# 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

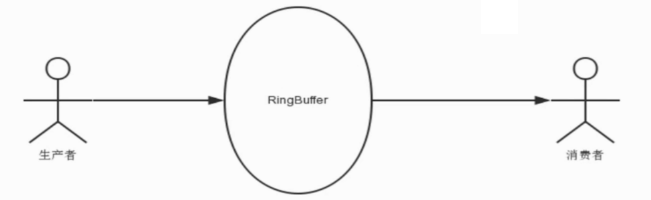
|  |
| --- |
| @Data @AllArgsConstructor **public class** OrderEventProducer {  **private** RingBuffer<OrderEvent> **ringBuffer**;  **public void** setData(ByteBuffer byteBuffer){  *// 1 在生产者发送消息的时候，首先需要从ringBuffer中获取一个可用的序号* **long** sequence = **ringBuffer**.next();  **try**{  *// 2 根据这个序号，找到具体OrderEvent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）

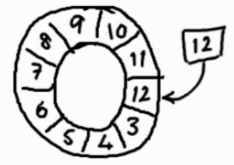
根据序号取出空对象；

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

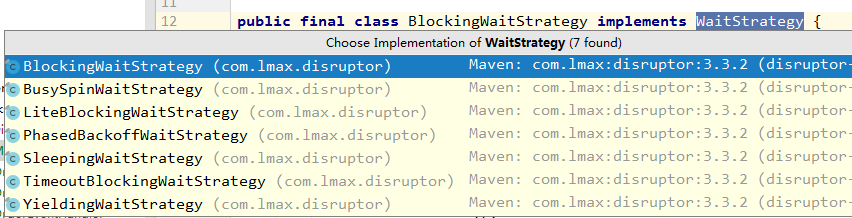
·MultiProducerSequencer

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；

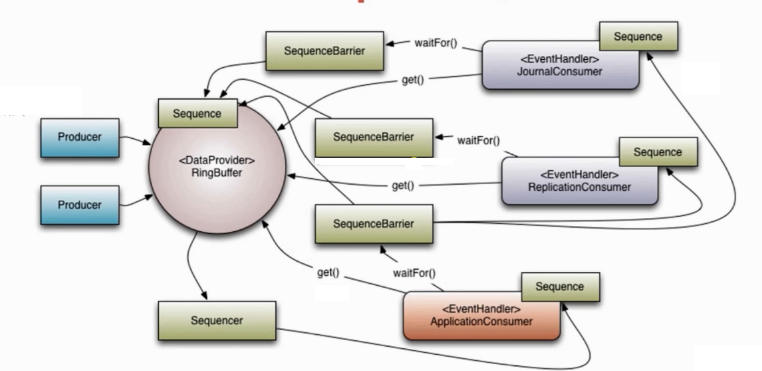
（3）Disruptor-核心-EventHandler

·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



## 第3节并发编程框架高级特性讲解

·Disruptor核心链路场景应用

·并行计算-串行操作

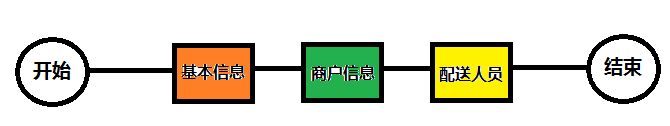
·并行计算-并行操作

·并行计算-多边形操作

·并行编程-多生产者模型

·并行编程-多消费者模型

### 3.1Disruptor核心链路场景应用



核心链路特点：至关重要且业务复杂，那么代码应该如何实现？

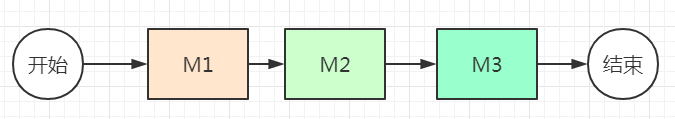
·传统方式完全解耦

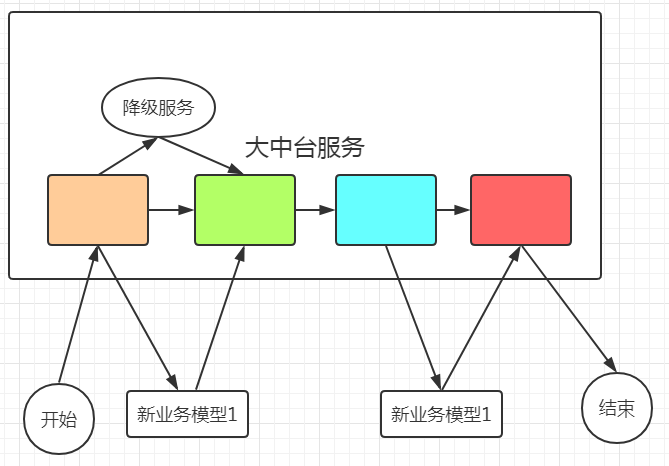
·模板方法

解决手段：

·领域模型的高度抽象

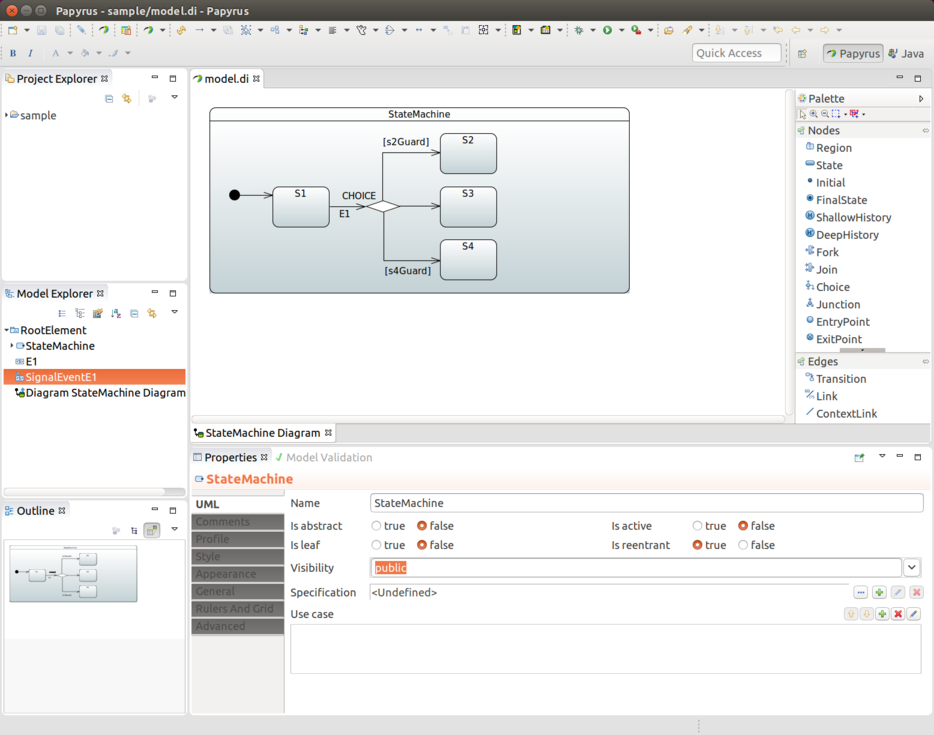
·寻找更好的编程框架





使用框架：

·有限状态机框架，例如Spring-StateMachine



·使用Disruptor

### 3.2并行计算--串、并行操作



|  |
| --- |
| */\*\*  \* Disruptor中的Event  \*/* @Data @NoArgsConstructor **public class** Trade {  **private** String **id**;  **private** String **name**;  **private double price**;  **private** AtomicInteger **count** = **new** AtomicInteger(0); } |

