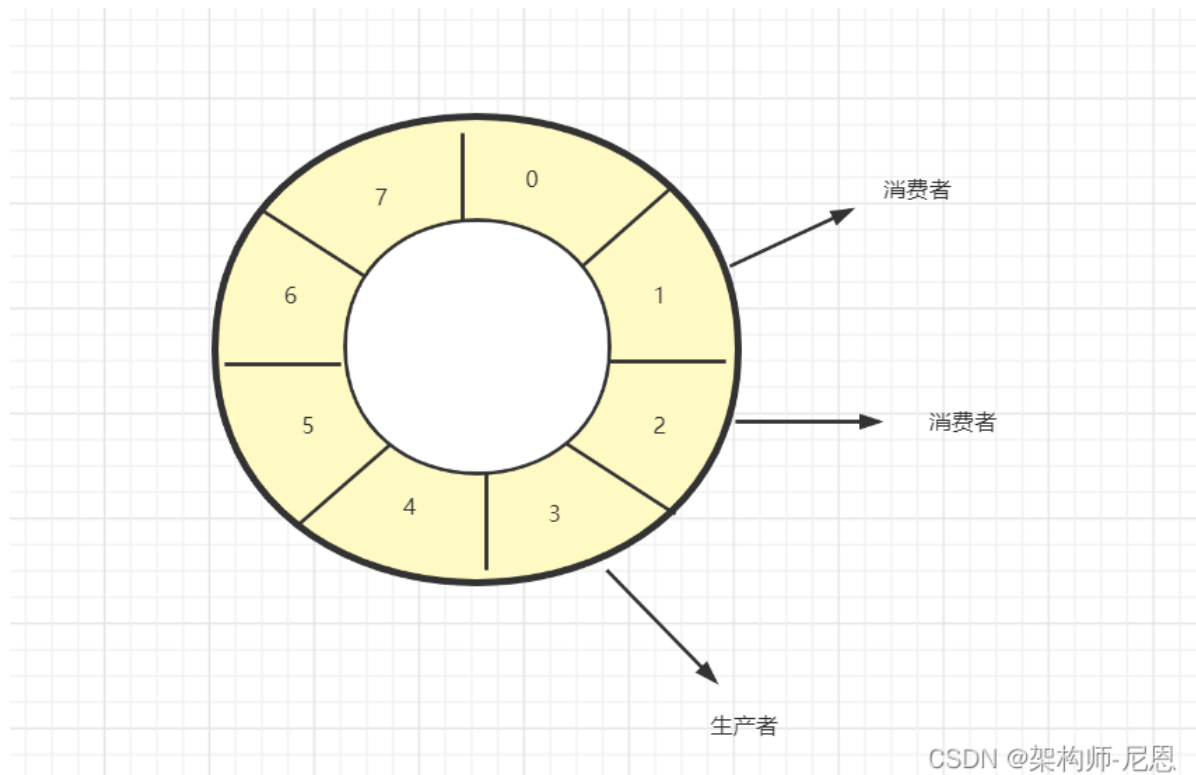
## 高性能的数组 + CAS 无锁队列解决访问

在大规模流量的高并发系统中，需要一个数组 + CAS 操作实现的无锁安全队列，有没有成熟的解决方案呢？

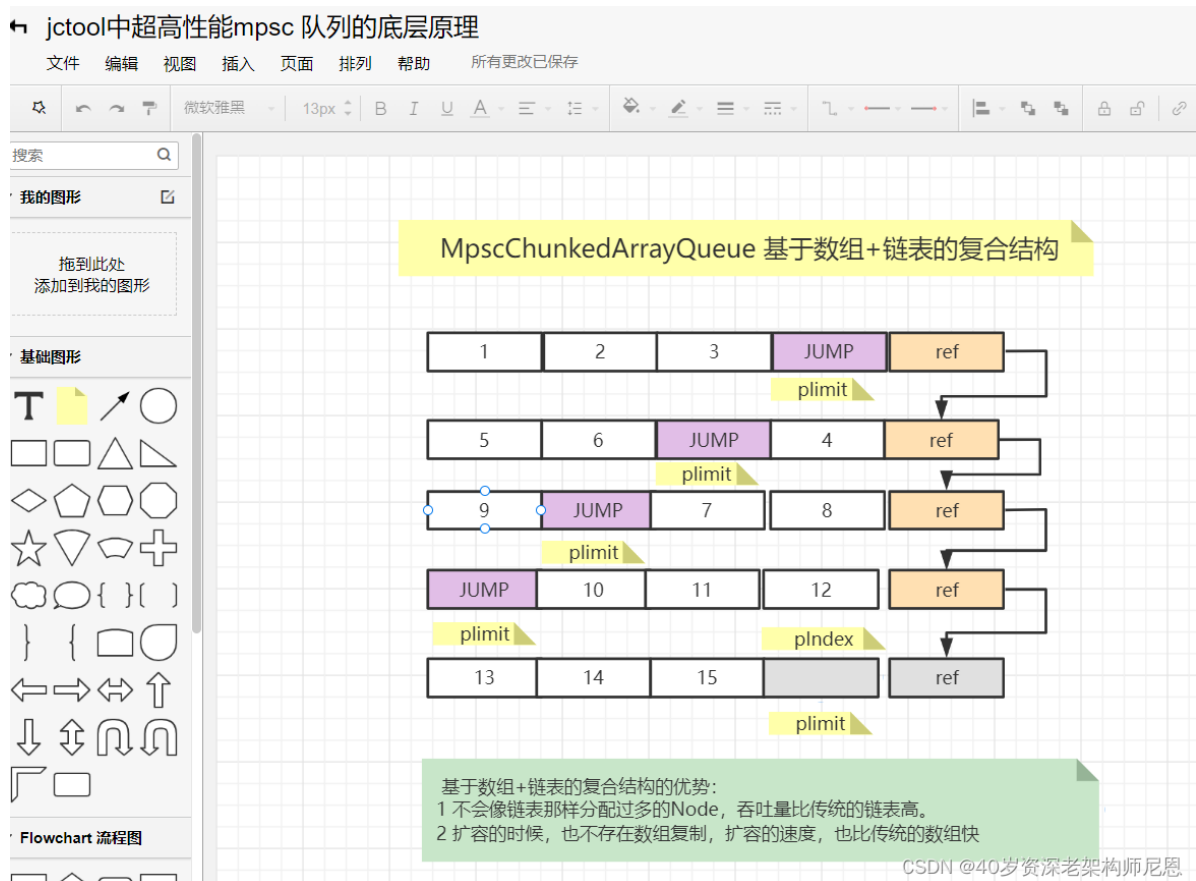
有两个高性能无锁队列：

- Disruptor
- JCTools

Disruptor 采用的是环形结构，进行无锁队列的设计



JCTools 采用的是 数组 + 链表的复合结构，进行无锁队列的设计



## Disruptor 如何实现高性能？

使用 Disruptor，主要用于对性能要求高、延迟低的场景，它通过“榨干”机器的性能来换取处理的高性能。

Disruptor 实现高性能主要体现了去掉了锁，采用 CAS 算法，同时内部通过环形队列实现有界队列。

- 环形数据结构数组元素不会被回收，避免频繁的 GC，所以，为了避免垃圾回收，采用数组而非链表。

同时，数组对处理器的缓存机制更加友好。

预分配数组空间

- 无锁设计

采用 CAS 无锁方式，保证线程的安全性

每个生产者或者消费者线程，会先申请可以操作的元素在数组中的位置，申请到之后，直接在该位置写入或者读取数据。

整个过程通过原子变量 CAS，保证操作的线程安全。

为啥 cas 性能高，请参见《java 高并发核心编程卷 2》

- 元素位置属性进行 cacheline 填充：

通过添加额外的无用信息，避免伪共享问题

性能提升 5 倍多

- 元素位置属性进行 volatile 变量延迟写入（有序写入）：

性能提升 10 倍多

- 位运算定位元素在数组当中的位置

数组长度  $2^n$ ，通过位运算，加快定位的速度。

下标采取递增的形式。



不用担心 index 溢出的问题。index 是 long 类型，即使 100 万 QPS 的处理速度，也需要 30 万年才能用完。

Disruptor 已经开源，详细可查阅 Github 地址 <https://github.com/LMAX-Exchange/disruptor>。

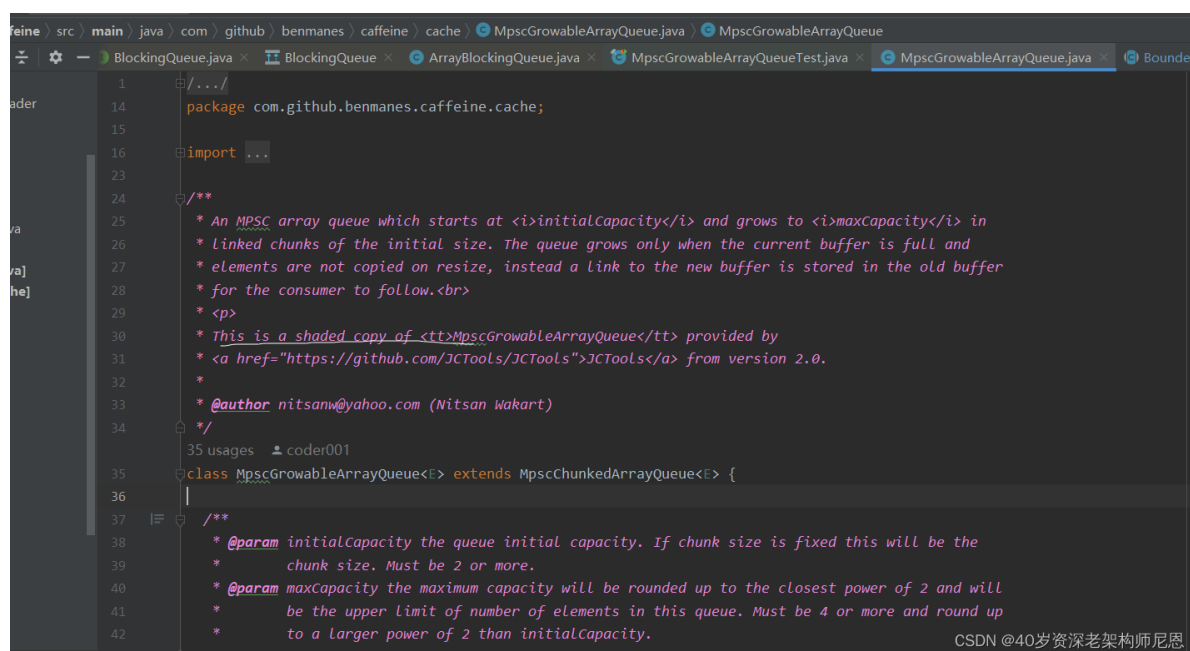
## JCTools 如何实现高性能？

JCTools 也是一个开源项目，Github 地址为 <https://github.com/JCTools/JCTools>。

JCTools 是适用于 JVM 并发开发的工具，主要提供了一些 JDK 缺失的并发数据结构，例如非阻塞 Map、非阻塞 Queue 等。其中非阻塞队列可以分为四种类型，可以根据不同的场景选择使用。

- Spsc 单生产者单消费者；
- Mpsc 多生产者单消费者；
- SPMC 单生产者多消费者；
- MPMC 多生产者多消费者。

Netty 中直接引入了 JCTools 的 Mpsc Queue，Caffeine 中引入了 JCTools 的 Mpsc Queue，复制了其代码，然后简单改了下。



```
1  package com.github.benmanes.caffeine.cache;
2
3  import ...
4
5  /**
6   * An MPSC array queue which starts at <i>initialCapacity</i> and grows to <i>maxCapacity</i> in
7   * linked chunks of the initial size. The queue grows only when the current buffer is full and
8   * elements are not copied on resize, instead a link to the new buffer is stored in the old buffer
9   * for the consumer to follow.
10  *
11  * This is a shaded copy of <code>MpscGrowableArrayQueue</code> provided by
12  * <a href="https://github.com/JCTools/JCTools">JCTools</a> from version 2.0.
13  *
14  * @author nitsanw@yahoo.com (Nitsan Wakart)
15  */
16
17  35 usages  coder001
18
19  class MpscGrowableArrayQueue<E> extends MpscChunkedArrayQueue<E> {
20
21  /**
22   * @param initialCapacity the queue initial capacity. If chunk size is fixed this will be the
23   * chunk size. Must be 2 or more.
24   * @param maxCapacity the maximum capacity will be rounded up to the closest power of 2 and will
25   * be the upper limit of number of elements in this queue. Must be 4 or more and round up
26   * to a larger power of 2 than initialCapacity.
27   */
28  }
```

相比于 JDK 原生的并发队列，Mpsc Queue 又有什么过人之处呢？

## Mpsc Queue 基础知识

Mpsc 的全称是 Multi Producer Single Consumer，多生产者单消费者。

Mpsc Queue 可以保证多个生产者同时访问队列是线程安全的，而且同一时刻只允许一个消费者从队列中读取数据。

多生产者单消费者的使用场景，最佳的案例有：

- Netty Reactor 线程中任务队列 taskQueue 必须满足多个生产者可以同时提交任务。
- Caffeine 中多个生产者业务线程，去进行缓存写入操作，而只有单个的维护线程，基于 W-TinyLRU 进行访问记录的维护

所以 JCTools 提供的 Mpsc Queue 非常适合：

- Netty 异步任务场景，
- Caffeine 的写入操作访问记录维护场景。

Mpsc Queue 有多种的实现类，例如 MpscArrayQueue、MpscUnboundedArrayQueue、MpscChunkedArrayQueue 等。

首先我们看下 MpscArrayQueue 的继承关系



除了顶层 JDK 原生的 AbstractCollection、AbstractQueue 接口之外，MpscArrayQueue 还继承了很多类似于 MpscXxxPad 以及 MpscXxxField 的类。

