

# Disruptor（一）Disruptor概念和RingBuffer数据结构

**谈到并发程序设计，有几个概念是避免不了的。**

**1、锁：** 锁是用来做并发最简单的方式，当然其代价也是最高的。内核态的锁的时候需要操作系统进行一次上下文切换，等待锁的线程会被挂起直至锁释放。在上下文切换的时候，cpu之前缓存的指令和数据都将失效，对性能有很大的损失。用户态的锁虽然避免了这些问题，但是其实它们只是在没有真实的竞争时才有效。

**2、CAS：** CAS的涵义不多介绍了。使用CAS时不像上锁那样需要一次上下文切换，但是也需要处理器锁住它的指令流水线来保证原子性，并且还要加上Memory Barrier来保证其结果可见。

**3、MemoryBarrier:** 大家都知道现代CPU是乱序执行的，也就是程序顺序与实际的执行顺序很可能是不一致的。在单线程执行时这不是个问题，但是在多线程环境下这种乱序就可能会对执行结果产生很大的影响了。memory barrier提供了一种控制程序执行顺序的手段, 关于其更多介绍，可以参考 <http://en.wikipedia.org/wiki/Memory_barrier>。

**4、CacheLine：** cacheLine解释起来其实很简单，就是CPU在做缓存的时候有个最小缓存单元，在同一个单元内的数据被同时被加载到缓存中，充分利用cacheLine可以大大降低数据读写的延迟，错误利用cache line也会导致缓存不同替换，反复失效。

**并发内存队列时需要考虑的问题**

**1、数据结构问题**

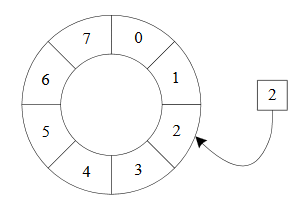
是选用定长的数组还是可变的链表

**2、并发控问题**

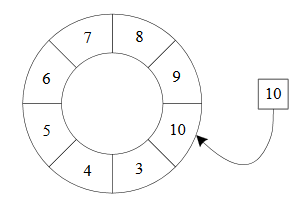
使用锁还是CAS操作，是使用粗粒度的一把锁，还是将队列的头、尾、和容量三个变量分开控制，即使分开，能不能避免它们落入同一个Cache line中。数据的入队和出队都是很耗时的，尤其在性能要求极高的场景中，这种消耗更显得奢侈。

**Disruptor数据结构RingBuffer**

RingBuffer是一个环（首尾相接的环），你可以把它用做在不同上下文（线程）间传递数据的buffer。基本来说，RingBuffer拥有一个序号，这个序号指向数组中下一个可用的元素。

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160904002507_89205.png)

随着你不停地填充这个buffer（可能也会有相应的读取），这个序号会一直增长，直到绕过这个环。

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160904002519_47624.png)

要找到数组中当前序号指向的元素，可以通过mod操作

sequence mod array.length = array.index

以上面的RingBuffer为例（java的mod语法）：11 % 8 = 3，RingBuffer槽的个数是2的N次方，更有利于基于二进制的计算机进行计算，，这样计算余数只需要通过位操作index & ( size – 1 )就能够得到实际的index。和维基百科里面的关于[环形buffer](https://en.wikipedia.org/wiki/Circular_buffer)的实现方式，与其最大的区别在于：没有尾指针。我们只维护了一个指向下一个可用位置的序号。

RingBuffer本身并不控制是否需要重叠，对于如何避免RingBuffer产生重叠，以及如何对RingBuffer进行读写操作，移到了数据结构（RingBuffer）的外（队列通常注重维护队列的头尾元素，添加和删除元素等)，由于生产者和消费者是分开考虑和控制的，因此有可能能够通过一个核心的环形队列来表示全部的依赖关系，可以大大提高吞吐，降低延迟。

**RingBuffer的优点**

**1、在可靠消息传递方面有很好的性能**

不删除buffer中的数据（这些数据一直存放在buffer中，直到新的数据覆盖他们），这样当另外一个服务通过nak (拒绝应答信号) 告诉我们没有成功收到消息时，我们能够重新发送给他们。

**2、底层使用数组，访问比链表快**

数组内元素的内存地址的连续性存储的，而且有一个容易预测的访问模式，这是对CPU缓存友好的—也就是说，在硬件级别，数组中的元素是会被预加载的，因此在RingBuffer当中，cpu无需时不时去主存加载数组中的下一个元素。（因为只要一个元素被加载到缓存行，其他相邻的几个元素也会被加载进同一个缓存行）。

