# 什么是Disruptor？

功能上来看，Disruptor 是实现了“队列”的功能，而且是一个有界队列。那么它的应用场景自然就是“生产者-消费者”模型的应用场合了。可以拿JDK BlockingQueue 做一个简单对比，以便更好地认识 Disruptor 是什么。我们知道 BlockingQueue 是一个 FIFO 队列，生产者(Producer)往队列里发布(publish)一项事件(或称之为“消息”也可以)时，消费者(Consumer)能获得通知；如果没有事件时，消费者被堵塞，直到生产者发布了新的事件。

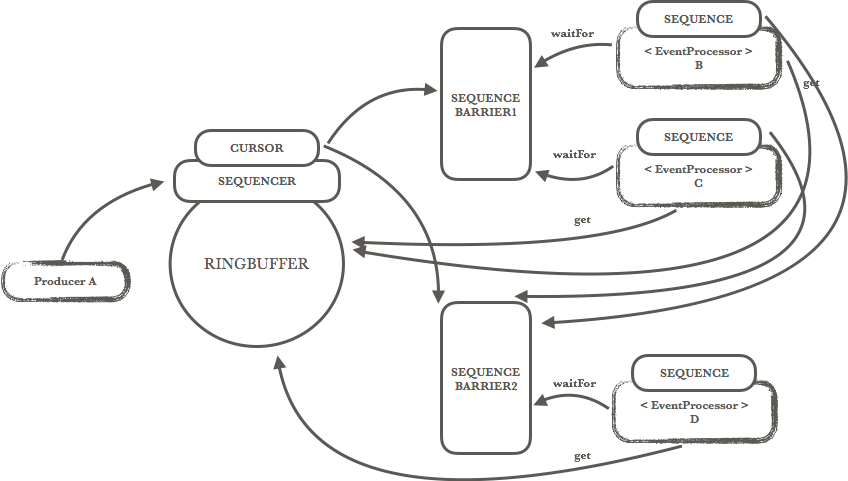
这些都是 Disruptor 能做到的，与之不同的是，Disruptor 能做更多：

* 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，也可以相互依赖形成处理的先后次序(形成一个依赖图)；
* 预分配用于存储事件内容的内存空间；
* 针对极高的性能目标而实现的极度优化和无锁的设计；

# Disruptor的核心概念

1. RingBuffer是Disruptor最核心的组件，负责生产者和消费组交互事件的存储，事件发布者发布事件放到RingBuffer 上面，事件处理者从RingBuffer上面取事件进行消费；
2. RingBuffer可以认为是一个环形队列，内部存储结构为数组，为了避免伪共享，数组前后做了缓存行的填充，在CPU层面提升了性能，数据长度必须为2的幂，可以高效的取数据；
3. RingBuffer创建时对事件做了预填充处理，事件一直可以复用，不会被GC，避免了GC带来的性能影响；

# Disruptor并发模型

并发领域的一个典型场景是生产者消费者模型，常规方式是使用queue作为生产者线程与消费者线程之间共享数据的方法，对于queue的读写避免不了读写锁的竞争。Disruptor使用环形缓冲区RingBuffer作为共享数据的媒介。生产者通过Sequencer控制RingBuffer，以及唤醒等待事件的消费者，消费者通过SequenceBarrier监听RingBuffer的可消费事件。考虑一个场景，一个生产者A与三个消费者B、C、D,同时D的事件处理需要B与C先完成。则该模型结构如下：  


在这个结构下，每个消费者拥有各自独立的事件序号Sequence，消费者之间不存在共享竞态。SequenceBarrier1监听RingBuffer的序号cursor，消费者B与C通过SequenceBarrier1等待可消费事件。SequenceBarrier2除了监听cursor，同时也监听B与C的序号Sequence，从而将最小的序号返回给消费者D，由此实现了D依赖B与C的逻辑。  
RingBuffer是Disruptor高性能的一个亮点。RingBuffer就是一个大数组，事件以循环覆盖的方式写入。与[常规RingBuffer](https://en.wikipedia.org/wiki/Circular_buffer)拥有2个首尾指针的方式不同，Disruptor的RingBuffer只有一个指针(或称序号)，指向数组下一个可写入的位置，该序号在Disruptor源码中就是Sequencer中的cursor，由生产者通过Sequencer控制RingBuffer的写入。为了避免未消费事件的写入覆盖，Sequencer需要监听所有消费者的消息处理进度，也就是gatingSequences。RingBuffer通过这种方式实现了事件缓存的无锁设计。

# disruptor-3.3.2源码解析

### 3.1 disruptor-3.3.2源码解析(1)-序列

* **Disruptor中的序列-Sequence：**

       disruptor中较为重要的一个类是Sequence。我们设想下，在disruptor运行过程中，事件发布者(生产者)和事件处理者(消费者)在ringbuffer上相互追逐，由什么来标记它们的相对位置呢？它们根据什么从ringbuffer上发布或者处理事件呢？就是这个Sequence-序列。

       我们看一下这个类的源代码，先看结构：

**Java代码  [藏代码](javascript:void())**

1. **class** LhsPadding{
2. **protected** **long** p1, p2, p3, p4, p5, p6, p7;
3. }
4. **class** Value **extends** LhsPadding{
5. **protected** **volatile** **long** value;
6. }
7. **class** RhsPadding **extends** Value{
8. **protected** **long** p9, p10, p11, p12, p13, p14, p15;
9. }
11. **public** **class** Sequence **extends** RhsPadding{
12. **static** **final** **long** INITIAL\_VALUE = -1L;
13. **private** **static** **final** Unsafe UNSAFE;
14. **private** **static** **final** **long** VALUE\_OFFSET;
15. **static**{
16. UNSAFE = Util.getUnsafe();
17. **try**{
18. VALUE\_OFFSET = UNSAFE.objectFieldOffset(Value.**class**.getDeclaredField("value"));
19. }**catch** (**final** Exception e){
20. **throw** **new** RuntimeException(e);
21. }
22. }
23. /\*\*
24. \* 默认初始值为-1
25. \*/
26. **public** Sequence(){
27. **this**(INITIAL\_VALUE);
28. }
30. **public** Sequence(**final** **long** initialValue){
31. UNSAFE.putOrderedLong(**this**, VALUE\_OFFSET, initialValue);
32. }

       我们可以注意到两点：

              1.通过Sequence的一系列的继承关系可以看到，它真正的用来计数的域是value，在value的前后各有7个long型的填充值，这些值在这里的作用是做cpu cache line填充，防止发生伪共享。

              2.value域本身由volatile修饰，而且又看到了Unsafe类，大概猜到是要做原子操作了。

       继续看一下Sequence中的方法：

**Java代码  [藏代码](javascript:void())**

1. **public** **long** get(){
2. **return** value;
3. }
4. /\*\*
5. \* ordered write
6. \* 在当前写操作和任意之前的读操作之间加入Store/Store屏障
7. \*/
8. **public** **void** set(**final** **long** value){
9. UNSAFE.putOrderedLong(**this**, VALUE\_OFFSET, value);
10. }
11. /\*\*
12. \* volatile write
13. \* 在当前写操作和任意之前的读操作之间加入Store/Store屏障
14. \* 在当前写操作和任意之后的读操作之间加入Store/Load屏障
15. \*/
16. **public** **void** setVolatile(**final** **long** value){
17. UNSAFE.putLongVolatile(**this**, VALUE\_OFFSET, value);
18. }
20. **public** **boolean** compareAndSet(**final** **long** expectedValue, **final** **long** newValue){
21. **return** UNSAFE.compareAndSwapLong(**this**, VALUE\_OFFSET, expectedValue, newValue);
22. }
24. **public** **long** incrementAndGet(){
25. **return** addAndGet(1L);
26. }
28. **public** **long** addAndGet(**final** **long** increment){
29. **long** currentValue;
30. **long** newValue;
31. **do**{
32. currentValue = get();
33. newValue = currentValue + increment;
34. }**while** (!compareAndSet(currentValue, newValue));
35. **return** newValue;
36. }

