24 Netty的队列有何不一样

更新时间：2020-08-12 09:33:08

> 人的差异在于业余时间。——爱因斯坦

前言

你好，我是彤哥。

上一节，我们一起学习了 Netty 中的快男 ——FastThreadLocal，通过源码剖析，我们知道，如果使用不当，FastThreadLocal 也可能会变成慢男，不过，这都不是事儿，因为我们只要记住跟着 FastThreadLocalThread 一起使用就可以了，这个原则很简单。

其实呢，追踪一下常用的 Spring 等框架，会发现正常运转的情况下，一个线程最多也就三四十个 ThreadLocal 变量，那么，Netty 为何还要大费周章搞一个 FastThreadLocal 呢？这是由于 Netty 的使用场景导致的，不管是对象池还是内存池，亦或者是前面讲到的请求处理的过程，都大量使用了线程本地变量，且操作频繁，而 Java 原生的 ThreadLocal 使用的是线性探测法实现的哈希表，使得哈希冲突的概率太大且解决冲突的方式也不友好，且解决冲突之后更容易引起哈希冲突，所以，Netty 必须定义一个全新的 ThreadLocal 用来存储本地变量，简单点说，就是 Java 原生的 ThreadLocal 太慢了，无法应对 Netty 这种多缓存高频率的场景。

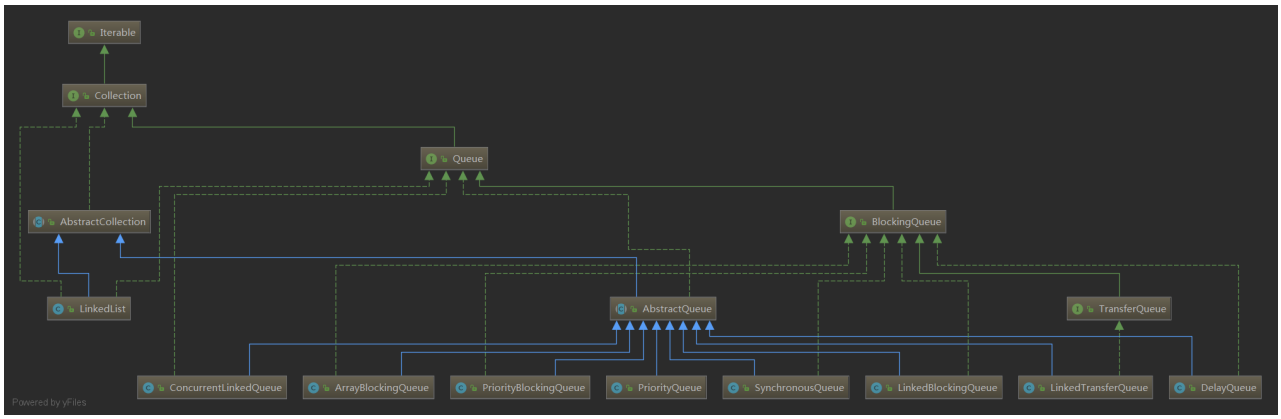
上面我们提到了“场景”两个字，其实，在 Netty 中，很多地方都针对特定的场景使用了特定的技术，比如，我们今天要说的一揽子队列 ——MpscArrayQueue、MpscChunkedArrayQueue、MpscUnboundedArrayQueue、MpscAtomicArrayQueue、MpscGrowableAtomicArrayQueue、MpscUnboundedAtomicArrayQueue 等。

可以发现，这些队列都有统一的前缀 Mpsc-，它是什么意思呢？这些队列又是使用在什么样的场景呢？相比于 Java 原生的队列，这些队列又有哪些好处呢？它们又是怎么实现的呢？

让我们带着这些问题进入今天的学习吧。

Java 原生队列回顾

首先，我们来回顾下 Java 原生的队列，也就是下面这张图，它覆盖了 Java 中所有的原生队列：



对于这些 Java 原生的队列，我把它们分成这么几大类：

1. 非并发安全的队列

    - LinkedList，没错，你没看错，LInkedList 确实实现了 Queue 接口，可以把它当作一个队列来使用；
    - PriorityQueue，优先级队列，使用 “堆” 这种数据结构实现的队列，主要用于堆排序、中位数、99% 位数等场景；

2. 阻塞队列

    - ArrayBlockingQueue，最简单的阻塞队列，使用数组实现，有界，使用一个 ReentrantLock 及两个 Condition 控制并发安全，效率低下；
    - LinkedBlockingQueue，使用链表实现，有界或无界，使用两个 ReentrantLock 分别控制入队和出队，效率相对 ArrayBlockingQueue 要高一些；
    - SynchronizedQueue，俗称无缓冲队列，里面不存储任何元素，所有入队的元素都移交给另一个线程来处理，如果放入元素时没有线程来消费，那么，调用者线程会阻塞；同样地，如果取元素时没有生产者放入元素，那么消费线程也会阻塞；
    - PriorityBlockingQueue，优先级队列的阻塞模式，可在多线程环境中用于堆排序、中位数、99% 位数等场景；
    - LinkedTransferQueue，这是个强大的阻塞队列，它使用了一种叫作 “双重队列” 的数据结构，而且它相当于是 LinkedBlockingQueue、SynchronousQueue（公平模式）、ConcurrentLinkedQueue 三者的集合体，且比它们更高效；
    - DelayQueue，延时队列，它在优先级队列的基础上加入了 “延时” 的概念，出队时，如果堆顶的元素还没有到期，是不会出队的，主要运用在定时任务的场景中，比如，Java 的定时任务线程池 ScheduledThreadPoolExecutor，不过它是自己又实现了一遍延时队列，叫作 DelayedWorkQueue，而没有使用现成的 DelayQueue。