## 高性能组件的 Cache Line Padding 缓存行填充 技术

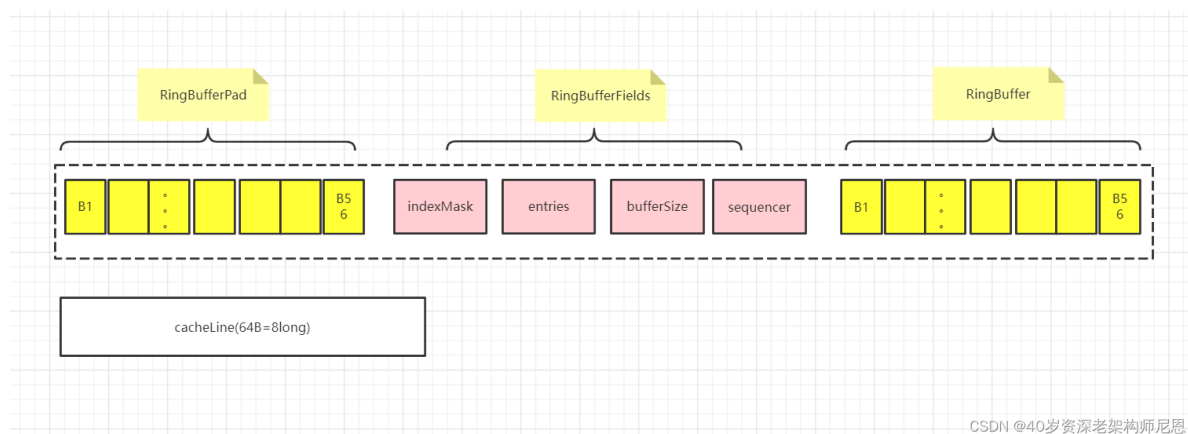
我们可以发现一个很有意思的规律，JCTool 每个有包含属性的类后面都会被 MpscXxxPad 类隔开。

MpscXxxPad 到底起到什么作用呢？

首先，回顾一下 Disruptor RingBuffer 的缓存行填充

### Disruptor RingBuffer 的缓存行填充

Disruptor RingBuffer（环形缓冲区）定义了 RingBufferFields 类，里面有 indexMask 和其他几个变量存放 RingBuffer 的内部状态信息。



CSDN @40岁资深老架构师尼恩

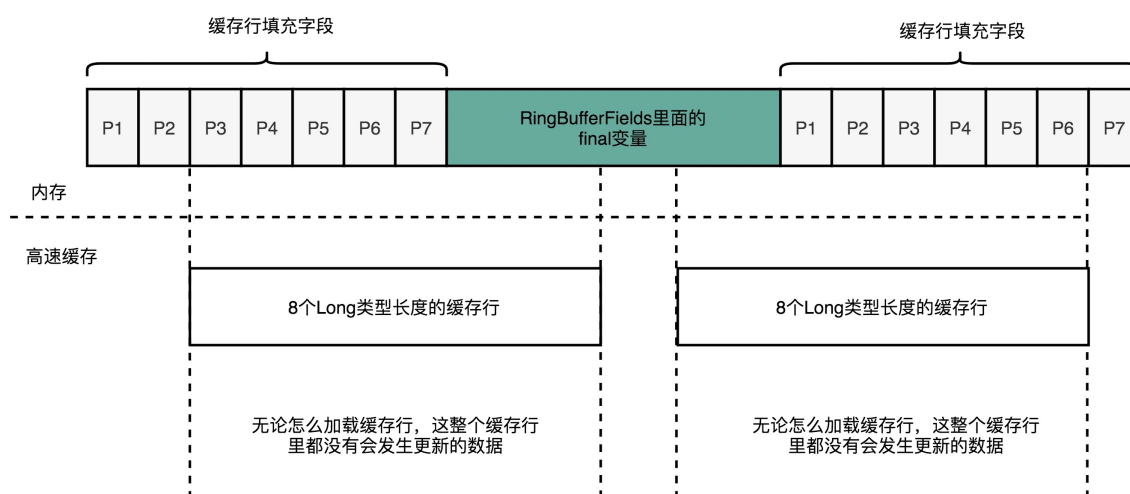
Disruptor 利用了缓存行填充，在 RingBufferFields 里面定义的变量的前后，分别定义了 7 个 long 类型的变量：

- 前面 7 个来自继承的 RingBufferPad 类
- 后面 7 个直接定义在 RingBuffer 类

这 14 个变量无任何实际用途。我们既不读他们，也不写他们。帮助 RingBufferFields 进行缓存行填充 Cache Line Padding

所以，一旦 RingBufferFields 被加载到 CPU Cache 后，只要被频繁读取访问，就不会再被换出 Cache。

这意味着，对于 RingBufferFields 的读取速度，会一直是 CPU Cache 的访问速度，而非内存的访问速度。



## MPSC 的 RingBuffer 的缓存行填充

我们自顶向下，将所有类的字段合并在一起，看下 MpscArrayQueue 的整体结构。

除了顶层 JDK 原生的 AbstractCollection、AbstractQueue，MpscArrayQueue 还继承了很多类似于 MpscXxxPad 以及 MpscXxxField 的类。

我们可以发现一个很有意思的规律：

每个有包含属性的类后面都会被 MpscXxxPad 类隔开。

MpscXxxPad 到底起到什么作用呢？

我们自顶向下，将所有类的字段合并在一起，看下 MpscArrayQueue 的整体结构。

- 填充类 BaseMpscLinkedArrayQueuePad1

```

abstract class BaseMpscLinkedArrayQueuePad1<E> extends AbstractQueue<E>
implements IndexedQueue {
    long p01, p02, p03, p04, p05, p06, p07;
    long p10, p11, p12, p13, p14, p15, p16, p17;
}

```

- 包含属性的类 BaseMpscLinkedArrayQueueProducerFields  
包含生产相关的属性

```

// $gen:ordered-fields
abstract class BaseMpscLinkedArrayQueueProducerFields<E> extends
BaseMpscLinkedArrayQueuePad1<E> {
    private final static long P_INDEX_OFFSET =
fieldOffset(BaseMpscLinkedArrayQueueProducerFields.class, "producerIndex");

    private volatile long producerIndex;

    // 获取生产者索引
    // lv = lazy value
    @Override
    public final long lvProducerIndex() {
        return producerIndex;
    }

    // so = set ordered / lazy set
    final void soProducerIndex(long newValue) {
        UNSAFE.putOrderedLong(this, P_INDEX_OFFSET, newValue);
    }

    final boolean casProducerIndex(long expect, long newValue) {
        return UNSAFE.compareAndSwapLong(this, P_INDEX_OFFSET, expect, newValue);
    }
}

```

- 填充类 BaseMpscLinkedArrayQueuePad2

```

abstract class BaseMpscLinkedArrayQueuePad2<E> extends
BaseMpscLinkedArrayQueueProducerFields<E> {
    long p01, p02, p03, p04, p05, p06, p07;
    long p10, p11, p12, p13, p14, p15, p16, p17;
}

```

- 包含属性的类 BaseMpscLinkedArrayQueueConsumerFields  
包含消费者相关的属性

```

// $gen:ordered-fields

```

```

abstract class BaseMpscLinkedListArrayQueueConsumerFields<E> extends
BaseMpscLinkedListArrayQueuePad2<E> {
    private final static long C_INDEX_OFFSET =
fieldOffset(BaseMpscLinkedListArrayQueueConsumerFields.class, "consumerIndex");

    private volatile long consumerIndex;
    protected long consumerMask;
    protected E[] consumerBuffer;

    // load volatile 消费者索引
    @Override
    public final long lvConsumerIndex() {
        return consumerIndex;
    }

    // load pain
    final long lpConsumerIndex() {
        return UNSAFE.getLong(this, C_INDEX_OFFSET);
    }

    // store ordered 消费者索引
    final void soConsumerIndex(long newValue) {
        UNSAFE.putOrderedLong(this, C_INDEX_OFFSET, newValue);
    }
}

```

可以看出，MpscXxxPad 类中使用了大量 long 类型的变量，其命名没有什么特殊的含义，只是起到填充的作用。

和 Disruptor 的源码，Mpsc Queue 和 Disruptor 之所以填充这些无意义的变量，是为了解决伪共享（false sharing）问题。

什么是伪共享呢？

请参考第 23 章：100w 级别 qps 日志平台，

那里介绍的极致详细，还有性能对比实操测试。已经是史诗级的详细了，这里不做介绍。

## MpscGrowableArrayQueue 源码分析

MpscArrayQueue 基本用法与 ArrayBlockingQueue 都是类似的：入队 offer() 和出队 poll()。

MpscArrayQueue 如何使用，示例代码如下：

```

package org.jctools.demo;

import org.jctools.queues.MpscArrayQueue;
import org.jctools.queues.MpscGrowableArrayQueue;

import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MpscArrayQueueTest {

```

```

    public static final MpscGrowableArrayQueue<String> MPSC_ARRAY_QUEUE = new
MpscGrowableArrayQueue<>(4);

    public static final ExecutorService pool= Executors.newFixedThreadPool(4);
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            int index = i;
            pool.submit(() -> MPSC_ARRAY_QUEUE.offer("疯狂创客圈 java 卷王 " +
index) ); // 入队操作
        }
        try {
            Thread.sleep(1000L);
            MPSC_ARRAY_QUEUE.add("疯狂创客圈 java 卷王 5"); // 入队操作，满则跑出异常
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("队列大小: " + MPSC_ARRAY_QUEUE.size() + ", 队列容量: " +
MPSC_ARRAY_QUEUE.capacity());
        System.out.println("出队: " + MPSC_ARRAY_QUEUE.remove()); // 出队操作，队列为
空则抛出异常
        System.out.println("出队: " + MPSC_ARRAY_QUEUE.poll()); // 出队操作，队列为空
则返回 NULL
    }
}

```

程序输出结果如下：

The screenshot shows an IDE with the following components:

- Project View:** Shows the project structure with a package `org.jctools.demo` containing `MpscArrayQueueTest`.
- Code Editor:** Displays the source code of `MpscArrayQueueTest`, showing the `main` method with a loop that enqueues elements and a `try` block that attempts to enqueue an additional element, which results in an exception.
- Debugger/Console:** Shows the execution output:
 

```

队列大小: 4, 队列容量: 4
出队: 疯狂创客圈 java 卷王 4
出队: 疯狂创客圈 java 卷王 1

```

## MpscUnboundedArrayQueue 的宏观架构

Caffeine 和 Netty，所使用的并不是原始的 `MpscArrayQueue`，而是 其高性能的子类 `MpscGrowableArrayQueue`，来看看其宏观架构。

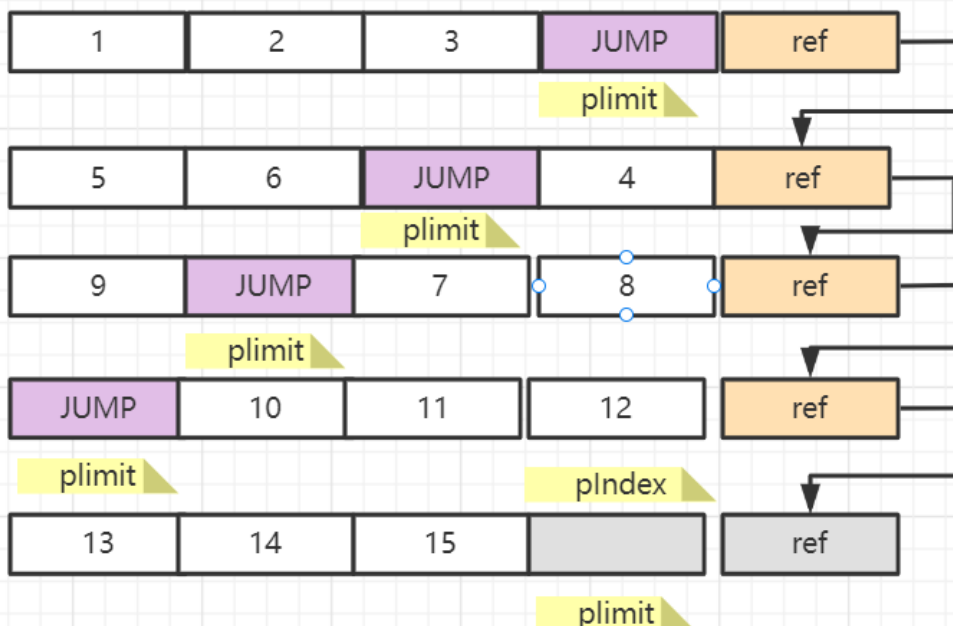
宏观上，`MpscGrowableArrayQueue` 基于数组 + 链表的复合结构。

基于数组 + 链表的复合结构的优势：

1 不会像链表那样分配过多的 Node，吞吐量比传统的链表高。

2 扩容的时候，也不存在数组复制，扩容的速度，也比传统的数组快

## MpscChunkedArrayQueue 基于数组+链表的复合结构



基于数组+链表的复合结构的优势：

1 不会像链表那样分配过多的Node，吞吐量比传统的链表高。

2 扩容的时候，也不存在数组复制，扩容的速度，也比传统的数组快

CSDN @40岁资深老架构师尼恩

## MpscUnboundedArrayQueue 的重要属性

MpscUnboundedArrayQueue 基本的数据结构由「数组 + 链表」组成，它有两个指针：

- producerBuffer 指向生产者生产的数组
- consumerBuffer，指向消费者消费的数组

它还有两个索引指针：

- producerIndex 指向生产者生产的索引
- consumerIndex，指向消费者消费的索引

这两个索引会以 2 为步长不断递增。

另外对于生产者，它还有一个 producerLimit 指针，它代表生产者生产消息的上限，

达到该 producerLimit 上限，Queue 就要扩容了，

扩容的方式是创建一个长度一样的新数组，然后旧数组的最后一个元素指向新数组，形成单向链表。

这些属性，为了高性能访问，都进行了填充，分布在不同的基类中

- BaseMpscLinkedArrayQueueColdProducerFields 生产者字段

```

abstract class BaseMpscLinkedArrayQueueColdProducerFields<E> extends
BaseMpscLinkedArrayQueuePad3<E> {
    private final static long P_LIMIT_OFFSET =
fieldOffset(BaseMpscLinkedArrayQueueColdProducerFields.class, "producerLimit");

    private volatile long producerLimit; //它代表生产者生产消息的上限
    protected long producerMask; // 计算生产者 数组下标的掩码
    protected E[] producerBuffer; // 计算生产者 数据的 数组

    .....