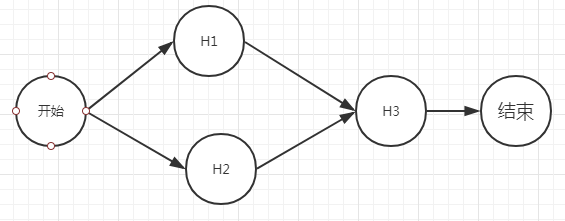
|  |
| --- |
| @AllArgsConstructor **public class** TradePublisher **implements** Runnable {  **private** Disruptor<Trade> **disruptor**;  **private** CountDownLatch **latch**;  **private static int** *PUBLISH\_COUNT* = 1;  @Override  **public void** run() {  TradeEventTranslator eventTranslator = **new** TradeEventTranslator();  **for** (**int** i = 0; i < *PUBLISH\_COUNT*; i++) {  **disruptor**.publishEvent(eventTranslator);  }  **latch**.countDown();  } }  **class** TradeEventTranslator **implements** EventTranslator<Trade>{  **private** Random **random** = **new** Random();  @Override  **public void** translateTo(Trade event, **long** sequence) {  **this**.generateTrade(event);  }   **private void** generateTrade(Trade event) {  event.setPrice(**random**.nextDouble() \* 9999);  } } |

|  |
| --- |
| @Slf4j **public class** Handler1 **implements** EventHandler<Trade>, WorkHandler<Trade> {  @Override  **public void** onEvent(Trade event, **long** l, **boolean** b) **throws** Exception {  **this**.onEvent(event);  }   @Override  **public void** onEvent(Trade event) **throws** Exception {  ***log***.info(**"Handler1 : SET NAME"**);  event.setName(**"H1"**);  Thread.*sleep*(1000);  } } |
| @Slf4j **public class** Handler2 **implements** EventHandler<Trade> {  @Override  **public void** onEvent(Trade event, **long** l, **boolean** b) **throws** Exception {  ***log***.info(**"Handler2 : SET ID"**);  event.setId(UUID.*randomUUID*().toString());  Thread.*sleep*(2000);  } } |
| @Slf4j **public class** Handler3 **implements** EventHandler<Trade> {  @Override  **public void** onEvent(Trade event, **long** l, **boolean** b) **throws** Exception {  ***log***.info(**"Handler3 : NAME:{}, ID:{}, INSTANCE:{}"**,event.getName(),event.getId(),event.toString());  Thread.*sleep*(3000);  } } |

|  |
| --- |
| @Slf4j **public class** Main {  **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = **new** ThreadPoolExecutor(8,16, 1000,TimeUnit.MILLISECONDS,  **new** LinkedBlockingDeque<>(1000));   Disruptor<Trade> disruptor = **new** Disruptor<>(  **new** EventFactory<Trade>() {  @Override  **public** Trade newInstance() {  **return new** Trade();  }  },  1024 \* 1024,  Executors.*newFixedThreadPool*(8),  ProducerType.***SINGLE***,  **new** BusySpinWaitStrategy()  );   *// 串行操作* **final** EventHandlerGroup<Trade> tradeEventHandlerGroup = disruptor  .handleEventsWith(**new** Handler1())  .handleEventsWith(**new** Handler2())  .handleEventsWith(**new** Handler3());   *// 并行操作  // disruptor.handleEventsWith(new Handler1(),new Handler2(),new Handler3());* ***log***.info(**"开始"**);  **long** begin = System.*currentTimeMillis*();  RingBuffer<Trade> ringBuffer = disruptor.start();  CountDownLatch latch = **new** CountDownLatch(1);  exec.submit(**new** TradePublisher(disruptor,latch));  latch.await();  disruptor.shutdown();  exec.shutdown();  ***log***.info(**"总耗时：{}"**,(System.*currentTimeMillis*()-begin));  } } |
| 21:02:54.866 [main] INFO com.byf.disruptor.advance.Main - 开始  21:02:55.019 [pool-2-thread-1] INFO com.byf.disruptor.advance.Handler1 - Handler1 : SET NAME  21:02:56.024 [pool-2-thread-2] INFO com.byf.disruptor.advance.Handler2 - Handler2 : SET ID  21:02:58.314 [pool-2-thread-3] INFO com.byf.disruptor.advance.Handler3 - Handler3 : NAME:H1, ID:dc446962-0c4f-4eae-8f4e-2104b0f6e6e5, INSTANCE:Trade(id=dc446962-0c4f-4eae-8f4e-2104b0f6e6e5, name=H1, price=5383.095610087307, count=0)  21:03:01.314 [main] INFO com.byf.disruptor.advance.Main - 总耗时：6302 |

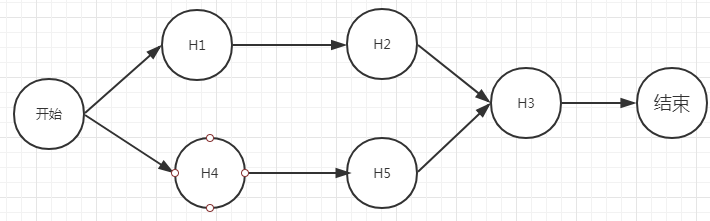
|  |
| --- |
| *// 并行操作* disruptor.handleEventsWith(**new** Handler1(),**new** Handler2(),**new** Handler3()); |
| 21:03:44.468 [main] INFO com.byf.disruptor.advance.Main - 开始  21:03:44.593 [pool-2-thread-2] INFO com.byf.disruptor.advance.Handler2 - Handler2 : SET ID  21:03:44.593 [pool-2-thread-1] INFO com.byf.disruptor.advance.Handler1 - Handler1 : SET NAME  21:03:44.593 [pool-2-thread-3] INFO com.byf.disruptor.advance.Handler3 - Handler3 : NAME:null, ID:null, INSTANCE:Trade(id=null, name=null, price=9969.155325920348, count=0)  21:03:47.620 [main] INFO com.byf.disruptor.advance.Main - 总耗时：3027 |

### 3.3并行计算--菱形操作

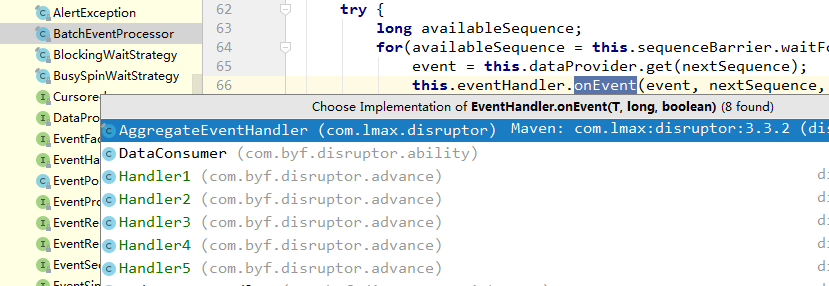


|  |
| --- |
| @Slf4j **public class** Main {  **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = **new** ThreadPoolExecutor(8,16, 1000,TimeUnit.MILLISECONDS,  **new** LinkedBlockingDeque<>(1000));   Disruptor<Trade> disruptor = **new** Disruptor<>(  **new** EventFactory<Trade>() {  @Override  **public** Trade newInstance() {  **return new** Trade();  }  },  1024 \* 1024,  Executors.*newFixedThreadPool*(8),  ProducerType.***SINGLE***,  **new** BusySpinWaitStrategy()  );   *// 串行操作  /\*final EventHandlerGroup<Trade> tradeEventHandlerGroup = disruptor  .handleEventsWith(new Handler1())  .handleEventsWith(new Handler2())  .handleEventsWith(new Handler3());\*/   // 并行操作  // disruptor.handleEventsWith(new Handler1(),new Handler2(),new Handler3());   // 菱形操作（一）  // disruptor.handleEventsWith(new Handler1(),new Handler2()).handleEventsWith(new Handler3());  // 菱形操作（二）* EventHandlerGroup<Trade> eventHandlerGroup = disruptor.handleEventsWith(**new** Handler1(), **new** Handler2());  eventHandlerGroup.then(**new** Handler3());  ***log***.info(**"开始"**);  **long** begin = System.*currentTimeMillis*();  RingBuffer<Trade> ringBuffer = disruptor.start();  CountDownLatch latch = **new** CountDownLatch(1);  exec.submit(**new** TradePublisher(disruptor,latch));  latch.await();  disruptor.shutdown();  exec.shutdown();  ***log***.info(**"总耗时：{}"**,(System.*currentTimeMillis*()-begin));  } } |
| 21:15:29.882 [main] INFO com.byf.disruptor.advance.Main - 开始  21:15:29.970 [pool-2-thread-2] INFO com.byf.disruptor.advance.Handler2 - Handler2 : SET ID  21:15:29.970 [pool-2-thread-1] INFO com.byf.disruptor.advance.Handler1 - Handler1 : SET NAME  21:15:33.502 [pool-2-thread-3] INFO com.byf.disruptor.advance.Handler3 - Handler3 : NAME:H1, ID:fc102675-7108-432a-8fde-82a257198510, INSTANCE:Trade(id=fc102675-7108-432a-8fde-82a257198510, name=H1, price=5537.542398500754, count=0)  21:15:33.506 [main] INFO com.byf.disruptor.advance.Main - 总耗时：3546 |

