# 1 disruptor 是什么?

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 Disruptor。同年 Disruptor 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。

要深入了解 disruptor ，咱们从 Java 的 内置队列开始介绍起。

# 2 Java 内置队列的问题

介绍 Disruptor 之前，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 |
| --- | --- | --- | --- |
| ArrayBlockingQueue | bounded | 加锁 | arraylist |
| LinkedBlockingQueue | optionally-bounded | 加锁 | linkedlist |
| ConcurrentLinkedQueue | unbounded | 无锁 | linkedlist |
| LinkedTransferQueue | unbounded | 无锁 | linkedlist |
| PriorityBlockingQueue | unbounded | 加锁 | heap |
| DelayQueue | unbounded | 加锁 | heap |

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，暂且不考虑。

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，而后者通过原子变量 compare and swap（以下简称 “CAS”）这种**无锁方式**来实现的。

和 ConcurrentLinkedQueue 一样，上面表格中的 LinkedTransferQueue 都是通过原子变量 compare and swap（以下简称 “CAS”）这种不加锁的方式来实现的

但是，对 volatile 类型的变量进行 CAS 操作，存在伪共享问题，具体请参考专门的文章：

1 [disruptor 史上最全 之 1：**伪共享 原理 & 性能对比实战**]

Disruptor 使用了类似上面的方案，解决了伪共享问题。

> 说明：本文会以 pdf 格式持续更新，更多最新尼恩 3 高 pdf 笔记，请从下面的链接获取：语雀 或者 码云

# 3 Disruptor 框架是如何解决伪共享问题的?

在 Disruptor 中有一个重要的类 Sequence，该类包装了一个 volatile 修饰的 long 类型数据 value，

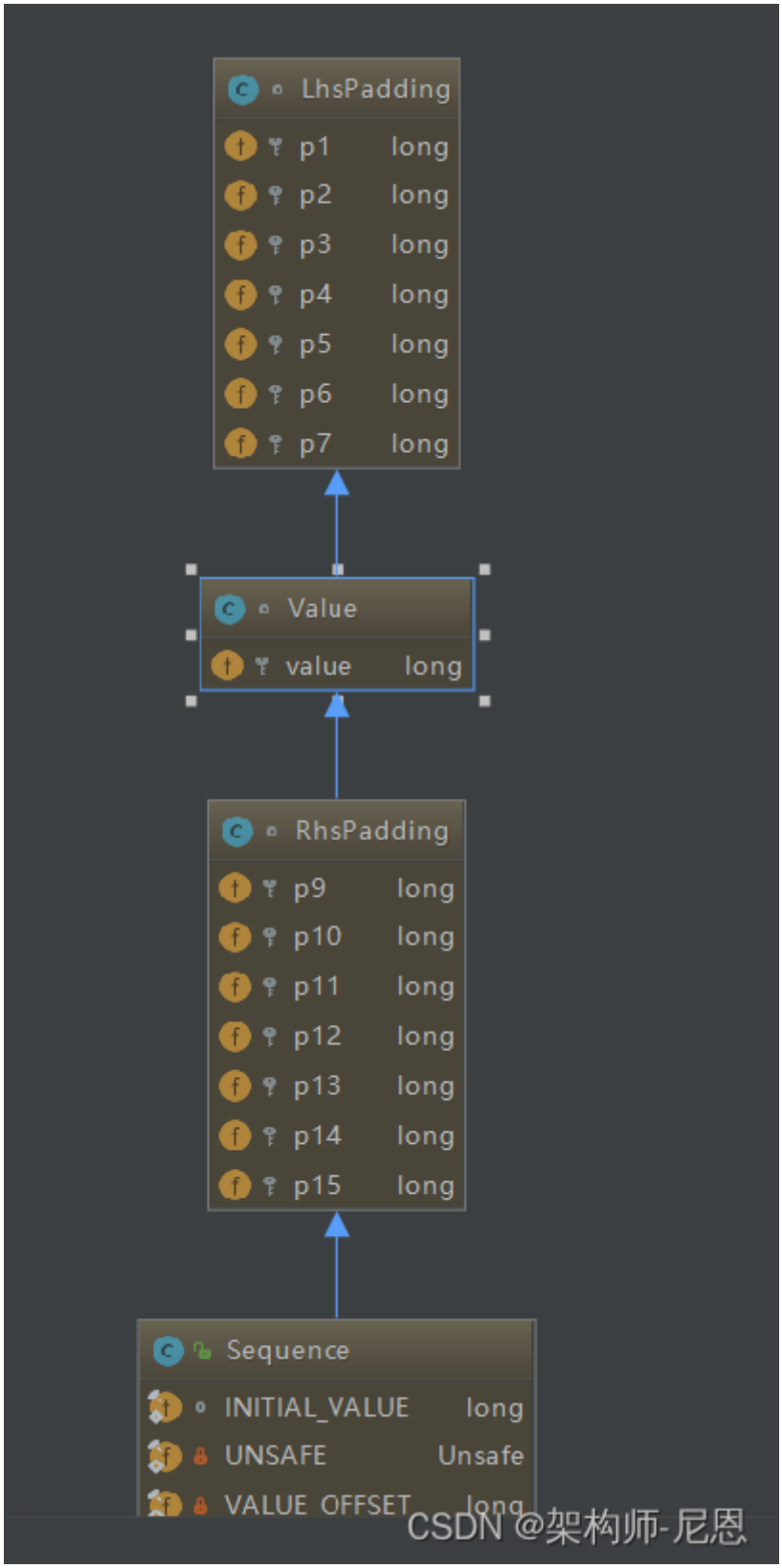
## Sequence 的结构和源码

无论是 Disruptor 中的基于数组实现的缓冲区 RingBuffer，还是生产者，消费者，都有各自独立的 Sequence，

Sequence 的用途是啥呢?

- 在 RingBuffer 缓冲区中，Sequence 标示着写入进度，例如每次生产者要写入数据进缓冲区时，都要调用 RingBuffer.next（）来获得下一个可使用的相对位置。
- 对于生产者和消费者来说，Sequence 标示着它们的事件序号。

Sequence 的结构图如下




来看看 Sequence 类的源码：

```
  class LhsPadding {
    protected long p1, p2, p3, p4, p5, p6, p7;
}

class Value extends LhsPadding {
    protected volatile long value;
}

class RhsPadding extends Value {
```

```
    protected long p9, p10, p11, p12, p13, p14, p15;
}

public class Sequence extends RhsPadding {
    static final long INITIAL_VALUE = -1L;
    private static final Unsafe UNSAFE;