```

- BaseMpscLinkedArrayQueueConsumerFields 消费者字段

```

abstract class BaseMpscLinkedArrayQueueConsumerFields<E> extends
BaseMpscLinkedArrayQueuePad2<E> {
    private final static long C_INDEX_OFFSET =
fieldOffset(BaseMpscLinkedArrayQueueConsumerFields.class, "consumerIndex");

    private volatile long consumerIndex; // 消费者索引
    protected long consumerMask; // 计算消费 数组下标的掩码
    protected E[] consumerBuffer; // 计算消费 数据的 数组

    .....

```

看到 mask 变量，是用于计算数组下标的掩码。

相当于取模的操作，只是这里使用位运算取模，所以，队列中数组的容量大小肯定是 2 的次幂。

因为 Mpsc 是多生产者单消费者队列，所以 producerIndex、producerLimit 都是用 volatile 进行修饰的，其中一个生产者线程的修改需要对其他生产者线程可见。

还有一些其他的属性

```

abstract class BaseMpscLinkedArrayQueue<E> extends
BaseMpscLinkedArrayQueueColdProducerFields<E>
    implements MessagePassingQueue<E>, QueueProgressIndicators {

    // 数组被生产者填满后，会填充一个JUMP，代表队列扩容了，消费者遇到JUMP会消费下一个数组。

    // No post padding here, subclasses must add
    private static final Object JUMP = new Object();

    // 消费者消费完一个完整的数组后，会将最后一个元素设为BUFFER_CONSUMED。
    private static final Object BUFFER_CONSUMED = new Object();

    .....

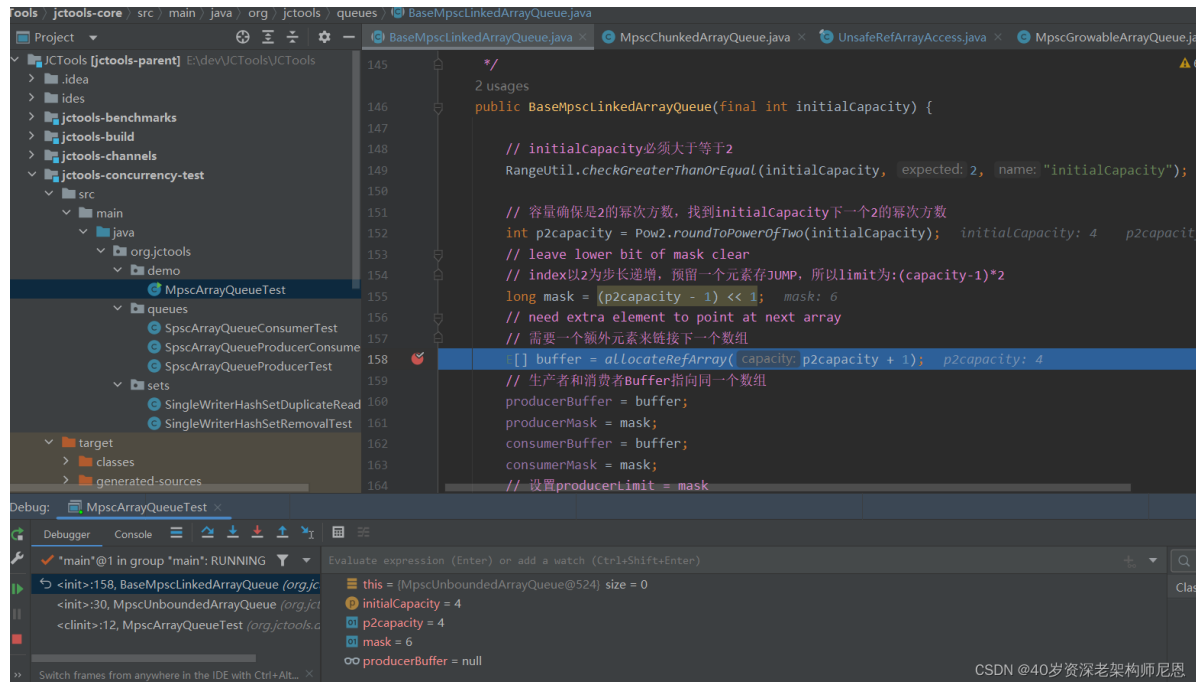
```



# MpscUnboundedArrayQueue 构造函数,

需要给定一个 chunkSize, 指定块大小,

```
public static final MpscUnboundedArrayQueue<String> MPSC_ARRAY_QUEUE = new
MpscUnboundedArrayQueue<>(4);
```



MpscQueue 由一系列数组构成, chunkSize 就是数组的大小, 它必须是一个 2 的幂次方数。

```
public class MpscUnboundedArrayQueue<E> extends BaseMpscLinkedArrayQueue<E>
{
    long p0, p1, p2, p3, p4, p5, p6, p7;
    long p10, p11, p12, p13, p14, p15, p16, p17;

    public MpscUnboundedArrayQueue(int chunkSize)
    {
        super(chunkSize);
    }
    ....
}
```

父类的构造函数

```
public BaseMpscLinkedArrayQueue(final int initialCapacity) {

    // initialCapacity必须大于等于2
    RangeUtil.checkGreaterThanOrEqualTo(initialCapacity, 2, "initialCapacity");

    // 容量确保是2的幂次方数, 找到initialCapacity下一个2的幂次方数
    int p2capacity = Pow2.roundToPowerOfTwo(initialCapacity);
    // leave lower bit of mask clear
    // index以2为步长递增, 预留一个元素存JUMP, 所以limit为:(capacity-1)*2
```

```

    long mask = (p2capacity - 1) << 1;
    // need extra element to point at next array
    // 需要一个额外元素来链接下一个数组
    E[] buffer = allocateRefArray(p2capacity + 1);
    // 生产者和消费者Buffer指向同一个数组
    producerBuffer = buffer;
    producerMask = mask;
    consumerBuffer = buffer;
    consumerMask = mask;
    // 设置producerLimit = mask
    soProducerLimit(mask); // we know it's all empty to start with
}

```

在父类的构造函数中，计算了 mask，初始化了一个数组，

并将 producerBuffer 和 consumerBuffer 都指向了同一个数组，然后根据 mask 设置 producerLimit。

假设 initialCapacity 为 4，数组的长度就是 5，

因为最后一个元素会用来存放扩容数组的地址，形成链表。

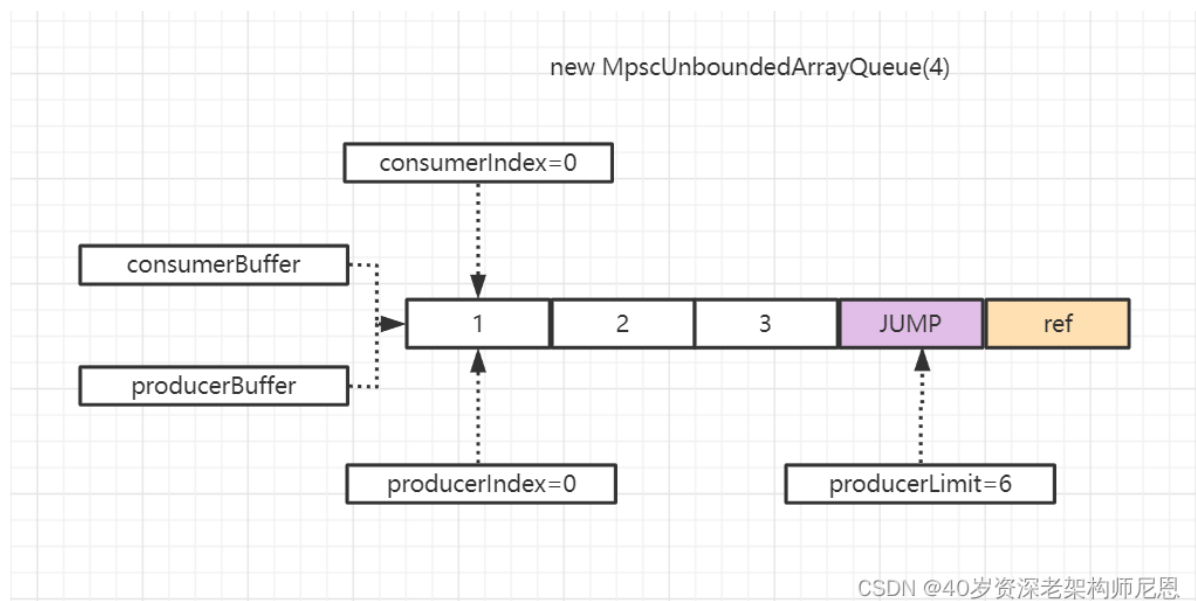
每个数组还会预留一个槽位存放 JUMP 元素，代表队列扩容了，消费者遇到 JUMP 元素就会通过最后一个元素找到扩容后的数组继续消费，因此一个数组最多保留 3 个元素。

## 入队、扩容源码分析

### 图解：入队的核心流程

这个结构比较复杂，大家慢慢看吧

### 新队列的属性值



```
src > main > java > org > jctools > queues > BaseMpscLinkedArrayQueue.java
BaseMpscLinkedArrayQueue.java  UnsafeRefArrayAccess.java  MpscGrowArrayQueue.java  MpscArrayQueueTest.java
parent] E:\dev\JCTools\JCTools 232 // 扩容的时候, 会在offerSlowPath()中扩容线程会将其设为奇数
233 // lower bit is indicative of resize, if we see it we spin until it's cleared
234 if ((pIndex & 1) == 1) { pIndex: 0
235 // 奇数代表正在扩容, 自旋, 等待扩容完成
236 continue;
237 }
238 // pIndex is even (lower bit is 0) -> actual index is (pIndex >> 1)
239 // pIndex 是偶数, 实际的索引值 需要 除以2
240
241 // mask/buffer may get changed by resizing -> only use for array access after successful
242
243 mask = this.producerMask;
244 buffer = this.producerBuffer;
245
246 // a successful CAS ties the ordering. lv(pIndex) - [mask/buffer] -> cas(pIndex)
```

group "main": RUNNING

MpscLinkedArrayQueue (org.jctools.queues)

UnboundedArrayQueue (org.jctools.queues)

ArrayQueueTest (org.jctools.demo)

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

this = (MpscUnboundedArrayQueue@526) size = 0

e = "疯狂创客圈 java 卷王 1"

producerLimit = 6

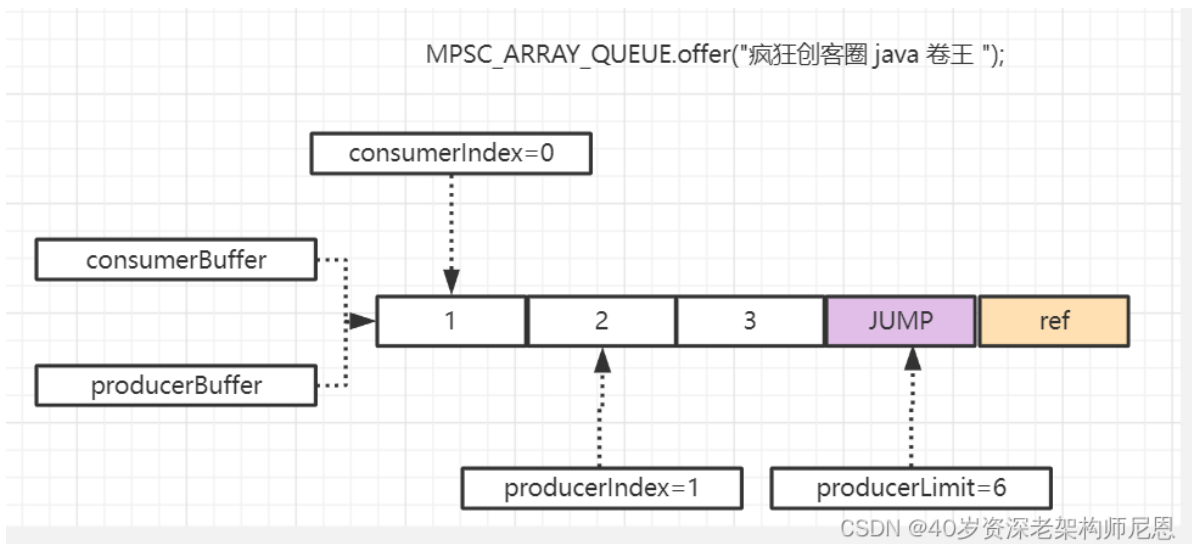
pIndex = 0

this.producerBuffer = (Object[5]@528)