### 3.4并行计算--多边形操作

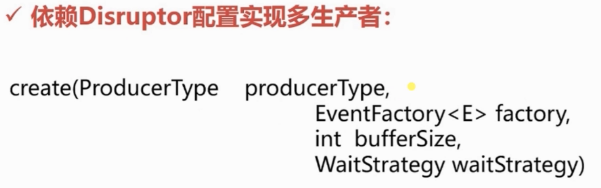


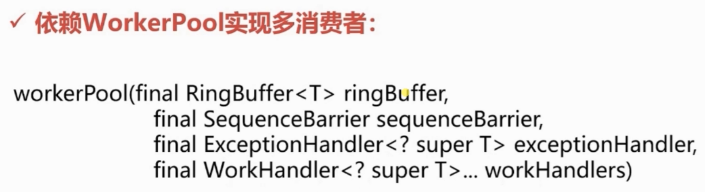
·EventProcessor：主要事件循环，处理Disruptor中的Event，拥有消费者的Sequence

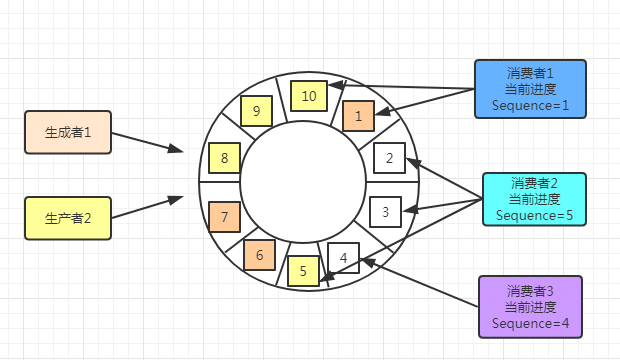


|  |
| --- |
| @Slf4j **public class** Main {  **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService es1 = **new** ThreadPoolExecutor(1,16, 1000,TimeUnit.MILLISECONDS,  **new** LinkedBlockingDeque<>(1000));  // 线程池大小为5的原因：BatchEventProcessor单线程处理Handler，单消费者模型  ExecutorService es2 = **new** ThreadPoolExecutor(5,16, 1000,TimeUnit.MILLISECONDS,  **new** LinkedBlockingDeque<>(1000));  Disruptor<Trade> disruptor = **new** Disruptor<>(  **new** EventFactory<Trade>() {  @Override  **public** Trade newInstance() {  **return new** Trade();  }  },  1024 \* 1024,  es2,  ProducerType.***SINGLE***,  **new** BusySpinWaitStrategy()  );   *// 串行操作  /\*final EventHandlerGroup<Trade> tradeEventHandlerGroup = disruptor  .handleEventsWith(new Handler1())  .handleEventsWith(new Handler2())  .handleEventsWith(new Handler3());\*/   // 并行操作  // disruptor.handleEventsWith(new Handler1(),new Handler2(),new Handler3());   // 菱形操作（一）  // disruptor.handleEventsWith(new Handler1(),new Handler2()).handleEventsWith(new Handler3());  // 菱形操作（二）  // EventHandlerGroup<Trade> eventHandlerGroup = disruptor.handleEventsWith(new Handler1(), new Handler2());  // eventHandlerGroup.then(new Handler3());   // 六边形操作* Handler1 handler1 = **new** Handler1();  Handler2 handler2 = **new** Handler2();  Handler3 handler3 = **new** Handler3();  Handler4 handler4 = **new** Handler4();  Handler5 handler5 = **new** Handler5();  disruptor.handleEventsWith(handler1,handler4);  disruptor.after(handler1).handleEventsWith(handler2);  disruptor.after(handler4).handleEventsWith(handler5);  disruptor.after(handler2,handler5).handleEventsWith(handler3);  ***log***.info(**"开始"**);  **long** begin = System.*currentTimeMillis*();  RingBuffer<Trade> ringBuffer = disruptor.start();  CountDownLatch latch = **new** CountDownLatch(1);  es1.submit(**new** TradePublisher(disruptor,latch));  latch.await(); *// disruptor.shutdown(); // es1.shutdown(); // es2.shutdown();* ***log***.info(**"总耗时：{}"**,(System.*currentTimeMillis*()-begin));  } } |
| 21:43:06.079 [main] INFO com.byf.disruptor.advance.Main - 开始  21:43:06.147 [pool-2-thread-1] INFO com.byf.disruptor.advance.Handler1 - Handler1 : SET NAME  21:43:06.147 [pool-2-thread-2] INFO com.byf.disruptor.advance.Handler4 - Handler4 : SET PRICE  21:43:06.148 [main] INFO com.byf.disruptor.advance.Main - 总耗时：9  21:43:06.147 [pool-2-thread-4] INFO com.byf.disruptor.advance.Handler5 - Handler5 : GET PRICE:17.2  21:43:07.147 [pool-2-thread-3] INFO com.byf.disruptor.advance.Handler2 - Handler2 : SET ID  21:43:09.623 [pool-2-thread-5] INFO com.byf.disruptor.advance.Handler3 - Handler3 : NAME:H1, ID:caa4ea56-eecc-4b80-87e6-5529b20a7264, INSTANCE:Trade(id=caa4ea56-eecc-4b80-87e6-5529b20a7264, name=H1, price=17.2, count=0) |

### 3.5并行计算--多生产者-多消费者模型讲解







|  |
| --- |
| */\*\*  \* Disruptor中的Event  \*/* @Data @NoArgsConstructor **public class** Order {  **private** String **id**;  **private** String **name**;  **private double price**; } |

|  |
| --- |
| @Data @Slf4j **public class** Customer **implements** WorkHandler<Order> {  **private** String **customerId**;  **private static** AtomicInteger *count* = **new** AtomicInteger(0);  **private** Random **random** = **new** Random();   **public** Customer(String customerId){  **this**.**customerId** = customerId;  }   @Override  **public void** onEvent(Order event) **throws** Exception {  Thread.*sleep*(1 \* **random**.nextInt(5));  ***log***.info(**"Customer:{}, event ID:{}"**,**this**.**customerId**,event.getId());  *count*.getAndIncrement();  }   **public int** getCount(){  **return** *count*.get();  } } |

|  |
| --- |
| @AllArgsConstructor **public class** Producer {  **private** RingBuffer<Order> **ringBuffer**;  **public void** sendData(String uuid){  **long** sequence = **ringBuffer**.next();  **try**{  Order order = **ringBuffer**.get(sequence);  order.setId(uuid);  } **finally** {  **ringBuffer**.publish(sequence);  }  } } |