    private static final long VALUE_OFFSET;
    static {
        UNSAFE = Util.getUnsafe();
        try {
            VALUE_OFFSET =
UNSAFE.objectFieldOffset(Value.class.getDeclaredField("value"));
        } catch(final Exception e) {
             throw new RuntimeException(e);
        }
    }


public Sequence() {
    this(INITIAL_VALUE);
}

public Sequence(final long initialValue) {
    UNSAFE.putOrderedLong(this, VALUE_OFFSET, initialValue);
}

}
```

# Disruptor 的使用场景

Disruptor 它可以用来作为高性能的有界内存队列， 适用于两大场景：

- 生产者消费者场景
- 发布订阅 场景

生产者消费者场景。Disruptor 的最常用的场景就是 “生产者 - 消费者” 场景，对场景的就是 “一个生产者、多个消费者” 的场景，并且要求顺序处理。

> 备注，这里和 JCTool 的 MPSC 队列，刚好相反， MPSC 使用于多生产者，单消费者场景

发布订阅 场景：Disruptor 也可以认为是观察者模式的一种实现， 实现发布订阅模式。

当前业界开源组件使用 Disruptor 的包括 Log4j2、Apache Storm 等，

> 说明：本文会以 pdf 格式持续更新，更多最新尼恩 3 高 pdf 笔记，请从下面的链接获取：语雀 或者 码云

# 实战：Disruptor 的 使用实例

我们从一个简单的例子开始学习 Disruptor：

生产者传递一个 long 类型的值给消费者，而消费者消费这个数据的方式仅仅是把它打印出来。

## 定义一个 Event 和工厂

首先定义一个 Event 来包含需要传递的数据：

```
public class LongEvent {
    private long value;
    public long getValue() {
        return value;
    }

    public void setValue(long value) {
        this.value = value;
    }
}
```

由于需要让 Disruptor 为我们创建事件，我们同时还声明了一个 EventFactory 来创建 Event 对象。

```
public class LongEventFactory implements EventFactory {
    @Override
    public Object newInstance() {
        return new LongEvent();
    }
}
```

## 定义事件处理器（消费者）

我们还需要一个事件消费者，也就是一个事件处理器。

这个例子中，事件处理器的工作，就是简单地把事件中存储的数据打印到终端：

```
    /**
     * 类似于消费者
     *  disruptor会回调此处理器的方法
     */
    static class LongEventHandler implements EventHandler<LongEvent> {
        @Override
        public void onEvent(LongEvent longEvent, long l, boolean b) throws
Exception {
            System.out.println(longEvent.getValue());
        }
    }
```

disruptor 会回调此处理器的方法

## 定义事件源 (生产者)

事件都会有一个生成事件的源，类似于 生产者的角色，

如何产生事件，然后发出事件呢?