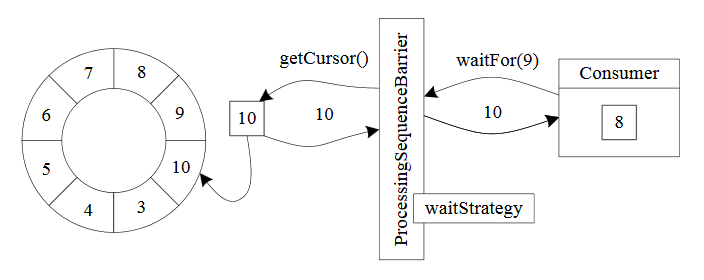
**3、避免GC问题**

可以为数组预先分配内存，使得数组对象一直存在（除非程序终止），这就意味着不需要花大量的时间用于垃圾回收。不像链表那样，需要为每一个添加到其上面的对象创造节点对象—对应的，当删除节点时，需要执行相应的内存清理操作。

# Disruptor（二）RingBuffer读取

**1、消费者通过ProcessingSequenceBarrier读取数据**

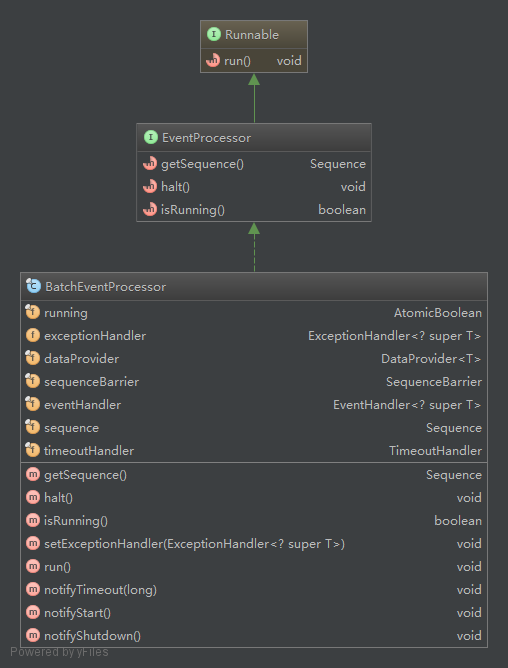
能够读取数据的前提是数据已经写入到Ring Buffer中，关于数据的写入，后面一章节会详细讲解。

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160904005456_25007.png)

RingBuffer的元素的大小是2的n次方（上面ringBufferSize为8，从序号0开始）。消费者(Consumer)是一个想从RingBuffer里读取数据的线程，它可以通过访问ProcessingSequenceBarrier对象和RingBuffer进行交互。消费者也需要知道它将要处理的序号，每个消费者都需要找到下一个它要访问的序号。在上面的例子中，消费者处理完了RingBuffer里序号8之前（包括8）的所有数据，那么它期待访问的下一个序号是9。

**2、消费者BatchEventProcessor**

关于消费者如何通过调用SequenceBarrier对象的waitFor()方法，传递它所需要的下一个序号。本章节以BatchEventProcessor批量事件处理器为例进行讲解，首先查看类图。



拿到了数据后，消费者(Consumer)会更新自己的标识(cursor)，消费者(Consumer)现在只需要通过简单通过ProcessingSequenceBarrier拿到可用的Ringbuffer中的Sequence序号就可以可以读取数据了。因为这些新的节点的确已经写入了数据（RingBuffer本身的序号已经更新），而且消费者对这些节点的唯一操作是读而不是写，因此访问不用加锁。不仅代码实现起来可以更加安全和简单，而且不用加锁使得速度更快。另一个好处是可以用多个消费者(Consumer)去读同一个RingBuffer，不需要加锁，也不需要用另外的队列来协调不同的线程(消费者)。这样你可以在Disruptor的协调下实现真正的并发数据处理。

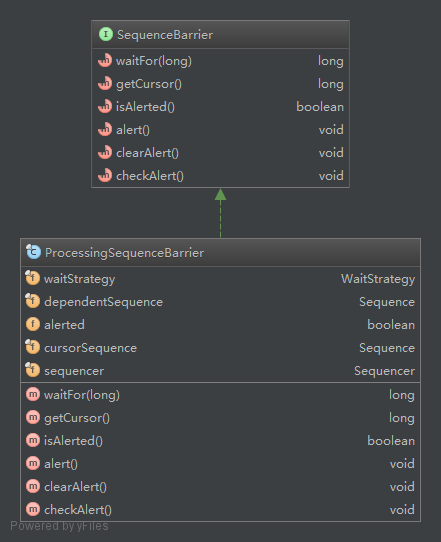
**3、ProcessingSequenceBarrier获取可用序号**

在上面的BatchEventProcessor中的run方法中有如下调用

final long availableSequence = sequenceBarrier.waitFor(nextSequence);

获取RingBuffer最大可访问的availableSequence序号，在上面的例子中是10。

首先看下ProcessingSequenceBarrier的类图。

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160904005744_40210.png)

