# 认识Disruptor

Disruptor是一个开源框架，研发的初衷是为了解决高并发下列队锁的问题，最早由

LMAX（一种新型零售金融交易平台）提出并使用，能够在无锁的情况下实现队列的并发操作，并号称能够在一个线程里每秒处理6百万笔订单(这个真假就不清楚了！牛皮谁都会

吹)。

框架最经典也是最多的应用场景：生产消费。

讲到生产消费模型，大家应该马上就能回忆起前面我们已经学习过的BlockingQueue 课程，里面我们学习过多种队列，但是这些队列大多是基于条件阻塞方式的，性能还不够优秀！

1. ArrayBlockingQueue：基于数组形式的队列，通过加锁的方式，来保证多线程情况下数据的安全；
2. LinkedBlockingQueue：基于链表形式的队列，也通过加锁的方式，来保证多线程情况下数据的安全；
3. ConcurrentLinkedQueue：基于链表形式的队列，通过compare and swap(简称CAS)协议的方式，
4. 来保证多线程情况下数据的安全，不加锁，主要使用了Java中的sun.misc.Unsa fe类来实现；

# 核心设计原理

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（回顾一下：

CPU加载**空间局部性原则**）。

元素位置定位：

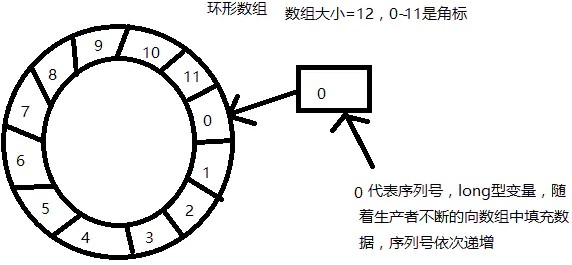
数组长度2^n，通过位运算，加快定位的速度。下标采取递增的形式。不用担心index溢出的问题。index是long类型，即使100万QPS的处理速度，也需要30万年才能用完。

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。

## 数据结构

框架使用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：



### Sequence

mark：Disruptor通过顺序递增的序号来编号管理通过其进行交换的数据（事件），对数据(事件)的处理过程总是沿着序号逐个递增处理。

### 数组+序列号设计的优势是什么呢？

回顾一下我们讲HashMap时，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 % table.length。当然也可以用位运算来计算效率更高，此时

table.length必须是2的幂次方(原理前面讲过)。

## 概念与作用

RingBuffer——Disruptor底层数据结构实现，核心类，是线程间交换数据的中转地；

Sequencer——序号管理器，生产同步的实现者，负责消费者/生产者各自序 号、序号栅栏的管理和协调,Sequencer有单生产者,多生产者两种不同的模式,里面实现了各种同步的算法；

Sequence——序号，声明一个序号，用于跟踪ringbuffer中任务的变化和消费者的消费情况，disruptor里面大部分的并发代码都是通过对Sequence的值同步修改实现的,而非锁,这是disruptor高性能的一个主要原因；

SequenceBarrier——序号栅栏，管理和协调生产者的游标序号和各个消费者的序号，确保生产者不会覆盖消费者未来得及处理的消息，确保存在依赖的消费者之间能够按照正确的顺序处理， Sequence Barrier是由Sequencer创建的,并被

Processor持有；

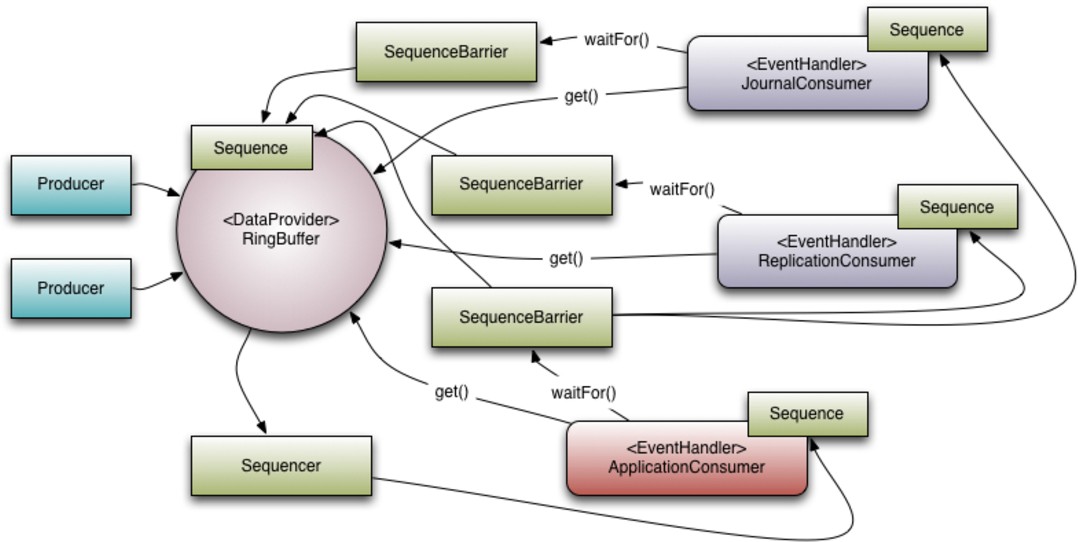
EventProcessor——事件处理器，监听RingBuffer的事件，并消费可用事件， 从RingBuffer读取的事件会交由实际的生产者实现类来消费；它会一直侦听下一个可用的序号，直到该序号对应的事件已经准备好。