|  |
| --- |
| @Slf4j **public class** Main {  **public static void** main(String[] args) **throws** InterruptedException {  RingBuffer<Order> ringBuffer =  RingBuffer.*create*(ProducerType.***MULTI***,  **new** EventFactory<Order>() {  @Override  **public** Order newInstance() {  **return new** Order();  }  },  1024 \* 1024,  **new** YieldingWaitStrategy());  *// 2 通过RingBuffer创建一个屏障* SequenceBarrier sequenceBarrier = ringBuffer.newBarrier();   *// 3 构建多消费者* Customer[] customers = **new** Customer[10];  **for** (**int** i = 0; i < customers.**length**; i++) {  customers[i] = **new** Customer(**"C"** + i);  }   *// 4 构建多消费者工作池* WorkerPool<Order> workerPool = **new** WorkerPool<>(  ringBuffer,  sequenceBarrier,  **new** EventExceptionHandler(),  customers  );   *// 5 设置多个消费者的Sequence序号，用于单独统计消费进度* ringBuffer.addGatingSequences(workerPool.getWorkerSequences());   *// 6 启动workerPool* workerPool.start(Executors.*newFixedThreadPool*(Runtime.*getRuntime*().availableProcessors()));   CountDownLatch latch = **new** CountDownLatch(1);  **for** (**int** i = 0; i < 100; i++) {  Producer producer = **new** Producer(ringBuffer);  **new** Thread(()->{  **try**{  latch.await();  } **catch** (Exception e){  log.error(**"exception"**,e);  }  **for** (**int** j = 0; j < 100; j++) {  producer.sendData(UUID.randomUUID().toString());  }  }).start();  }   Thread.*sleep*(2000);  ***log***.info(**"-----------线程创建完毕，开始生产数据--------------"**);  latch.countDown();  Thread.*sleep*(5000);  ***log***.info(**"第3个消费者处理总数：{}"**,customers[2].getCount());  }  **static class** EventExceptionHandler **implements** ExceptionHandler<Order>{   @Override  **public void** handleEventException(Throwable throwable, **long** l, Order order) {   }   @Override  **public void** handleOnStartException(Throwable throwable) {   }   @Override  **public void** handleOnShutdownException(Throwable throwable) {   }  } } |
| ......  23:55:29.432 [pool-1-thread-5] INFO com.byf.disruptor.advance.multi.Customer - Customer:C4, event ID:dcc0ee83-c11f-45b4-bc11-61105de7b8d4  23:55:29.432 [pool-1-thread-2] INFO com.byf.disruptor.advance.multi.Customer - Customer:C1, event ID:61d7cb1e-90d2-4af5-8e3a-7a566142a1c7  23:55:29.434 [pool-1-thread-8] INFO com.byf.disruptor.advance.multi.Customer - Customer:C7, event ID:30117119-0a7c-42a1-8c85-cbc066dc4850  23:55:29.434 [pool-1-thread-7] INFO com.byf.disruptor.advance.multi.Customer - Customer:C6, event ID:9ef28254-b01e-467d-bc76-76edf9114499  23:55:29.434 [pool-1-thread-3] INFO com.byf.disruptor.advance.multi.Customer - Customer:C2, event ID:faf6f0d3-db2e-4634-91e6-35e0d4f91827  23:55:29.435 [pool-1-thread-6] INFO com.byf.disruptor.advance.multi.Customer - Customer:C5, event ID:9b05790a-b142-4809-9ad9-9d851128f96d  23:55:29.436 [pool-1-thread-4] INFO com.byf.disruptor.advance.multi.Customer - Customer:C3, event ID:f482f04c-de49-41a8-b853-95a52c573e89  23:55:31.397 [main] INFO com.byf.disruptor.advance.multi.Main - 第3个消费者处理总数：10000 |

## 第4节并发编程深入学习

·并发编程类

·Volatile、Atomic、UnSafe

·JUC工具类：CountDownLatch、CyclicBarrier、Semaphone、Future、Exchange、ForkJoin

·ReentrantLock、Condition、ReadWriteLock、LockSupport

·AQS架构详解

·AQS底层代码深入分析

### 4.1并发容器类

（1）ConcurrentHashMap

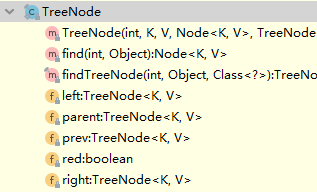
初始容量大小

|  |
| --- |
| */\*\*  \* The default initial table capacity. Must be a power of 2  \* (i.e., at least 1) and at most MAXIMUM\_CAPACITY.  \*/* **private static final int *DEFAULT\_CAPACITY*** = 16; |

分段加锁

|  |
| --- |
| */\*\*  \* Stripped-down version of helper class used in previous version,  \* declared for the sake of serialization compatibility  \*/* **static class** Segment<K,V> **extends** ReentrantLock **implements** Serializable {  **private static final long *serialVersionUID*** = 2249069246763182397L;  **final float loadFactor**;  Segment(**float** lf) { **this**.**loadFactor** = lf; } } |

红黑树



JDK1.7和JDK1.8中ConcurrentHashMap的区别

1、在JDK1.8中ConcurrentHashMap的实现方式有了很大的改变，在JDK1.7中采用的是Segment + HashEntry，而Sement继承了ReentrantLock，所以自带锁功能，而在JDK1.8中则取消了Segment，作者认为Segment太过臃肿，采用Node + CAS + Synchronized（ps:Synchronized一直以来被各种吐槽性能差，但java一直没有放弃Synchronized，也一直在改进，既然作者在这里采用了Synchronized，可见Synchronized的性能应该是有所提升的，当然只是猜想哈哈哈。。。）

2、在上篇HashMap中我们知道，在JDK1.8中当HashMap的链表个数超过8时，会转换为红黑树，在ConcurrentHashMap中也不例外。这也是新能的一个小小提升。

3、在JDK1.8版本中，对于size的计算，在扩容和addCount()时已经在处理了。JDK1.7是在调用时才去计算。

为了帮助统计size，ConcurrentHashMap提供了baseCount和counterCells两个辅助变量和CounterCell辅助类，1.8中使用一个volatile类型的变量baseCount记录元素的个数，当插入新数据或则删除数据时，会通过addCount()方法更新baseCount。

1. CopyOnWriteArrayList

写操作加锁，并拷贝副本操作，再setArray

|  |
| --- |
| */\*\*  \* Appends the specified element to the end of this list.  \*  \** ***@param e*** *element to be appended to this list  \** ***@return*** *{****@code*** *true} (as specified by {****@link*** *Collection#add})  \*/* **public boolean** add(E e) {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  Object[] elements = getArray();  **int** len = elements.**length**;  Object[] newElements = Arrays.*copyOf*(elements, len + 1);  newElements[len] = e;  setArray(newElements);  **return true**;  } **finally** {  lock.unlock();  } } |

1. ArrayBlockingQueue、LinkedBlockingQueue

有界队列、无界队列

1. SynchronousQueue、PriorityBlockingQueue

A、B线程操作一个队列，A放数据，B取数据；实现Comparable接口比较优先级；

（5）DelayQueue

超时失效的设置

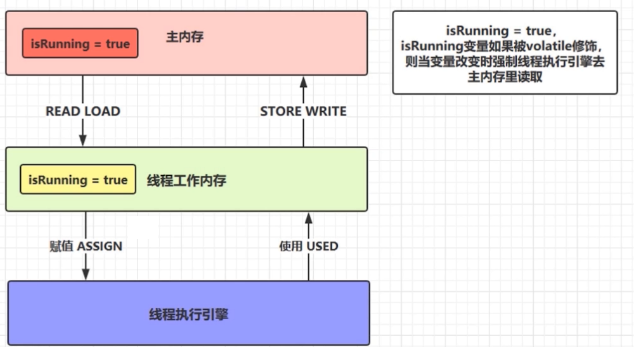
|  |
| --- |
| @ToString **public class** DelayTask **implements** Delayed {  **private** String **name**;  **private long start** = System.*currentTimeMillis*();  **private long time**;  **public** DelayTask(String name,**long** time) {  **this**.**name** = name;  **this**.**time** = time;  }  */\*\*  \* 需要实现的接口，获得延迟时间 用过期时间-当前时间  \** ***@param unit*** *\** ***@return*** *\*/* @Override  **public long** getDelay(TimeUnit unit) {  **return** unit.convert((**start**+**time**) - System.*currentTimeMillis*(),TimeUnit.***MILLISECONDS***);  }    @Override  **public int** compareTo(Delayed o) {  DelayTask task = (DelayTask) o;  **return** (**int**) (**this**.getDelay(TimeUnit.***MILLISECONDS***) - o.getDelay(TimeUnit.***MILLISECONDS***));  } } |

|  |
| --- |
| **public class** Main {  **private static** DelayQueue *delayQueue* = **new** DelayQueue();   **public static void** main(String[] args) **throws** InterruptedException {  DelayTask dt1 = **new** DelayTask(**"dt1"**, 1000);  DelayTask dt2 = **new** DelayTask(**"dt2"**, 2000);  DelayTask dt3 = **new** DelayTask(**"dt3"**, 4000);  DelayTask dt4 = **new** DelayTask(**"dt4"**, 6000);  DelayTask dt5 = **new** DelayTask(**"dt5"**, 3000);  DelayTask dt6 = **new** DelayTask(**"dt6"**, 5000);   **new** Thread(()->{  delayQueue.offer(dt1);  delayQueue.offer(dt2);  delayQueue.offer(dt3);  delayQueue.offer(dt4);  delayQueue.offer(dt5);  delayQueue.offer(dt6);  }).start();   **while** (**true**){  Delayed take = *delayQueue*.take();  System.***out***.println(take);  **if** (*delayQueue*.size() == 0) {  **break**;  }  }  } } |
| **DelayTask(name=dt1, start=1563636427741, time=1000)**  **DelayTask(name=dt2, start=1563636427741, time=2000)**  **DelayTask(name=dt5, start=1563636427741, time=3000)**  **DelayTask(name=dt3, start=1563636427741, time=4000)**  **DelayTask(name=dt6, start=1563636427741, time=5000)**  **DelayTask(name=dt4, start=1563636427741, time=6000)** |

