# JAVA并发编程深度学习-无锁并行计算框架

## 并发编程与无锁并行计算框架初探

|  |
| --- |
| **public interface** Constants {   **int *EVENT\_NUM\_OHM*** = 100000000;    **int *EVENT\_NUM\_FM*** = 50000000;    **int *EVENT\_NUM\_OM*** = 10000000;   } |

|  |
| --- |
| **public class** ArrayBlockingQueue4Test {  **public static void** main(String[] args) {  **final** ArrayBlockingQueue<Message> queue = **new** ArrayBlockingQueue<>(Constants.***EVENT\_NUM\_OM***);  **final long** startTime = System.*currentTimeMillis*();  **new** Thread(()->{  **long** i = 0;  **while** (i < Constants.EVENT\_NUM\_OHM) {  Message message = **new** Message(i, **"c"** + i);  **try** {  queue.put(message);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  i++;  }  }).start();   **new** Thread(()->{  **int** k = 0;  **while** (k < Constants.EVENT\_NUM\_OHM) {  **try** {  queue.take();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  k++;  }  **long** endTime = System.currentTimeMillis();  System.out.println(**"ArrayBlockingQueue costTime = "** + (endTime - startTime) + **"ms"**);  }).start();  } } |
| ArrayBlockingQueue costTime = 27912ms |

|  |
| --- |
| **public class** DataConsumer **implements** EventHandler<Message> {   **private long startTime**;  **private int i**;   **public** DataConsumer() {  **this**.**startTime** = System.*currentTimeMillis*();  }   **public void** onEvent(Message data, **long** seq, **boolean** bool)  **throws** Exception {  **i**++;  **if** (**i** == Constants.***EVENT\_NUM\_FM***) {  **long** endTime = System.*currentTimeMillis*();  System.***out***.println(**"Disruptor costTime = "** + (endTime - **startTime**) + **"ms"**);  }  } } |

|  |
| --- |
| **public class** DisruptorSingle4Test {  **public static void** main(String[] args) {  **int** ringBufferSize = 65536;  **final** Disruptor<Message> disruptor = **new** Disruptor<>(  **new** EventFactory<Message>() {  @Override  **public** Message newInstance() {  **return new** Message();  }  },  ringBufferSize,  Executors.*newSingleThreadExecutor*(),  ProducerType.***SINGLE***,  **new** YieldingWaitStrategy()  );  DataConsumer consumer = **new** DataConsumer();  disruptor.handleEventsWith(consumer);  disruptor.start();  **new** Thread(()->{  RingBuffer<Message> ringBuffer = disruptor.getRingBuffer();  **for** (**long** i = 0; i < Constants.EVENT\_NUM\_FM; i++) {  **long** seq = ringBuffer.next();  Message data = ringBuffer.get(seq);  data.setId(i);  data.setName(**"c"** + i);  ringBuffer.publish(seq);  }  }).start();  } } |
| Disruptor costTime = 7458ms |

## 并发编程框架核心讲解

### 2.1 Disruptor-QuickStart-基础元素工厂类

**Disruptor实践简介**

·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

·这个系统建立在JVM平台上，其核心是一个业务逻辑处理器

**Disruptor性能及核心**

·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，它使用事件源驱动方式

·业务逻辑处理器的核心是Disruptor

**Disruptor基本使用**

·建立一个工厂Event类，用于创建Event类实例对象

·需要有一个监听事件类，用于处理数据（Event类）

·实例化Disruptor实例，配置一系列参数，编写Disruptor核心组件

·编写生产者组件，向Disruptor容器中去投递数据

**生成-消费模型**

|  |
| --- |
| @Data **public class** OrderEvent {  **private long value**; } |

基础元素工厂类

|  |
| --- |
| **public class** OrderEventFactory **implements** EventFactory<OrderEvent> {  @Override  **public** OrderEvent newInstance() {  *// 这个方法就是为了返回空的OrderEvent对象* **return new** OrderEvent();  } } |