3. 并发安全的队列

    - ConcurrentLinkedQueue，它是并发安全的队列却不是阻塞队列，内部使用 自旋 + CAS 实现，是一种无

锁队列，但是它无法使用在线程池中。

> 阻塞队列一定是并发安全的队列，关于以上所有队列的源码分析，可以参考文末链接解锁。

大部分情况下，使用 Java 原生的队列就能够达到我们的要求了，但是，对于一些特殊的场景，使用 Java 原生的队列性能就略显低下，所以，又衍生了一些第三方的框架专门实现特定的队列，来提高特定场景下的性能问题。

这些第三方框架中比较著名的有两个：Disruptor 和 jctools。

Disruptor，基于环形数组和 LMAX 架构实现，性能杠杠滴。

jctools，它把队列分成四种使用场景：

- 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

针对这四种场景 jctools 又实现了各种口味不同的队列，比如，我们今天的主角 ——MPSC 队列，没错，Netty 中使用的就是 jctools 中的 MPSC 队列，而且 Netty 只使用了 MPSC 这一种队列，更过分地，在 Netty 打包的时候使用了一个叫作 "shade" 的 maven 插件，直接把使用到的 jctools 中代码打包到了 Netty 的 `io.netty.util.internal.shaded.org.jctools.queues` 包下面，而没有使用到的其它的代码并没有打包过来，所以，在 `io.netty.util.internal.shaded.org.jctools.queues` 包下面只能看到跟 MPSC 队列相关的代码。

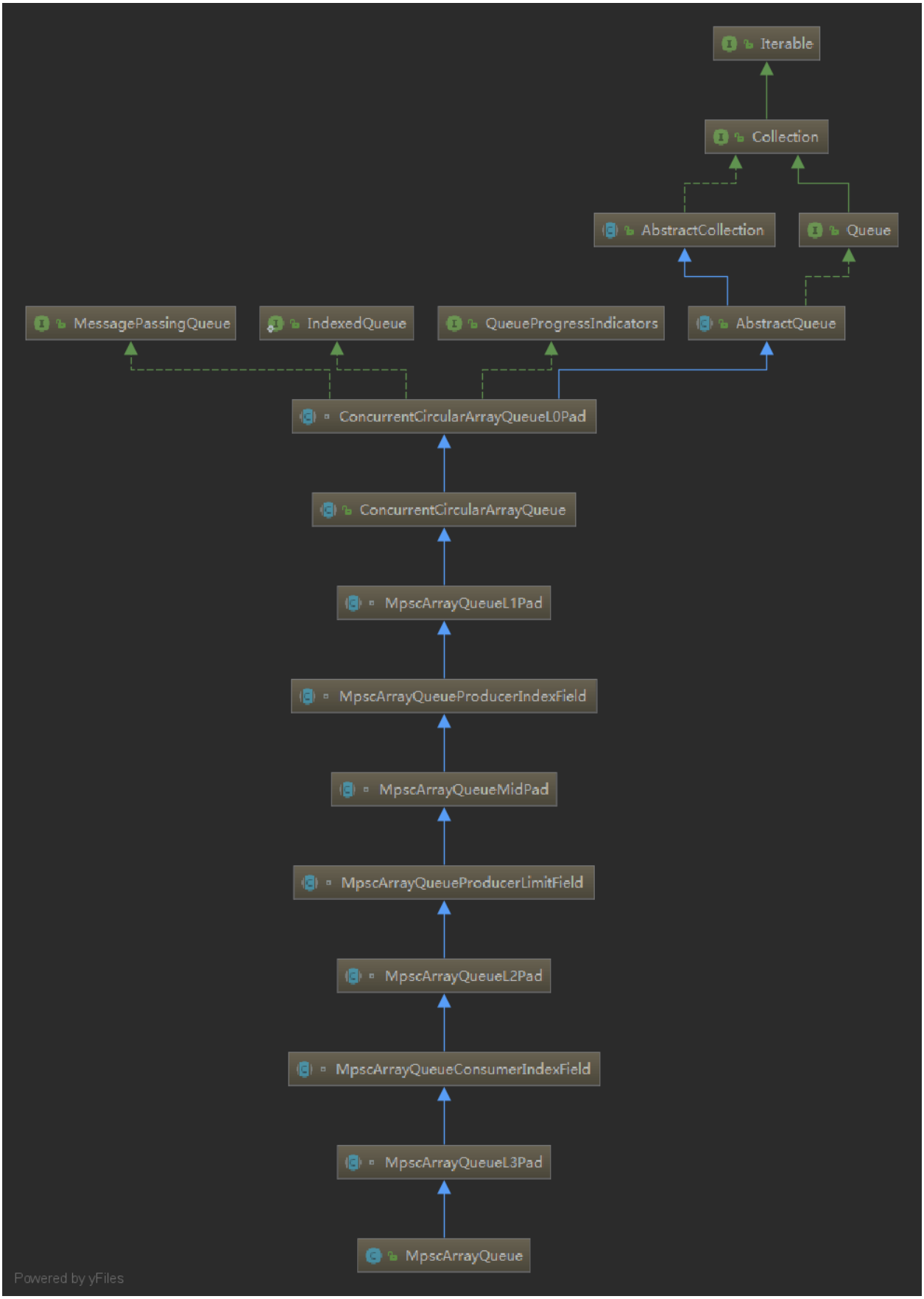
> shade 插件的全称为 maven-shade-plugin，参数 minimizeJar 用于控制只打包用到的类文件，详情见 netty-common 工程的 pom.xml 文件的插件部分。

