主讲老师：Fox

课前须知：

1. Disruptor主要是用于金融领域的交易撮合系统，同学们了解会用就行
2. 关于 Disruptor的一些针对高性能比较精髓的设计，同学们务必掌握。

、

有道云链接：http://note.youdao.com/noteshare?id=4f9aa74cba58a258347a7913a1ebd8bf&sub=CDC29898C44F489BAD4A6C72A8BBDE30

Disruptor简介

Disruptor是英国外汇交易公司LMAX开发的一个高性能队列，研发的初衷是解决内存队列的延迟问题（在性能测试中发现竟然与I/O操作处于同样的数量级）。基于Disruptor开发的系统单线程能支撑每秒600万订单，2010年在QCon演讲后，获得了业界关注。2011年，企业应用软件

专家Martin Fowler专门撰写长文介绍。同年它还获得了Oracle官方的Duke大奖。目前，包括Apache Storm、Camel、Log4j 2在内的很多知名项目都应用了Disruptor以获取高性能。注意，这里所说的队列是系统内部的内存队列，而不是Kafka这样的分布式队列。

Github：https://github.com/LMAX-Exchange/disruptor

Disruptor实现了队列的功能并且是一个有界队列，可以用于生产者-消费者模型。

juc下队列存在的问题

队列	描述
ArrayBlockingQueue	基于数组结构实现的一个有界阻塞队列
LinkedBlockingQueue	基于链表结构实现的一个无界阻塞队列，指定容量为有界阻塞队列
PriorityBlockingQueue	支持按优先级排序的无界阻塞队列
DelayQueue	基于优先级队列（PriorityBlockingQueue）实现的无界阻塞队列
SynchronousQueue	不存储元素的阻塞队列
LinkedTransferQueue	基于链表结构实现的一个无界阻塞队列
LinkedBlockingDeque	基于链表结构实现的一个双端阻塞队列

https://www.processon.com/view/link/618ce3941e0853689b0818e2

1. juc下的队列大部分采用加ReentrantLock锁方式保证线程安全。在稳定性要求特别高的系统中，为了防止生产者速度过快，导致内存溢出，只能选择有界队列。

2. 加锁的方式通常会严重影响性能。线程会因为竞争不到锁而被挂起，等待其他线程释放锁而唤醒，这个过程存在很大的开销，而且存在死锁的隐患。

3. 有界队列通常采用数组实现。但是采用数组实现又会引发另外一个问题false sharing(伪共享)。

Disruptor的设计方案

Disruptor通过以下设计来解决队列速度慢的问题：

- **环形数组结构**

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（空间局部性原理）。

- **元素位置定位**

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。

- **无锁设计**

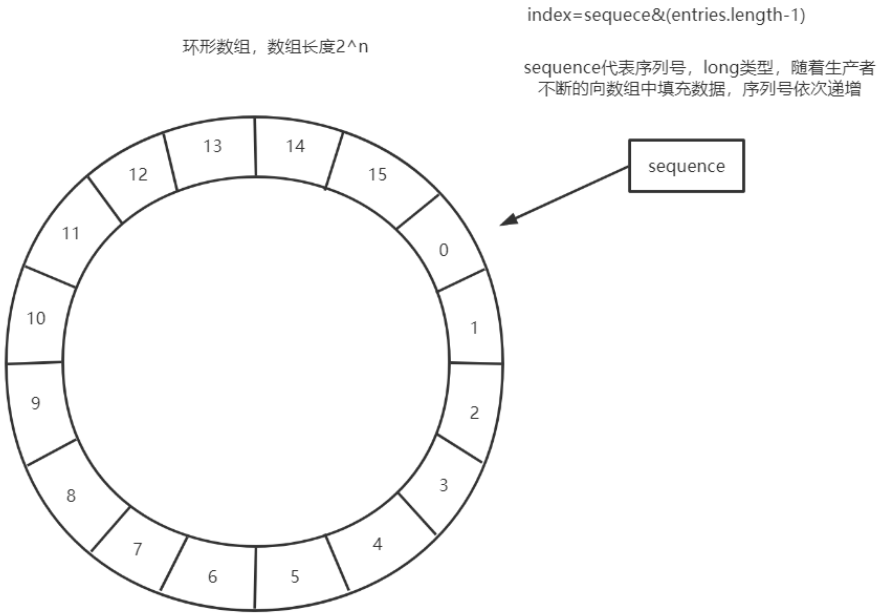
每个生产者或者消费者线程，会先申请可以操作的元素在数组中的位置，申请到之后，直接在该位置写入或者读取数据。