  可见Sequence是一个"原子"的序列。

   总结一下：Sequence是一个做了缓存行填充优化的原子序列。

       再看个FixedSequenceGroup类：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** FixedSequenceGroup **extends** Sequence{
3. **private** **final** Sequence[] sequences;
5. **public** FixedSequenceGroup(Sequence[] sequences){
6. **this**.sequences = Arrays.copyOf(sequences, sequences.length);
7. }
9. @Override
10. **public** **long** get(){
11. **return** Util.getMinimumSequence(sequences);
12. }
14. @Override
15. **public** **void** set(**long** value){
16. **throw** **new** UnsupportedOperationException();
17. }
19. ...
21. }

        FixedSequenceGroup相当于包含了若干序列的一个包装类，尽管本身继承了Sequence，但只是重写了get方法，获取内部序列组中最小的序列值，但其他的"写"方法都不支持。

* **上面看了序列的内容，框架中也针对序列的使用，提供了专门的功能接口Sequencer：**

       Sequencer接口扩展了Cursored和Sequenced。先看下这两个货：

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** Cursored{
2. **long** getCursor();
3. }

       Cursored接口只提供了一个获取当前序列值(游标)的方法。

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** Sequenced{
2. /\*\*
3. \* 数据结构中事件槽的个数(就是RingBuffer的容量)
4. \*/
5. **int** getBufferSize();
6. /\*\*
7. \* 判断是否还有给定的可用容量。
8. \*/
9. **boolean** hasAvailableCapacity(**final** **int** requiredCapacity);
10. /\*\*
11. \* 获取剩余容量。
12. \*/
13. **long** remainingCapacity();
14. /\*\*
15. \* 申请下一个序列值，用来发布事件。
16. \*/
17. **long** next();
18. /\*\*
19. \* 申请下N个序列值，用来发布事件。
20. \*/
21. **long** next(**int** n);
22. /\*\*
23. \* 尝试申请下一个序列值用来发布事件，这个是无阻塞的方法。
24. \*/
25. **long** tryNext() **throws** InsufficientCapacityException;
26. /\*\*
27. \* 尝试申请下N个序列值用来发布事件，这个是无阻塞的方法。
28. \*/
29. **long** tryNext(**int** n) **throws** InsufficientCapacityException;
30. /\*\*
31. \* 在给定的序列值上发布事件，当填充好事件后会调用这个方法。
32. \*/
33. **void** publish(**long** sequence);
34. /\*\*
35. \* 在给定的序列返回上发布事件，当填充好事件后会调用这个方法。
36. \*/
37. **void** publish(**long** lo, **long** hi);
38. }

       最后看下Sequencer接口：

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** Sequencer **extends** Cursored, Sequenced{
2. /\*\* 序列初始值 \*/
3. **long** INITIAL\_CURSOR\_VALUE = -1L;
4. /\*\*
5. \* 声明一个序列，这个方法在初始化RingBuffer的时候被调用。
6. \*/
7. **void** claim(**long** sequence);
8. /\*\*
9. \* 判断一个序列是否被发布，并且发布到序列上的事件是可处理的。非阻塞方法。
10. \*/
11. **boolean** isAvailable(**long** sequence);
12. /\*\*
13. \* 添加一些追踪序列到当前实例，添加过程是原子的。
14. \* 这些控制序列一般是其他组件的序列，当前实例可以通过这些
15. \* 序列来查看其他组件的序列使用情况。
16. \*/
17. **void** addGatingSequences(Sequence... gatingSequences);
18. /\*\*
19. \* 移除控制序列。
20. \*/
21. **boolean** removeGatingSequence(Sequence sequence);
22. /\*\*
23. \* 基于给定的追踪序列创建一个序列栅栏，这个栅栏是提供给事件处理者
24. \* 在判断Ringbuffer上某个事件是否能处理时使用的。
25. \*/
26. SequenceBarrier newBarrier(Sequence... sequencesToTrack);
27. /\*\*
28. \* 获取控制序列里面当前最小的序列值。
29. \*/
30. **long** getMinimumSequence();
31. /\*\*
32. \* 获取RingBuffer上安全使用的最大的序列值。
33. \* 具体实现里面，这个调用可能需要序列上从nextSequence到availableSequence之间的值。
34. \* 如果没有比nextSequence大的可用序列，会返回nextSequence - 1。
35. \* 为了保证正确，事件处理者应该传递一个比最后的序列值大1个单位的序列来处理。
36. \*/
37. **long** getHighestPublishedSequence(**long** nextSequence, **long** availableSequence);
39. /\*
40. \* 通过给定的数据提供者和控制序列来创建一个EventPoller
41. \*/
42. <T> EventPoller<T> newPoller(DataProvider<T> provider, Sequence...gatingSequences);
43. }

       这里要注意一下：

              Sequencer接口的很多功能是提供给事件发布者用的。

              通过Sequencer可以得到一个SequenceBarrier，这货是提供给事件处理者用的。

**框架中针对Sequencer接口提供了2种实现：SingleProducerSequencer和MultiProducerSequencer。**

       看这两个类之前，先看下它们的基类AbstractSequencer：

**Java代码  [藏代码](javascript:void())**

1. **public** **abstract** **class** AbstractSequencer **implements** Sequencer{
3. **private** **static** **final** AtomicReferenceFieldUpdater<AbstractSequencer, Sequence[]> SEQUENCE\_UPDATER =
4. AtomicReferenceFieldUpdater.newUpdater(AbstractSequencer.**class**, Sequence[].**class**, "gatingSequences");
5. **protected** **final** **int** bufferSize;
6. **protected** **final** WaitStrategy waitStrategy;
7. **protected** **final** Sequence cursor = **new** Sequence(Sequencer.INITIAL\_CURSOR\_VALUE);
8. **protected** **volatile** Sequence[] gatingSequences = **new** Sequence[0];
10. **public** AbstractSequencer(**int** bufferSize, WaitStrategy waitStrategy){
11. **if** (bufferSize < 1){
12. **throw** **new** IllegalArgumentException("bufferSize must not be less than 1");
13. }
14. **if** (Integer.bitCount(bufferSize) != 1){
15. **throw** **new** IllegalArgumentException("bufferSize must be a power of 2");
16. }
17. **this**.bufferSize = bufferSize;
18. **this**.waitStrategy = waitStrategy;
19. }
21. @Override
22. **public** **final** **long** getCursor(){
23. **return** cursor.get();
24. }
26. @Override
27. **public** **final** **int** getBufferSize(){
28. **return** bufferSize;
29. }
31. @Override
32. **public** **final** **void** addGatingSequences(Sequence... gatingSequences){
33. SequenceGroups.addSequences(**this**, SEQUENCE\_UPDATER, **this**, gatingSequences);
34. }
36. @Override
37. **public** **boolean** removeGatingSequence(Sequence sequence){
38. **return** SequenceGroups.removeSequence(**this**, SEQUENCE\_UPDATER, sequence);
39. }
41. @Override
42. **public** **long** getMinimumSequence(){
43. **return** Util.getMinimumSequence(gatingSequences, cursor.get());
44. }
46. @Override
47. **public** SequenceBarrier newBarrier(Sequence... sequencesToTrack){
48. **return** **new** ProcessingSequenceBarrier(**this**, waitStrategy, cursor, sequencesToTrack);
49. }
51. @Override
52. **public** <T> EventPoller<T> newPoller(DataProvider<T> dataProvider, Sequence... gatingSequences){
53. **return** EventPoller.newInstance(dataProvider, **this**, **new** Sequence(), cursor, gatingSequences);
54. }
55. }