今天，我们就从这些 MPSC 队列中挑一个来讲解，看看它的实现原理以及源码，挑来挑去挑谁呢，就 MpscArrayQueue 吧，因为它相对来说比较简单，看懂了它，其它的都不在话下，好了，开始喽 ^^

MpscArrayQueue

按照我们以往的套路，对于这样的单个类，最适合使用从宏观到微观的分析方法，所以，我们先来看看 MpscArrayQueue 的继承体系。

继承体系



纳尼！What！什么！点解咁复杂！没错，就是这么复杂，让我们把这些类分个类：

- -Pad 结尾的类，它们里面全都是一堆的 long 型变量，是用来避免伪共享的；
- -Field 结尾的类，它们里面存储着主要的字段，这些字段通过 - Pad 结尾的类通过 long 型变量隔开，以达到避免伪共享的目的；
- ConcurrentCircularArrayQueue，数据存储的地方，从名字可以看出，它也是通过环形数组实现的；

- MpscArrayQueue，对外暴露的可使用的类，里面不包含任何字段；
- 其它，最上层的就是一些接口了，可以看到，最终，MpscArrayQueue 是实现了 Java 原生的 Queue 接口的；

既然，是通过继承这种手段来避免伪共享的，那么，我们把这些字段压缩到一个类中看看长什么样子呢？

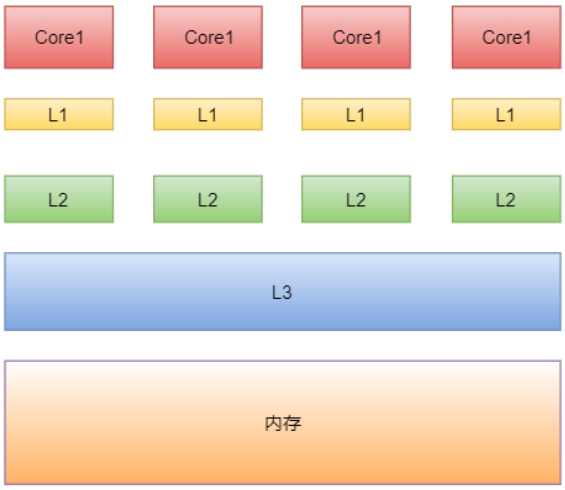
```
public class FakeMpscArrayQueue<E> {
    long p01, p02, p03, p04, p05, p06, p07;
    long p10, p11, p12, p13, p14, p15, p16, p17;
    // 掩码，用来计算数组下标，加1就成了容量
    protected final long mask;
    // 存储数据的环形数组
    protected final E[] buffer;
    long p00, p01, p02, p03, p04, p05, p06, p07;
    long p10, p11, p12, p13, p14, p15, p16;
    // 生产者索引，有volatile
    private volatile long producerIndex;
    long p01, p02, p03, p04, p05, p06, p07;
    long p10, p11, p12, p13, p14, p15, p16, p17;
    // 生产者索引的最大值，有volatile
    private volatile long producerLimit;
    long p00, p01, p02, p03, p04, p05, p06, p07;
    long p10, p11, p12, p13, p14, p15, p16;
    // 消费者索引，无volatile
    protected long consumerIndex;
    long p01, p02, p03, p04, p05, p06, p07;
    long p10, p11, p12, p13, p14, p15, p16, p17;
}
```

既然，这里又提到了伪共享，那我们就简单介绍下伪共享到底是何方神圣。

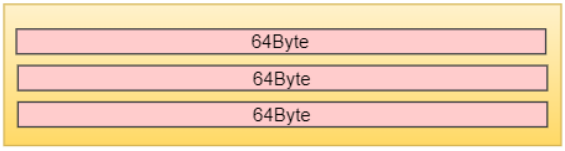
题外话 —— 伪共享

在现在的 CPU 架构下，一般地，一台计算机都有多个 CPU 核心，叫作多核 CPU，这些 CPU 都要从一块叫作内存的地方读取数据，经过加工处理，再写回到内存中，如果每次读写数据都跟内存进行交互，太慢了，你可以想像成内存跟硬盘的关系，所以，为了加快 CPU 的处理速度，人们就给 CPU 安上了缓存，一般地，现代处理器都具有三级缓存，这三缓存也有个关系，越接近 CPU 的缓存越快越贵容量越小，越远离 CPU 的缓存越慢越便宜容量越大。