消费端事件处理器

|  |
| --- |
| @Slf4j **public class** OrderEventHandler **implements** EventHandler<OrderEvent> {  @Override  **public void** onEvent(OrderEvent orderEvent, **long** l, **boolean** b) **throws** Exception {  ***log***.info(**"Customer event value:{}"**, orderEvent.getValue());  } } |

生产者组件投递数据

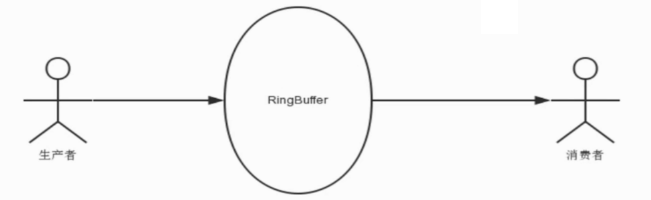
|  |
| --- |
| @Data @AllArgsConstructor **public class** OrderEventProducer {  **private** RingBuffer<OrderEvent> **ringBuffer**;  **public void** setData(ByteBuffer byteBuffer){  *// 1 在生产者发送消息的时候，首先需要从ringBuffer中获取一个可用的序号* **long** sequence = **ringBuffer**.next();  **try**{  *// 2 根据这个序号，找到具体OrderEvent元素，注意：此时获取的OrderEvent对象是一个没有被赋值空对象* OrderEvent event = **ringBuffer**.get(sequence);  *// 3 进行实际的赋值处理* event.setValue(byteBuffer.getLong(0));  } **finally** {  *// 4 提交发布操作* **ringBuffer**.publish(sequence);  }  } } |

|  |
| --- |
| **public class** Main {  **public static void** main(String[] args) {  *// 1.实例化Disruptor对象* OrderEventFactory orderEventFactory = **new** OrderEventFactory();  **int** ringBufferSize = 1024 \* 1024;  ExecutorService executor = Executors.*newFixedThreadPool*(Runtime.*getRuntime*().availableProcessors());  */\*\*  \* 1 eventFactory: 消息(event)工厂对象  \* 2 ringBufferSize: 容器的长度  \* 3 executor: 线程池(建议使用自定义线程池) RejectedExecutionHandler  \* 4 ProducerType: 单生产者 还是 多生产者  \* 5 waitStrategy: 等待策略  \*/* Disruptor<OrderEvent> disruptor = **new** Disruptor<>(orderEventFactory,  ringBufferSize,  executor,  ProducerType.***SINGLE***,  **new** BlockingWaitStrategy());    *// 2. 添加消费者的监听(构建disruptor 与 消费者的一个关联关系)，处理OrderEvent类* disruptor.handleEventsWith(**new** OrderEventHandler());   *// 3. 启动disruptor* disruptor.start();   *// 4. 获取实际存储数据的容器：RingBuffer* RingBuffer<OrderEvent> ringBuffer = disruptor.getRingBuffer();   OrderEventProducer producer = **new** OrderEventProducer(ringBuffer);  ByteBuffer byteBuffer = ByteBuffer.*allocate*(8);  **for** (**int** i = 0; i < 100; i++) {  byteBuffer.putLong(0,i);  producer.setData(byteBuffer);  }   executor.shutdown();  disruptor.shutdown();  } } |
| **......**  **22:28:04.561 [pool-1-thread-1] INFO com.byf.disruptor.quickstart.OrderEventHandler - Customer event value:96**  **22:28:04.561 [pool-1-thread-1] INFO com.byf.disruptor.quickstart.OrderEventHandler - Customer event value:97**  **22:28:04.561 [pool-1-thread-1] INFO com.byf.disruptor.quickstart.OrderEventHandler - Customer event value:98**  **22:28:04.561 [pool-1-thread-1] INFO com.byf.disruptor.quickstart.OrderEventHandler - Customer event value:99** |