       可见，基类基本上的作用就是管理追踪序列和关联当前序列。

       先看下SingleProducerSequencer，还是先看结构：

**Java代码  [藏代码](javascript:void())**

1. bstract **class** SingleProducerSequencerPad **extends** AbstractSequencer{
2. **protected** **long** p1, p2, p3, p4, p5, p6, p7;
3. **public** SingleProducerSequencerPad(**int** bufferSize, WaitStrategy waitStrategy){
4. **super**(bufferSize, waitStrategy);
5. }
6. }
7. **abstract** **class** SingleProducerSequencerFields **extends** SingleProducerSequencerPad{
8. **public** SingleProducerSequencerFields(**int** bufferSize, WaitStrategy waitStrategy){
9. **super**(bufferSize, waitStrategy);
10. }
11. **protected** **long** nextValue = Sequence.INITIAL\_VALUE;
12. **protected** **long** cachedValue = Sequence.INITIAL\_VALUE;
13. }
15. **public** **final** **class** SingleProducerSequencer **extends** SingleProducerSequencerFields{
16. **protected** **long** p1, p2, p3, p4, p5, p6, p7;
18. **public** SingleProducerSequencer(**int** bufferSize, **final** WaitStrategy waitStrategy){
19. **super**(bufferSize, waitStrategy);
20. }

       又是缓存行填充，真正使用的值是nextValue和cachedValue。

       再看下里面的方法实现：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **boolean** hasAvailableCapacity(**final** **int** requiredCapacity){
3. **long** nextValue = **this**.nextValue;
4. **long** wrapPoint = (nextValue + requiredCapacity) - bufferSize;
5. **long** cachedGatingSequence = **this**.cachedValue;
6. **if** (wrapPoint > cachedGatingSequence || cachedGatingSequence > nextValue){
7. **long** minSequence = Util.getMinimumSequence(gatingSequences, nextValue);
8. **this**.cachedValue = minSequence;
9. **if** (wrapPoint > minSequence){
10. **return** **false**;
11. }
12. }
13. **return** **true**;
14. }

  hasAvailableCapacity方法可以这样理解：

 当前序列的nextValue + requiredCapacity是事件发布者要申请的序列值。

 当前序列的cachedValue记录的是之前事件处理者申请的序列值。

 想一下一个环形队列，事件发布者在什么情况下才能申请一个序列呢？

 事件发布者当前的位置在事件处理者前面，并且不能从事件处理者后面追上事件处理者(因为是环形)，

 即事件发布者要申请的序列值大于事件处理者之前的序列值且事件发布者要申请的序列 值减去环的长度要小于事件处理者的序列值

如果满足这个条件，即使不知道当前事件处理者的序列值，也能确保事件发布者可以申请给定的序列。

如果不满足这个条件，就需要查看一下当前事件处理者的最小的序列值(因为可能有多个事件处理者)，如果当前要申请的序列值比当前事件处理者的最小序列值大了一圈(从后面追上了)，那就不能申请了(申请的话会覆盖没被消费的事件)，也就是说没有可用的空间(用来发布事件)了，也就是hasAvailableCapacity方法要表达的意思。

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **long** next(){
3. **return** next(1);
4. }
6. @Override
7. **public** **long** next(**int** n){
8. **if** (n < 1) {
9. **throw** **new** IllegalArgumentException("n must be > 0");
10. }
11. **long** nextValue = **this**.nextValue;
12. **long** nextSequence = nextValue + n;
13. **long** wrapPoint = nextSequence - bufferSize;
14. **long** cachedGatingSequence = **this**.cachedValue;
15. **if** (wrapPoint > cachedGatingSequence || cachedGatingSequence > nextValue){
16. **long** minSequence;
17. **while** (wrapPoint > (minSequence = Util.getMinimumSequence(gatingSequences, nextValue))){
18. LockSupport.parkNanos(1L); // TODO: Use waitStrategy to spin?
19. }
20. **this**.cachedValue = minSequence;
21. }
22. **this**.nextValue = nextSequence;
23. **return** nextSequence;
24. }

   next方法是真正申请序列的方法，里面的逻辑和hasAvailableCapacity一样，只是在不能申请序列的时候会阻塞等待一下，然后重试。

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **long** tryNext() **throws** InsufficientCapacityException{
3. **return** tryNext(1);
4. }
6. @Override
7. **public** **long** tryNext(**int** n) **throws** InsufficientCapacityException{
8. **if** (n < 1){
9. **throw** **new** IllegalArgumentException("n must be > 0");
10. }
11. **if** (!hasAvailableCapacity(n)){
12. **throw** InsufficientCapacityException.INSTANCE;
13. }
14. **long** nextSequence = **this**.nextValue += n;
15. **return** nextSequence;
16. }

   tryNext方法是next方法的非阻塞版本，不能申请就抛异常。

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **long** remainingCapacity(){
3. **long** nextValue = **this**.nextValue;
4. **long** consumed = Util.getMinimumSequence(gatingSequences, nextValue);
5. **long** produced = nextValue;
6. **return** getBufferSize() - (produced - consumed);
7. }

  remainingCapacity方法就是环形队列的容量减去事件发布者与事件处理者的序列差。

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **void** claim(**long** sequence){
3. **this**.nextValue = sequence;
4. }

   claim方法是声明一个序列，在初始化的时候用。

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **void** publish(**long** sequence){
3. cursor.set(sequence);
4. waitStrategy.signalAllWhenBlocking();
5. }
6. @Override
7. **public** **void** publish(**long** lo, **long** hi){
8. publish(hi);
9. }

  发布一个序列，会先设置内部游标值，然后唤醒等待的事件处理者。

  看下剩下的方法：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **boolean** isAvailable(**long** sequence){
3. **return** sequence <= cursor.get();
4. }
5. @Override
6. **public** **long** getHighestPublishedSequence(**long** lowerBound, **long** availableSequence){
7. **return** availableSequence;
8. }

  再看下MultiProducerSequencer，还是先看结构：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** MultiProducerSequencer **extends** AbstractSequencer{
2. **private** **static** **final** Unsafe UNSAFE = Util.getUnsafe();
3. **private** **static** **final** **long** BASE  = UNSAFE.arrayBaseOffset(**int**[].**class**);
4. **private** **static** **final** **long** SCALE = UNSAFE.arrayIndexScale(**int**[].**class**);
5. **private** **final** Sequence gatingSequenceCache = **new** Sequence(Sequencer.INITIAL\_CURSOR\_VALUE);
6. // availableBuffer是用来记录每一个ringbuffer槽的状态。
7. **private** **final** **int**[] availableBuffer;
8. **private** **final** **int** indexMask;
9. **private** **final** **int** indexShift;
11. **public** MultiProducerSequencer(**int** bufferSize, **final** WaitStrategy waitStrategy){
12. **super**(bufferSize, waitStrategy);
13. availableBuffer = **new** **int**[bufferSize];
14. indexMask = bufferSize - 1;
15. indexShift = Util.log2(bufferSize);
16. initialiseAvailableBuffer();
17. }