EventHandler——业务处理器，是实际消费者的接口，完成具体的业务逻辑实现，第三方实现该接口；代表着消费者。

Producer——生产者接口，第三方线程充当该角色，producer向RingBuffer写入事件。

Wait Strategy：Wait Strategy决定了一个消费者怎么等待生产者将事件

（Event）放入Disruptor中。



## 等待策略

### BlockingWaitStrategy

Disruptor的默认策略是BlockingWaitStrategy。在BlockingWaitStrategy内部是使 用锁和condition来控制线程的唤醒。BlockingWaitStrategy是最低效的策略，但其对CPU 的消耗最小并且在各种不同部署环境中能提供更加一致的性能表现。**SleepingWaitStrategy**

SleepingWaitStrategy 的性能表现跟 BlockingWaitStrategy 差不多，对 CPU 的消耗也类似，但其对生产者线程的影响最小，通过使用LockSupport.parkNanos(1)来实现循环等待。一般来说Linux系统会暂停一个线程约60µs，这样做的好处是，生产线程不需要采取任何其他行动就可以增加适当的计数器，也不需要花费时间信号通知条件变量。但是，在生产者线程和使用者线程之间移动事件的平均延迟会更高。它在不需要低延迟并且对生产线程的影响较小的情况最好。一个常见的用例是异步日志记录。

### YieldingWaitStrategy

YieldingWaitStrategy是可以使用在低延迟系统的策略之一。YieldingWaitStrategy 将自旋以等待序列增加到适当的值。在循环体内，将调用Thread.yield（），以允许其他排队的线程运行。在要求极高性能且事件处理线数小于 CPU 逻辑核心数的场景中，推荐使用此策略；例如，CPU开启超线程的特性。