其实现了SequenceBarrier接口，用于和RingBuffer之间进行交互，下面主要看下构造函数和waitFor函数。

**4、WaitStrategy策略**

waitFor函数的主要功能为获取到可用的sequence并返回给事件处理器。SequenceBarrier内部有一个WaitStrategy方法来决定它如何等待这个序号，我现在不会去描述它的细节，代码的注释里已经概括了每一种WaitStrategy的优点和缺点，目前的实现方式主要有以下几种，后续会做详细介绍。

* BlockingWaitStrategy
* BusySpinWaitStrategy
* LiteBlockingWaitStrategy
* PhasedBackoffWaitStrategy
* SleepingWaitStrategy
* TimeoutBlockingWaitStrategy
* YieldingWaitStrategy

# Disruptor（三）RingBuffer单生产者写入

上一章主要介绍了消费者从RingBuffer读取数据，本章主要介绍单个生产者如何向RingBuffer数据写入数据。在RingBuffer数据写入过程中如何不要让Ring重叠，写入后通知消费者，生产者一端的批处理，以及多个生产者如何协同工作。

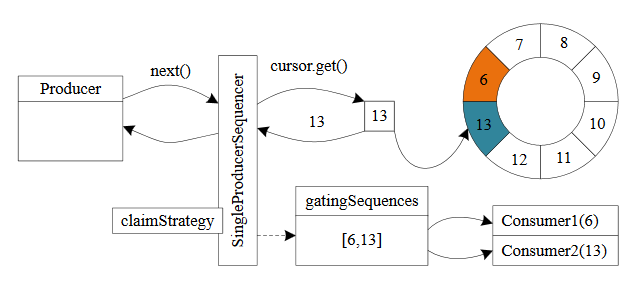
在RingBuffer写入数据的过程涉及到两阶段提交(two-phasecommit)

1）生产者需要申请buffer里的下一个节点。

2）当生产者向节点写完数据，需要调用调用publish发布数据。

**1、单个生产者SingleProducerSequencer数据写入**

在后台由ProducerSequencer负责所有的交互细节，来从RingBuffer中找到下一个节点，然后才允许生产者向它写入数据。

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160905170454_94205.png)

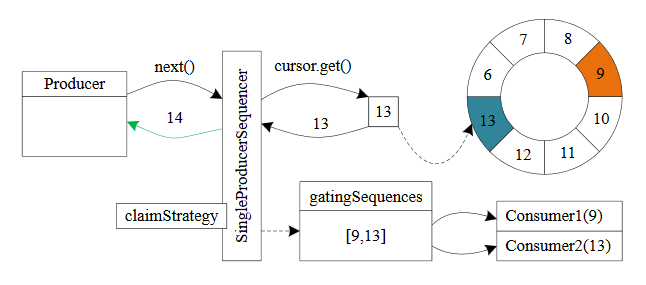
在图中一个生产者写入RingBuffer，SingleProducerSequencer对象拥有所有正在访问RingBuffer的消费者gatingSequences列表（区别于队列需要追踪队列的头和尾，而且它们有时候会指向相同的位置）。，Disruptor中由消费者负责通知它们处理到了哪个序列号，而不是RingBuffer。

如果想确定我们没有让RingBuffer重叠，需要检查所有的消费者们都读到了哪里。在上图中有2个消费者，一个消费者顺利的读到了最大序号13（用蓝色高亮），第二个消费者有点儿落后停在序号6。因此消费者2在赶上消费者1之前要跑完整个RingBuffer一圈的距离。

现在生产者想要写入RingBuffer中序号6占据的节点，因为它是RingBuffer当前游标的下一个节点。但是SingleProducerSequencer明白现在不能写入，因为有一个消费者正在占用它。所以SingleProducerSequencer停下来自旋(spins)，等待，直到那个消费者离开。

**2、申请下一个节点**

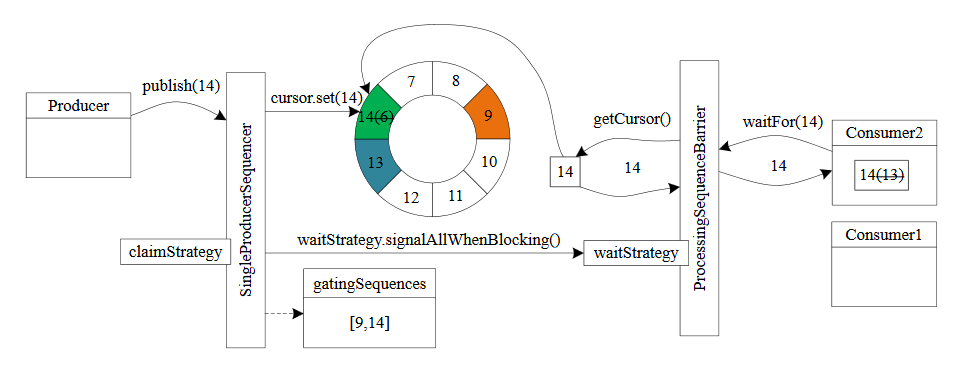
现在可以想像消费者2已经处理完了一批节点，并且向前移动了它的序号。可能它挪到了序号9（因为消费端的批处理方式，现实中我会预计它到达13）

