并发编程框架篇（四）

讲师：白鹤翔

目录

1.1 Disruptor并发框架简介

- Martin Fowler在自己网站上写了一篇LMAX架构的文章，在文章中他介绍了LMAX是一种新型零售金融交易平台，它能够以很低的延迟产生大量交易。这个系统是建立在JVM平台上，其核心是一个业务逻辑处理器，它能够在一个线程里每秒处理6百万订单。业务逻辑处理器完全是运行在内存中，使用事件源驱动方式。业务逻辑处理器的核心是Disruptor。
- Disruptor它是一个开源的并发框架，并获得2011 Duke’s 程序框架创新奖，能够在无锁的情况下实现网络的Queue并发操作。
- Disruptor是一个高性能的异步处理框架，或者可以认为是最快的消息框架（轻量的JMS），也可以认为是一个观察者模式的实现，或者事件监听模式的实现。

1.2 Disruptor并发框架使用

- 目前我们使用**disruptor**已经更新到了**3.x**版本，比之前的**2.x**版本性能更加的优秀，提供更多的**API**使用方式。
- 下载**disruptor-3.3.2.jar**引入我们的项目既可以开始**disruptor**之旅。
- 在使用之前，首先说明**disruptor**主要功能加以说明，你可以理解为他是一种高效的"生产者-消费者"模型。也就性能远远高于传统的**BlockingQueue**容器。
- 官方学习网站：**http://ifeve.com/disruptor-getting-started/**

1.3 Disruptor Hello World

- 在Disruptor中，我们想实现hello world 需要如下几步骤：
  - 第一：建立一个Event类
  - 第二：建立一个工厂Event类，用于创建Event类实例对象
  - 第三：需要有一个监听事件类，用于处理数据（Event类）
  - 第四：我们需要进行测试代码编写。实例化Disruptor实例，配置一系列参数。然后我们对Disruptor实例绑定监听事件类，接受并处理数据。
  - 第五：在Disruptor中，真正存储数据的核心叫做RingBuffer，我们通过Disruptor实例拿到它，然后把数据生产出来，把数据加入到RingBuffer的实例对象中即可。
- 我们一起来看下这个HelloWorld程序：com.bjsxt.base