### 4.2Volatile与内存分析

·可见性

·防止指令重排序



·热部署、动态更新

·给自己独立应用推荐使用Volatile，给别人公共服务公开调整推荐Zookeeper

### 4.3Atomic系列类与UnSafe

·Atomic系列类提供了原子性操作，保障多线程下的安全；

·UnSafe类的四大作用：

·内存操作（AllocateMemory、FreeMemory、RellocateMemoy）

·字段的定位于修改

·挂起与恢复（LockSupport：Pack、UnPack）

·CAS操作（乐观锁）

|  |
| --- |
| **boolean** flag = atomicInteger.compareAndSet(0,1); System.***out***.println(flag); System.***out***.println(atomicInteger.get()); |
| **public final boolean** compareAndSet(**int** expect, **int** update) {  **return *unsafe***.compareAndSwapInt(**this**, ***valueOffset***, expect, update); } |

### 4.4J.U.C工具类

·CountDownLatch & CyclicBarrier

·Future模式与Caller接口

·Exchanger线程数据交换器（对账）

·ForkJoin并行计算框架（JDK1.7）

·Semaphone信号量

### 4.5AQS各种锁

·ReentrantLock重入锁

·ReentrantReadWriteLock读写锁

·Condition条件判断

·LockSupport基于线程的锁

|  |
| --- |
| **public class** Review {  **public static void** main(String[] args) **throws** InterruptedException {  */\*ConcurrentHashMap<String,String> map = new ConcurrentHashMap<>();  CopyOnWriteArrayList list = new CopyOnWriteArrayList();  SynchronousQueue synchronousQueue = new SynchronousQueue();  AtomicInteger atomicInteger = new AtomicInteger(0);  boolean flag = atomicInteger.compareAndSet(0,1);  System.out.println(flag);  System.out.println(atomicInteger.get());\*/* Object lock = **new** Object();  Thread A = **new** Thread(()->{  **int** sum = 0;  **for** (**int** i = 0; i < 10; i++) {  sum += i;  }  **try** {  Thread.sleep(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  LockSupport.park(); *// 后执行* System.out.println(**"sum: "** + sum);  });  A.start();  Thread.*sleep*(1000);  LockSupport.*unpark*(A); *// 限执行* } } |
| **sum: 45** |

### 4.6线程池最佳使用

·Executors工厂类（没有上限）

|  |
| --- |
| **public static** ExecutorService newCachedThreadPool() {  **return new** ThreadPoolExecutor(0, Integer.***MAX\_VALUE***,  60L, TimeUnit.***SECONDS***,  **new** SynchronousQueue<Runnable>()); } |

|  |
| --- |
| **public static** ExecutorService newFixedThreadPool(**int** nThreads) {  **return new** ThreadPoolExecutor(nThreads, nThreads,  0L, TimeUnit.***MILLISECONDS***,  **new** LinkedBlockingQueue<Runnable>()); } |

|  |
| --- |
| ThreadPoolExecutor executor = **new** ThreadPoolExecutor(  5,  Runtime.*getRuntime*().availableProcessors() \* 2,  60,  TimeUnit.***SECONDS***,  **new** ArrayBlockingQueue<>(200),  **new** ThreadFactory() {  @Override  **public** Thread newThread(Runnable r) {  Thread t = **new** Thread(r);  t.setName(**"order-thread"**);  **if** (t.isDaemon()) {  t.setDaemon(**false**);  }  **if** (Thread.***NORM\_PRIORITY*** == t.getPriority()) {  t.setPriority(Thread.***NORM\_PRIORITY***);  }  **return** t;  }  },  **new** RejectedExecutionHandler() {  @Override  **public void** rejectedExecution(Runnable r, ThreadPoolExecutor executor) {  System.***out***.println(**"拒绝策略"** + r);  }  } );  executor.shutdown(); |

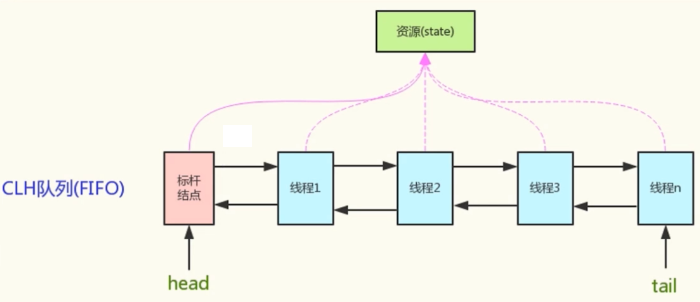
·计算机密集型（cores + 1 或 cores \* 2）与IO密集型（cores /（1-阻塞系数0.9））

·如何正确地使用线程池（关闭使用hook，在bean中使用线程池）

### 4.7AQS架构核心

·AQS维护了一个volatile int state（代表共享资源）和一个FIFO线程等待队列（多线程争用资源被阻塞时进入次队列）

·AQS定义两种资源共享方式：Exclusive、Share



·isHeldExclusively方法：该线程是否正在独占资源

|  |
| --- |
| **protected final boolean** isHeldExclusively() {  *// While we must in general read state before owner,  // we don't need to do so to check if current thread is owner* **return** getExclusiveOwnerThread() == Thread.*currentThread*(); } |
| **final** Thread getOwner() {  **return** getState() == 0 ? **null** : getExclusiveOwnerThread(); } |
| **final boolean** isLocked() {  **return** getState() != 0; } |

·tryAcquire / tryRelease：独占的方式尝试获取和释放资源state

·tryAcquireShared / tryReleaseShared：共享方式尝试获取和释放资源

成功返回true，失败返回false

负数：存在未释放资源的线程

0：获取到，但无可用资源

正数：获取到，有可用资源

### 4.8AQS-ReentrantLock底层源码分析

（1）重入锁ReentrantLock

·以ReentrantLock重入锁为例，state初始化为0，表示未锁定状态

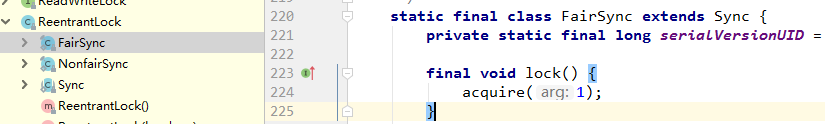
·A线程lock()时，会调用tryAcquire()独占该锁，并state+1

·此后，其他线程tryAcquire()时失败，直到A线程unlock()到state=0（即所有重入的计数都释放锁）为止，其他线程才有机会获得该锁

·当然，A线程释放之前，A线程自己是可以重复获取此锁（state会累加），这就是可重入的概念

·最后，要保证有多少次的重入state的计数，就需要有释放rtryRelease与之对应，是其能够将至0,；

（2）ReentrantLock源码



|  |
| --- |
| **public final void** acquire(**int** arg) {  **if** (!tryAcquire(arg) &&  acquireQueued(addWaiter(Node.***EXCLUSIVE***), arg))  *selfInterrupt*(); } |
|  |
| **protected final boolean** tryAcquire(**int** acquires) {  **final** Thread current = Thread.*currentThread*();  **int** c = getState();  **if** (c == 0) {  **if** (!hasQueuedPredecessors() &&  compareAndSetState(0, acquires)) {  setExclusiveOwnerThread(current);  **return true**;  }  }  **else if** (current == getExclusiveOwnerThread()) {  **int** nextc = c + acquires;  **if** (nextc < 0)  **throw new** Error(**"Maximum lock count exceeded"**);  setState(nextc);  **return true**;  }  **return false**; } |
|  |
| **public final boolean** hasQueuedPredecessors() {  *// The correctness of this depends on head being initialized  // before tail and on head.next being accurate if the current  // thread is first in queue.* Node t = **tail**; *// Read fields in reverse initialization order* Node h = **head**;  Node s;  **return** h != t &&  ((s = h.**next**) == **null** || s.**thread** != Thread.*currentThread*()); } |
| **protected final boolean** compareAndSetState(**int** expect, **int** update) {  *// See below for intrinsics setup to support this* **return *unsafe***.compareAndSwapInt(**this**, ***stateOffset***, expect, update); } |
| **protected final void** setExclusiveOwnerThread(Thread thread) {  **exclusiveOwnerThread** = thread; } |