  MultiProducerSequencer内部多了一个availableBuffer，是一个int型的数组，size大小和RingBuffer的Size一样大，用来追踪Ringbuffer每个槽的状态，构造MultiProducerSequencer的时候会进行初始化，availableBuffer数组中的每个元素会被初始化成-1。

  再看下里面的方法实现：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **boolean** hasAvailableCapacity(**final** **int** requiredCapacity){
3. **return** hasAvailableCapacity(gatingSequences, requiredCapacity, cursor.get());
4. }
5. **private** **boolean** hasAvailableCapacity(Sequence[] gatingSequences, **final** **int** requiredCapacity, **long** cursorValue){
6. **long** wrapPoint = (cursorValue + requiredCapacity) - bufferSize;
7. **long** cachedGatingSequence = gatingSequenceCache.get();
8. **if** (wrapPoint > cachedGatingSequence || cachedGatingSequence > cursorValue){
9. **long** minSequence = Util.getMinimumSequence(gatingSequences, cursorValue);
10. gatingSequenceCache.set(minSequence);
11. **if** (wrapPoint > minSequence){
12. **return** **false**;
13. }
14. }
15. **return** **true**;
16. }

  逻辑和前面SingleProducerSequencer内部一样，区别是这里使用了cursor.get()，里面获取的是一个volatile的value值。

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **long** next(**int** n){
3. **if** (n < 1){
4. **throw** **new** IllegalArgumentException("n must be > 0");
5. }
6. **long** current;
7. **long** next;
8. **do**{
9. current = cursor.get();
10. next = current + n;
11. **long** wrapPoint = next - bufferSize;
12. **long** cachedGatingSequence = gatingSequenceCache.get();
13. **if** (wrapPoint > cachedGatingSequence || cachedGatingSequence > current){
14. **long** gatingSequence = Util.getMinimumSequence(gatingSequences, current);
15. **if** (wrapPoint > gatingSequence){
16. LockSupport.parkNanos(1); // TODO, should we spin based on the wait strategy?
17. **continue**;
18. }
19. gatingSequenceCache.set(gatingSequence);
20. }**else** **if** (cursor.compareAndSet(current, next)){
21. **break**;
22. }
23. }**while** (**true**);
24. **return** next;
25. }

  逻辑还是一样，区别是里面的增加当前序列值是原子操作。

  其他的方法都类似，都能保证多线程下的安全操作，唯一有点不同的是publish方法，看下：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **void** publish(**final** **long** sequence){
3. setAvailable(sequence);
4. waitStrategy.signalAllWhenBlocking();
5. }
6. **private** **void** setAvailable(**final** **long** sequence){
7. setAvailableBufferValue(calculateIndex(sequence), calculateAvailabilityFlag(sequence));
8. }
9. **private** **void** setAvailableBufferValue(**int** index, **int** flag){
10. **long** bufferAddress = (index \* SCALE) + BASE;
11. UNSAFE.putOrderedInt(availableBuffer, bufferAddress, flag);
12. }
13. **private** **int** calculateAvailabilityFlag(**final** **long** sequence){
14. **return** (**int**) (sequence >>> indexShift);
15. }
16. **private** **int** calculateIndex(**final** **long** sequence){
17. **return** ((**int**) sequence) & indexMask;
18. }

 方法中会将当前序列值的可用状态记录到availableBuffer里面，而记录的这个值其实就是sequence除以bufferSize，也就是当前sequence绕buffer的圈数。

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **boolean** isAvailable(**long** sequence){
3. **int** index = calculateIndex(sequence);
4. **int** flag = calculateAvailabilityFlag(sequence);
5. **long** bufferAddress = (index \* SCALE) + BASE;
6. **return** UNSAFE.getIntVolatile(availableBuffer, bufferAddress) == flag;
7. }
8. @Override
9. **public** **long** getHighestPublishedSequence(**long** lowerBound, **long** availableSequence){
10. **for** (**long** sequence = lowerBound; sequence <= availableSequence; sequence++){
11. **if** (!isAvailable(sequence)){
12. **return** sequence - 1;
13. }
14. }
15. **return** availableSequence;
16. }

 isAvailable方法也好理解了，getHighestPublishedSequence方法基于isAvailable实现。

* **大概了解了Sequencer的功能和实现，接下来看一下序列相关的一些类：**

 首先看下SequenceBarrier这个接口：

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** SequenceBarrier{
3. /\*\*
4. \* 等待一个序列变为可用，然后消费这个序列。.
5. \* 这货明显是给事件处理者使用的。
6. \*/
7. **long** waitFor(**long** sequence) **throws** AlertException, InterruptedException, TimeoutException;
8. /\*\*
9. \* 获取当前可以读取的序列值。
10. \*/
11. **long** getCursor();
12. /\*\*
13. \* 当前栅栏是否发过通知。
14. \*/
15. **boolean** isAlerted();
16. /\*\*
17. \* 通知事件处理者状态变化，然后停留在这个状态上，直到状态被清除。
18. \*/
19. **void** alert();
20. /\*\*
21. \* 清楚通知状态。
22. \*/
23. **void** clearAlert();
24. /\*\*
25. \* 检测是否发生了通知，如果已经发生了抛出AlertException异常。
26. \*/
27. **void** checkAlert() **throws** AlertException;
28. }

  来看一下SequenceBarrier的实现ProcessingSequenceBarrier：

**Java代码  [藏代码](javascript:void())**

1. **final** **class** ProcessingSequenceBarrier **implements** SequenceBarrier{
2. //等待策略。
3. **private** **final** WaitStrategy waitStrategy;
4. //这个域可能指向一个序列组。
5. **private** **final** Sequence dependentSequence;
6. **private** **volatile** **boolean** alerted = **false**;
7. **private** **final** Sequence cursorSequence;
8. **private** **final** Sequencer sequencer;
9. **public** ProcessingSequenceBarrier(**final** Sequencer sequencer,
10. **final** WaitStrategy waitStrategy,
11. **final** Sequence cursorSequence,
12. **final** Sequence[] dependentSequences){
13. **this**.sequencer = sequencer;
14. **this**.waitStrategy = waitStrategy;
15. **this**.cursorSequence = cursorSequence;
16. **if** (0 == dependentSequences.length){
17. dependentSequence = cursorSequence;
18. }**else**{
19. dependentSequence = **new** FixedSequenceGroup(dependentSequences);
20. }
21. }
22. @Override
23. **public** **long** waitFor(**final** **long** sequence)
24. **throws** AlertException, InterruptedException, TimeoutException{
25. //先检测报警状态。
26. checkAlert();
27. //然后根据等待策略来等待可用的序列值。
28. **long** availableSequence = waitStrategy.waitFor(sequence, cursorSequence, dependentSequence, **this**);
29. **if** (availableSequence < sequence){
30. **return** availableSequence; //如果可用的序列值小于给定的序列，那么直接返回。
31. }
32. //否则，要返回能安全使用的最大的序列值。
33. **return** sequencer.getHighestPublishedSequence(sequence, availableSequence);
34. }
35. @Override
36. **public** **long** getCursor(){
37. **return** dependentSequence.get();
38. }
39. @Override
40. **public** **boolean** isAlerted(){
41. **return** alerted;
42. }
43. @Override
44. **public** **void** alert(){
45. alerted = **true**; //设置通知标记
46. waitStrategy.signalAllWhenBlocking();//如果有线程以阻塞的方式等待序列，将其唤醒。
47. }
48. @Override
49. **public** **void** clearAlert(){
50. alerted = **false**;
51. }
52. @Override
53. **public** **void** checkAlert() **throws** AlertException{
54. **if** (alerted){
55. **throw** AlertException.INSTANCE;
56. }
57. }
58. }