- **利用缓存行填充解决了伪共享的问题**
- **实现了基于事件驱动的生产者消费者模型（观察者模式）**

消费者时刻关注着队列里有没有消息，一旦有新消息产生，消费者线程就会立刻把它消费

RingBuffer数据结构

使用RingBuffer来作为队列的数据结构，RingBuffer就是一个可自定义大小的环形数组。除数组外还有一个序列号(sequence)，用以指向下一个可用的元素，供生产者与消费者使用。原理图如下所示：



- Disruptor要求设置数组长度为2的n次幂。在知道索引(index)下标的情况下，存与取数组上的元素时间复杂度只有O(1)，而这个index我们可以通过序列号与数组的长度取模来计算得出，index=sequence % entries.length。也可以用位运算来计算效率更高，此时array.length必须是2的幂次方，index=sequece&(entries.length-1)
- 当所有位置都放满了，再放下一个时，就会把0号位置覆盖掉

思考：能覆盖数据是否会导致数据丢失呢?

当需要覆盖数据时，会执行一个策略，Disruptor给提供多种策略，比较常用的：

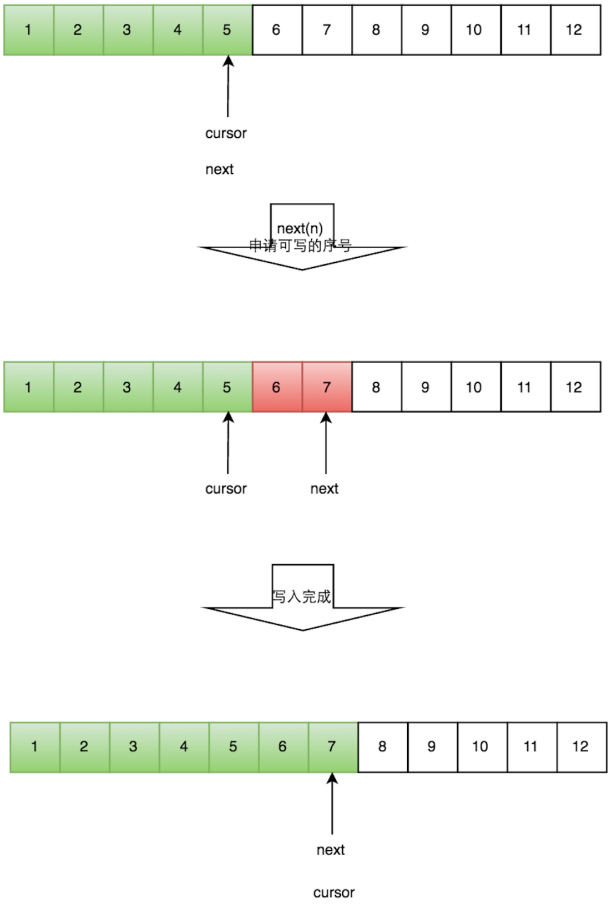
- **BlockingWaitStrategy策略**，常见且默认的等待策略，当这个队列里满了，不执行覆盖，而是阻塞等待。使用ReentrantLock+Condition实现阻塞，最节省cpu，但高并发场景下性能最差。适合CPU资源紧缺，吞吐量和延迟并不重要的场景
- **SleepingWaitStrategy策略**，会在循环中不断等待数据。先进行自旋等待如果不成功，则使用Thread.yield()让出CPU,并最终使用LockSupport.parkNanos(1L)进行线程休眠，以确保不占用太多的CPU资源。因此这个策略会产生比较高的平均延时。典型的应用场景就是异步日志。
- **YieldingWaitStrategy策略**，这个策略用于低延时的场合。消费者线程会不断循环监控缓冲区变化，在循环内部使用Thread.yield()让出CPU给别的线程执行时间。如果需要一