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160905170514_64706.png)

上图显示了当消费者2挪动到序号9时发生的情况。SingleProducerSequencer会看到下一个节点序号6那个已经可以用了。它会抢占这个节点上的Entry（我还没有特别介绍Entry对象，基本上它是一个放写入到某个序号的RingBuffer数据的桶），把下一个序号（14）更新成Entry的序号，然后把Entry返回给生产者。生产者可以接着往Entry里写入数据。

3、**提交新的数据**

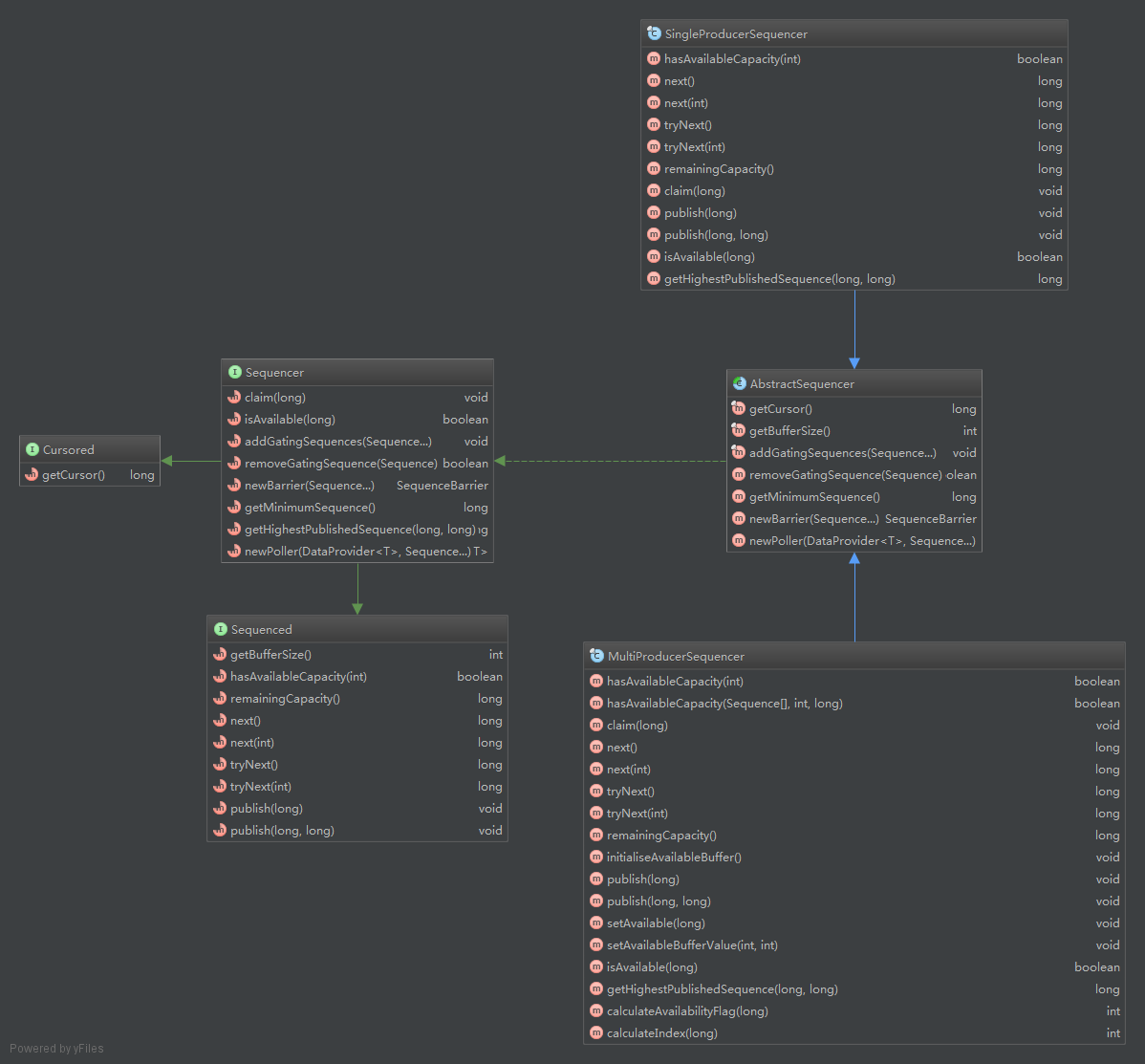
将生产的数据提交，通知消费之。

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160905170530_95347.png)