（3）非公平锁

|  |
| --- |
| **final void** lock() {  // 不管线程阻塞等待队列是否有人排队，先抢抢试试  **if** (compareAndSetState(0, 1))  setExclusiveOwnerThread(Thread.*currentThread*());  **else**  **// 没有取到锁入队** acquire(1); } |

（4）tryRelease释放锁

|  |
| --- |
| **protected final boolean** tryRelease(**int** releases) {  **int** c = getState() - releases;  **if** (Thread.*currentThread*() != getExclusiveOwnerThread())  **throw new** IllegalMonitorStateException();  **boolean** free = **false**;  **if** (c == 0) {  free = **true**;  setExclusiveOwnerThread(**null**);  }  setState(c);  **return** free; } |

（5）Node等待队列

|  |
| --- |
| **public final void** acquire(**int** arg) {  **if** (!tryAcquire(arg) &&  acquireQueued(addWaiter(Node.***EXCLUSIVE***), arg))  *selfInterrupt*(); } |
| **private** Node addWaiter(Node mode) {  Node node = **new** Node(Thread.*currentThread*(), mode);  *// Try the fast path of enq; backup to full enq on failure* Node pred = **tail**;  **if** (pred != **null**) {  node.**prev** = pred;  **if** (compareAndSetTail(pred, node)) {  pred.**next** = node;  **return** node;  }  }  enq(node);  **return** node; } |
| **final boolean** acquireQueued(**final** Node node, **int** arg) {  **boolean** failed = **true**;  **try** {  **boolean** interrupted = **false**;  // 自旋，当前线程的前一个节点是头结点head，尝试获取锁，看是否有新入队的元素（非公平的情况），没有则占用锁，并将头节点移除队列  **for** (;;) {  **final** Node p = node.predecessor();  **if** (p == **head** && tryAcquire(arg)) {  setHead(node);  p.**next** = **null**; *// help GC* failed = **false**;  **return** interrupted;  }  **if** (*shouldParkAfterFailedAcquire*(p, node) &&  parkAndCheckInterrupt())  interrupted = **true**;  }  } **finally** {  **if** (failed)  cancelAcquire(node);  } } |

### 4.9AQS-CountDownLatch-底层源码

·以CountDownLatch为例，任务分为N个子线程去执行，state也初始化为N

·这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS减去1

·等到所有子线程都执行完后（即state==0），会unpack()调用线程，然后主线程会调从await()方法返回，继续后续动作。

## 第5节Disruptor深度提升-源码分析

·Disruptor为何底层性能如此高

·数据结构-内存预加载机制

·内核-使用单线程

·系统内存优化-内存屏障

·算法优化-序号栅栏机制

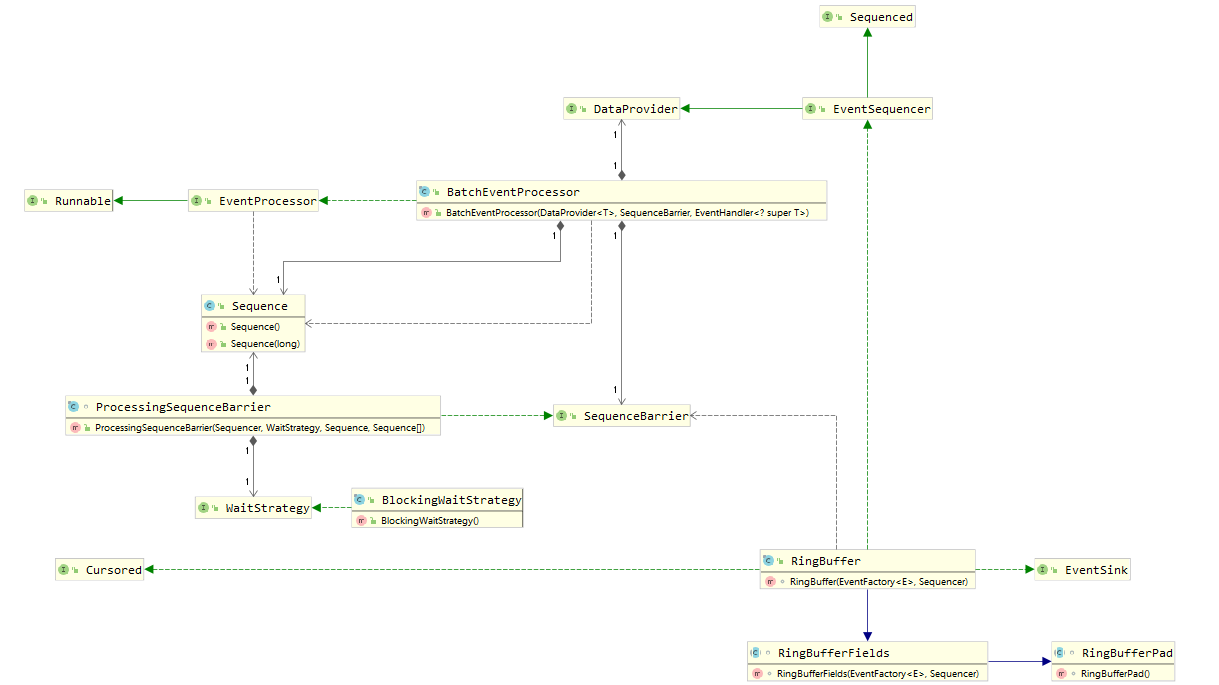
·获取下一个可用序号深度分析

·WaitStrategy等待策略深度剖析

·EventProcessor核心机制深度剖析

·EventHandler深度分析

### 5.1Disruptor核心架构UML图



### 5.2Disruptor底层性能高的原因

·数据结构层面：使用环形结构、数组、内存预加载

·使用单线程当时、内存屏障

·消除共享（填充缓存行）

·序号栅栏和序号配合使用消除锁和CAS

### 5.3数据结构设计原理与底层源码深度分析

（1）下载Disruptor源码

$ git clone <https://github.com/LMAX-Exchange/disruptor.git>

Gradle转Maven

|  |
| --- |
| G:\disruptor\_coding\disruptor-src\disruptor>gradle install  BUILD SUCCESSFUL in 11s  7 actionable tasks: 7 executed |

修改disruptor\build\poms下的pom-default.xml为pom.xml放置disruptor根目录。

修改插件编译版本和项目名称

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**>  <**modelVersion**>4.0.0</**modelVersion**>  <**groupId**>com.lmax.src</**groupId**>  <**artifactId**>disruptor</**artifactId**>  <**version**>3.4.3</**version**>  <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**configuration**>  <**source**>7</**source**>  <**target**>7</**target**>  </**configuration**>  </**plugin**>  </**plugins**>  </**build**>  <**name**>Disruptor Framework</**name**>  <**description**>Disruptor - Concurrent Programming Framework</**description**>  <**url**>http://lmax-exchange.github.com/disruptor</**url**>  <**properties**>  <**java.version**>1.8</**java.version**>  </**properties**>  <**licenses**>  <**license**>  <**name**>The Apache Software License, Version 2.0</**name**>  <**url**>http://www.apache.org/licenses/LICENSE-2.0.txt</**url**>  <**distribution**>repo</**distribution**>  </**license**>  </**licenses**>  <**developers**>  <**developer**>  <**id**>team</**id**>  <**name**>LMAX Disruptor Development Team</**name**>  <**email**>lmax-disruptor@googlegroups.com</**email**>  </**developer**>  </**developers**>  <**scm**>  <**connection**>scm:git@github.com:LMAX-Exchange/disruptor.git</**connection**>  <**url**>scm:git@github.com:LMAX-Exchange/disruptor.git</**url**>  </**scm**>  <**dependencies**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  <**scope**>test</**scope**>  </**dependency**>  </**dependencies**> </**project**> |