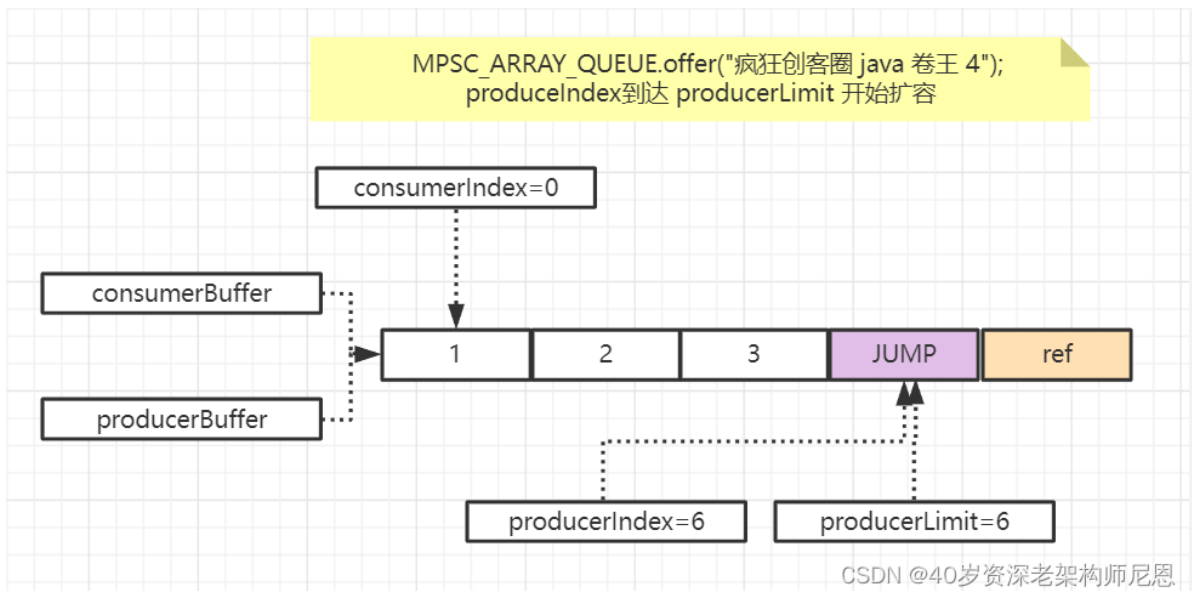
this.producerMask = 6

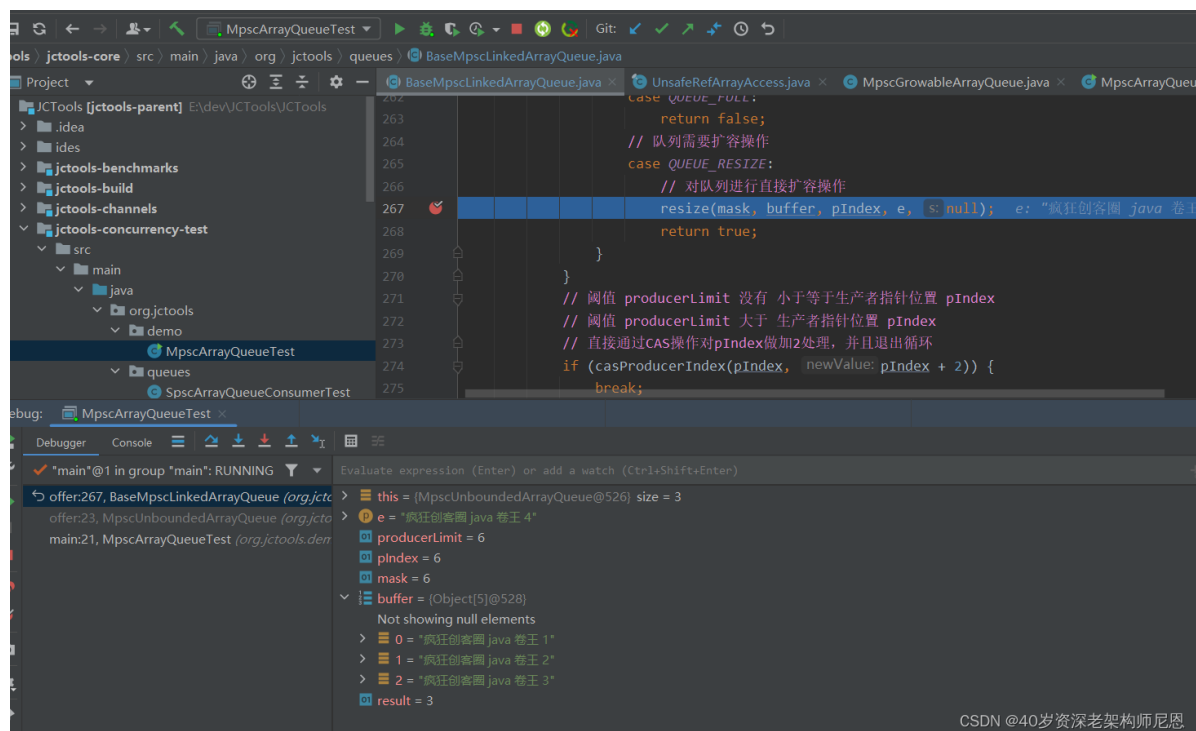
CSDN @40岁资深老架构师尼恩

## 通过 offer 插入一个元素

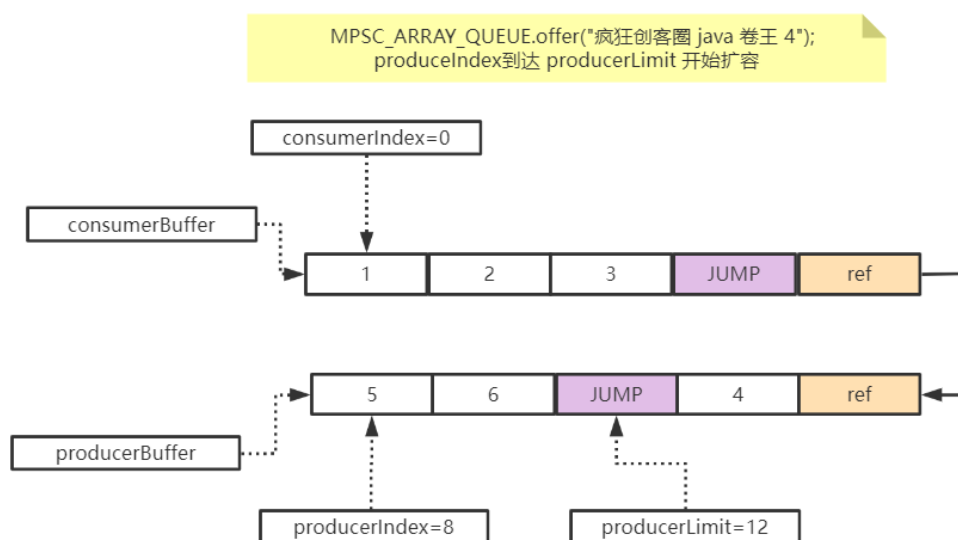


## 第一次 produceIndex 到达 producerLimit 开始扩容

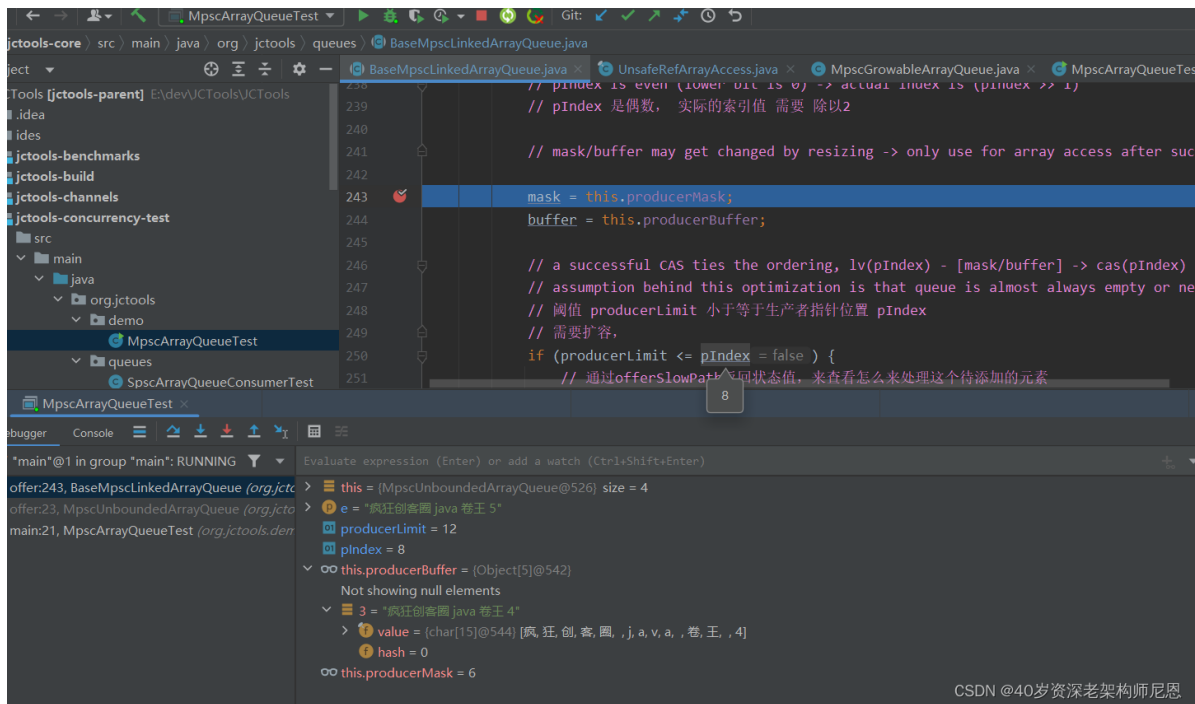




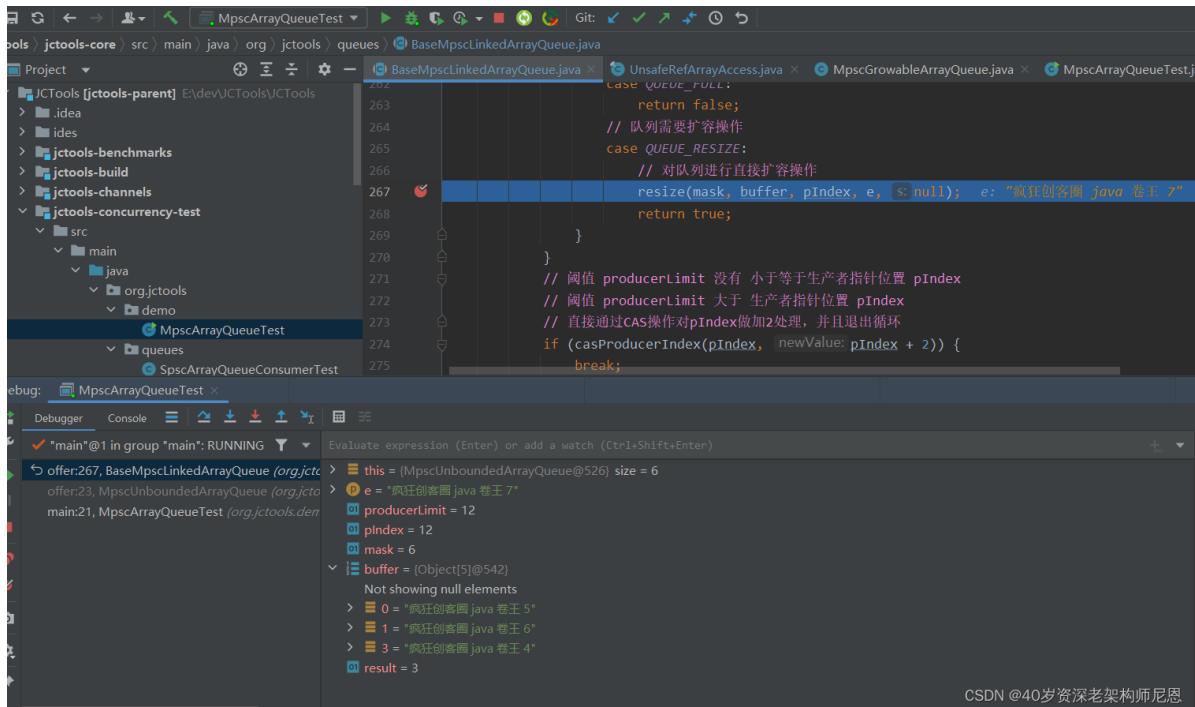
扩容之后，插入一个新的元素，之后的属性值



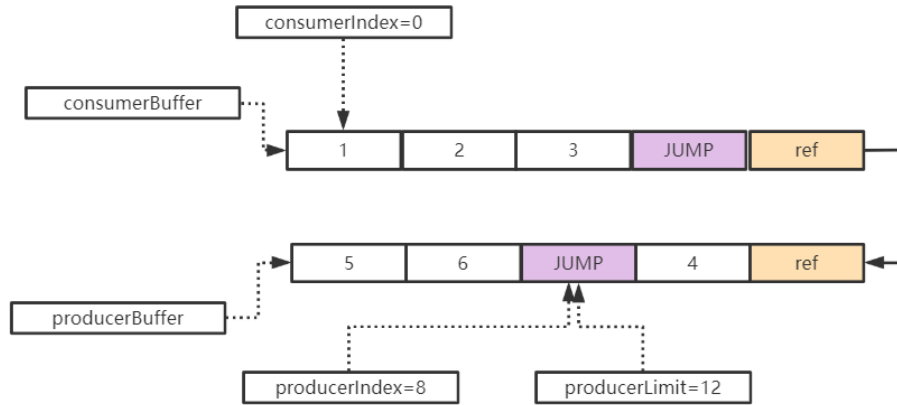
CSDN @40岁资深老架构师尼恩



## 第二次 produceIndex 到达 producerLimit



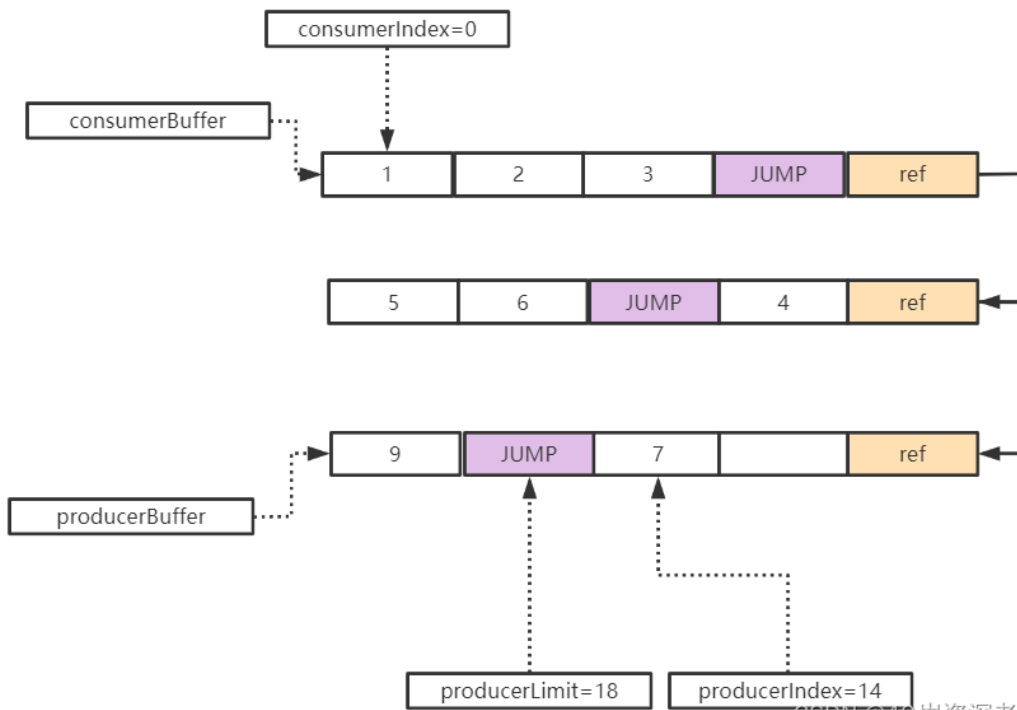
MPSC\_ARRAY\_QUEUE.offer("疯狂创客圈 java 卷王 7");  
produceIndex到达 producerLimit 开始扩容