绿色表示最近写入的Entry，序号是14，通过publish方法提交，设置RingBuffer的cursor为14，通知消费者14被更新了，可以读取了（不同的WaitStrategy实现以不同的方式来实现提醒，取决于它是否采用阻塞模式）。现在消费者2可以读Entry14的数据进行消费了。

看完上面的原理后下面分析SingleProducerSequencer是如何获取序号和提交数据的。

**4、SingleProducerSequencer生产者类图**

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160905170543_89702.png)

SingleProducerSequencer继承AbstractSequencer，实现了Sequencer接口。

Sequencer提供增加删除消费者序列，创建SequenceBarrier，获取最小序号，和最大发布的序号。

Cursored获取当前的游标。

Sequenced获取当前ringbuffer大小，获取想一个序号，以及提交数据接口。

**5、消费者和生产者直接的关联**

首先看下AbstractSequencer中定义

// 生产者的当前的游标位置

protected final Sequence cursor = new Sequence(Sequencer.INITIAL\_CURSOR\_VALUE);

// 消费者当前处理的序号集合

protected volatile Sequence[] gatingSequences = new Sequence[0];

**由于volatile只能保存可见性和禁止编译器优化，当时不能保证互斥性，多线程并发读写的话会有问题。**

private static final AtomicReferenceFieldUpdater<AbstractSequencer, Sequence[]> SEQUENCE\_UPDATER = AtomicReferenceFieldUpdater.newUpdater(AbstractSequencer.class, Sequence[].class, "gatingSequences");

**使用AtomicReferenceFieldUpdater原子字段更新解决多线程更新gatingSequences问题**

具体实现参照SequenceGroups中使用CAS进行更新。

public final void addGatingSequences(Sequence... gatingSequences) {

SequenceGroups.addSequences(this, SEQUENCE\_UPDATER, this, gatingSequences);

}

public boolean removeGatingSequence(Sequence sequence) {

return SequenceGroups.removeSequence(this, SEQUENCE\_UPDATER, sequence);

}

**6、生产者使用next获取下一个可用的序号**

public long next(int n) {

if (n < 1) {

throw new IllegalArgumentException("n must be > 0");

}

// 当前的最小序号（单个生产者为生产者的游标）

long nextValue = this.nextValue;

// 下一个序号

long nextSequence = nextValue + n;

// 重叠点位置

long wrapPoint = nextSequence - bufferSize;

// 缓存的消费者处理的序号

long cachedGatingSequence = this.cachedValue;

// wrapPoint > cachedGatingSequence,

// 重叠位置大于缓存的消费者处理的序号，说明有消费者没有处理完成，不能够防止数据

// cachedGatingSequence > nextValue

// 只会在https://github.com/LMAX-Exchange/disruptor/issues/76情况下存在

if (wrapPoint > cachedGatingSequence || cachedGatingSequence > nextValue) {

long minSequence;

// 等待不重叠后退出循环

while (wrapPoint > (minSequence = Util.getMinimumSequence(gatingSequences, nextValue))) {

// 通知消费者处理事件

waitStrategy.signalAllWhenBlocking();

// 生产者等待的时候后自旋，后续需要使用策略

LockSupport.parkNanos(1L);

}

// 缓存消费者和生产者的最小序号

this.cachedValue = minSequence;

}

// 设置生产者下一个可用的的序号

this.nextValue = nextSequence;

return nextSequence;

}

**7、生产者使用publish发布数据**

public void publish(long sequence) { // 设置生产者的游标序号

cursor.set(sequence);

// 通知消费者处理事件

waitStrategy.signalAllWhenBlocking();

}

当发布数据后，消费者sequenceBarrier.waitFor(nextSequence)就能够获取RingBuffer最大可访问的availableSequence序号，处理数据了。

**8、消费者消费数据**

再回忆下ProcessingSequenceBarrier的waitFor函数，其中调用到了sequencer.getHighestPublishedSequence(sequence,  
availableSequence);

public long waitFor(final long sequence)

throws AlertException, InterruptedException, TimeoutException {

// 检查clert异常

checkAlert();

// 通过waitStrategy策略获取可用的序号,cursorSequence为当前的Sequence，dependentSequence为依赖的Sequence[]

long availableSequence = waitStrategy.waitFor(sequence, cursorSequence, dependentSequence, this);

// 产生比预期的sequence小,可能序号被重置回老的的oldSequence值

//可参考https://github.com/LMAX-Exchange/disruptor/issues/76

if (availableSequence < sequence) {

return availableSequence;