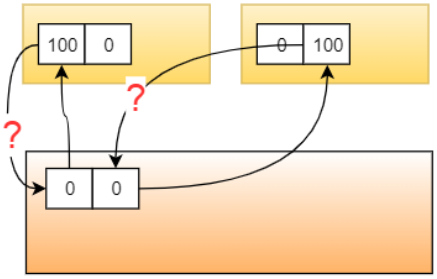
比如，对于一台 4 核 CPU 的计算机，它的缓存布局可能是这样的：



这样，在处理数据的时候，CPU 就加载内存中的一小块数据到 CPU 缓存中，处理完毕并不用立马写回内存，等下次再读取或修改同一片内存区域的数据时，直接走缓存就好了，这样就极大地提高了数据处理的速度。刚才有提到每次加载一小块数据，那么，这个“一小块”是多大呢？通常地，现代 CPU 架构为 64 个字节。



似乎很完美，试想这样一个问题，在内存中有两个相临的变量 x 和 y，一个线程一直在对 x 进行 ++ 操作，一个线程一直在对 y 进行 ++ 操作，会出现怎样地后果呢？



假设两个变量初始值为 0，各自增 100 次，因为是两个不同的线程处理，所以这两个线程可能处于不同的 CPU 核中，根据上面的理论，CPU 每次加载 64 字节的数据到缓存中，所以，x 和 y 始终一起被加载到不同的缓存中，那么，各自修改完了如何写回主内存呢？发现没法写回了是不是？因为写回也是整个缓存行一起写回的，不管先写回哪个，都会被后写回的覆盖。

为了解决这种问题，有两种策略：

1. 给这个缓存行对应的内存块加锁，每次读写数据的时候都从主内存重新读取，写完之后立马写回主内存，多个线程处理同一块内存区域数据的时候排队进行，这样数据肯定就准确了；
2. 把 x 和 y 分隔开，不要让它们相临，让它们始终不会同时被加载到同一个缓存行中，只需要在它们之间补足 64 字节，它们自然就被隔开了，永远不会加载到同一个缓存行中；

两种方案都是可行的。

对于第一种方案，相当于缓存行永远失效，形同虚设了，这种锁又有另外一个名字 —— 内存屏障（Memory Barrier）或者内存栅栏 （Memory Fence），在现代 CPU 架构下，内存屏障主要分为读屏障（Load Memory Barrier）和写屏障（Store Memory Barrier）：

- 读屏障，每次都从主内存读取最新的数据；
- 写屏障，将缓存写入到主内存；

内存屏障还有个重要的功能，防止重排序，即不会把内存屏障前后的指令进行重排序。

使用内存屏障这种技术，又引来了新的问题，每次对 x 的操作，同时对 y 产生了影响，反之亦然，相当于 x 和 y 变成了一种共生的状态，但是实际上他们却没有任何关系，这种不同线程对同一块内存区域（缓存行）的不同变量的操作产生了互相影响的现象，就叫作**伪共享**（False Sharing）。为了解决伪共享带来的问题，就引出了第二种方案。

对于第二种方案，这样的玩法叫作加 Padding，在两个变量之间加一系列无用的变量，使得两个变量永远不会被加载到同一个缓存行，但是，它也有个问题，试想如果两个线程同时修改 x，它就无法处理了，此时，就只能使用第一种方案了。

在 Java 中，这种加 Padding 的玩法主要有三种实现方式：

> 变量前后添加 N 个 long 类型，N 的取值有两种说法，一种是 7，一种是 15，因为内存布局是按 8 字节对齐的，所以加上 7 个 long 正好等于 64 字节，也就是一个缓存行的大小，可以保证这个变量与其它变量分隔开，15 的说法是为了避免相邻扇区预取导致的伪共享冲突，在 Disruptor 框架中使用的是 7，在 jctools 中使用的是 15；
>
> 使用继承且在父子类中加上 padding，这样是为了防止内存布局重排序，比如，下面这个类，会把 byte 类型的 b 存储在 long 类型的前面，因为对象头占用 12 字节（压缩后），byte 类型占用 1 字节，这样只需要被 3 个字节就可以了，如果不做这种重排序，对象头需要补齐 4 个字节，而 byte 类型需要补齐 7 个字节，造成空间浪费；

```
class MemoryLayout{
    private long a;
    private byte b;
}
```

- 使用 @sun.misc.Contended 注解，不过这是 Java8 新增的注解，所以，无法兼容之前的版本，现在大部分开源框架还没有使用这个注解；

好了，针对伪共享的问题，我们就简单介绍这么多内容，还是回到正题。

回归正题

如果把避免伪共享添加的这些字段去掉，那么，MpscArrayQueue 就只剩下这么几个字段了：

```
public class FakeMpscArrayQueue<E> {
    // 掩码，用来计算数组下标，加1就成了容量
    protected final long mask;
    // 存储数据的环形数组
    protected final E[] buffer;
    // 生产者索引，有volatile
    private volatile long producerIndex;
    // 生产者索引的最大值，有volatile
    private volatile long producerLimit;
    // 消费者索引，无volatile
    protected long consumerIndex;
}
```