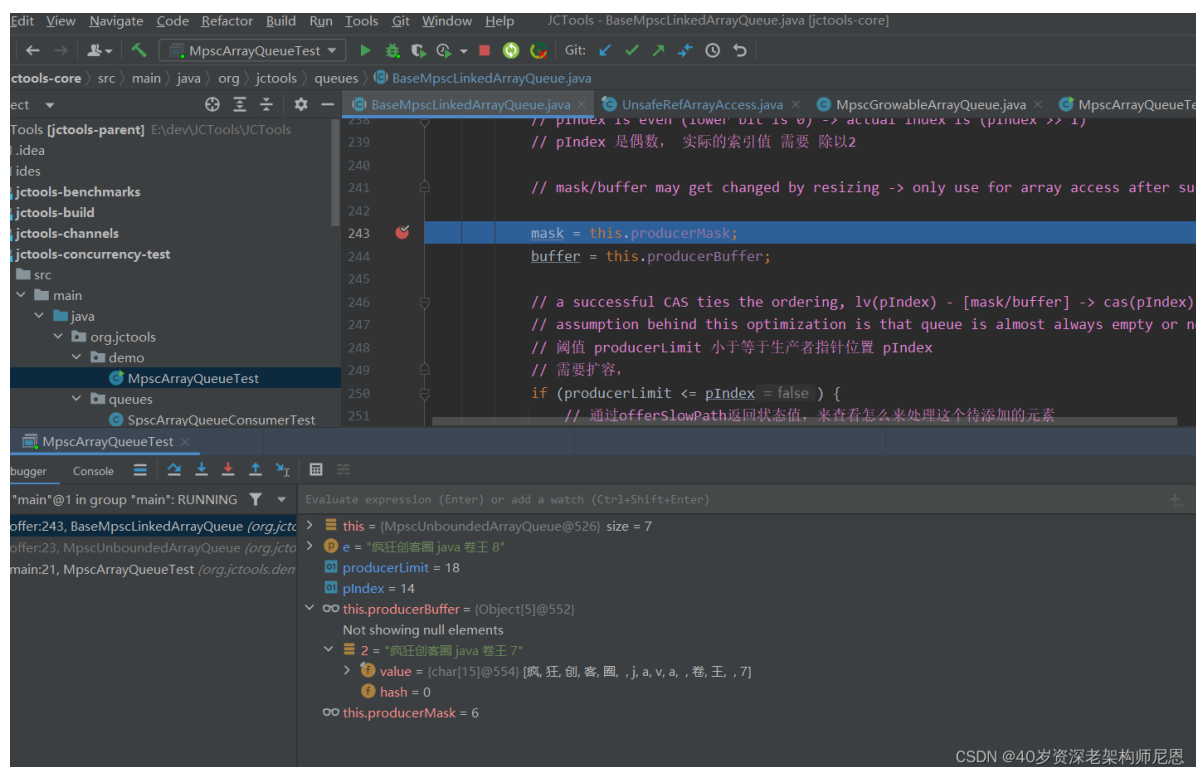
CSDN @40岁资深老架构师尼恩

扩容之后，插入一个新的元素，之后的属性值

MPSC\_ARRAY\_QUEUE.offer("疯狂创客圈 java 卷王 7");

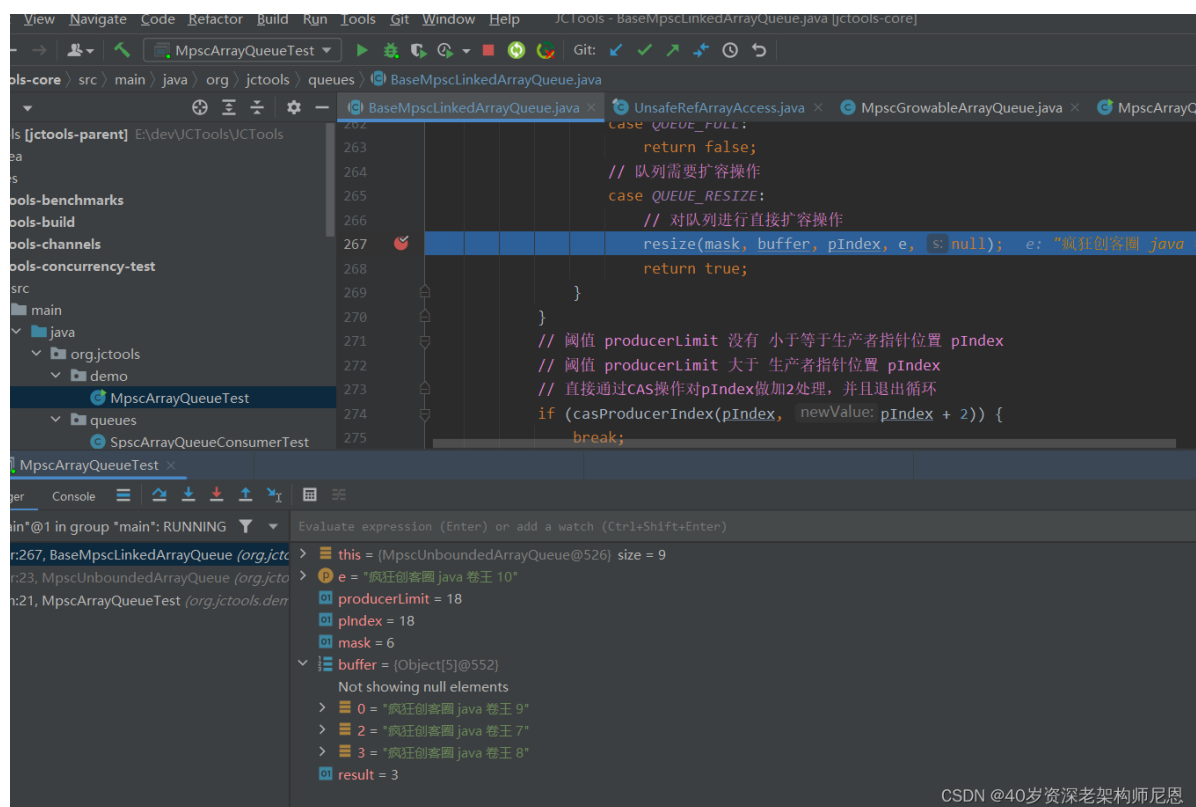


CSDN @40岁资深老架构师尼恩



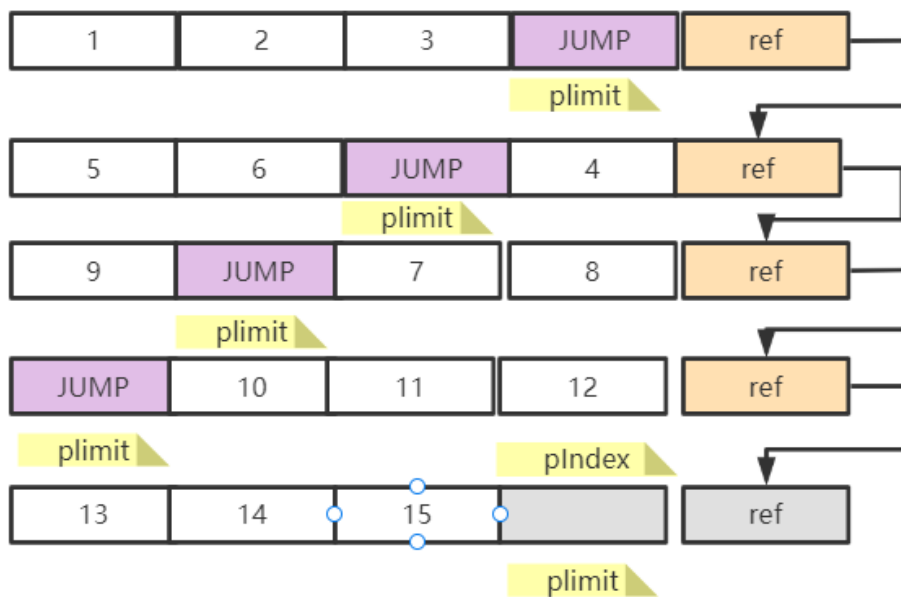
CSDN @40岁资深老架构师尼恩

再插入两个元素



CSDN @40岁资深老架构师尼恩

## MpscChunkedArrayQueue 基于数组+链表的复合结构



基于数组+链表的复合结构的优势:

- 1 不会像链表那样分配过多的Node, 吞吐量比传统的链表高。
- 2 扩容的时候, 也不存在数组复制, 扩容的速度, 也比传统的数组快

CSDN @40岁资深老架构师尼恩

### 入队 offer 的源码分析

`offer(e)` 会将元素 `e` 添加到队列中, 即生产数据。

在 `MpscQueue` 中, 线程通过 `CAS` 的方式以步长为 2 递增 `producerIndex`, `CAS` 会保证只有一个线程操作成功, `CAS` 成功就代表线程抢到了数组中的槽位, 它可以将元素 `e` 添加到数组的指定槽位。

`CAS` 失败代表并发失败了, 会自旋重试。

如果 `producerIndex` 达到 `producerLimit`, 代表生产达到上限, 队列可能需要扩容了。

`offerSlowPath()` 方法会判断队列是否需要扩容, 如果需要扩容, 也只会交给一个线程去扩容, 这里又是一个 `CAS` 操作, 线程以 1 为步长递增 `producerIndex`, 只有 `CAS` 成功的线程才会去执行扩容逻辑。

因此, 在 `offer(e)` 的逻辑中, 还会判断 `producerIndex` 是否是奇数, 如果为奇数就代表队列正在扩容。

因为 `MpscQueue` 的扩容非常快速, 它不需要迁移元素, 只需要创建一个新数组, 再和旧数组建立连接就可以了,

所以没有必要让其他线程挂起, 线程发现队列在扩容时, 会进行自旋重试, 直到扩容完成。

```
/**
 * , 生产数据
 * 向队列中添加一个元素e
 *
 * @param e not {@code null}, will throw NPE if it is
 * @return
 */
@Override
public boolean offer(final E e) {
```



```

        if (null == e) {
            throw new NullPointerException();
        }

        long mask;
        E[] buffer; //生产者指向的数组
        long pIndex; //生产索引

        while (true) {
            long producerLimit = lvProducerLimit(); // 获取生产者索引最大限制

            pIndex = lvProducerIndex();          // 获取生产者索引

            // 生产索引以2为步长递增,
            // 第0位标识为resize, 所以非扩容场景, 不会是奇数,
            // 扩容的时候, 会在offerSlowPath()中扩容线程会将其设为奇数
            // lower bit is indicative of resize, if we see it we spin until it's
cleared

            if ((pIndex & 1) == 1) {
                // 奇数代表正在扩容, 自旋, 等待扩容完成
                continue;
            }
            // pIndex is even (lower bit is 0) -> actual index is (pIndex >> 1)
            // pIndex 是偶数, 实际的索引值 需要 除以2

            // mask/buffer may get changed by resizing -> only use for array
access after successful CAS.

            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
            // 阈值 producerLimit 小于等于生产者指针位置 pIndex
            // 需要扩容,
            if (producerLimit <= pIndex) {
                // 通过offerSlowPath返回状态值, 来查看怎么处理这个待添加的元素
                int result = offerSlowPath(mask, pIndex, producerLimit);
                switch (result) {
                    // producerLimit虽然达到了limit,
                    // 但是当前数组已经被消费了部分数据, 暂时不会扩容, 会使用已被消费的槽位。
                    case CONTINUE_TO_P_INDEX_CAS:
                        break;
                    // 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
                    case RETRY:
                        continue;
                    // 队列已满, 直接返回false操作
                    case QUEUE_FULL:
                        return false;
                    // 队列需要扩容操作
                    case QUEUE_RESIZE:
                        // 对队列进行直接扩容操作
                        resize(mask, buffer, pIndex, e, null);
                        return true;
                }
            }
        }
    }

```

```

    }
    // 阈值 producerLimit 大于 生产者指针位置 pIndex
    // 直接通过CAS操作对pIndex做加2处理
    if (casProducerIndex(pIndex, pIndex + 2)) {
        break;
    }
}
// INDEX visible before ELEMENT
final long offset = modifiedCalcCircularRefElementOffset(pIndex, mask);
// 将buffer数组的指定位置替换为e,
// 不是根据下标来设置的, 是根据槽位的地址偏移量offset, UNSAFE操作。
soRefElement(buffer, offset, e); // release element e
return true;
}

```

`offerSlowPath()` 会告诉线程队列是满了, 还是需要扩容, 还是需要自旋重试。

总之, 这个一条慢路径, 所以叫做 slow path。

虽然 `producerIndex` 达到了 `producerLimit`, 但不代表队列就非得扩容,

如果消费者已经消费了部分生产者指向的数组元素, 就意味这当前数组还是有槽位可以继续用的, 暂时不用扩容。

```

/**
 * @param mask
 * @param pIndex 生产者索引
 * @param producerLimit 生产者limit
 * @return
 */
private int offerSlowPath(long mask, long pIndex, long producerLimit) {
    // 消费者索引
    final long cIndex = lvConsumerIndex();
    // 数组缓冲的容量, (长度-1) * 2
    long bufferCapacity = getCurrentBufferCapacity(mask);
    // 消费索引 + 当前数组的容量 > 生产索引, 代表当前数组已有部分元素被消费了,
    // 不会扩容, 会使用已被消费的槽位。
    // cIndex + bufferCapacity => producerLimit
    if (cIndex + bufferCapacity > pIndex) {
        if (!casProducerLimit(producerLimit, cIndex + bufferCapacity)) {
            // CAS失败, 自旋重试
            // retry from top
            return RETRY;
        } else {
            // continue to pIndex CAS
            // 重试 CAS修改 生产索引
            return CONTINUE_TO_P_INDEX_CAS;
        }
    }
    // full and cannot grow
    // 根据生产者和消费者索引判断Queue是否已满, 无界队列永不会满
    else if (availableInQueue(pIndex, cIndex) <= 0) {
        // offer should return false;
        return QUEUE_FULL;
    }
}