通过从 环形队列中 获取 序号， 通过序号获取 对应的 事件对象， 将数据填充到 事件对象，再通过 序号将 事件对象 发布出去。

一段生产者的代码如下:

```
    //  事件生产者: 业务代码
    // 通过从 环形队列中 获取 序号， 通过序号获取 对应的 事件对象， 将数据填充到 事件对象，再通过 序号将 事件对象 发布出去。
    static class LongEventProducer {
        private final RingBuffer<LongEvent> ringBuffer;

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

        /**
         * onData用来发布事件，每调用一次就发布一次事件事件
         * 它的参数会通过事件传递给消费者
         *
         * @param data
         */
        public void onData(long data) {

            // step1: 通过从 环形队列中 获取 序号
            //可以把ringBuffer看做一个事件队列，那么next就是得到下面一个事件槽
            long sequence = ringBuffer.next();

            try {

                //step2: 通过序号获取 对应的 事件对象， 将数据填充到 事件对象，
                //用上面的索引，取出一个空的事件用于填充
                LongEvent event = ringBuffer.get(sequence);// for the sequence
                event.setValue(data);
            } finally {

                //step3: 再通过 序号将 事件对象 发布出去。
                //发布事件
                ringBuffer.publish(sequence);
            }
        }
    }
```

很明显的是:

当用一个简单队列来发布事件的时候会牵涉更多的细节，这是因为事件对象还需要预先创建。

发布事件最少需要三步:

step1：获取下一个事件槽。

如果我们使用 RingBuffer.next() 获取一个事件槽，那么一定要发布对应的事件。

step2: 通过序号获取 对应的 事件对象， 将数据填充到 事件对象,

step3: 再通过 序号将 事件对象 发布出去。

发布事件的时候要使用 try/finnally 保证事件一定会被发布

如果不能发布事件，那么就会引起 Disruptor 状态的混乱。

尤其是在多个事件生产者的情况下会导致事件消费者失速，从而不得不重启应用才能会恢复。

Disruptor 3.0 提供了 lambda 式的 API。

这样可以把一些复杂的操作放在 Ring Buffer，所以在 Disruptor3.0 以后的版本最好使用 Event Publisher 或者 Event Translator(事件转换器) 来发布事件。

## 组装起来

最后一步就是把所有的代码组合起来完成一个完整的事件处理系统。

```
  @org.junit.Test
    public  void testSimpleDisruptor() throws InterruptedException {
        // 消费者线程池
        Executor executor = Executors.newCachedThreadPool();
        // 事件工厂
        LongEventFactory eventFactory = new LongEventFactory();
        // 环形队列大小，2的指数
        int bufferSize = 1024;

        // 构造  分裂者 （事件分发者）
        Disruptor<LongEvent> disruptor = new Disruptor<LongEvent>(eventFactory,
bufferSize, executor);

        // 连接 消费者 处理器
        disruptor.handleEventsWith(new LongEventHandler());
        // 开启 分裂者（事件分发）
        disruptor.start();

        // 获取环形队列，用于生产 事件
        RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();

        LongEventProducer producer = new LongEventProducer(ringBuffer);

        for (long i = 0; true; i++) {
            //发布事件
            producer.onData(i);
            Thread.sleep(1000);
        }
    }
```

# 事件转换器

Disruptor3.0 以后 , 提供了事件转换器， 帮助填充 LongEvent 的业务数据

下面是一个例子

```
  static class LongEventProducerWithTranslator {
        //一个translator可以看做一个事件初始化器，publicEvent方法会调用它
        //填充Event
        private static final EventTranslatorOneArg<LongEvent, Long> TRANSLATOR =
                new EventTranslatorOneArg<LongEvent, Long>() {
                    public void translateTo(LongEvent event, long sequence, Long
data) {
                        event.setValue(data);
                    }
                };

        private final RingBuffer<LongEvent> ringBuffer;

        public LongEventProducerWithTranslator(RingBuffer<LongEvent> ringBuffer)
{
            this.ringBuffer = ringBuffer;
        }

        public void onData(Long data) {
            ringBuffer.publishEvent(TRANSLATOR, data);
        }
    }
```