  再看一个SequenceGroup类：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** SequenceGroup **extends** Sequence{
3. **private** **static** **final** AtomicReferenceFieldUpdater<SequenceGroup, Sequence[]> SEQUENCE\_UPDATER =
4. AtomicReferenceFieldUpdater.newUpdater(SequenceGroup.**class**, Sequence[].**class**, "sequences");
5. **private** **volatile** Sequence[] sequences = **new** Sequence[0];
7. **public** SequenceGroup(){
8. **super**(-1);
9. }
10. /\*\*
11. \* 获取序列组中最小的序列值。
12. \*/
13. @Override
14. **public** **long** get(){
15. **return** Util.getMinimumSequence(sequences);
16. }
17. /\*\*
18. \* 将序列组中所有的序列设置为给定值。
19. \*/
20. @Override
21. **public** **void** set(**final** **long** value){
22. **final** Sequence[] sequences = **this**.sequences;
23. **for** (**int** i = 0, size = sequences.length; i < size; i++){
24. sequences[i].set(value);
25. }
26. }
27. /\*\*
28. \* 添加一个序列到序列组，这个方法只能在初始化的时候调用。
29. \* 运行时添加的话，使用addWhileRunning(Cursored, Sequence)
30. \*/
31. **public** **void** add(**final** Sequence sequence){
32. Sequence[] oldSequences;
33. Sequence[] newSequences;
34. **do**{
35. oldSequences = sequences;
36. **final** **int** oldSize = oldSequences.length;
37. newSequences = **new** Sequence[oldSize + 1];
38. System.arraycopy(oldSequences, 0, newSequences, 0, oldSize);
39. newSequences[oldSize] = sequence;
40. } **while** (!SEQUENCE\_UPDATER.compareAndSet(**this**, oldSequences, newSequences));
41. }
42. /\*\*
43. \* 将序列组中出现的第一个给定的序列移除。
44. \*/
45. **public** **boolean** remove(**final** Sequence sequence){
46. **return** SequenceGroups.removeSequence(**this**, SEQUENCE\_UPDATER, sequence);
47. }
48. /\*\*
49. \* 获取序列组的大小。
50. \*/
51. **public** **int** size(){
52. **return** sequences.length;
53. }
54. /\*\*
55. \* 在线程已经开始往Disruptor上发布事件后，添加一个序列到序列组。
56. \* 调用这个方法后，会将新添加的序列的值设置为游标的值。
57. \*/
58. **public** **void** addWhileRunning(Cursored cursored, Sequence sequence){
59. SequenceGroups.addSequences(**this**, SEQUENCE\_UPDATER, cursored, sequence);
60. }
61. }

  还有一个SequenceGroups类，是针对SequenceGroup的帮助类，里面提供了 addSequences和removeSequence方法，都是原子操作。

* **最后总结：**

**上面看了这么多序列的相关类，其实只需要记住三点：**

**1.真正的序列是Sequence。**

**2.事件发布者通过Sequencer的大部分功能来使用序列。**

**3.事件处理者通过SequenceBarrier来使用序列。**

### 3.2 disruptor-3.3.2源码解析(2)-队列

* **Disruptor中的队列-RingBuffer：**

       RingBuffer是disruptor最重要的核心组件，如果以生产者/消费者模式来看待disruptor框架的话，那RingBuffer就是生产者和消费者的工作队列了。RingBuffer可以理解为是一个环形队列，那内部是怎么实现的呢？看下源码。

       首先，RingBuffer实现了一系列接口，Cursored、EventSequencer和EventSink，Cursored上篇提过了，这里看下后面两个：

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** EventSequencer<T> **extends** DataProvider<T>, Sequenced{
2. }

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** DataProvider<T>{
2. T get(**long** sequence);
3. }

       EventSequencer扩展了Sequenced，提供了一些序列功能；同时扩展了DataProvider，提供了按序列值来获取数据的功能。

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** EventSink<E>{
2. **void** publishEvent(EventTranslator<E> translator);
4. **boolean** tryPublishEvent(EventTranslator<E> translator);
6. <A> **void** publishEvent(EventTranslatorOneArg<E, A> translator, A arg0);
8. <A> **boolean** tryPublishEvent(EventTranslatorOneArg<E, A> translator, A arg0);
10. <A, B> **void** publishEvent(EventTranslatorTwoArg<E, A, B> translator, A arg0, B arg1);
12. <A, B> **boolean** tryPublishEvent(EventTranslatorTwoArg<E, A, B> translator, A arg0, B arg1);
14. <A, B, C> **void** publishEvent(EventTranslatorThreeArg<E, A, B, C> translator, A arg0, B arg1, C arg2);
16. <A, B, C> **boolean** tryPublishEvent(EventTranslatorThreeArg<E, A, B, C> translator, A arg0, B arg1, C arg2);
18. **void** publishEvent(EventTranslatorVararg<E> translator, Object... args);
20. **boolean** tryPublishEvent(EventTranslatorVararg<E> translator, Object... args);
22. **void** publishEvents(EventTranslator<E>[] translators);
24. **void** publishEvents(EventTranslator<E>[] translators, **int** batchStartsAt, **int** batchSize);
26. **boolean** tryPublishEvents(EventTranslator<E>[] translators);
28. **boolean** tryPublishEvents(EventTranslator<E>[] translators, **int** batchStartsAt, **int** batchSize);
30. <A> **void** publishEvents(EventTranslatorOneArg<E, A> translator, A[] arg0);
32. <A> **void** publishEvents(EventTranslatorOneArg<E, A> translator, **int** batchStartsAt, **int** batchSize, A[] arg0);
34. <A> **boolean** tryPublishEvents(EventTranslatorOneArg<E, A> translator, A[] arg0);
36. <A> **boolean** tryPublishEvents(EventTranslatorOneArg<E, A> translator, **int** batchStartsAt, **int** batchSize, A[] arg0);
38. <A, B> **void** publishEvents(EventTranslatorTwoArg<E, A, B> translator, A[] arg0, B[] arg1);
40. <A, B> **void** publishEvents(EventTranslatorTwoArg<E, A, B> translator, **int** batchStartsAt, **int** batchSize, A[] arg0, B[] arg1);
42. <A, B> **boolean** tryPublishEvents(EventTranslatorTwoArg<E, A, B> translator, A[] arg0, B[] arg1);
44. <A, B> **boolean** tryPublishEvents(EventTranslatorTwoArg<E, A, B> translator, **int** batchStartsAt, **int** batchSize, A[] arg0, B[] arg1);
46. <A, B, C> **void** publishEvents(EventTranslatorThreeArg<E, A, B, C> translator, A[] arg0, B[] arg1, C[] arg2);
48. <A, B, C> **void** publishEvents(EventTranslatorThreeArg<E, A, B, C> translator, **int** batchStartsAt, **int** batchSize, A[] arg0, B[] arg1, C[] arg2);
50. <A, B, C> **boolean** tryPublishEvents(EventTranslatorThreeArg<E, A, B, C> translator, A[] arg0, B[] arg1, C[] arg2);
52. <A, B, C> **boolean** tryPublishEvents(EventTranslatorThreeArg<E, A, B, C> translator, **int** batchStartsAt, **int** batchSize, A[] arg0, B[] arg1, C[] arg2);
54. **void** publishEvents(EventTranslatorVararg<E> translator, Object[]... args);
56. **void** publishEvents(EventTranslatorVararg<E> translator, **int** batchStartsAt, **int** batchSize, Object[]... args);
58. **boolean** tryPublishEvents(EventTranslatorVararg<E> translator, Object[]... args);
60. **boolean** tryPublishEvents(EventTranslatorVararg<E> translator, **int** batchStartsAt, **int** batchSize, Object[]... args);
61. }