```

```

        // grab index for resize -> set lower bit
        // CAS的方式将producerIndex加1, 奇数代表正在resize
        else if (casProducerIndex(pIndex, pIndex + 1)) {
            // trigger a resize
            return QUEUE_RESIZE;
        } else {
            // failed resize attempt, retry from top
            return RETRY;
        }
    }
}

```

如果需要扩容，线程会 CAS 操作将 producerIndex 改为奇数，让其它线程能感知到队列正在扩容，要生产数据的线程先自旋，等待扩容完成再继续操作。

offer() 过程中，通过 CAS 抢槽位，CAS 失败的线程自旋重试。

如果遇到队列需要扩容，则将 producerIndex 设为奇数，其他线程自旋等待，一直等到扩容完成，扩容后再设为偶数，通知其它线程继续生产。

## 入队扩容源码分析

resize() 是扩容的核心方法，

它首先会创建一个相同长度的新数组，将 producerBuffer 指向新数组，然后将元素 e 放到新数组中，旧元素的最后一个元素指向新数组，形成链表。

还会将旧元素的槽位填充 JUMP 元素，代表队列扩容了。

```

// 扩容：
// 新建一个E[], 将oldBuffer和newBuffer建立连接。
// 将producerBuffer指向新数组，然后将元素e放到新数组中，
// 旧元素的最后一个元素指向新数组，形成链表。
// 还会将旧元素的槽位填充JUMP元素，代表队列扩容了。
private void resize(long oldMask, E[] oldBuffer, long pIndex, E e,
Supplier<E> s) {
    assert (e != null && s == null) || (e == null || s != null);

    // 下一个Buffer的长度，MpscQueue会构建一个相同长度的Buffer
    int newBufferLength = getNextBufferSize(oldBuffer);
    final E[] newBuffer;
    try {
        // 创建一个新的E[]

        newBuffer = allocateRefArray(newBufferLength);
    } catch (OutOfMemoryError oom) {
        assert lvProducerIndex() == pIndex + 1;
        soProducerIndex(pIndex);
        throw oom;
    }

    // 生产者Buffer指向新的E[]
    producerBuffer = newBuffer;
    // 计算新的Mask, Buffer长度不变的情况下，Mask也不变
    final int newMask = (newBufferLength - 2) << 1;
    producerMask = newMask;
}

```

```

        // 根据该偏移量设置oldBuffer的JUMP元素, 会递增然后重置, 不断循环
        final long offsetInOld = modifiedCalcCircularRefElementOffset(pIndex,
oldMask);
        // Mask不变的情况下, oldBuffer的JUMP对应的位置, 就是newBuffer中要消费的位置.
        final long offsetInNew = modifiedCalcCircularRefElementOffset(pIndex,
newMask);

        // 元素e放到新数组中
        soRefElement(newBuffer, offsetInNew, e == null ? s.get() : e); // element
in new array
        // 旧数组和新数组建立连接, 旧数组的最后一个元素保存新数组的地址。
        soRefElement(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
        soProducerLimit(pIndex + Math.min(newMask, availableInQueue));

        // make resize visible to the other producers
        soProducerIndex(pIndex + 2);

        // INDEX visible before ELEMENT, consistent with consumer expectation

        // make resize visible to consumer
        soRefElement(oldBuffer, offsetInOld, JUMP);
    }

```

## 出队 poll 的核心流程

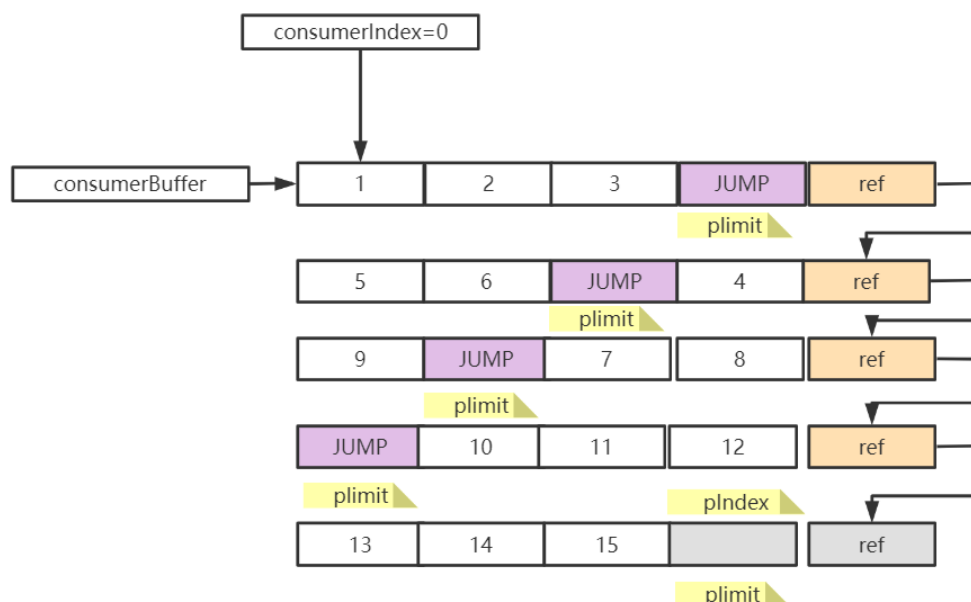
### 图解：poll 的核心流程

这个结构比较复杂, 大家慢慢看吧

```

core | src | main | java | org | jctools | queues | BaseMpscLinkedArrayQueue.java | BaseMpscLinkedArrayQueue | poll
BaseMpscLinkedArrayQueue.java | MpscChunkedArrayQueue.java | UnsafeRefArrayAccess.java | MpscGrowArrayQueue.java
ctools-parent] E:\dev\JCTools\JCTools
- benchmarks
- build
- channels
- concurrency-test
main
java
org.jctools
demo
MpscArrayQueueTest
MpscArrayQueueTest
Console
@1 in group "main": RUNNING
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
> this = (MpscUnboundedArrayQueue@524) size = 15
> buffer = (Object[5]@531)
> 0 = "疯狂创客圈 java 卷王 1"
> 1 = "疯狂创客圈 java 卷王 2"
> 2 = "疯狂创客圈 java 卷王 3"
> 3 = (Object@534)
> 4 = (Object[5]@551)
> index = 0
> mask = 6
> offset = 16
> e = "疯狂创客圈 java 卷王 1"
> value = (char[15]@550) [疯, 狂, 创, 客, 圈, , j, a, v, a, , 卷, 王, , 1]
> hash = 0
CSDN @40岁资深老架构师尼恩