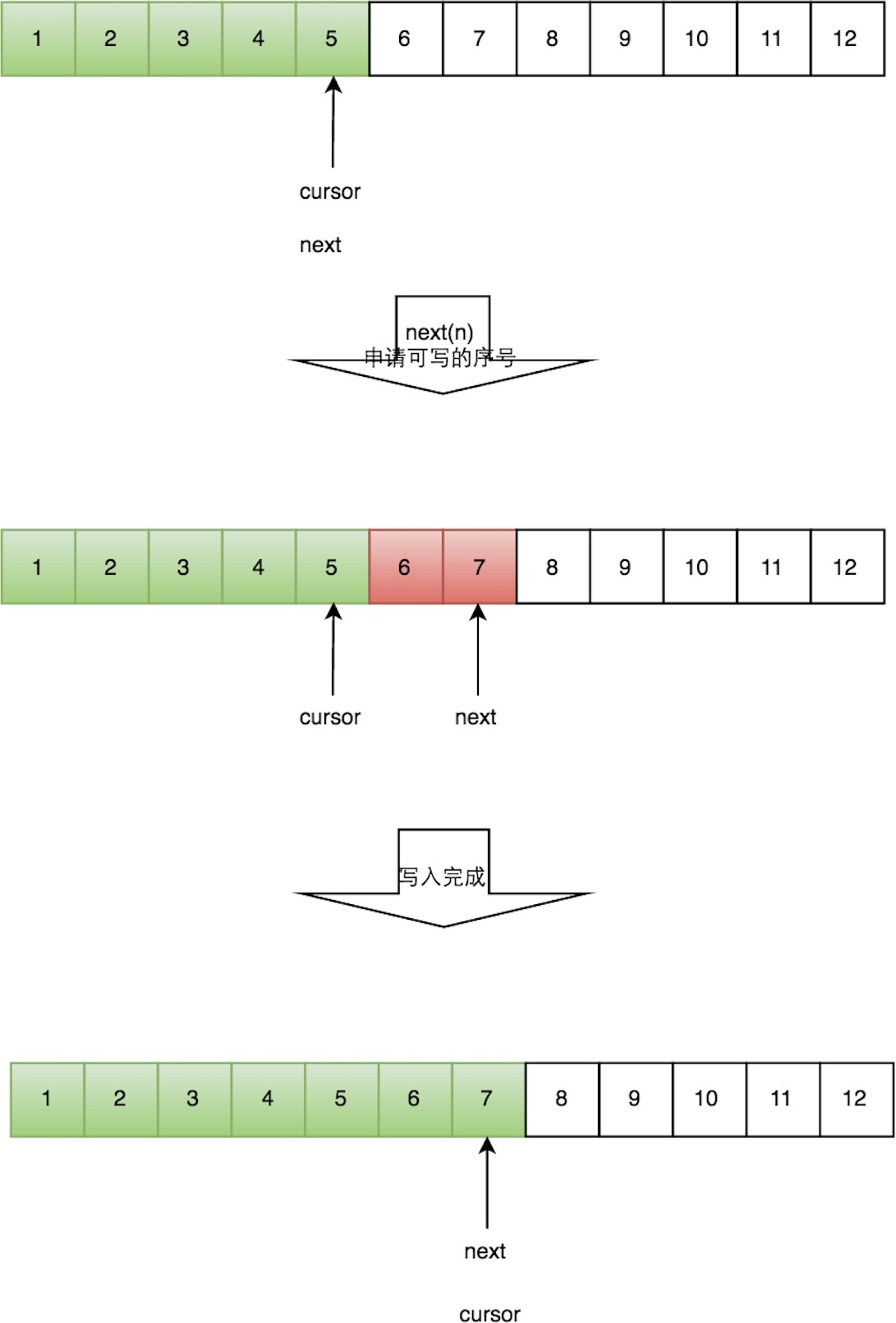
### BusySpinWaitStrategy

性能最好，适合用于低延迟的系统。在要求极高性能且事件处理线程数小于CPU逻辑核心数的场景中，推荐使用此策略；例如，CPU开启超线程的特性。

**写数据**

单线程写数据的流程：

1. 申请写入m个元素；
2. 若是有m个元素可以入，则返回最大的序列号。这儿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。



# 框架的使用

## 生产消费模型的应用

### 1、引入依赖

1. <dependencies>
2. <dependency>
3. <groupId>com.lmax</groupId>
4. <artifactId>disruptor</artifactId>
5. <version>3.2.1</version>
6. </dependency>
7. </dependencies>

**2、定义Event**

1. //定义事件event 通过Disruptor 进行交换的数据类型。
2. public class LongEvent {

3

4 private Long value;

5

1. public Long getValue() {
2. return value;

8 }

9

1. public void setValue(Long value) {
2. this.value = value;

12 }

13

14 }

**3、定义EventFactory**

我们需要Disruptor为我们创建Event，所以这里我们需要定义事件工厂，实现框架定义的接口

1. public class LongEventFactory implements EventFactory<LongEvent>

{

1. public LongEvent newInstance() {
2. return new LongEvent();

4 }

5 }

### 4、定义事件消费者

1. public class LongEventHandler implements EventHandler<LongEvent>

{

1. public void onEvent(LongEvent event, long sequence, boolean end OfBatch) throws Exception {
2. System.out.println("消费者:"+event.getValue());

4 }

5 }

**5、定义生产者**

1 public class LongEventProducer {

2

3 public final RingBuffer<LongEvent> ringBuffer;

4

1. public LongEventProducer(RingBuffer<LongEvent> ringBuffer) {
2. this.ringBuffer = ringBuffer;

7 }

8

1. public void onData(ByteBuffer byteBuffer) {
2. // 1.ringBuffer 事件队列 下一个槽
3. long sequence = ringBuffer.next();
4. Long data = null;
5. try {
6. //2.取出空的事件队列
7. LongEvent longEvent = ringBuffer.get(sequence);
8. data = byteBuffer.getLong(0);
9. //3.获取事件队列传递的数据
10. longEvent.setValue(data);
11. try {
12. Thread.sleep(10);
13. } catch (InterruptedException e) {
14. // TODO Auto‐generated catch block
15. e.printStackTrace();

24 }

1. } finally {
2. System.out.println("生产这准备发送数据");
3. //4.发布事件

28 ringBuffer.publish(sequence);

29 }

30 }

31 }

**6、定义Main入口**

1 public class DisruptorMain {

2

1. public static void main(String[] args) {
2. // 1.创建一个可缓存的线程 提供线程来出发Consumer 的事件处理
3. ExecutorService executor = Executors.newCachedThreadPool();
4. // 2.创建工厂
5. EventFactory<LongEvent> eventFactory = new LongEventFactory();
6. // 3.创建ringBuffer 大小
7. int ringBufferSize = 1024 \* 1024; // ringBufferSize大小一定要是2

的N次方

1. // 4.创建Disruptor
2. Disruptor<LongEvent> disruptor = new Disruptor<LongEvent>(even tFactory, ringBufferSize, executor,
3. ProducerType.SINGLE, new YieldingWaitStrategy());
4. // 5.连接消费端方法
5. disruptor.handleEventsWith(new LongEventHandler());

15 // 6.启动

1. disruptor.start();
2. // 7.创建RingBuffer容器
3. RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();
4. // 8.创建生产者
5. LongEventProducer producer = new LongEventProducer(ringBuffer);
6. // 9.指定缓冲区大小
7. ByteBuffer byteBuffer = ByteBuffer.allocate(8);

23 for (int i = 1; i <= 100; i++) {

1. byteBuffer.putLong(0, i);
2. producer.onData(byteBuffer);

26 }

1. //10.关闭disruptor和executor
2. disruptor.shutdown();

29 executor.shutdown();

30 }

31

32 }

[有 道 链 接 ：http://note.youdao.com/noteshare? id=3b0ee63fac4353cc134d1f9e87116f5c&sub=E9DC5677DEEB46BF9F98E851695E33A9](http://note.youdao.com/noteshare?id=3b0ee63fac4353cc134d1f9e87116f5c&amp;sub=E9DC5677DEEB46BF9F98E851695E33A9)