       可见，EventSink主要是提供发布事件(就是往队列上放数据)的功能，接口上定义了以各种姿势发布事件的方法。

       了解了RingBuffer的接口功能，下面看下RingBuffer的结构：

**Java代码  [藏代码](javascript:void())**

1. **abstract** **class** RingBufferPad{
2. **protected** **long** p1, p2, p3, p4, p5, p6, p7;
3. }
4. **abstract** **class** RingBufferFields<E> **extends** RingBufferPad{
5. **private** **static** **final** **int** BUFFER\_PAD;
6. **private** **static** **final** **long** REF\_ARRAY\_BASE;
7. **private** **static** **final** **int** REF\_ELEMENT\_SHIFT;
8. **private** **static** **final** Unsafe UNSAFE = Util.getUnsafe();
9. **static**{
10. **final** **int** scale = UNSAFE.arrayIndexScale(Object[].**class**);
11. **if** (4 == scale){
12. REF\_ELEMENT\_SHIFT = 2;
13. } **else** **if** (8 == scale){
14. REF\_ELEMENT\_SHIFT = 3;
15. }**else**{
16. **throw** **new** IllegalStateException("Unknown pointer size");
17. }
18. BUFFER\_PAD = 128 / scale;
19. // Including the buffer pad in the array base offset
20. REF\_ARRAY\_BASE = UNSAFE.arrayBaseOffset(Object[].**class**) + (BUFFER\_PAD << REF\_ELEMENT\_SHIFT);
21. }
22. **private** **final** **long** indexMask;
23. **private** **final** Object[] entries;
24. **protected** **final** **int** bufferSize;
25. **protected** **final** Sequencer sequencer;
26. RingBufferFields(EventFactory<E> eventFactory,
27. Sequencer       sequencer){
28. **this**.sequencer  = sequencer;
29. **this**.bufferSize = sequencer.getBufferSize();
30. **if** (bufferSize < 1){
31. **throw** **new** IllegalArgumentException("bufferSize must not be less than 1");
32. }
33. **if** (Integer.bitCount(bufferSize) != 1){
34. **throw** **new** IllegalArgumentException("bufferSize must be a power of 2");
35. }
36. **this**.indexMask = bufferSize - 1;
37. **this**.entries   = **new** Object[sequencer.getBufferSize() + 2 \* BUFFER\_PAD];
38. fill(eventFactory);
39. }
40. **private** **void** fill(EventFactory<E> eventFactory){
41. **for** (**int** i = 0; i < bufferSize; i++){
42. entries[BUFFER\_PAD + i] = eventFactory.newInstance();
43. }
44. }
45. @SuppressWarnings("unchecked")
46. **protected** **final** E elementAt(**long** sequence){
47. **return** (E) UNSAFE.getObject(entries, REF\_ARRAY\_BASE + ((sequence & indexMask) << REF\_ELEMENT\_SHIFT));
48. }
49. }
51. **public** **final** **class** RingBuffer<E> **extends** RingBufferFields<E> **implements** Cursored, EventSequencer<E>, EventSink<E>{
52. **public** **static** **final** **long** INITIAL\_CURSOR\_VALUE = Sequence.INITIAL\_VALUE;
53. **protected** **long** p1, p2, p3, p4, p5, p6, p7;
55. RingBuffer(EventFactory<E> eventFactory, Sequencer sequencer){
56. **super**(eventFactory, sequencer);
57. }

       RingBuffer的内部结构明确了：内部用数组来实现，同时有保存数组长度的域bufferSize和下标掩码indexMask，还有一个sequencer。

       这里要注意几点：

       1.整个RingBuffer内部做了大量的缓存行填充，前后各填充了56个字节，entries本身也根据引用大小进行了填充，假设引用大小为4字节，那么entries数组两侧就要个填充32个空数组位。也就是说，实际的数组长度比bufferSize要大。所以可以看到根据序列从entries中取元素的方法elementAt内部做了一些调整，不是单纯的取模。

       2.bufferSize必须是2的幂，indexMask就是bufferSize-1，这样取模更高效(sequence&indexMask)。

       3.初始化时需要传入一个EventFactory，用来做队列内事件的预填充。

***总结下RingBuffer的特点：内部做了细致的缓存行填充，避免伪共享；内部队列基于数组实现，能很好的利用程序的局部性；队列上的事件槽会不断重复利用，不受JavaGC的影响。***

       从上面的代码看到，RingBuffer对本包外屏蔽了构造方法，那么怎创建RingBuffer实例呢？

**Java代码  [藏代码](javascript:void())**

1. **public** **static** <E> RingBuffer<E> createMultiProducer(EventFactory<E> factory,
2. **int**             bufferSize,
3. WaitStrategy    waitStrategy){
4. MultiProducerSequencer sequencer = **new** MultiProducerSequencer(bufferSize, waitStrategy);
5. **return** **new** RingBuffer<E>(factory, sequencer);
6. }
8. **public** **static** <E> RingBuffer<E> createMultiProducer(EventFactory<E> factory, **int** bufferSize){
9. **return** createMultiProducer(factory, bufferSize, **new** BlockingWaitStrategy());
10. }
12. **public** **static** <E> RingBuffer<E> createSingleProducer(EventFactory<E> factory,
13. **int**             bufferSize,
14. WaitStrategy    waitStrategy){
15. SingleProducerSequencer sequencer = **new** SingleProducerSequencer(bufferSize, waitStrategy);
16. **return** **new** RingBuffer<E>(factory, sequencer);
17. }
19. **public** **static** <E> RingBuffer<E> createSingleProducer(EventFactory<E> factory, **int** bufferSize){
20. **return** createSingleProducer(factory, bufferSize, **new** BlockingWaitStrategy());
21. }
23. **public** **static** <E> RingBuffer<E> create(ProducerType    producerType,
24. EventFactory<E> factory,
25. **int**             bufferSize,
26. WaitStrategy    waitStrategy){
27. **switch** (producerType){
28. **case** SINGLE:
29. **return** createSingleProducer(factory, bufferSize, waitStrategy);
30. **case** MULTI:
31. **return** createMultiProducer(factory, bufferSize, waitStrategy);
32. **default**:
33. **throw** **new** IllegalStateException(producerType.toString());
34. }
35. }

  可见，RingBuffer提供了静态工厂方法分别针对单事件发布者和多事件发布者的情况进行RingBuffer实例创建。

  RingBuffer方法实现也比较简单，看几个：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** E get(**long** sequence){
3. **return** elementAt(sequence);
4. }

   事件发布者和事件处理者申请到序列后，都会通过这个方法从序列上获取事件槽来生产或者发布事件。

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **long** next(**int** n){
3. **return** sequencer.next(n);
4. }

  next系列的方法是通过内部的sequencer来实现的。

**Java代码  [藏代码](javascript:void())**

1. **public** **boolean** isPublished(**long** sequence){
2. **return** sequencer.isAvailable(sequence);
3. }