```

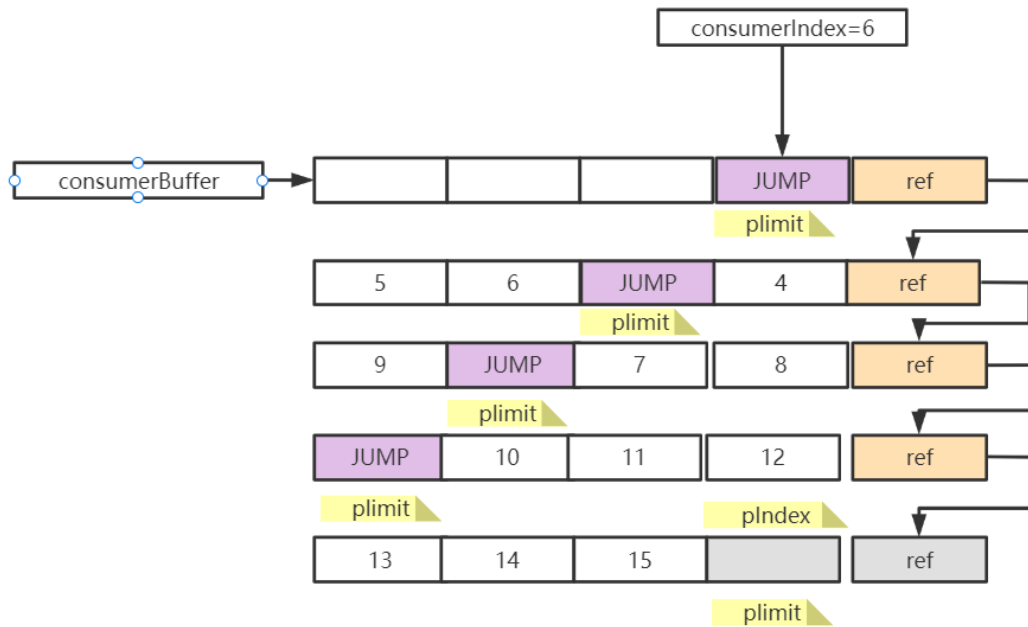


CSDN @40岁资深老架构师尼恩

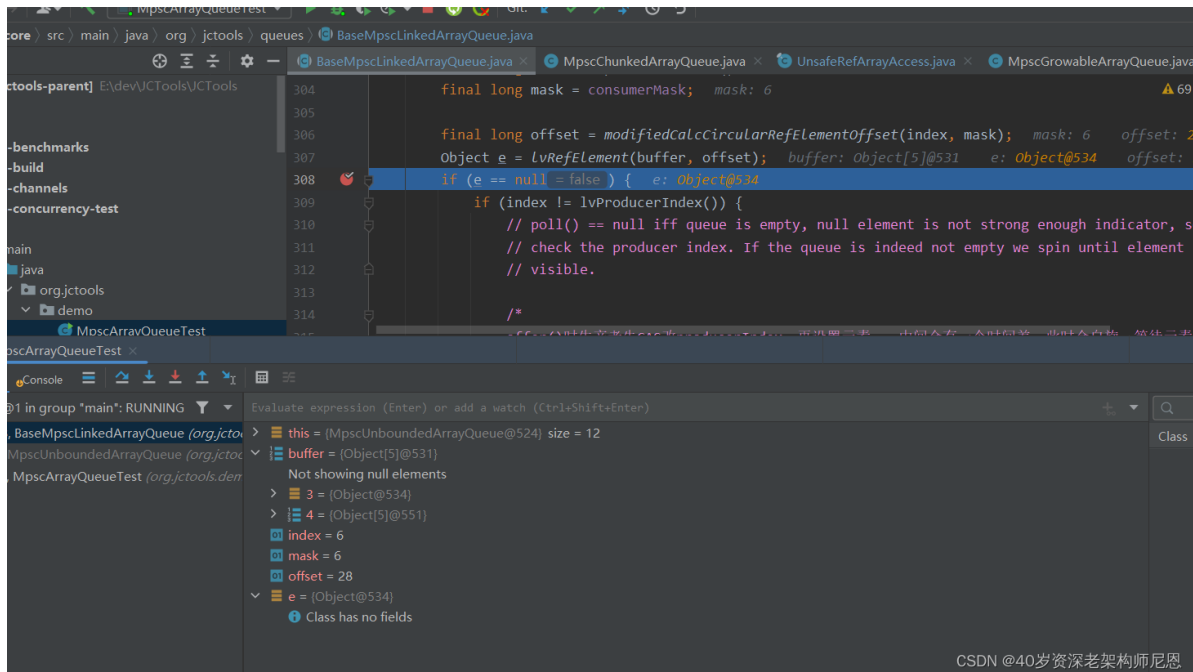
queue.poll() 消费后会将槽位设 null

一个 **Buffer** 消费完毕，消费遇到 **JUMP** 节点

一个 Buffer 消费完毕，消费遇到 JUMP 节点，

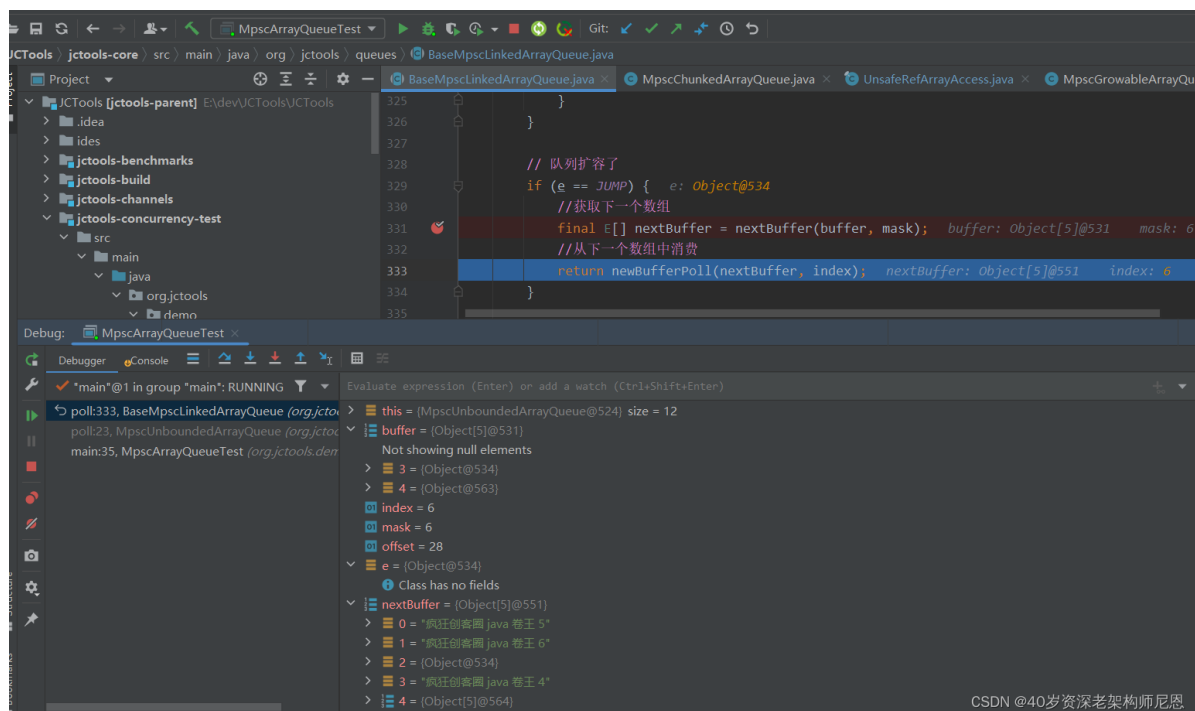


CSDN @40岁资深老架构师尼恩

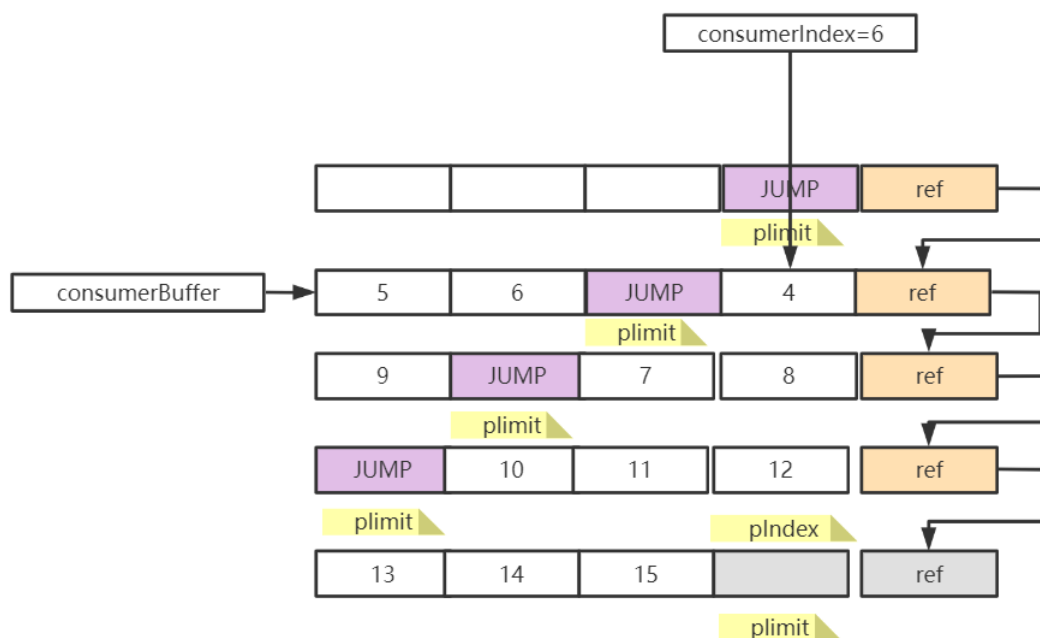


CSDN @40岁资深老架构师尼恩

通过 nextbuffer 找下一个数组，消费相同索引位的元素

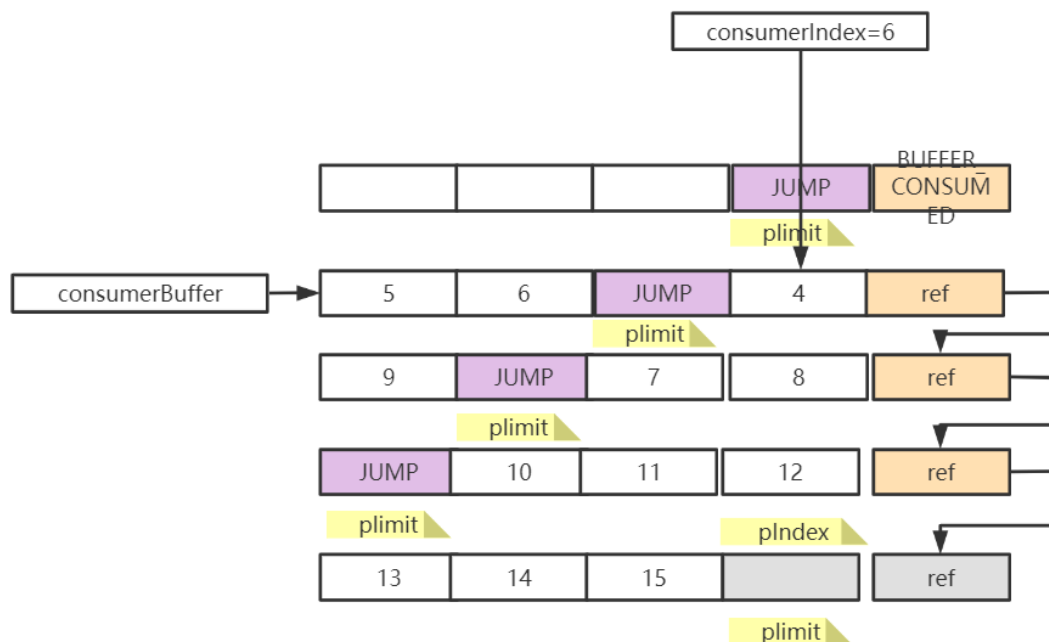


CSDN @40岁资深老架构师尼恩



CSDN @40岁资深老架构师尼恩

整个数组消费完了，将最后一位设为 BUFFER CONSUMED，从链表中剔除。



CSDN @40岁资深老架构师尼恩

## 出队 poll 的源码分析

poll() 方法核心思路是获取消费者索引 consumerIndex，然后根据 consumerIndex 计算得出数组对应的偏移量，然后将数组对应位置的元素取出并返回，最后将 consumerIndex 移动到环形数组下一个位置。

如果元素为 null，并不代表队列是空的，还要比较 consumerIndex 和 producerIndex，如果两者索引不同，那么 producerIndex 肯定是大于 consumerIndex 的，说明生产者已经在生产了，移动了 producerIndex，只是还没来得及将元素填充到数组而已。

因为生产者是先 CAS 递增 producerIndex，再将元素填充到数组的，两步之间存在一个非常短的时间差，如果消费者恰好在这个时间差内去消费数据，那么就自旋等待一下，等待生产者填充元素到数组。

如果元素为 JUMP，说明队列扩容了，消费者需要根据数组的最后一个元素找到扩容后的新数组，消费新数组的元素。

```
// poll()没有做并发控制，MpscQueue是多生产单消费者的Queue，只有一个消费者，这个也是netty
换成 mspcqueue的主要原因
@Override
public E poll() {
    final E[] buffer = consumerBuffer;
    final long index = lpConsumerIndex();
    final long mask = consumerMask;

    final long offset = modifiedCalcCircularRefElementOffset(index, mask);
    Object e = lvRefElement(buffer, offset);
    if (e == null) {
        if (index != lvProducerIndex()) {
            // poll() == null iff queue is empty, null element is not strong
            enough indicator, so we must
            // check the producer index. If the queue is indeed not empty we
            spin until element is
            // visible.

            /*
```



offer()时生产者先CAS改producerIndex, 再设置元素。中间会有一个时间差, 此时会自旋, 等待元素设置完成。

```
        */
        do {
            e = lvRefElement(buffer, offset);
        }
        while (e == null);
    } else {

        //元素已经消费完
        return null;
    }
}

// 队列扩容了
if (e == JUMP) {
    //获取下一个数组
    final E[] nextBuffer = nextBuffer(buffer, mask);
    //从下一个数组中消费
    return newBufferPoll(nextBuffer, index);
}

// 取出元素后, 将原来的槽位设为null
soRefElement(buffer, offset, null); // release element null
// 递增consumerIndex
soConsumerIndex(index + 2); // release cIndex
return (E) e;
}
```

如果队列扩容了, nextBuffer() 会找到扩容后的新数组, 同时它还会将旧数组的最后一个元素设为 BUFFER\_CONSUMED, 代表当前数组已经被消费完了, 也就从链表中剔除了。

```
@SuppressWarnings("unchecked")
private E[] nextBuffer(final E[] buffer, final long mask) {

    /*
        通过当前数组的最后一个元素, 获取下一个待消费的数组,
        将当前数组最后一个元素设为BUFFER_CONSUMED, 代表当前数组已经消费完毕。
    */
    final long offset = nextArrayOffset(mask);
    final E[] nextBuffer = (E[]) lvRefElement(buffer, offset);
    consumerBuffer = nextBuffer;
    consumerMask = (length(nextBuffer) - 2) << 1;
    soRefElement(buffer, offset, BUFFER_CONSUMED);
    return nextBuffer;
}
```

## 队列中的有序写入、有序读取

### 有序写入数组元素

offer 写入的时候，jctool 根据 pIndex 进行位运算计算得到数组对应的下标，然后通过 soRefElement 方法将数据写入到数组中，源码如下所示。

```
    }
    // INDEX visible before ELEMENT
    final long offset = modifiedCalcCircularRefElementOffset(pIndex, mask);
    // 将buffer数组的指定位置替换为e,
    // 不是根据下标来设置的，是根据槽位的地址偏移量offset，UNSAFE操作。
    soRefElement(buffer, offset, e); // An ordered store of an element to a
given offset
    return true;
```

```
/**
 * An ordered store of an element to a given offset
 *
 * @param buffer this.buffer
 * @param offset computed via {@link
UnsafeRefArrayAccess#calcCircularRefElementOffset}
 * @param e      an orderly kitty
 */
public static <E> void soRefElement(E[] buffer, long offset, E e)
{
    UNSAFE.putOrderedObject(buffer, offset, e);
}
```

putOrderedObject() 和 putObject() 都可以用于更新对象的值，但是 putOrderedObject() 并不会立刻将数据更新到内存中，同时，也不会去保障不同 CPU 内核的 Cache Line 数据强一致。

这部分原理比较复杂，具体请参见《Java 高并发核心编程卷 2》。

putOrderedObject() 使用的是 LazySet 延迟更新机制，所以性能方面 putOrderedObject() 要比 putObject() 高很多。

Java 中有四种类型的内存屏障，分别为 LoadLoad、StoreStore、LoadStore 和 StoreLoad。

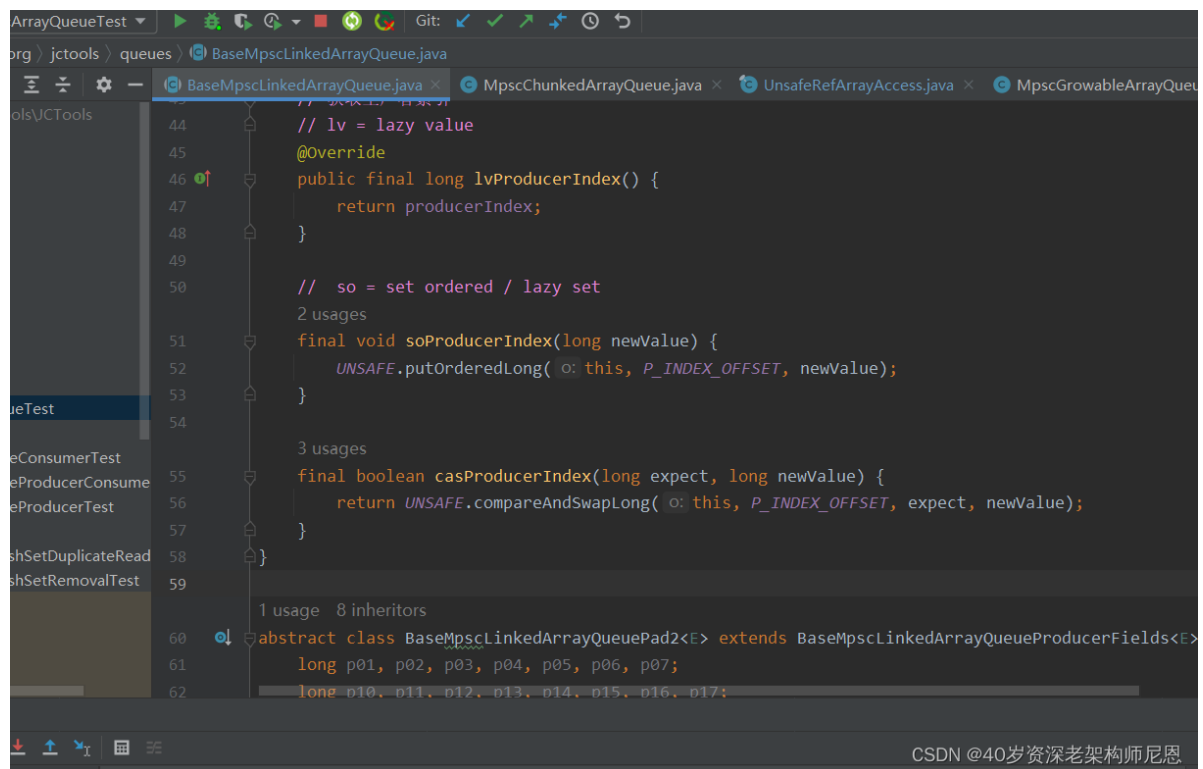
putOrderedObject() 使用了 StoreStore Barrier，对于 Store1，StoreStore，Store2 这样的操作序列，在 Store2 进行写入之前，会保证 Store1 的写操作对其他处理器可见。

有序写入的优势，就是性能高，写操作结果有纳秒级的延迟。

但是，有序写入是有代价的，不会立刻被其他线程以及自身线程可见。

因为在 Mpsc Queue 的使用场景中，多个生产者只负责写入数据，并没有写入之后立刻读取的需求，所以使用 LazySet 机制是没有问题的，只要 StoreStore Barrier 保证多线程写入的顺序即可。