可以发现，producerIndex 和 producerLimit 加了 volatile，而 consumerIndex 却没有，这是为什么呢？请记住我们的场景是 MPSC，多生产者单消费者，所以，生产者的索引修改必须立马对其它线程可见，而只有一个消费者，它并不需要对别的线程立马可见，当然，生产者在特定情况下也是需要 consumerIndex 的最新值的，比如，环形数组的头性相接了，它是怎么实现的呢？让我们跟着源码来一起学习吧。

源码剖析

调试用例

这一节的用例就比较好写了，因为 MpscArrayQueue 实现了 Queue 接口，所以，我们只要按其它的队列一样来写用例就可以了：

```
public class MpscArrayQueueTest {

    public static final MpscArrayQueue<String> QUEUE = new MpscArrayQueue<>(5);

    public static void main(String[] args) {
        // 入队，如果队列满了则会抛出异常
        QUEUE.add("1");
        // 入队，返回是否成功
        QUEUE.offer("2");
        QUEUE.offer("3");
        QUEUE.offer("4");

        // 存储了多少元素
        System.out.println("队列大小：" + QUEUE.size());
        // 容量，可以存储多少元素，会按2次方对齐，所以这里为8
        System.out.println("队列容量" + QUEUE.capacity());

        // 出队，如果队列为空则会抛出异常
        System.out.println("出队：" + QUEUE.remove());
        // 出队，如果队列为空返回null
        System.out.println("出队：" + QUEUE.poll());
        // 查看队列头元素，如果队列为空则会抛出异常
        System.out.println("查看队列头元素：" + QUEUE.element());
        // 查看队列头元素，如果队列为空则返回null
        System.out.println("查看队列头元素：" + QUEUE.peek());
    }
}
```

接口 Queue 中对于入队、出队、查看队首元素各定义了两种方法，一类是抛出异常，一类是返回特定值：

	抛出异常	返回特定值
入队	add(e)	offer(e)
出队	remove()	poll()
查看队首元素	element()	peek()

其中，抛出异常的方法最终也还是调用的返回特定值的方法，而查看队首元素跟出队方法是比较类似的，所以，这里我们主要看 offer (e)、poll () 这两个方法的源码实现。

入队 offer (e)

好了，让我们先来看看入队方法 offer (e) 的实现，在 `QUEUE.offer("2");` 处打一个断点，此时，已经入队一个元素了，跟踪进去：

```
// 入队
@Override
public boolean offer(final E e)
{
    // 空值检查
    if (null == e)
    {
        throw new NullPointerException();
    }

    // 掩码
    final long mask = this.mask;
    // 生产者的最大的索引值，初始时为传入的容量，即5
    long producerLimit = lvProducerLimit(); // LoadLoad
    long pIndex;
    do
```

```
    {
        // 生产者索引
        pIndex = lvProducerIndex(); // LoadLoad
        // 如果生产者索引达到了最大值，防止追尾
        if (pIndex >= producerLimit)
        {
            // 消费者索引，以volatile的形式获取，保证获取的是最新的值
            final long cIndex = lvConsumerIndex(); // LoadLoad
            // 修改为当前消费者的索引加上数组的大小
            producerLimit = cIndex + mask + 1;

            // 如果依然达到了最大值，则返回false，表示队列满了，再放元素就追尾了
            if (pIndex >= producerLimit)
            {
                return false; // FULL :(
            }
            else
            {
                // 否则更新最大索引为新值
                soProducerLimit(producerLimit);
            }
        }
    }
    // CAS更新生产者索引，更新成功了则跳出循环，说明数组中这个下标被当前这个生产者占有了
    // 此时即使更新索引成功了，数组中依然还没有放入元素
    // 如果更新失败，说明其它生产者（线程）先占用了这个位置，重新来过
    while (!casProducerIndex(pIndex, pIndex + 1));
    // 计算这个索引在数组中的下标偏移量
    final long offset = calcElementOffset(pIndex, mask);
    // 将元素放到这个位置
    soElement(buffer, offset, e); // StoreStore
    // 入队成功
    return true; // AWESOME :)
}
// lv=load valatile
// 读取producerLimit
protected final long lvProducerLimit()
{   // producerLimit本身就是volatile修饰的
    // 所以不用像下面的consumerIndex一样通过UNSAFE.getLongVolatile()一样来读取
    return producerLimit;
}
// 读取producerIndex
public final long lvProducerIndex()
{
    // producerIndex本身就用volatile修饰了
    return producerIndex;
}
// 读取consumerIndex
public final long lvConsumerIndex()
{   // 以volatile的形式加载consumerIndex
    // 此时，可以把consumerIndex想像成前面加了volatile
    // 会从内存读取最新的值
    return UNSAFE.getLongVolatile(this, C_INDEX_OFFSET);
}
// so=save ordered
// 保存producerLimit
protected final void soProducerLimit(long newValue)
{
    // 这个方法会加StoreStore屏障
    // 会把最新值直接更新到主内存中，但其它线程不会立即可见
    // 其它线程需要使用volatile语义才能读取到最新值
    // 这相当于是一种延时更新的方法，比volatile语义的性能要高一些
    UNSAFE.putOrderedLong(this, P_LIMIT_OFFSET, newValue);
}
// 修改数组对应偏移量的值
public static <E> void soElement(E[] buffer, long offset, E e)
{
    // 与上面同样的方法，比使用下标更新数组元素有两个优势
    // 1. 使用Unsafe操作内存更新更快
```

```
    // 2. 使用putOrderedObject会直接更新到主内存，而使用下标不会立马更新到主内存
    UNSAFE.putOrderedObject(buffer, offset, e);
}
// CAS更新producerIndex
protected final boolean casProducerIndex(long expect, long newValue)
{
    // CAS更新
    return UNSAFE.compareAndSwapLong(this, P_INDEX_OFFSET, expect, newValue);
}
```