### 2.2Disruptor核心原理

1. Disruptor核心原理

·初看Disruptor，给人的印象就是RingBuffer是核心，生产者想RingBuffer中写入元素，消费者从RingBuffer中消费元素



1. RingBuffer到底是啥

·正如名字所说一样，他是一个环（首尾相接的环）

·它在做不同上下文（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



（3）Producer的关键步骤

ringBuffer.next(); 取出数组中下一个可用元素（空Event对象的序号）

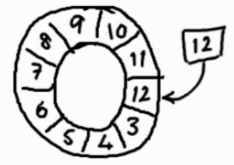
根据序号取出空对象；

OrderEvent event = ringBuffer.get(sequence);

event.setValue(...);

### 2.3Disruptor-仍芝麻与捡芝麻小故事

1. 消费者捡的比生产者扔的快，那么消费者要停下来，等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，生成者到末尾的时候回再从数据的开始位置继续。这时可能会追上消费者，消费者还没从哪个地方捡走芝麻，这个时候生产者要等待消费者捡走芝麻，然后继续；
3. 随着生产者不停填充这个Buffer（可能也会有对应的读取），这个序号会一直增长，直到绕过这个环（覆盖消费者还未取走的元素）；



注意：RingBuffer的槽的数量通常是2的n次方，有利于基于二进制的取模运算；

### 2.4Disruptor核心-RingBuffer

·RingBuffer：基于数组的缓存实现，也是创建sequence与定义WaitStrategy的入口；

·Disruptor：持有RingBuffer、消费者线程池Executor、消费者集合ConsumerRepository等引用；

### 2.5Disruptor-核心-Sequence、Sequencer、SequenceBarrier

1. Disruptor-核心-Sequence

·通过顺序递增的序号来编号，管理进行交换的数据（事件）

·对数据（事件）的处理过程总是沿着序号诸葛递增处理

·一个Sequence用于跟踪某个特定的事件处理者（RingBuffer/Producer/Consumer）的处理进度

（2）Disruptor-核心-Sequencer

·Sequence可以看成一个AtomicLong用于标识进度；

·还有另外一个目的就是防止不同Sequence之间CPU缓存伪共享（Flase Sharing）的问题

**Sequencer是Disruptor的真正核心**

此接口有两个实现类

·SingleProducerSequencer

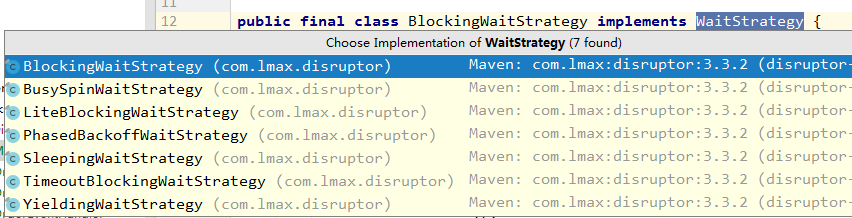
·MultiProducerSequencer

主要实现生产者和消费者之间快速、正确地传递数据的并发算法

（3）Disruptor核心-Sequence Barrier

·用于保持对RingBuffer的Main Published Sequence（Producer）和Consumer之间的平衡关系；Sequence Barrier还定义了决定Consumer是否还有可处理的事件的逻辑

### 2.6Disruptor核心-WaitStrategy消费者等待策略



·BlockingWaitStartegy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；

### 2.7Disruptor-核心-Event、EventProcessor、EventHandler、WorkProcessor

（1）Disruptor-核心-Event

·Event：从生产者到消费者过程中所处理的数据单元；

·Disruptor中没有代码标识Event，因为它完全是由用户定义的

（2）Disruptor-核心-EventProcessor

·EventProcessor：主要事件循环，处理Disruptor中的Event，拥有消费者的Sequence；

·它有一个实现类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；

（3）Disruptor-核心-EventHandler

·EventHandler：由用户实现并代表了Disruptor中的一个消费者的接口，也就是我们的消费者逻辑都需要写在这里；

（4）Disruptor-核心-WorkProcessor

·WorkProcessor：确保每个sequence只被一个processor消费，在同一个WorkPool中处理多个WorkProcessor不会消费同样的sequence；

### 2.8Disruptor-核心概念整体图解