使用事件转换器的好处，省了从 环形队列 获取 序号， 然后拿到事件 填充数据， 再发布序号 中的第二步骤

给 事件 填充 数据 的动作，在 EventTranslatorOneArg 完成

Disruptor 提供了不同的接口去产生一个 Translator 对象：

- EventTranslator,
- EventTranslatorOneArg,
- EventTranslatorTwoArg,

很明显，Translator 中方法的参数是通过 RingBuffer 来传递的。

使用 事件转换器 转换器的进行事件的 生产与消费 代码，大致如下：

```
   @org.junit.Test
    public void testSimpleDisruptorWithTranslator() throws InterruptedException {
        // 消费者线程池
        Executor executor = Executors.newCachedThreadPool();
        // 事件工厂
        LongEventFactory eventFactory = new LongEventFactory();
        // 环形队列大小，2的指数
        int bufferSize = 1024;

        // 构造  分裂者 （事件分发者）
        Disruptor<LongEvent> disruptor = new Disruptor<LongEvent>(eventFactory,
bufferSize, executor);
```

```
        // 连接 消费者 处理器
        disruptor.handleEventsWith(new LongEventHandler());
        // 开启 分裂者（事件分发）
        disruptor.start();

        // 获取环形队列，用于生产 事件
        RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();

        LongEventProducerWithTranslator producer = new
LongEventProducerWithTranslator(ringBuffer);

        for (long i = 0; true; i++) {
            //发布事件
            producer.onData(i);
            Thread.sleep(1000);
        }
    }
```

上面写法的另一个好处是，Translator 可以分离出来并且更加容易单元测试。

## 通过 Java 8 Lambda 使用 Disruptor

Disruptor 在自己的接口里面添加了对于 Java 8 Lambda 的支持。

大部分 Disruptor 中的接口都符合 Functional Interface 的要求（也就是在接口中仅仅有一个方法）。

所以在 Disruptor 中，可以广泛使用 Lambda 来代替自定义类。

```
 @org.junit.Test
    public void testSimpleDisruptorWithLambda() throws InterruptedException {
        // 消费者线程池
        Executor executor = Executors.newCachedThreadPool();
        // 环形队列大小，2的指数
        int bufferSize = 1024;

        // 构造  分裂者 （事件分发者）
        Disruptor<LongEvent> disruptor = new Disruptor<LongEvent>(LongEvent::new,
bufferSize, executor);

        // 连接 消费者 处理器
        // 可以使用lambda来注册一个EventHandler
        disruptor.handleEventsWith((event, sequence, endOfBatch) ->
System.out.println("Event: " + event.getValue()));
        // 开启 分裂者（事件分发）
        disruptor.start();

        // 获取环形队列，用于生产 事件
        RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();

        LongEventProducerWithTranslator producer = new
LongEventProducerWithTranslator(ringBuffer);

        for (long i = 0; true; i++) {
            //发布事件
            producer.onData(i);
```

```
            Thread.sleep(1000);
        }
    }
```

由于在 Java 8 中方法引用也是一个 lambda，因此还可以把上面的代码改成下面的代码：

```
    public static void handleEvent(LongEvent event, long sequence, boolean
endOfBatch)
    {
        System.out.println(event.getValue());
    }

    @org.junit.Test
    public void testSimpleDisruptorWithMethodRef() throws InterruptedException {
        // 消费者线程池
        Executor executor = Executors.newCachedThreadPool();
        // 环形队列大小，2的指数
        int bufferSize = 1024;

        // 构造  分裂者 （事件分发者）
        Disruptor<LongEvent> disruptor = new Disruptor<LongEvent>(LongEvent::new,
bufferSize, executor);

        // 连接 消费者 处理器
        // 可以使用lambda来注册一个EventHandler
        disruptor.handleEventsWith(LongEventDemo::handleEvent);
        // 开启 分裂者（事件分发）
        disruptor.start();

        // 获取环形队列，用于生产 事件
        RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();

        LongEventProducerWithTranslator producer = new
LongEventProducerWithTranslator(ringBuffer);

        for (long i = 0; true; i++) {
            //发布事件
            producer.onData(i);
            Thread.sleep(1000);
        }
    }
}
```

> 说明：本文会以 pdf 格式持续更新，更多最新尼恩 3 高 pdf 笔记，请从下面的链接获取：语雀 或者 码云

# 构造 Disruptor 对象的几个要点

在构造 Disruptor 对象，有几个核心的要点：
1：事件工厂 (Event Factory) 定义了如何实例化事件(Event)，Disruptor 通过 EventFactory 在 RingBuffer 中预创建 Event 的实例。
2：ringBuffer 这个数组的大小，一般根据业务指定成 2 的指数倍。