这段入队的方法看似简单，实则蕴含大量的底层知识和优化技巧，让我们来看几个问题：

- 为什么需要 producerLimit，拿 producerIndex 与 consumerIndex 直接比较行不行？
- 很多方法后面写了 LoadLoad、StoreStore，它们是什么意思？
- Unsafe 的新方法 putOrderedObject () 和 getLongVolatile ()？

我们先来看第一个问题：**为什么要使用 producerLimit 呢？**

其实不使用 producerLimit 也是可以的，只不过这样的话，就需要每次都使用 volatile 语义获取 consumerIndex 的值，再用这个值加上数组的大小，就是 producerIndex 能达到的最大值，这跟把 consumerIndex 声明为 volatile 就没有什么分别了，也就无法达到提高性能的目的了。使用 producerLimit 的好处是明显的，差不多一轮才需要获取一次 consumerIndex 的值，相当于减少了消费端的竞争。

接着看看第二个问题：**LoadLoad、StoreStore 是什么意思？**

可以把 LoadLoad 看成是读屏障，表示每次都从主内存读取最新值，StoreStore 看成是写屏障，每次都把最新值写入到主内存。如果一个线程使用 StoreStore 屏障把最新值写入主内存，另一个线程只需要使用 LoadLoad 屏障就可以读取到最新值了，它们俩往往结合着来使用。

最后一个问题：**Unsafe 的新方法 putOrderedObject () 和 getLongVolatile ()**？

其实，在 Unsafe 中有五组相似的方法：

- putOrderedXxx ()，使用 StoreStore 屏障，会把最新值更新到主内存，但不会立即失效其它缓存行中的数据，是一种延时更新机制；
- putXxxVolatile ()，使用 StoreLoad 屏障，会把最新值更新到主内存，同时会把其它缓存行的数据失效，或者说会刷新其它缓存行的数据；
- putXxx (obj, offset)，不使用任何屏障，更新对象对应偏移量的值；
- getXxxVolatile ()，使用 LoadLoad 屏障，会从主内存获取最新值；
- getXxx，不使用任何屏障，读取对象对应偏移量的值；

从性能方面来说的话，putOrderedXxx () 用得好的话，性能会比 putXxxVolatile () 高一些，但是，如果用的不好的话，可能会出现并发安全的问题，所以，个人请谨慎使用，即使使用了，也要做好并发安全的测试。

OK，基础知识也补齐了，如果还看不懂，不要紧，先跳过去，我们再来看看出队方法，等看完出队方法了，我们使用脑补法来模拟一下入队出队的实现。

出队 **poll ()**

同样地，在 `System.out.println("出队：" + QUEUE.poll());` 这行打一个断点，并跟踪进去：

```
// 出队
@Override
public E poll()
{
    // 读取consumerIndex的值，注意这里是lp不是lv
    final long cIndex = lpConsumerIndex();
    // 计算在数组中的偏移量
    final long offset = calcElementOffset(cIndex);
    // 存储元素的数组
    final E[] buffer = this.buffer;

    // 取元素，前面通过StoreStore写入的，这里通过LoadLoad取出来的就是最新值
    E e = lvElement(buffer, offset); // LoadLoad
    if (null == e)
    {
        // 有一种例外，还记得上面入队的时候吗？
        // 是先更新了producerIndex的值，再把更新元素到数组中的。
        // 如果在两者之间，进行了消费，则此处是无法获取到元素的
        // 所以需要进入下面的判断
        // 判断consumerIndex是否等于producerIndex
        // 只要两则不相等，就可以再消费元素
        if (cIndex != lvProducerIndex())
        {
            // 使用死循环来取元素，直到取到为止
            do
            {
                e = lvElement(buffer, offset);
            }
            while (e == null);
        }
        else
        {
            // 如果两个索引相等了，说明没有元素了，返回null
            return null;
        }
    }

    // 更新取出的位置元素为null，注意是sp，不是so
    spElement(buffer, offset, null);
    // 修改consumerIndex的索引为新值，使用StoreStore屏障，直接更新到主内存
    soConsumerIndex(cIndex + 1); // StoreStore
    // 返回出队的元素
    return e;
}
// lp=load plain，简单读取
protected final long lpConsumerIndex()
{
    return consumerIndex;
}
// sp=store plain，简单存储
public static <E> void spElement(E[] buffer, long offset, E e)
{
    UNSAFE.putObject(buffer, offset, e);
}
```