}

// 获取最大的可用的已经发布的sequence，可能比sequence小

// 会在多生产者中出现，当生产者1获取到序号13，生产者2获取到14；生产者1没发布，生产者2发布，会导致获取的可用序号为12，而sequence为13

return sequencer.getHighestPublishedSequence(sequence, availableSequence);

}

public long getHighestPublishedSequence(long lowerBound, long availableSequence) {

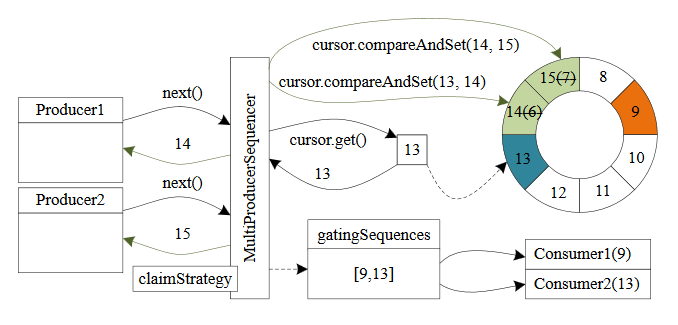
return availableSequence;

}

在SingleProducerSequencer的getHighestPublishedSequence方法中直接返回可用的availableSequence，通知消费者消费数据。通过以上步骤，生产者和消费者就协同起来了。

# Disruptor（四）RingBuffer多生产者写入

**1、多生产者MultiProducerSequencer申请下一个节点**

**[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160905225141_94026.png)**

和单生产者不同的是在next方法中会直接通过cursor.compareAndSet(current,  
next)设置生产者的游标cursor的sequence。大家很可能会问设置了生产者的游标后，没有提交数据之前，多生产者场景中消费者是否就能够获取到数据，答案是否定的，在MultiProducerSequencer实现的getHighestPublishedSequence的方法和单生产者有所区别，后面会详细讲解。

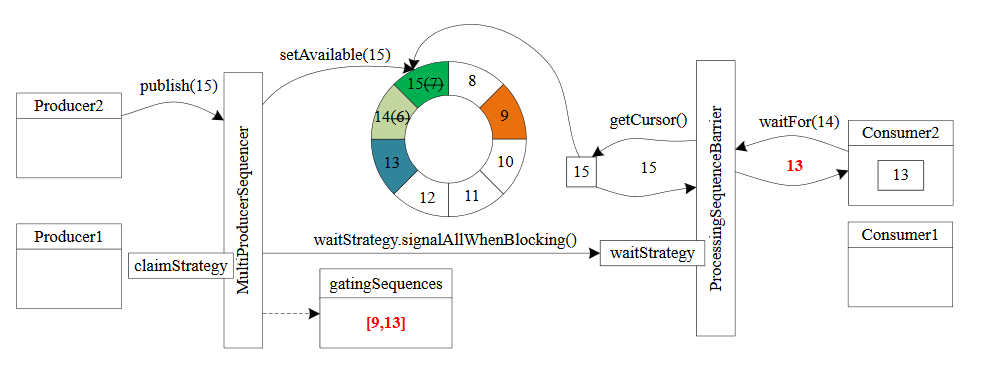
2、**多生产者MultiProducerSequencer提交数据**

和单生产者的区别是使用setAvailable将数据设置成可用状态。

在多个生产者的场景下，还需要其他东西来追踪序号。这个序号是指当前可写入的序号。注意这和“向RingBuffer的游标加1”不一样，如果你有一个以上的生产者同时在向RingBuffer写入，就有可能出现某些Entry正在被生产者写入但还没有提交的情况。

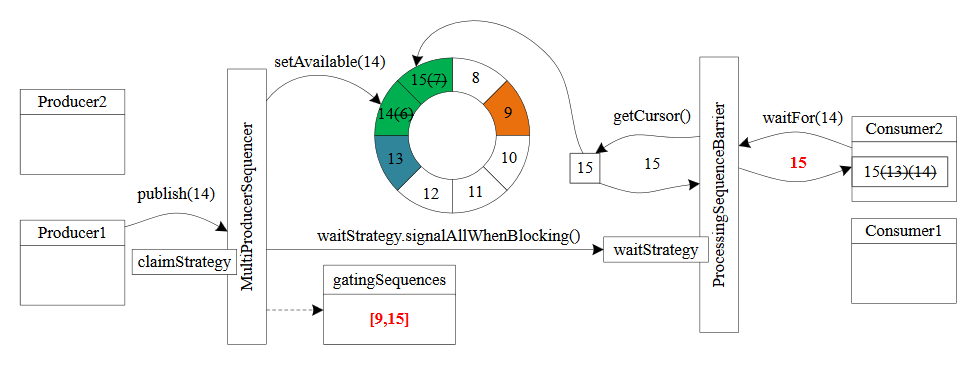
生产者1拿到序号14，生产者2拿到序号15。现在假设生产者1因为某些原因没有来得及提交数据。