使用IDEA导入Maven项目，指向disruptor-src目录。

使用Maven编译后，用disruptor-quickstart的maven工程引用源码：

|  |
| --- |
| <**dependency**>  <**groupId**>com.lmax.src</**groupId**>  <**artifactId**>disruptor</**artifactId**>  <**version**>3.4.3</**version**> </**dependency**> |

重新载入Maven的依赖

（2）RingBuffer

|  |
| --- |
| RingBufferFields(  EventFactory<E> eventFactory,  Sequencer sequencer) {  **this**.**sequencer** = sequencer;  **this**.**bufferSize** = sequencer.getBufferSize();   **if** (**bufferSize** < 1)  {  **throw new** IllegalArgumentException(**"bufferSize must not be less than 1"**);  }  **if** (Integer.*bitCount*(**bufferSize**) != 1)  {  **throw new** IllegalArgumentException(**"bufferSize must be a power of 2"**);  }   **this**.**indexMask** = **bufferSize** - 1;  **this**.**entries** = **new** Object[sequencer.getBufferSize() + 2 \* ***BUFFER\_PAD***];  fill(eventFactory); } |
| */\*\*  \* 内存预加载机制，默认填充eventFactory创建的实例  \** ***@param eventFactory*** *\*/* **private void** fill(EventFactory<E> eventFactory) {  **for** (**int** i = 0; i < **bufferSize**; i++)  {  **entries**[***BUFFER\_PAD*** + i] = eventFactory.newInstance();  } } |

1. Disruptor

|  |
| --- |
| **public static** <E> RingBuffer<E> create(  ProducerType producerType,  EventFactory<E> factory,  **int** bufferSize,  WaitStrategy waitStrategy) {  **switch** (producerType)  {  **case *SINGLE***:  **return** *createSingleProducer*(factory, bufferSize, waitStrategy);  **case *MULTI***:  **return** *createMultiProducer*(factory, bufferSize, waitStrategy);  **default**:  **throw new** IllegalStateException(producerType.toString());  } } |

### 5.4内存-使用单线程写

·Disruptor的RingBuffer，之所以可以做到完全无锁，也是因为“单线程写”，这是所有“前提的前提”；

·离开这个前提条件，没有任何技术可以做到完全无锁

·Redis、Netty（Reactor）等高性能技术框架的设计都是这个思想

·多个需求增加Disruptor池

### 5.5系统内存优化-内存屏障

·要正确的实现无锁，还需要另外一个关键技术：内存屏障

·对应到Java语言，就是volatile变量与happens-before语义；

·内存屏障：Linux的smp\_wmb() / smp\_rmb()



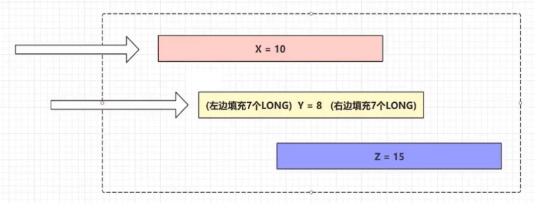
### 5.6系统缓存优化-消除伪共享-填充缓存行

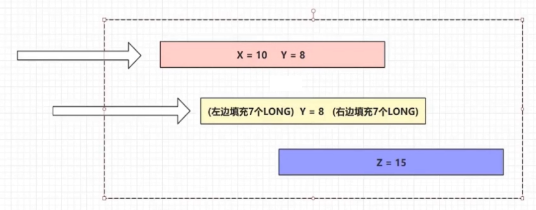
·缓存系统中以缓存行（cache line）为单位存储；

·缓存行是2的整数幂个连续字节，一般为32-256个字节；

·最常见的缓存行大小是64个字节；

·当多线程修改相互独立的变量时，如果这些变量共享一个缓存行，就会无意中影响彼此的性能，这就是伪共享





解决原理

为了避免由于false sharing 导致CacheLine从L1,L2,L3到主存之间重复载入，我们可以使用数据填充的方式来避免，即单个数据填充满一个CacheLine。这本质是一种空间换时间的做法。

Java8中已经提供了官方的解决方案，Java8中新增了一个注解：@sun.misc.Contended。加上这个注解的类会自动补齐缓存行，需要注意的是此注解默认是无效的，需要在jvm启动时设置-XX:-RestrictContended才会生效。

|  |
| --- |
| @sun.misc.Contended  public final static class VolatileLong {  public volatile long value = 0L;  *//public long p1, p2, p3, p4, p5, p6;*  } |

### 5.7算法优化-序号栅栏机制

·我们在生产者投递Event的时候，总是会使用：

Long sequence = ringBuffer.next();

·Disruptor3.0中，序号栅栏SequenceBarrier和序号Sequence搭配使用

·协调和管理消费者与生产者的工作节奏，避免了锁和CAS的使用

序号栅栏机制：

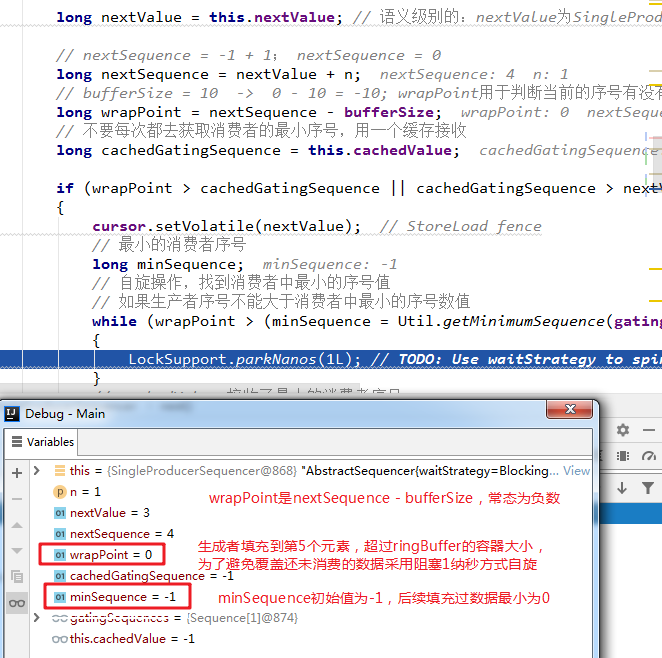
·消费者序号数值必须小于生产者序号数值

·消费者序号必须小于其前置（依赖关系）消费者的序号数值

·生产者序号数值不能大于消费者中最小的序号数值，以避免生产者速度过快，将还未来得及消费的消息覆盖

|  |
| --- |
| **int** ringBufferSize = 4;  ...  **for** (**int** i = 0; i < 5; i++) {  byteBuffer.putLong(0,i);  producer.setData(byteBuffer); } |
| @Override **public void** onEvent(OrderEvent orderEvent, **long** l, **boolean** b) **throws** Exception {  Thread.*sleep*(Integer.***MAX\_VALUE***);  // 阻塞消费  ***log***.info(**"Customer event value:{}"**, orderEvent.getValue()); } |

|  |
| --- |
| @Override **public long** next(**int** n) *// 1* {  **if** (n < 1 || n > **bufferSize**) *// 初始值sequence = -1* {  **throw new** IllegalArgumentException(**"n must be > 0 and < bufferSize"**);  }   **long** nextValue = **this**.**nextValue**; *// 语义级别的：nextValue为SingleProducerSequencer的变量   // nextSequence = -1 + 1； nextSequence = 0* **long** nextSequence = nextValue + n;  *// bufferSize = 10 -> 0 - 10 = -10; wrapPoint用于判断当前的序号有没有绕过整个ringBuffer容器* **long** wrapPoint = nextSequence - **bufferSize**;  *// 不要每次都去获取消费者的最小序号，用一个缓存接收* **long** cachedGatingSequence = **this**.**cachedValue**;   **if** (wrapPoint > cachedGatingSequence || cachedGatingSequence > nextValue)  {  **cursor**.setVolatile(nextValue); *// StoreLoad fence  // 最小的消费者序号* **long** minSequence;  *// 自旋操作，找到消费者中最小的序号值  // 如果生产者序号不能大于消费者中最小的序号数值* **while** (wrapPoint > (minSequence = Util.*getMinimumSequence*(**gatingSequences**, nextValue)))  {  LockSupport.*parkNanos*(1L); *//* ***TODO: Use waitStrategy to spin?*** }  *// cachedValue 接收了最小的消费者序号* **this**.**cachedValue** = minSequence;  }   **this**.**nextValue** = nextSequence;   **return** nextSequence; } |