个高性能的系统，并且对延时比较有严格的要求，可以考虑这种策略。

- **BusySpinWaitStrategy策略**: 采用死循环，消费者线程会尽最大努力监控缓冲区的变化。对延时非常苛刻的场景使用，cpu核数必须大于消费者线程数量。推荐在线程绑定到固定的CPU的场景下使用

一个生产者单线程写数据的流程

1. 申请写入m个元素；
2. 若是有m个元素可以写入，则返回最大的序列号。这里主要判断是否会覆盖未读的元素；
3. 若是返回的正确，则生产者开始写入元素。



多个生产者写数据的流程

多个生产者的情况下，会遇到“如何防止多个线程重复写同一个元素”的问题。Disruptor的解决方法是每个线程获取不同的一段数组空间进行操作。这个通过CAS很容易达到。只需要在分配元素的时候，通过CAS判断一下这段空间是否已经分配出去即可。

但是会遇到一个新问题：如何防止读取的时候，读到还未写的元素。Disruptor在多个生产者的情况下，引入了一个与Ring Buffer大小相同的buffer：available Buffer。当某个位置写入成功的时候，便把availble Buffer相应的位置置位，标记为写入成功。读取的时候，会遍历available Buffer，来判断元素是否已经就绪。

消费者读数据

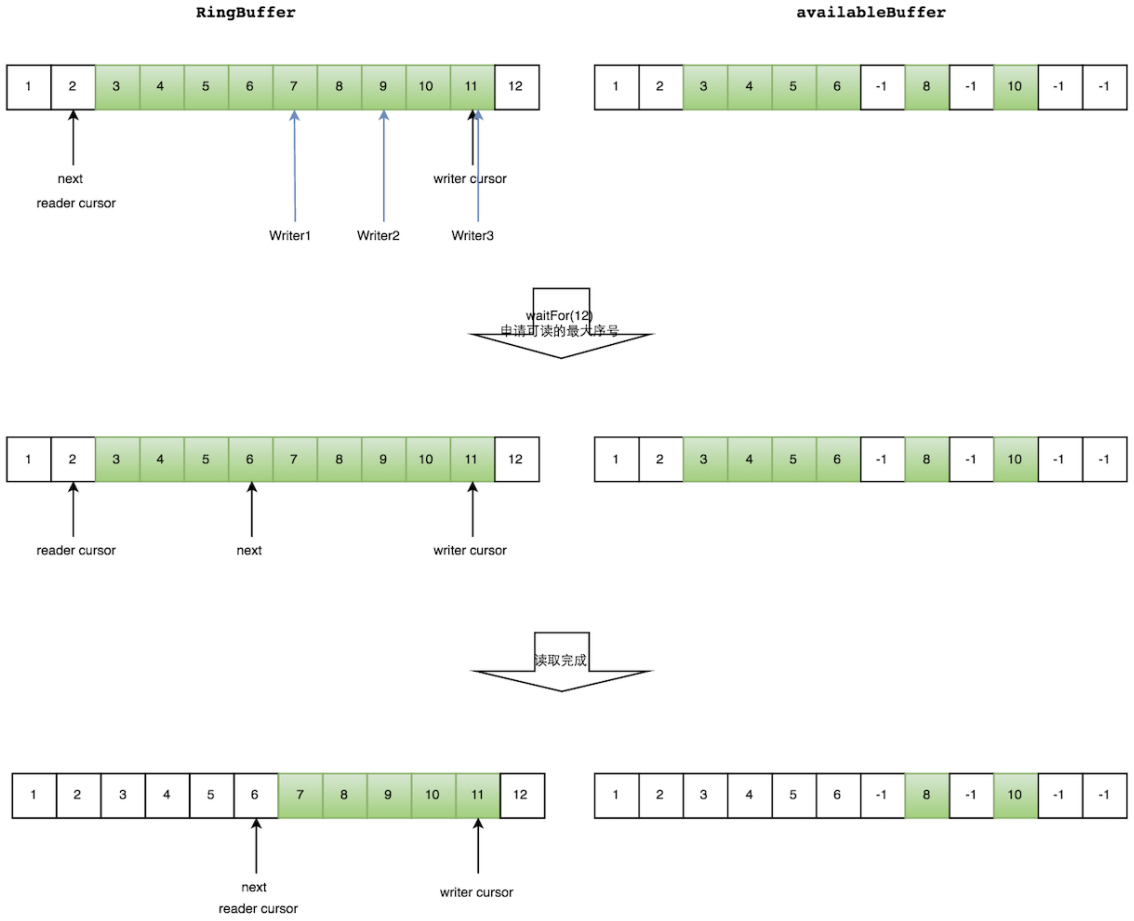
生产者多线程写入的情况下读数据会复杂很多：

1. 申请读取到序号n；

2. 若writer cursor >= n，这时仍然无法确定连续可读的最大下标。从reader cursor开始读取available Buffer，一直查到第一个不可用的元素，然后返回最大连续可读元素的位置；

3. 消费者读取元素。

如下图所示，读线程读到下标为2的元素，三个线程Writer1/Writer2/Writer3正在向RingBuffer相应位置写数据，写线程被分配到的最大元素下标是11。读线程申请读取到下标从3到11的元素，判断writer cursor>=11。然后开始读取availableBuffer，从3开始，往后读取，发现下标为7的元素没有生产成功，于是WaitFor(11)返回6。然后，消费者读取下标从3到6共计4个元素。