生产者2通过setAvailable(15)请求完成提交数据，如图所示。

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160905225159_84973.png)

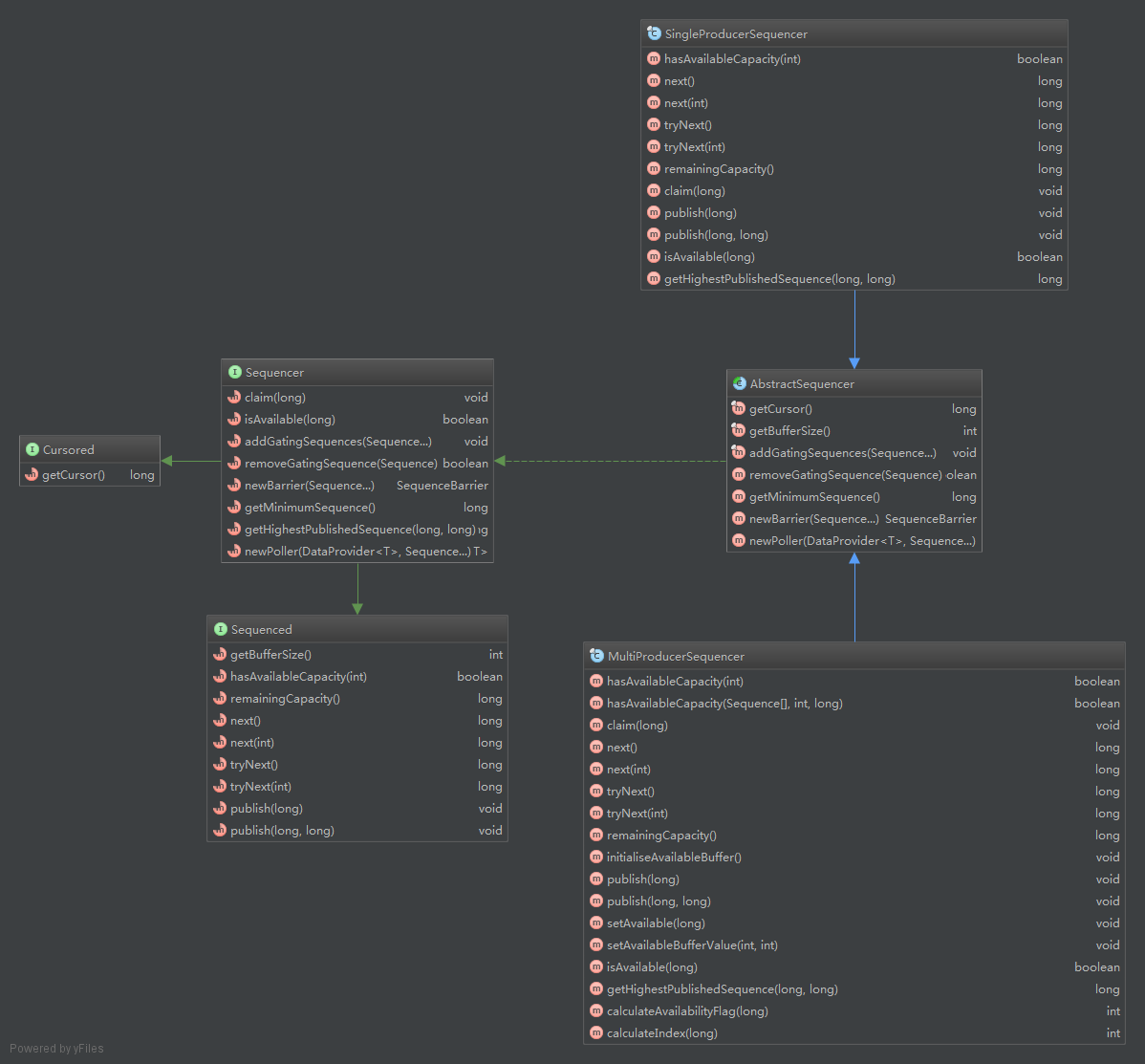
当这个时候消费者通过waitFor(14)，返回的结果会为13，不错任何事件处理。

当生产者1通过setAvailable(14)请求完成提交数据，如图所示。

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160905225215_39379.png)

以BatchEventProcessor的run实现会处理14和15位置上的数据，在下一次通过waitFor(16)获取可用的数据。

3、**MutiProducerSequencer生产者类图。**

[](http://style.alicharles.com/alicharles/wp-content/uploads/2016/09/20160905225327_64542.png)