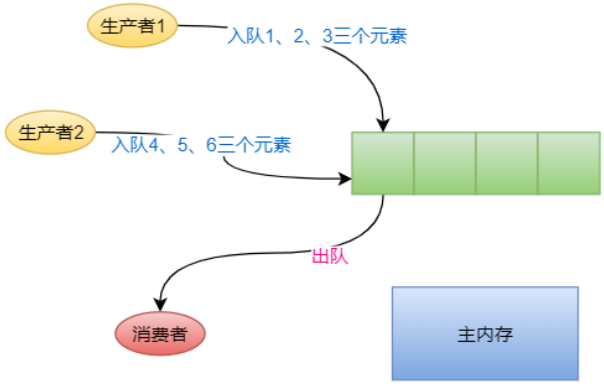
时刻要记住消费者只有一个，所以，消费端完全不需要使用任何锁或者 CAS 操作，但是，生产者端是有可能读取 consumerIndex 的值的，所以，使用 StoreStore 屏障修改它的值即可。

还有一种例外，是生产者端先更新 producerIndex，再更新数组元素，这里使用死循环不断读取直到读取到为止。

入队出队的代码都分析完毕了，可以看到，整体的逻辑非常少，我算了下，入队出队两者加一起的主体逻辑都不到 100 行，但是，里面蕴含了大量的底层知识，为了更好地理解这种队列，我决定使用脑补法来模拟一下入队出队的过程。

脑补入队出队过程

为了简单点，我们假设队列的长度为 4，一共入队 5 个元素，并出队 2 个元素，2 个生产者：



假设入队的过程为 1、4 同时请求入队，3 入队，2、5 同时请求入队，连续出队 4 次，6 入队，让我们分析下整个过程：

1. 初始时，pIndex=0，cIndex=0，pLimit=4；（p=producer，c=consumer，下同)
2. 1、4 同时请求入队，两者拿到 pIndex 都是 0，pLimit 都是 4，所以，都判断为小于 pLimit，可以入队，但是在 CAS 更新 pIndex 的时候必然会一个成功一个失败，假设生产者 1 成功了，1 成功入队，4 入队失败，进入循环，重新读取 pIndex 为 1（因为 pIndex 为 volatile，所以 CAS 更新为 1 后立即对生产者 2 可见），判断依然小于 pLimit，可以入队，CAS 更新 pIndex 为 2；
3. 3 请求入队，没人跟它竞争，直接入队成功，pIndex 被更新为 3；
4. 2、5 同时请求入队，两者拿到的 pIndex 都是 3，pLimit 都是 4，所以，都判断为小于 pLimit，可以入队，但是在 CAS 更新 pIndex 的时候必然会有一个成功一个失败，假设生产者 2 成功了，5 成功入队，2 入队失败，进入循环，重新读取 pIndex 为 4，等于 pLimit 了（重新计算 pLimit 还是 4），所以直接返回 false，2 入队失败了；
5. 出队，此时，读取到 cIndex 为 0，读取数组下标为 0 位置的元素，也就是 1，更新下标 0 处元素为 null，并更新 cIndex 为 1，因为使用的是 StoreStore 屏障，所以，主内存中的 cIndex 也为 1；
6. 出队，出队，出队，同样地，最后，主内存中的 cIndex 为 4；
7. 6 请求入队，此时，读取到的 pIndex 为 4，pLimit 也为 4，pIndex 大于等于了 pLimit，重新读取 cIndex 的值为 4，并重新计算 pLimit 为 8，再判断 pIndex 小于 pLimit 了，更新主内存中的 pLimit 为新值 8，CAS 更新 pIndex 的值为 5 成功，6 入队成功，且在数组下标为 0 的位置。

整个过程就是这样，咦，6 是怎么跑到下标为 0 的位置的？

这个其实是通过计算偏移量算出来的，即下面这段代码：

```
public static long calcElementOffset(long index, long mask)
{
  // REF_ARRAY_BASE，基础地址，数组在内存中的地址
  // REF_ELEMENT_SHIFT，可以简单地看作一个元素占用多少字节
  // 64位系统中一个引用对象占用64位，也就是8字节，但是压缩模式下占用4字节
  // index & mask 计算数组下标
  // 比如数组大小为4，mask就为3，pIndex为4时，4&3=100&11=0
  return REF_ARRAY_BASE + ((index & mask) << REF_ELEMENT_SHIFT);
}
```

好了，到这里，关于 MpscArrayQueue 的剖析就结束了，开篇我们说了，它还有很多兄弟类 —— MpscChunkedArrayQueue、MpscUnboundedArrayQueue、MpscAtomicArrayQueue、MpscGrowableAtomicArrayQueue、MpscUnboundedAtomicArrayQueue，相信有了本节的基础，分析这些类对你来说不是什么难事了。