  判断某个序列是否已经被事件发布者发布事件。

**Java代码  [藏代码](javascript:void())**

1. **public** **void** addGatingSequences(Sequence... gatingSequences){
2. sequencer.addGatingSequences(gatingSequences);
3. }
5. **public** **long** getMinimumGatingSequence(){
6. **return** sequencer.getMinimumSequence();
7. }
9. **public** **boolean** removeGatingSequence(Sequence sequence){
10. **return** sequencer.removeGatingSequence(sequence);
11. }

   追踪序列相关的方法。

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **void** publishEvent(EventTranslator<E> translator){
3. **final** **long** sequence = sequencer.next();
4. translateAndPublish(translator, sequence);
5. }
6. **private** **void** translateAndPublish(EventTranslator<E> translator, **long** sequence){
7. **try**{
8. translator.translateTo(get(sequence), sequence);
9. }**finally**{
10. sequencer.publish(sequence);
11. }
12. }

  可见，发布事件分三步：

   1.申请序列。

   2.填充事件。

   3.提交序列。

  其他方法实现都很简单，这里不啰嗦了。

* **最后总结：**

**有关RingBuffer要记住以下几点：**

**1.RingBuffer是协调事件发布者和事件处理者的中间队列，事件发布者发布事件放到RingBuffer，事件处理者从RingBuffer上拿事件进行消费。**

**2.RingBuffer可以认为是一个环形队列，底层由数组实现。内部做了大量的缓存行填充，保存事件使用的数组的长度必须是2的幂，这样可以高效的取模(取模本身就包含绕回逻辑，按照序列不断的增长，形成一个环形轨迹)。由于RingBuffer这样的特性，也避免了GC带来的性能影响，因为RingBuffer本身永远不会被GC。**

**3.RingBuffer和普通的FIFO队列相比还有一个重要的区别就是，RingBuffer避免了头尾节点的竞争，多个事件发布者/事件处理者之间不必竞争同一个节点，只需要安全申请序列值各自存取事件就好了。**

### 3.3 disruptor-3.3.2源码解析(3)-发布事件

* **Disruptor中如何发布事件：**

       前面两篇看了disruptor中的序列和队列，这篇说一下怎么往RingBuffer中发布事件。这里也需要明确一下，和一般的生产者/消费者模式不同(如果以生产者/消费者的模式来看待disruptor的话)，disruptor中队列里面的数据一般称为事件，RingBuffer中提供了发布事件的方法，另外也提供了专门的处理事件的类。

       其实在disruptor中，RingBuffer也提供了一部分生产的功能，里面提供了大量的发布事件的方法。

       上篇看到的RingBuffer的构造方法，需要传一个EventFactory做事件的预填充：

**Java代码  [藏代码](javascript:void())**

1. RingBuffer(EventFactory<E> eventFactory,
2. Sequencer       sequencer){
3. **super**(eventFactory, sequencer);
4. }

**Java代码  [藏代码](javascript:void())**

1. RingBufferFields(EventFactory<E> eventFactory,
2. Sequencer       sequencer){
3. ...
4. //最后要填充事件
5. fill(eventFactory);
6. }
7. **private** **void** fill(EventFactory<E> eventFactory){
8. **for** (**int** i = 0; i < bufferSize; i++){
9. entries[BUFFER\_PAD + i] = eventFactory.newInstance();
10. }
11. }

  看下EventFactory：

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** EventFactory<T>{
2. /\*
3. \* Implementations should instantiate an event object, with all memory already allocated where possible.
4. \*/
5. T newInstance();
6. }

       再看下RingBuffer的发布事件方法：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **void** publishEvent(EventTranslator<E> translator){
3. **final** **long** sequence = sequencer.next();
4. translateAndPublish(translator, sequence);
5. }
6. **private** **void** translateAndPublish(EventTranslatorVararg<E> translator, **long** sequence){
7. **try**{
8. translator.translateTo(get(sequence), sequence);
9. }**finally**{
10. sequencer.publish(sequence);
11. }
12. }

       在发布事件时需要传一个事件转换的接口，内部用这个接口做一下数据到事件的转换。看下这个接口：

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** EventTranslator<T>{
2. /\*\*
3. \* Translate a data representation into fields set in given event
4. \*
5. \* @param event into which the data should be translated.
6. \* @param sequence that is assigned to event.
7. \*/
8. **void** translateTo(**final** T event, **long** sequence);
9. }

       可见，具体的生产者可以实现这个接口，将需要发布的数据放到这个事件里面，一般是设置到事件的某个域上。

       好，来看个例子理解一下。

       首先我们定义好数据：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** MyData {
2. **private** **int** id;
3. **private** String value;
4. **public** MyData(**int** id, String value) {
5. **this**.id = id;
6. **this**.value = value;
7. }
9. ...getter setter...
11. @Override
12. **public** String toString() {
13. **return** "MyData [id=" + id + ", value=" + value + "]";
14. }
16. }

       然后针对我们的数据定义事件：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** MyDataEvent {
2. **private** MyData data;
3. **public** MyData getData() {
4. **return** data;
5. }
6. **public** **void** setData(MyData data) {
7. **this**.data = data;
8. }
10. }

       接下来需要给出一个EventFactory提供给RingBuffer做事件预填充：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** MyDataEventFactory **implements** EventFactory<MyDataEvent>{
2. @Override
3. **public** MyDataEvent newInstance() {
4. **return** **new** MyDataEvent();
5. }
6. }

       好了，可以初始化RingBuffer了：

**Java代码  [藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. RingBuffer<MyDataEvent> ringBuffer =
3. RingBuffer.createSingleProducer(**new** MyDataEventFactory(), 1024);
4. MyDataEvent dataEvent = ringBuffer.get(0);
5. System.out.println("Event = " + dataEvent);
6. System.out.println("Data = " + dataEvent.getData());
7. }

       输出如下：

**Java代码  [藏代码](javascript:void())**

1. Event = com.mjf.disruptor.product.MyDataEvent@5c647e05
2. Data = **null**

       首先要注意，RingBuffer里面是MydataEvent，而不是MyData；其次我们构造好了RingBuffer，里面就已经填充了事件，我们可以取一个事件出来，发现里面的数据是空的。

       下面就是怎么往RingBuffer里面放数据了，也就是事件发布者要干活了。我们上面看到了，要使用RingBuffer发布一个事件，需要一个事件转换器接口，针对我们的数据实现一个：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** MyDataEventTranslator **implements** EventTranslator<MyDataEvent>{
2. @Override
3. **public** **void** translateTo(MyDataEvent event, **long** sequence) {
4. //新建一个数据
5. MyData data = **new** MyData(1, "holy shit!");
6. //将数据放入事件中。
7. event.setData(data);
8. }
9. }

       有了转换器，我们就可以嗨皮的发布事件了：

**Java代码  [藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. RingBuffer<MyDataEvent> ringBuffer =
3. RingBuffer.createSingleProducer(**new** MyDataEventFactory(), 1024);
4. //发布事件!!!
5. ringBuffer.publishEvent(**new** MyDataEventTranslator());
7. MyDataEvent dataEvent0 = ringBuffer.get(0);
8. System.out.println("Event = " + dataEvent0);
9. System.out.println("Data = " + dataEvent0.getData());
10. MyDataEvent dataEvent1 = ringBuffer.get(1);
11. System.out.println("Event = " + dataEvent1);
12. System.out.println("Data = " + dataEvent1.getData());
13. }