MutiProducerSequencer继承AbstractSequencer，实现了Sequencer接口。

Sequencer提供增加删除消费者序列，创建SequenceBarrier，获取最小序号，和最大发布的序号。

Cursored获取当前的游标。

Sequenced获取当前ringbuffer大小，获取想一个序号，以及提交数据接口。

消费者和生产者之间的关联和单生产者一样，不做重复介绍。

**4、多生产者通过next获取下一个可用的序号**

public long next(int n) {

if (n < 1) {

throw new IllegalArgumentException("n must be > 0");

}

long current;

long next;

do {

// ringbuffer当前生产者cursor

current = cursor.get();

// 下一个可用的序号

next = current + n;

// 重叠点位置

long wrapPoint = next - bufferSize;

// 缓存的消费者处理的序号

long cachedGatingSequence = gatingSequenceCache.get();

// wrapPoint > cachedGatingSequence,

// 重叠位置大于缓存的消费者处理的序号，说明有消费者没有处理完成，不能够防止数据

// cachedGatingSequence > nextValue

// 只会在https://github.com/LMAX-Exchange/disruptor/issues/76情况下存在

if (wrapPoint > cachedGatingSequence || cachedGatingSequence > current) {

// 获取消费者和生产者最小的序号

long gatingSequence = Util.getMinimumSequence(gatingSequences, current);

// 仍然重叠

if (wrapPoint > gatingSequence) {

// 通知消费者处理事件

waitStrategy.signalAllWhenBlocking();

// 生产者等待的时候后自旋，后续需要使用策略

LockSupport.parkNanos(1);

continue;

}

// 没有重叠的话，设置消费者缓存

gatingSequenceCache.set(gatingSequence);

}

// 没有重叠，直接将RingBuffer的序号设置成next

else if (cursor.compareAndSet(current, next)) {

break;

}

}

while (true);

// 返回可用的序号

return next;

}

**5、多生产者通过publish提交数据**

public void publish(final long sequence) {

// 将sequence设置为可用状态

setAvailable(sequence);

// 通知消费者处理事件

waitStrategy.signalAllWhenBlocking();

}

多生产者在获取序号next方法中就已经设置了cusor，提交数据的时候是将该sequence设置成可用状态，才能够被消费者使用。

**6、消费者消费数据**

再回忆下ProcessingSequenceBarrier的waitFor函数，其中调用到了sequencer.getHighestPublishedSequence(sequence,availableSequence);

public long waitFor(final long sequence)

throws AlertException, InterruptedException, TimeoutException {

// 检查clert异常

checkAlert();

// 通过waitStrategy策略获取可用的序号,cursorSequence为当前的Sequence，dependentSequence为依赖的Sequence[]

long availableSequence = waitStrategy.waitFor(sequence, cursorSequence, dependentSequence, this);

// 产生比预期的sequence小,可能序号被重置回老的的oldSequence值

//可参考https://github.com/LMAX-Exchange/disruptor/issues/76

if (availableSequence < sequence) {

return availableSequence;

}

// 获取最大的可用的已经发布的sequence，可能比sequence小

// 会在多生产者中出现，当生产者1获取到序号13，生产者2获取到14；生产者1没发布，生产者2发布，会导致获取的可用序号为12，而sequence为13

return sequencer.getHighestPublishedSequence(sequence, availableSequence);

}

获取最大的可用的已经发布的sequence

public long getHighestPublishedSequence(long lowerBound, long availableSequence) {

for (long sequence = lowerBound; sequence <= availableSequence; sequence++) {

// 判断是否可用

if (!isAvailable(sequence)) {

return sequence - 1;

}

}

return availableSequence;

}

其中判断isAvailable通过availableBuffer进行判断

public boolean isAvailable(long sequence) {

// 计算((int) sequence) & indexMask的索引index

int index = calculateIndex(sequence);

// 计算(int) (sequence >>> indexShift) ringbuffer的slot的设置次数

int flag = calculateAvailabilityFlag(sequence);

// index在数组中的偏移量

long bufferAddress = (index \* SCALE) + BASE;

// 如果和flag相等，说明可用

return UNSAFE.getIntVolatile(availableBuffer, bufferAddress) == flag;

}

内部使用的变量如下。

// availableBuffer跟踪每个ringbuffer的slot槽的状态，是否可用

private final int[] availableBuffer = new int[bufferSize]; // 初始值为-1

private final int indexMask = bufferSize - 1;

private final int indexShift = Util.log2(bufferSize);

通过以上方式就能够判断当前的sequence是否可用了。

通过在MutiProducerSequencer的getHighestPublishedSequence方法中直接返回可用的availableSequence，通知消费者消费数据，生产者和消费者就协同起来了。