多个生产者写数据

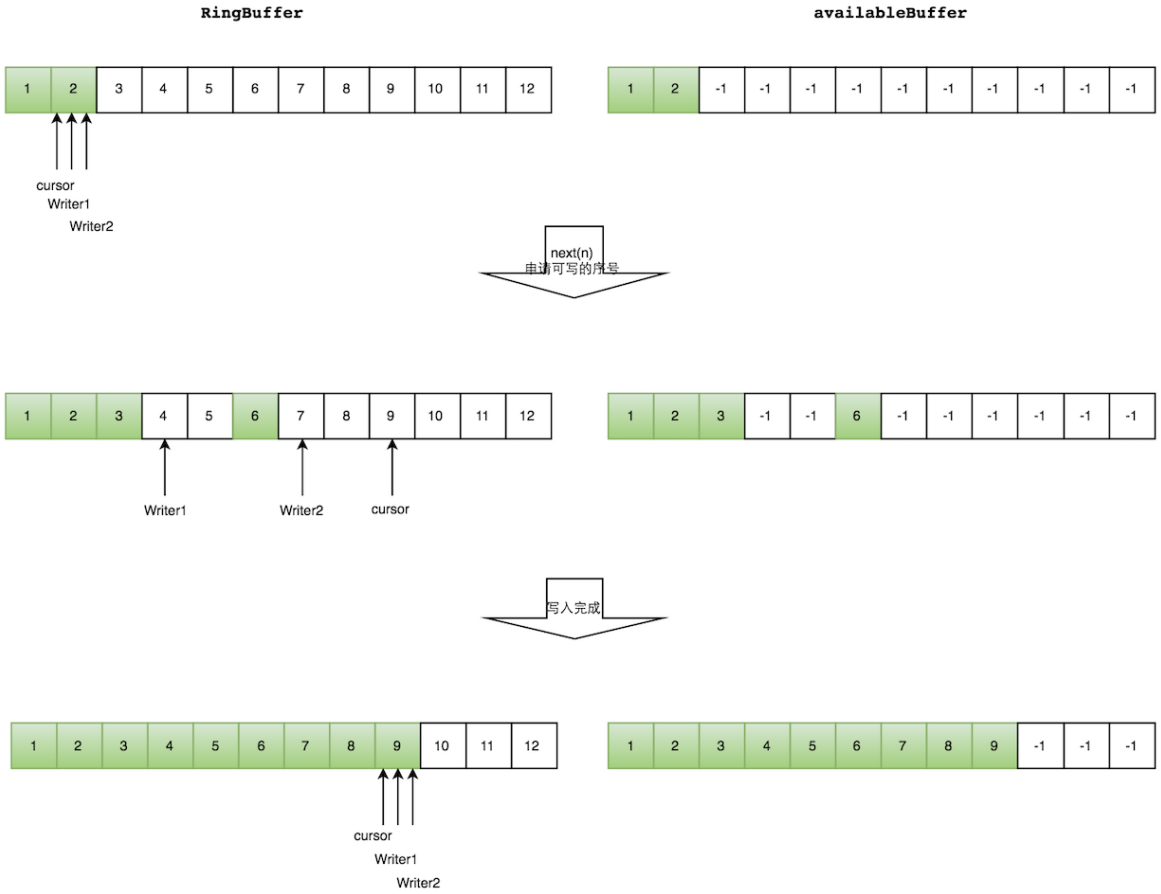
多个生产者写入的时候：

1. 申请写入m个元素；

2. 若是有m个元素可以写入，则返回最大的序列号。每个生产者会被分配一段独享的空间；

3. 生产者写入元素，写入元素的同时设置available Buffer里面相应的位置，以标记自己哪些位置是已经写入成功的。

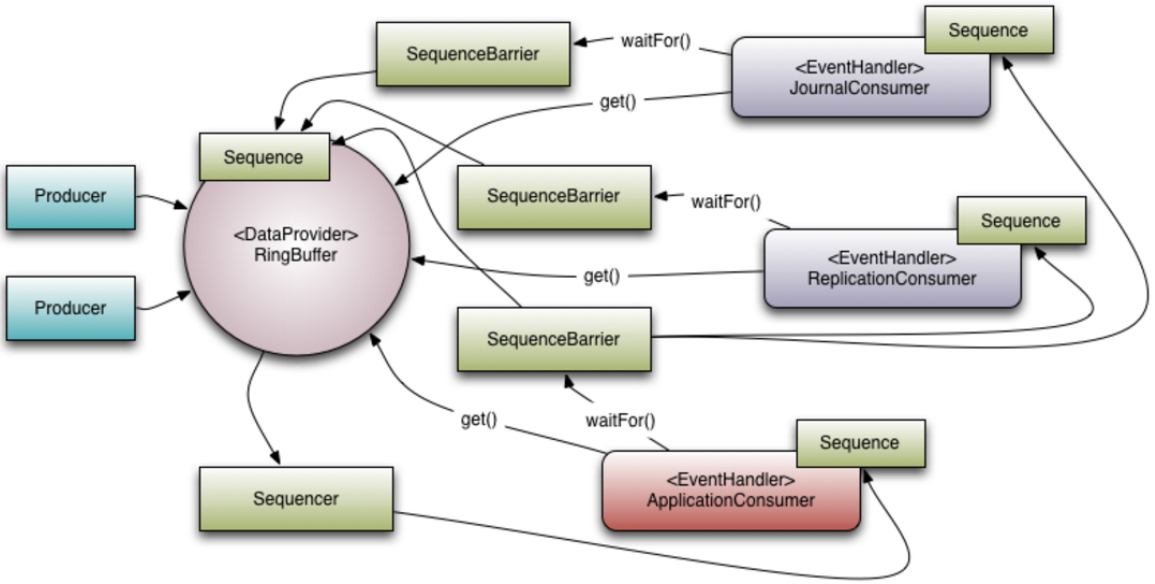
如下图所示，Writer1和Writer2两个线程写入数组，都申请可写的数组空间。Writer1被分配了下标3到下表5的空间，Writer2被分配了下标6到下标9的空间。Writer1写入下标3位置的元素，同时把available Buffer相应位置置位，标记已经写入成功，往后移一位，开始写下标4位置的元素。Writer2同样的方式。最终都写入完成。



Disruptor核心概念

- RingBuffer（环形缓冲区）：基于数组的内存级别缓存，是创建sequencer(序号)与定义WaitStrategy(拒绝策略)的入口。
- Disruptor（总体执行入口）：对RingBuffer的封装，持有RingBuffer、消费者线程池Executor、消费之集合ConsumerRepository等引用。
- Sequence（序号分配器）：对RingBuffer中的元素进行序号标记，通过顺序递增的方式来管理进行交换的数据(事件/Event)，一个Sequence可以跟踪标识某个事件的处理进度，同时还能消除伪共享。
- Sequencer（数据传输器）：Sequencer里面包含了Sequence，是Disruptor的核心，Sequencer有两个实现类：SingleProducerSequencer(单生产者实现)、MultiProducerSequencer(多生产者实现)，Sequencer主要作用是实现生产者和消费者之间快速、正确传递数据的并发算法
- SequenceBarrier（消费者屏障）：用于控制RingBuffer的Producer和Consumer之间的平衡关系，并且决定了Consumer是否还有可处理的事件的逻辑。
- WaitStrategy（消费者等待策略）：决定了消费者如何等待生产者将Event生产进Disruptor，WaitStrategy有多种实现策略
- Event：从生产者到消费者过程中所处理的数据单元，Event由使用者自定义。
- EventHandler：由用户自定义实现，就是我们写消费者逻辑的地方，代表了Disruptor中的一个消费者的接口。