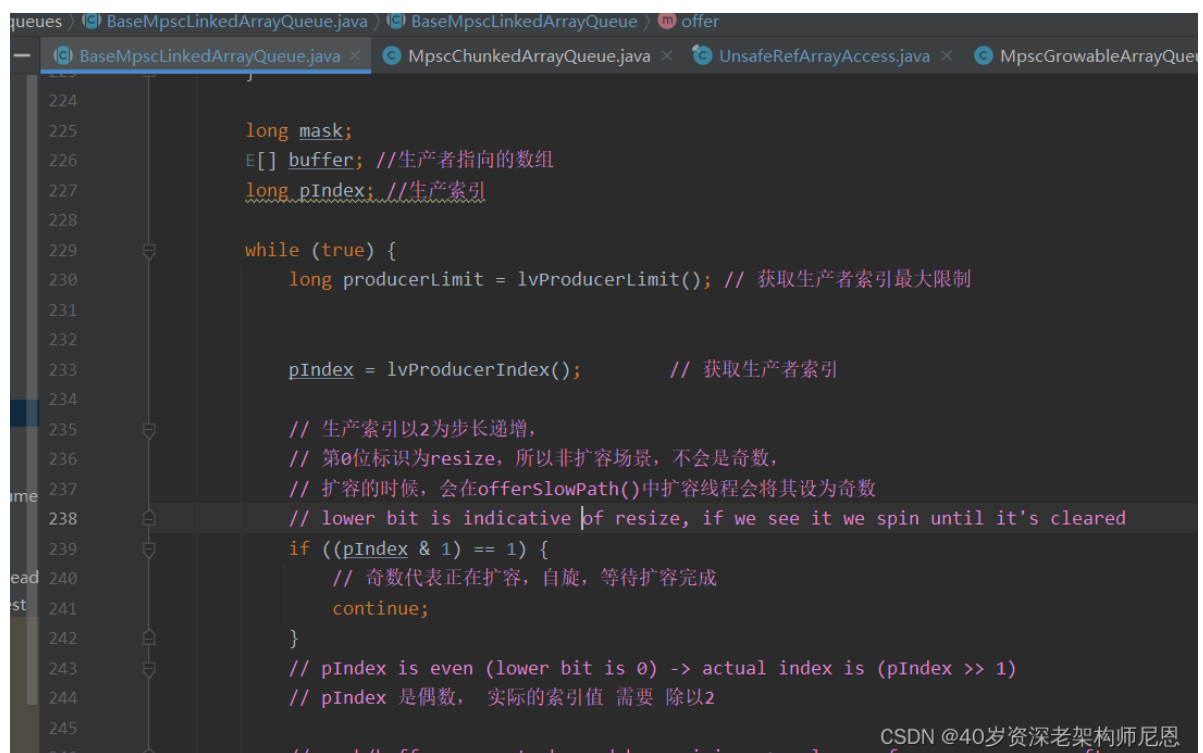
虽然有序写入数组元素，但是，对生产者的 pIndex 的写入，用的是 cas 写入，这个是有强一致性的。通过 StoreLoad Barrier 保证有序性和可见性。



```
org > jctools > queues > BaseMpscLinkedArrayQueue.java
BaseMpscLinkedArrayQueue.java x MpscChunkedArrayQueue.java x UnsafeRefArrayAccess.java x MpscGrowArrayQueue
toolsVCTools
44 // lv = lazy value
45 @Override
46 public final long lvProducerIndex() {
47     return producerIndex;
48 }
49
50 // so = set ordered / lazy set
51 2 usages
52 final void soProducerIndex(long newValue) {
53     UNSAFE.putOrderedLong( o: this, P_INDEX_OFFSET, newValue);
54 }
55 3 usages
56 final boolean casProducerIndex(long expect, long newValue) {
57     return UNSAFE.compareAndSwapLong( o: this, P_INDEX_OFFSET, expect, newValue);
58 }
59
60 1 usage 8 inheritors
61 abstract class BaseMpscLinkedArrayQueuePad2<E> extends BaseMpscLinkedArrayQueueProducerFields<E>
62     long p01, p02, p03, p04, p05, p06, p07;
63     long p10, p11, p12, p13, p14, p15, p16, p17;
```

CSDN @40岁资深老架构师尼恩

因为生产者有多个线程，所以 MpscArrayQueue 采用了 UNSAFE.getLongVolatile() 方法保证获取 pIndex 索引，保证其可见性、准确性。



```
queues > BaseMpscLinkedArrayQueue.java > BaseMpscLinkedArrayQueue > offer
BaseMpscLinkedArrayQueue.java x MpscChunkedArrayQueue.java x UnsafeRefArrayAccess.java x MpscGrowArrayQueue
224
225 long mask;
226 E[] buffer; //生产者指向的数组
227 long pIndex; //生产索引
228
229 while (true) {
230     long producerLimit = lvProducerLimit(); // 获取生产者索引最大限制
231
232
233     pIndex = lvProducerIndex(); // 获取生产者索引
234
235     // 生产索引以2为步长递增，
236     // 第0位标识为resize，所以非扩容场景，不会是奇数，
237     // 扩容的时候，会在offerSlowPath()中扩容线程会将其设为奇数
238     // lower bit is indicative of resize, if we see it we spin until it's cleared
239     if ((pIndex & 1) == 1) {
240         // 奇数代表正在扩容，自旋，等待扩容完成
241         continue;
242     }
243     // pIndex is even (lower bit is 0) -> actual index is (pIndex >> 1)
244     // pIndex 是偶数， 实际的索引值 需要 除以2
245
246     // mask/buffer may get changed by resizing -> only use for array access after success
```

CSDN @40岁资深老架构师尼恩

```
caffeine / src / main / java / com / github / benmanes / caffeine / cache / MpscGrowableArrayQueue.java / BaseMpscLinkedArrayQueue / lvProducerIndex
BlockingQueue.java x BlockingQueue x ArrayBlockingQueue.java x MpscGrowableArrayQueueTest.java x MpscGrowableArrayQueue
560 /**
561  * @return current buffer capacity for elements (excluding next pointer and jump entry) * 2
562  */
563 1 usage 2 implementations coder001
564 protected abstract long getCurrentBufferCapacity(long mask);
565 6 usages coder001
566 static long lvProducerIndex(BaseMpscLinkedArrayQueue<?> self) {
567     return (long) P_INDEX.getVolatile(self);
568 }
569 6 usages coder001
570 static long lvConsumerIndex(BaseMpscLinkedArrayQueue<?> self) {
571     return (long) C_INDEX.getVolatile(self);
572 }
573 1 usage coder001
574 static void soProducerIndex(BaseMpscLinkedArrayQueue<?> self, long v) { P_INDEX.setRelease(self, v); }
575 2 usages coder001
576 static boolean casProducerIndex(BaseMpscLinkedArrayQueue<?> self, long expect, long newValue) {
577     return P_INDEX.compareAndSet(self, expect, newValue);
578 }
579 3 usages coder001
CSDN @40岁资深老架构师尼恩
```

getLongVolatile() 使用了 StoreLoad Barrier，对于 Store1，StoreLoad，Load2 的操作序列，在 Load2 以及后续的读取操作之前，都会保证 Store1 的写入操作对其他处理器可见。

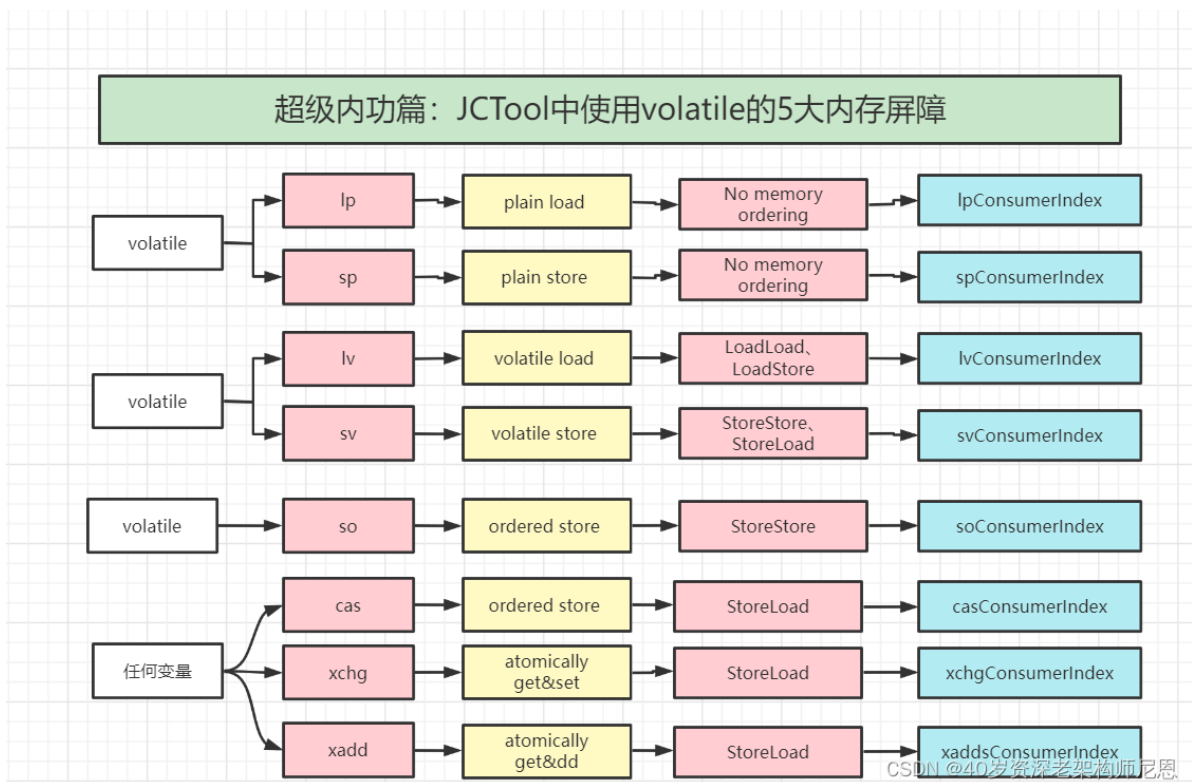
StoreLoad 是四种内存屏障开销最大的，

## JCTool 中的宽松读写命名规约

排他读写：这里把对 volatile 变量的可见写入和可见读取，叫做排他读写，需要保证 强一致，性能低

宽松读写：这里把对 volatile 变量的有序写入和有序读取，叫做宽松读写，不需要保证 强一致，但是纳秒级操作

JCTool 中的宽松读写命名规约，如下：



具体请参考 第 25 章视频 《第 25 章讲义：彻底穿透“缓存之王”Caffeine 底层原理、架构、源码》

## MpscArrayQueue 的优点总结：

MpscQueue 由一系列数组构成，数组的最后一个元素指向下一个数组，形成单向链表。

MpscQueue 全程无锁化，非阻塞，相较于 JDK 提供的同步阻塞队列，性能有很好的提升，这也是 Netty 后来的版本将任务队列替换为 JCTools 的重要原因。

在生产过程中，它用到了大量的 CAS 操作，对于需要做并发控制的地方，确保只有一个线程会执行成功，其他 CAS 失败的线程会自旋重试，全程都是无锁非阻塞的。

不管是扩容，还是等待元素被填充到数组，这些过程都是会极快完成的，因此短暂的自旋会比线程挂起再唤醒效率更高。数组扩容后会在原槽位填充 **JUMP** 元素，消费者遇到该元素就知道要寻找新数组继续消费了。

- 通过 cacheline padding 解决核心属性的伪共享问题。
- 数组的容量设置为 2 的次幂，可以通过位运算快速定位到数组对应下标。
- 入队操作通过 CAS 无锁编程实现，并且通过链式数组结构，和数组节点的动态增加，解决扩容时的元素复制的问题，完成数组的快速扩容，减少 CAS 空自旋。
- 在消费者 poll 过程中，因为只有一个消费者线程，所以整个 poll() 的过程没有 CAS 操作。
- 通过有序的写入元素，去掉 volatile 的 StoreLoad 屏障，实现纳米级别的写入。当然，读取的时候，如果间隔太短，需要进行短时间的自旋。

这个是非常高性能的。这也是 Netty、Caffeine 使用 mpsc 队列的原因

## 参考文献

1. [疯狂创客圈 JAVA 高并发 总目录](#)  
[ThreadLocal（史上最全）](#)
2. [3000 页《尼恩 Java 面试宝典》的 35 个面试专题](#)
3. [价值 10W 的架构师知识图谱](#)
4. [尼恩 架构师哲学](#)
5. [尼恩 3 高架构知识宇宙](#)  
<https://blog.csdn.net/bz120413/article/details/122107790>  
<https://blog.csdn.net/javaesandyou/article/details/123918852>  
<https://blog.csdn.net/javaesandyou/article/details/123918852>  
<https://blog.csdn.net/FreeeLinux/article/details/54897192>  
[https://blog.csdn.net/weixin\\_41605937/article/details/121972371](https://blog.csdn.net/weixin_41605937/article/details/121972371)

## 推荐阅读：

- 《[尼恩 Java 面试宝典](#)》
- 《[Springcloud gateway 底层原理、核心实战 \(史上最全\)](#)》
- 《[Flux、Mono、Reactor 实战（史上最全）](#)》
- 《[sentinel（史上最全）](#)》
- 《[Nacos \(史上最全\)](#)》
- 《[分库分表 Sharding-JDBC 底层原理、核心实战（史上最全）](#)》
- 《[TCP 协议详解 \(史上最全\)](#)》
- 《[clickhouse 超底层原理 + 高可用实操（史上最全）](#)》
- 《[nacos 高可用（图解 + 秒懂 + 史上最全）](#)》
- 《[队列之王：Disruptor 原理、架构、源码 一文穿透](#)》

- [《环形队列、条带环形队列 Striped-RingBuffer（史上最全）》](#)
- [《一文搞定：SpringBoot、SLF4j、Log4j、Logback、Netty 之间混乱关系（史上最全）》](#)
- [《单例模式（史上最全）》](#)
- [《红黑树（图解 + 秒懂 + 史上最全）》](#)
- [《分布式事务（秒懂）》](#)
- [《缓存之王：Caffeine 源码、架构、原理（史上最全，10W 字 超级长文）》](#)
- [《缓存之王：Caffeine 的使用（史上最全）》](#)
- [《Java Agent 探针、字节码增强 ByteBuddy（史上最全）》](#)
- [《Docker 原理（图解 + 秒懂 + 史上最全）》](#)
- [《Redis 分布式锁（图解 - 秒懂 - 史上最全）》](#)
- [《Zookeeper 分布式锁 - 图解 - 秒懂》](#)
- [《Zookeeper Curator 事件监听 - 10 分钟看懂》](#)
- [《Netty 粘包 拆包 | 史上最全解读》](#)
- [《Netty 100 万级高并发服务器配置》](#)
- [《Springcloud 高并发 配置（一文全懂）》](#)