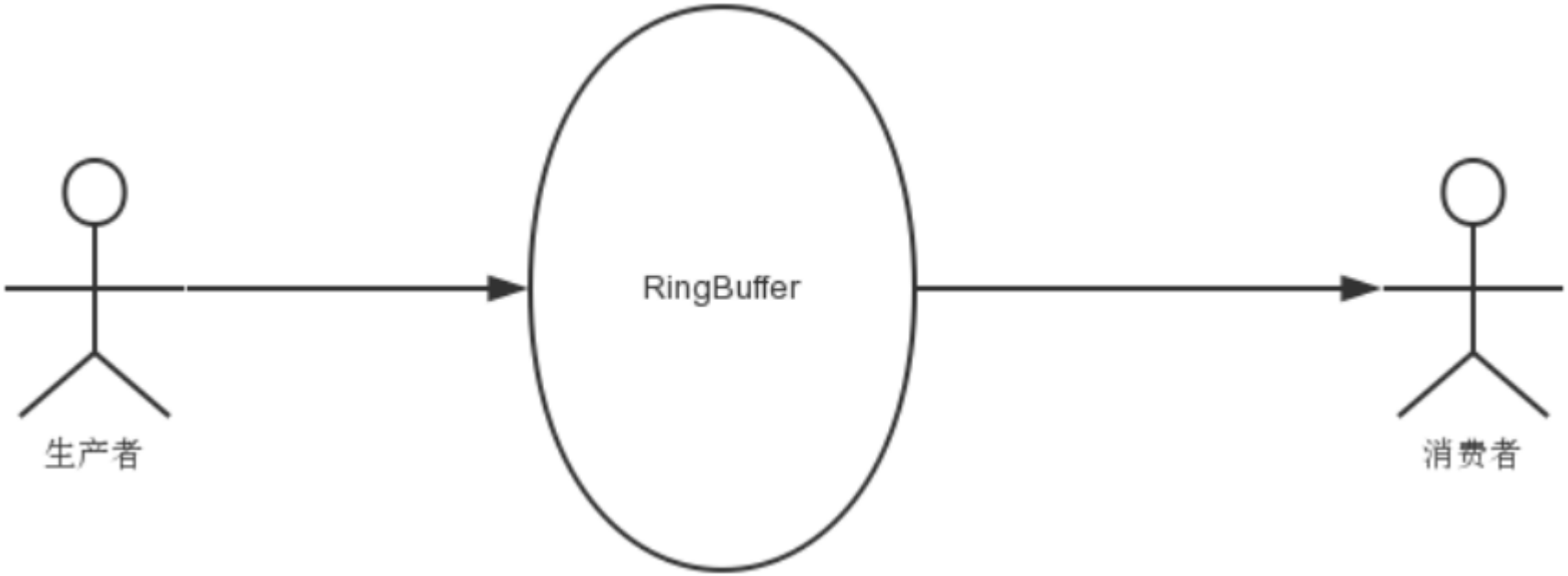
2.1 Disruptor术语说明（一）

- **RingBuffer:** 被看作Disruptor最主要的组件，然而从3.0开始RingBuffer仅仅负责存储和更新在Disruptor中流通的数据。对一些特殊的使用场景能够被用户(使用其他数据结构)完全替代。
- **Sequence:** Disruptor使用Sequence来表示一个特殊组件处理的序号。和Disruptor一样，每个消费者(EventProcessor)都维持着一个Sequence。大部分的并发代码依赖这些Sequence值的运转，因此Sequence支持多种当前为AtomicLong类的特性。
- **Sequencer:** 这是Disruptor真正的核心。实现了这个接口的两种生产者（单生产者和多生产者）均实现了所有的并发算法，为了在生产者和消费者之间进行准确快速的数据传递。
- **SequenceBarrier:** 由Sequencer生成，并且包含了已经发布的Sequence的引用，这些的Sequence源于Sequencer和一些独立的消费者的Sequence。它包含了决定是否有供消费者来消费的Event的逻辑。

2.2 Disruptor术语说明（二）

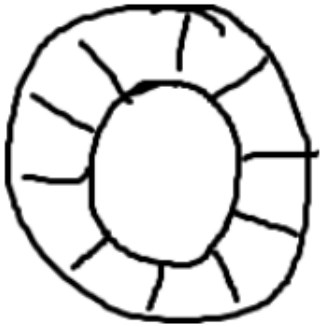
- **WaitStrategy**：决定一个消费者将如何等待生产者将Event置入Disruptor。
- **Event**：从生产者到消费者过程中所处理的数据单元。Disruptor中没有代码表示Event，因为它完全是由用户定义的。
- **EventProcessor**：主要事件循环，处理Disruptor中的Event，并且拥有消费者的Sequence。它有一个实现类是BatchEventProcessor，包含了event loop有效的实现，并且将回调到一个EventHandler接口的实现对象。
- **EventHandler**：由用户实现并且代表了Disruptor中的一个消费者的接口。
- **Producer**：由用户实现，它调用RingBuffer来插入事件(Event)，在Disruptor中没有相应的实现代码，由用户实现。
- **WorkProcessor**：确保每个sequence只被一个processor消费，在同一个WorkPool中的处理多个WorkProcessor不会消费同样的sequence。
- **WorkerPool**：一个WorkProcessor池，其中WorkProcessor将消费Sequence，所以任务可以在实现WorkHandler接口的worker吃间移交
- **LifecycleAware**：当BatchEventProcessor启动和停止时，于实现这个接口用于接收通知。

◆ 初看Disruptor，给人的印象就是RingBuffer是其核心，生产者向RingBuffer中写入元素，消费者从RingBuffer中消费元素，如下图：

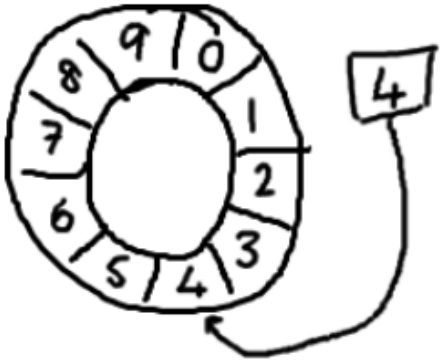


2.4 理解RingBuffer（一）

- ringbuffer到底是什么？
- 答：嗯，正如名字所说的一样，它是一个环（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元素。