- EventProcessor：这是个事件处理器接口，实现了Runnable，处理主要事件循环，处理Event，拥有消费者的Sequence



Disruptor构造器

```
public Disruptor(
 final EventFactory<T> eventFactory,
 final int ringBufferSize,
 final ThreadFactory threadFactory,
 final ProducerType producerType,
 final WaitStrategy waitStrategy)
```

- EventFactory：创建事件（任务）的工厂类。
- ringBufferSize：容器的长度。
- ThreadFactory ：用于创建执行任务的线程。
- ProductType：生产者类型：单生产者、多生产者。
- WaitStrategy：等待策略。

Disruptor的使用

引入依赖

```
<!-- disruptor -->
<dependency>
 <groupId>com.lmax</groupId>
 <artifactId>disruptor</artifactId>
 <version>3.3.4</version>
</dependency>
```

单生产者单消费者模式

1.创建Event(消息载体/事件)和EventFactory（事件工厂）

创建 OrderEvent 类，这个类将会被放入环形队列中作为消息内容。创建 OrderEventFactory类，用于创建OrderEvent事件

```
@Data
public class OrderEvent {
 private long value;
 private String name;
}

public class OrderEventFactory implements EventFactory<OrderEvent> {

 @Override
  public OrderEvent newInstance() {
  return new OrderEvent();
  }
}
```

2. 创建消息（事件）生产者

创建 OrderEventProducer 类，它将作为生产者使用

```
public class OrderEventProducer {
 //事件队列
 private RingBuffer<OrderEvent> ringBuffer;

 public OrderEventProducer(RingBuffer<OrderEvent> ringBuffer) {
 this.ringBuffer = ringBuffer;
 }

 public void onData(long value,String name) {
  // 获取事件队列 的下一个槽
  long sequence = ringBuffer.next();
  try {
  //获取消息（事件）
  OrderEvent orderEvent = ringBuffer.get(sequence);
  // 写入消息数据
  orderEvent.setValue(value);
  orderEvent.setName(name);
  } catch (Exception e) {
  // TODO 异常处理
  e.printStackTrace();
  } finally {
  System.out.println("生产者发送数据value:"+value+",name:"+name);
```

```
23   //发布事件
24   ringBuffer.publish(sequence);
25   }
26   }
27  }
```

3.创建消费者

创建 OrderEventHandler 类，并实现 EventHandler<T> ，作为消费者。

```
1  public class OrderEventHandler implements EventHandler<OrderEvent> {
2
3   @Override
4   public void onEvent(OrderEvent event, long sequence, boolean endOfBatch) throw
s Exception {
5   // TODO 消费逻辑
6   System.out.println("消费者获取数据value:"+ event.getValue()+",name:"+event.getN
ame());
7   }
8  }
```

4. 测试

```
1  public class DisruptorDemo {
2
3   public static void main(String[] args) throws Exception {
4
5   //创建disruptor
6   Disruptor<OrderEvent> disruptor = new Disruptor<>(
7   new OrderEventFactory(),
8   1024 * 1024,
9   Executors.defaultThreadFactory(),
10   ProducerType.SINGLE, //单生产者
11   new YieldingWaitStrategy() //等待策略
12   );
13   //设置消费者用于处理RingBuffer的事件
14   disruptor.handleEventsWith(new OrderEventHandler());
15   disruptor.start();
16
17   //创建ringbuffer容器
18   RingBuffer<OrderEvent> ringBuffer = disruptor.getRingBuffer();
19   //创建生产者
20   OrderEventProducer eventProducer = new OrderEventProducer(ringBuffer);
21   //发送消息
22   for(int i=0;i<100;i++){
23   eventProducer.onData(i,"Fox"+i);
24   }
25
```

```
26   disruptor.shutdown();
27   }
28 }
```