> 带 Atomic 的类，是表示在 Netty 无法使用 Unsafe 的情况下使用 Atomic 原子类来做替代方案。

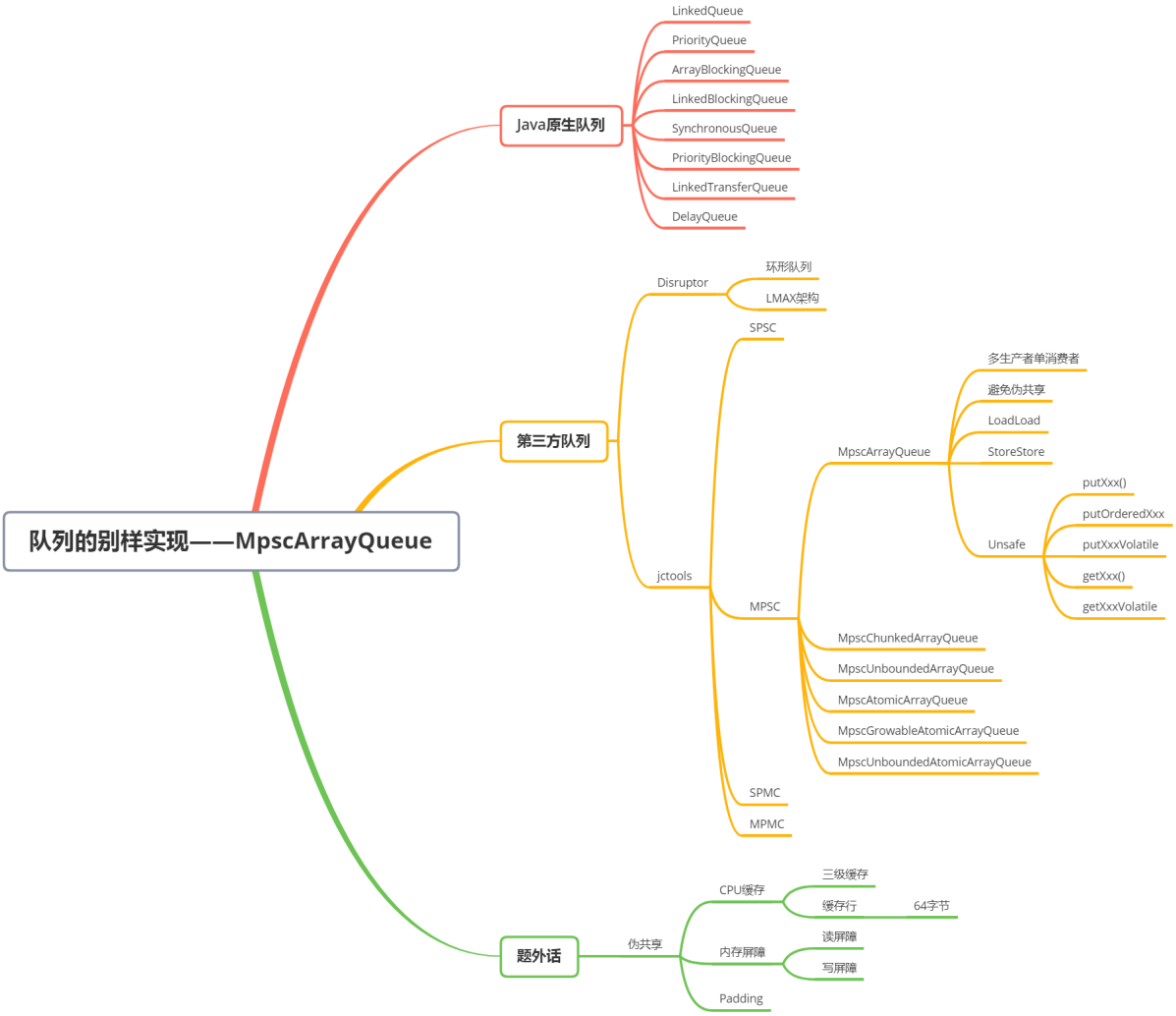
后记

本节，我们一起回顾了 Java 原生队列，并从源码级别剖析了一个 MPSC 队列 ——MpscArrayQueue，从中学到了很多 Java 底层的知识，相信有了本节的学习，下次再看到跟 Unsafe、伪共享相关的代码，你一定能够自己剖析地很好。

其实，Netty 中不仅使用了非常高效的 jctools 提供的队列，它还对 Java 原生的很多功能做了增强，比如前面学过的 ByteBuf、即将学到的 Future、线程池等等。

下一节，我们就来一起学习 Netty 增强的 Future，提前告诉你，Netty 虽然增强了 Future，但是它一不小心就搞出了个 Bug 呢，敬请期待。

思维导图



特别说明

在上一节中，InternalThreadLocalMap 中也使用了 padding 来消除伪共享，但是，那个用法实在不知道它是为了保护哪个变量被伪共享，而且，经过测试，也没发现加这段 padding 有明显的性能提升。

所以，正确的用法是本节介绍的这种用法，大家后面自己使用的时候也可以参考本节的用法，或者使用 Java8 的新注解 @sun.misc.Contended 来实现。

参考链接

死磕 java 集合之 LinkedList 源码分析

死磕 java 集合之 PriorityQueue 源码分析

死磕 java 集合之 ArrayBlockingQueue 源码分析

死磕 java 集合之 LinkedBlockingQueue 源码分析

死磕 java 集合之 SynchronousQueue 源码分析

死磕 java 集合之 PriorityBlockingQueue 源码分析

死磕 java 集合之 LinkedTransferQueue 源码分析

死磕 java 集合之 DelayQueue 源码分析

死磕 java 集合之 ConcurrentLinkedQueue 源码分析

杂谈 什么是伪共享？

}

23 Netty的FastThreadLocal到底快在哪里

25 Netty的Future是如何做到简捷易用的