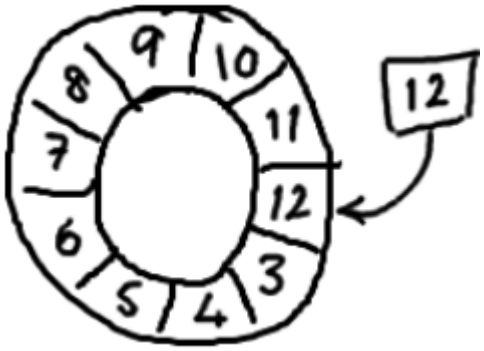
2.5 理解RingBuffer（二）

小故事：

Disruptor说的是生产者和消费者的故事. 有一个数组.生产者往里面扔芝麻.消费者从里面捡芝麻. 但是扔芝麻和捡芝麻也要考虑速度的问题. 1 消费者捡的比扔的快 那么消费者要停下来.生产者扔了新的芝麻,然后消费者继续. 2 数组的长度是有限的,生产者到末尾的时候会再从数组的开始位置继续.这个时候可能会追上消费者,消费者还没从那个地方捡走芝麻,这个时候生产者要等待消费者捡走芝麻,然后继续.

2.6 理解RingBuffer（三）

- 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。



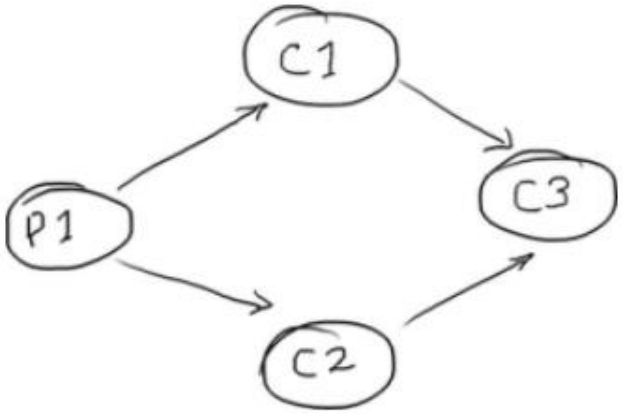
- 要找到数组中当前序号指向的元素，可以通过**mod**操作：**sequence mod array length = array index**（取模操作）以上面的**ringbuffer**为例（**java**的**mod**语法）：**12 % 10 = 2**。很简单吧。
- 事实上，上图中的**ringbuffer**只有**10**个槽完全是个意外。如果槽的个数是**2**的**N**次方更有利于基于二进制的计算机进行计算。

2.7 理解RingBuffer（三）

- 如果你看了维基百科里面的关于环形buffer的词条，你就会发现，我们的实现方式，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。这种实现是经过深思熟虑的—我们选择用环形buffer的最初原因就是想要提供可靠的消息传递。
- 我们实现的ring buffer和大家常用的队列之间的区别是，我们不删除buffer中的数据，也就是说这些数据一直存放在buffer中，直到新的数据覆盖他们。这就是和维基百科版本相比，我们不需要尾指针的原因。ringbuffer本身并不控制是否需要重叠。
- 因为它是数组，所以要比链表快，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。
- 其次，你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。此外，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。

3.1 场景使用

- 在helloWorld的实例中，我们创建Disruptor实例，然后调用getRingBuffer方法去获取RingBuffer，其实在很多时候，我们可以直接使用RingBuffer，以及其他的API操作。我们一起熟悉下示例：
- com.bjsxt.generate1
  - 使用EventProcessor消息处理器。
  - 使用WorkerPool消息处理器。
- com.bjsxt.generate2
  - 在复杂场景下使用RingBuffer（希望P1生产的数据给C1、C2并行执行，最后C1、C2执行结束后C3执行）



- com.bjsxt.multi，多生产者、消费者使用。

END