### 5.8WaitStrategy等待策略底层源码

·Disruptor之所以是高性能，其实也有一部分原因取决于他的等待策略的实现：

|  |
| --- |
| **long** sequence = **ringBuffer**.next(); **try**{  Order order = **ringBuffer**.get(sequence);  order.setId(uuid); } **finally** {  **ringBuffer**.publish(sequence); } |
|  |
| **public void** publish(**long** sequence) {  **cursor**.set(sequence);  // 唤醒消费者等待处理的方法  **waitStrategy**.signalAllWhenBlocking(); } |
|  |
|  |
| *// 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()); |
|  |
|  |

### 5.9EventProcessor核心架构设计与底层源码

|  |
| --- |
|  |
|  |
|  |
|  |

消费者BatchEventProcessor的等待

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

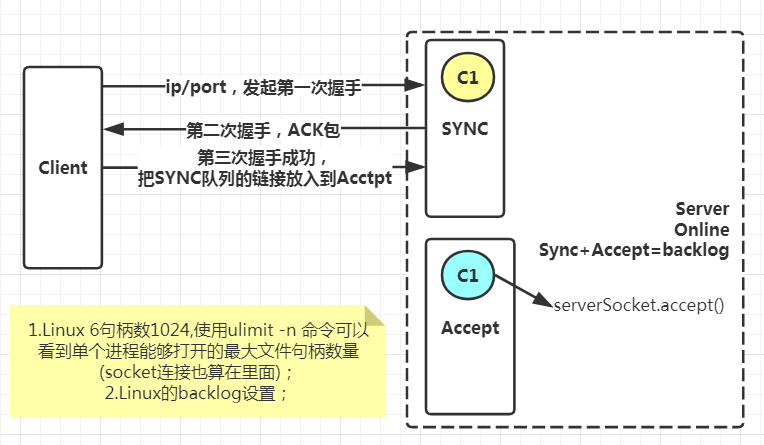
## 第6节Netty整合并发编程框架Disruptor实战百万长链接服务构建

·Disruptor与Netty实现百万长连接接入

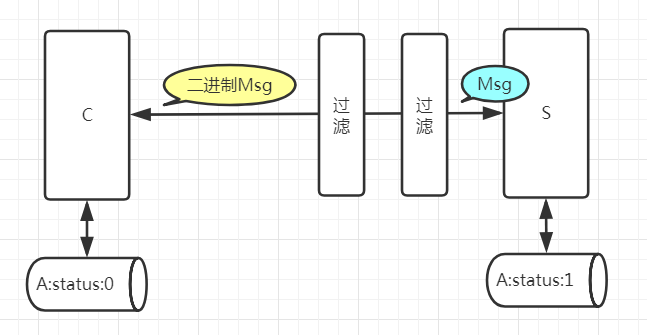
·分布式统一ID生成策略抗压

Zookeeper并发写超1000性能下降

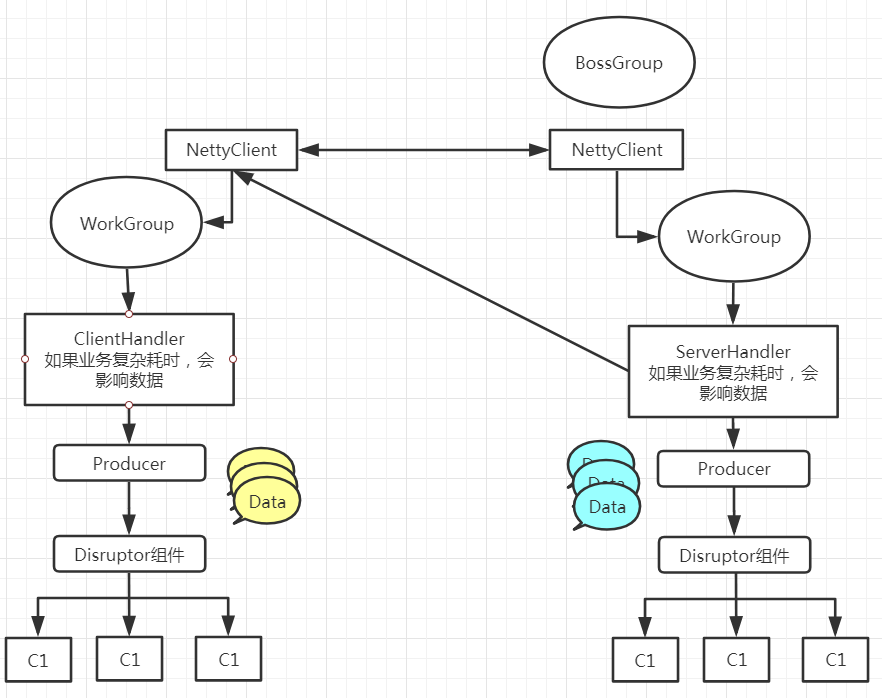
### 6.1Disruptor与Netty整合--服务端



### 6.2Disruptor与Netty整合--客户端



### 6.3Disruptor核心池化封装实现



## 第7节分布式统一ID生成策略

·对于ID的生成，在我们日常开发里应该是一个基本的问题

·最简单的就是利用java.util.UUID工具类进行生成，ID没有排序策略，这种方式的问题就是比如要查询一批数据，进行入库时间做排序的时候，只能在自己表里设置一个create\_time，给这个字段添加索引，然后进行排序。

·（雪花算法/数据库sequence序列、自增ID等）

### 7.1顺序ID生成方式

KeyUtil里生成的ID是有时间先后顺序的，可以使用ID天然进行排序，这种做法比较好的就是没必要浪费一个索引字段了。从数据库的角度来讲，一般能降低减少索引，因为索引虽然可以提升查询性能，但也要占用空间，一张表最好不要超过3个，随意在做索引优化的时候，往往也是要根据业务进行考量

### 7.2业务ID生成方式：

最实用带有业务含义的ID生成策略，这种方式也在传统应用系统、特定的场景下非常的好用。比如一张商品货架表，这张表的数据维度是这样的，比如按照城市和区域来划分。

比如北京按照100000为基本维度数据，100010为北京的一个区域，100020为另一个区域，以此类推，200000可能是另一个城市，200010则为另一个城市的区域。生成货架信息ID的时候，可以按照前6位为城市和区域。

### 7.3高并发下的统一ID生成策略服务

·ID生成在高并发下的重复生成问题

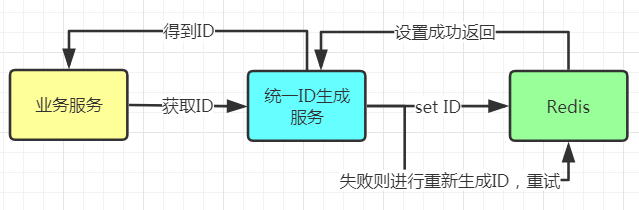
·如何承载高并发ID生成的性能瓶颈问题

·使用Zookeeper的分布式锁实现：

1. QPS上千有性能问题；
2. 选举过半数才能提供服务

·使用Redis缓存，利用Redis分布式锁

1. 调用链开销；
2. 可能失败；
3. 存在延时；



### 7.4主流的分布式ID生成器的策略

·实现一：提前加载，也就是预加载的机制；

1. 并发的获取，采用Disruptor框架提升性能
2. 按业务规则划分，如何配置（Zookeeper生成配置规则），根据规则生成对应的ID范围；

·实现二：单点生成方式；

（1）固定的一个机器节点来生成一个唯一的ID，好处是能够做到全家唯一；

（2）需要响应的业务规则拼接：机器码+时间戳+自增序列

（3）NTP问题

### 7.5高并发下分布式ID生成策略NTP问题