单生产者多消费者模式

如果消费者是多个，只需要在调用 handleEventsWith 方法时将多个消费者传递进去。

```
1 - disruptor.handleEventsWith(new OrderEventHandler());
2 + disruptor.handleEventsWith(new OrderEventHandler(), new OrderEventHandler());
```

上面传入的两个消费者会重复消费每一条消息，如果想实现一条消息在有多个消费者的情况下，只会被一个消费者消费，那么需要调用 handleEventsWithWorkerPool 方法。

```
1 - disruptor.handleEventsWith(new OrderEventHandler());
2 + disruptor.handleEventsWithWorkerPool(new OrderEventHandler(), new OrderEventH
andler());
```

注意：消费者要实现WorkHandler接口

```
1 public class OrderEventHandler implements EventHandler<OrderEvent>,
WorkHandler<OrderEvent> {
2
3  @Override
4  public void onEvent(OrderEvent event, long sequence, boolean endOfBatch) throw
s Exception {
5  // TODO 消费逻辑
6  System.out.println("消费者"+ Thread.currentThread().getName()
7  +"获取数据value:"+ event.getValue()+",name:"+event.getName());
8  }
9
10  @Override
11  public void onEvent(OrderEvent event) throws Exception {
12  // TODO 消费逻辑
13  System.out.println("消费者"+ Thread.currentThread().getName()
14  +"获取数据value:"+ event.getValue()+",name:"+event.getName());
15  }
16 }
```

多生产者多消费者模式

在实际开发中，多个生产者发送消息，多个消费者处理消息才是常态。

```
1 public class DisruptorDemo2 {
2
3  public static void main(String[] args) throws Exception {
4
5  //创建disruptor
6  Disruptor<OrderEvent> disruptor = new Disruptor<>(
7  new OrderEventFactory(),
```

```
 1024 * 1024,
 Executors.defaultThreadFactory(),
  ProducerType.MULTI, //多生产者
  new YieldingWaitStrategy() //等待策略
  );

  //设置消费者用于处理RingBuffer的事件
  //disruptor.handleEventsWith(new OrderEventHandler());
  //设置多消费者,消息会被重复消费
  //disruptor.handleEventsWith(new OrderEventHandler(),new OrderEventHandler
());
  //设置多消费者,消费者要实现WorkHandler接口，一条消息只会被一个消费者消费
  disruptor.handleEventsWithWorkerPool(new OrderEventHandler(), new OrderEventH
andler());

  //启动disruptor
  disruptor.start();

  //创建ringbuffer容器
  RingBuffer<OrderEvent> ringBuffer = disruptor.getRingBuffer();

  new Thread(()->{
  //创建生产者
  OrderEventProducer eventProducer = new OrderEventProducer(ringBuffer);
  // 发送消息
  for(int i=0;i<100;i++){
  eventProducer.onData(i,"Fox"+i);
  }
  },"producer1").start();

  new Thread(()->{
  //创建生产者
  OrderEventProducer eventProducer = new OrderEventProducer(ringBuffer);
  // 发送消息
  for(int i=0;i<100;i++){
  eventProducer.onData(i,"monkey"+i);
  }
  },"producer2").start();


  //disruptor.shutdown();

  }
}
```

消费者优先级模式

在实际场景中，我们通常会因为业务逻辑而形成一条消费链。比如一个消息必须由 消费者A -> 消费者B -> 消费者C 的顺序依次进行消费。在配置消费者时，可以通过 .then 方法去实现顺序消费。

```
disruptor.handleEventsWith(new OrderEventHandler())
  .then(new OrderEventHandler())
  .then(new OrderEventHandler());
```

handleEventsWith 与 handleEventsWithWorkerPool 都是支持 .then 的，它们可以结合使用。比如可以按照 消费者A -> (消费者B 消费者C) -> 消费者D 的消费顺序

```
disruptor.handleEventsWith(new OrderEventHandler())
  .thenHandleEventsWithWorkerPool(new OrderEventHandler(), new
OrderEventHandler())
  .then(new OrderEventHandler());
```