3：消费者线程池，事件的处理是在构造的线程池里来进行处理的。
4：指定等待策略，Disruptor 定义了 com.lmax.disruptor.WaitStrategy 接口用于抽象 **Consumer 如何等待 Event 事件**。

Disruptor 提供了多个 WaitStrategy 的实现，每种策略都具有不同性能和优缺点，根据实际运行环境的 CPU 的硬件特点选择恰当的策略，并配合特定的 JVM 的配置参数，能够实现不同的性能提升。

- 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 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，适合用于异步日志类似的场景；
- YieldingWaitStrategy 的性能是最好的，适合用于低延迟的系统。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；。

# 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。
- 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

  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。
- 属性填充：

  通过添加额外的无用信息，避免伪共享问题

## Disruptor 和 BlockingQueue 比较:

- **BlockingQueue:** FIFO 队列. 生产者 Producer 向队列中发布 publish 一个事件时, 消费者 Consumer 能够获取到通知. 如果队列中没有消费的事件, 消费者就会被阻塞, 直到生产者发布新的事件
- Disruptor 可以比 BlockingQueue 做到更多:
  - Disruptor 队列中同一个事件可以有多个消费者, 消费者之间既可以并行处理, 也可以形成依赖图相互依赖, 按照先后次序进行处理
  - Disruptor 可以预分配用于存储事件内容的内存空间
  - Disruptor 使用极度优化和无锁的设计实现极高性能的目标

如果你的项目有对性能要求高，对延迟要求低的需求，并且需要一个无锁的有界队列，来实现生产者 / 消费者模式，那么 Disruptor 是你的不二选择。

> 说明：本文会以 pdf 格式持续更新，更多最新尼恩 3 高 pdf 笔记，请从下面的链接获取：语雀 或者 码云

# 原理：Disruptor 的内部 Ring Buffer 环形队列

## RingBuffer 是什么

RingBuffer 是一个环 (首尾相连的环)，用做在不同上下文(线程) 间传递数据的 buffer。

RingBuffer 拥有一个序号，这个序号指向数组中下一个可用元素。


## Disruptor 使用环形队列的优势：

Disruptor 框架就是一个使用 CAS 操作的内存队列，与普通的队列不同，

Disruptor 框架使用的是一个基于数组实现的环形队列，无论是生产者向缓冲区里提交任务，还是消费者从缓冲区里获取任务执行，都使用 CAS 操作。

使用环形队列的优势：

第一，简化了多线程同步的复杂度。

学数据结构的时候，实现队列都要两个指针 head 和 tail 来分别指向队列的头和尾，对于一般的队列是这样，

想象下，如果有多个生产者同时往缓冲区队列中提交任务，某一生产者提交新任务后，tail 指针都要做修改的，那么多个生产者提交任务，头指针不会做修改，但会对 tail 指针产生冲突，

例如某一生产者 P1 要做写入操作，在获得 tail 指针指向的对象值 V 后，执行 compareAndSet（）方法前，tail 指针被另一生产者 P2 修改了，这时生产者 P1 执行 compareAndSet（）方法，发现 tail 指针指向的值 V 和期望值 E 不同，导致冲突。

同样，如果多个消费者不断从缓冲区中获取任务，不会修改尾指针，但会造成队列头指针 head 的冲突问题（因为队列的 FIFO 特点，出列会从头指针出开始）。

环形队列的一个特点就是只有一个指针，只通过一个指针来实现出列和入列操作。

如果使用两个指针 head 和 tail 来管理这个队列，有可能会出现 “伪共享” 问题（伪共享问题在下面我会详细说），

因为创建队列时，head 和 tail 指针变量常常在同一个缓存行中，多线程修改同一缓存行中的变量就容易出现伪共享问题。

第二，由于使用的是环形队列，那么队列创建时大小就被固定了，

Disruptor 框架中的环形队列本来也就是基于数组实现的，使用数组的话，减少了系统对内存空间管理的压力，

因为数组不像链表，Java 会定期回收链表中一些不再引用的对象，而数组不会出现空间的新分配和回收问题。

# 关闭 Disruptor

- **disruptor.shutdown() :** 关闭 **Disruptor.** 方法会阻塞, 直至所有的事件都得到处理
- **executor.shutdown() :** 关闭 **Disruptor** 使用的线程池. 如果线程池需要关闭, 必须进行手动关闭 **,Disruptor** 在 **shutdown** 时不会自动关闭使用的线程池