       输出如下：

**Java代码  [藏代码](javascript:void())**

1. Event = com.mjf.disruptor.product.MyDataEvent@5c647e05
2. Data = MyData [id=1, value=holy shit!]
3. Event = com.mjf.disruptor.product.MyDataEvent@33909752
4. Data = **null**

       可见，我们已经成功了发布了一个事件到RingBuffer，由于是从序列0开始发布，所以我们从序列0可以读出这个数据。因为只发布了一个，所以序列1上还是没有数据。

       当然也有其他姿势的转换器：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** MyDataEventTranslatorWithIdAndValue **implements** EventTranslatorTwoArg<MyDataEvent, Integer, String>{
2. @Override
3. **public** **void** translateTo(MyDataEvent event, **long** sequence, Integer id,
4. String value) {
5. MyData data = **new** MyData(id, value);
6. event.setData(data);
7. }
8. }

       当然也可以直接利用RingBuffer来发布事件，不需要转换器：

**Java代码  [藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. RingBuffer<MyDataEvent> ringBuffer =
3. RingBuffer.createSingleProducer(**new** MyDataEventFactory(), 1024);
4. **long** sequence = ringBuffer.next();
5. **try**{
6. MyDataEvent event = ringBuffer.get(sequence);
7. MyData data = **new** MyData(2, "R u kidding me?");
8. event.setData(data);
9. }**finally**{
10. ringBuffer.publish(sequence);
11. }
12. }

* **单线程发布事件和多线程发布事件：**

       前面我们构造RingBuffer使用的是单线程发布事件的模式：

**Java代码  [藏代码](javascript:void())**

1. RingBuffer<MyDataEvent> ringBuffer =
2. RingBuffer.createSingleProducer(**new** MyDataEventFactory(), 1024);

       RingBuffer也支持多线程发布事件模式，还记得上一篇分析的RingBuffer代码吧：

**Java代码  [藏代码](javascript:void())**

1. **public** **static** <E> RingBuffer<E> createMultiProducer(EventFactory<E> factory, **int** bufferSize){
2. **return** createMultiProducer(factory, bufferSize, **new** BlockingWaitStrategy());
3. }

       当然也提供了比较全面的构造方法：

**Java代码  [藏代码](javascript:void())**

1. **public** **static** <E> RingBuffer<E> create(ProducerType    producerType,
2. EventFactory<E> factory,
3. **int**             bufferSize,
4. WaitStrategy    waitStrategy){
5. **switch** (producerType){
6. **case** SINGLE:
7. **return** createSingleProducer(factory, bufferSize, waitStrategy);
8. **case** MULTI:
9. **return** createMultiProducer(factory, bufferSize, waitStrategy);
10. **default**:
11. **throw** **new** IllegalStateException(producerType.toString());
12. }
13. }

       这个方法支持传入一个枚举来选择使用哪种模式：

**Java代码  [藏代码](javascript:void())**

1. **public** **enum** ProducerType{
2. /\*\* Create a RingBuffer with a single event publisher to the RingBuffer \*/
3. SINGLE,
4. /\*\* Create a RingBuffer supporting multiple event publishers to the one RingBuffer \*/
5. MULTI
6. }

       上面看过了单线程发布事件的例子，接下来看个多线程发布事件的：

**Java代码  [藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. **final** RingBuffer<MyDataEvent> ringBuffer =
3. RingBuffer.createMultiProducer(**new** MyDataEventFactory(), 1024);
4. **final** CountDownLatch latch = **new** CountDownLatch(100);
5. **for**(**int** i=0;i<100;i++){
6. **final** **int** index = i;
7. //开启多个线程发布事件。
8. **new** Thread(**new** Runnable() {
9. @Override
10. **public** **void** run() {
11. **long** sequence = ringBuffer.next();
12. **try**{
13. MyDataEvent event = ringBuffer.get(sequence);
14. MyData data = **new** MyData(index, index+"s");
15. event.setData(data);
16. }**finally**{
17. ringBuffer.publish(sequence);
18. latch.countDown();
19. }
20. }
21. }).start();
22. }
23. **try** {
24. latch.await();
25. //最后观察下发布的时间。
26. **for**(**int** i=0;i<100;i++){
27. MyDataEvent event = ringBuffer.get(i);
28. System.out.println(event.getData());
29. }
30. } **catch** (InterruptedException e) {
31. e.printStackTrace();
32. }
33. }

       如果多线程环境下使用单线程发布模式会有上面问题呢？

**Java代码  [藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. **final** RingBuffer<MyDataEvent> ringBuffer =
3. //这里是单线程模式!!!
4. RingBuffer.createSingleProducer(**new** MyDataEventFactory(), 1024);
5. **final** CountDownLatch latch = **new** CountDownLatch(100);
6. **for**(**int** i=0;i<100;i++){
7. **final** **int** index = i;
8. //开启多个线程发布事件。
9. **new** Thread(**new** Runnable() {
10. @Override
11. **public** **void** run() {
12. **long** sequence = ringBuffer.next();
13. **try**{
14. MyDataEvent event = ringBuffer.get(sequence);
15. MyData data = **new** MyData(index, index+"s");
16. event.setData(data);
17. }**finally**{
18. ringBuffer.publish(sequence);
19. latch.countDown();
20. }
21. }
22. }).start();
23. }
24. **try** {
25. latch.await();
26. //最后观察下发布的时间。
27. **for**(**int** i=0;i<100;i++){
28. MyDataEvent event = ringBuffer.get(i);
29. System.out.println(event.getData());
30. }
31. } **catch** (InterruptedException e) {
32. e.printStackTrace();
33. }
34. }

       输出如下：

**Java代码  [藏代码](javascript:void())**

1. ...
2. MyData [id=92, value=92s]
3. MyData [id=93, value=93s]
4. MyData [id=94, value=94s]
5. MyData [id=95, value=95s]
6. MyData [id=96, value=96s]
7. MyData [id=97, value=97s]
8. MyData [id=99, value=99s]
9. MyData [id=98, value=98s]
10. **null**
11. **null**

       会发现，如果多线程发布事件的环境下，使用单线程发布事件模式，会有数据被覆盖的情况。所以使用时应该按照具体情况选择合理发布模式。

* **最后总结：**

**如何往RingBuffer中发布事件：**

**1.定义好要生产的数据和相应的事件类(里面存放数据)。**

**2.定于好事件转换器或者直接用RingBuffer进行事件发布。**

**3.明确发布场景，合理的选择发布模式(单线程还是多线程)。**

### 3.4 disruptor-3.3.2源码解析(4)-处理事件

* **Disruptor中如何处理事件：**

       disruptor中提供了专门的事件处理器接口，先看下接口定义：

**Java代码  [藏代码](javascript:void())**

1. /\*\*
2. \* 事件处理器会等待RingBuffer中的事件变为可用(可处理)，然后处理可用的事件。
3. \* 一个事件处理器通常会关联一个线程。
4. \*/
5. **public** **interface** EventProcessor **extends** Runnable{
6. /\*\*
7. \* 获取一个事件处理器使用的序列引用。
8. \*/
9. Sequence getSequence();
11. **void** halt();
12. **boolean** isRunning();
13. }

       接下来看一个实现，BatchEventProcessor。先看下内部结构：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** BatchEventProcessor<T>
2. **implements** EventProcessor{
4. //表示当前事件处理器的运行状态
5. **private** **final** AtomicBoolean running = **new** AtomicBoolean(**false**);
6. //异常处理器。
7. **private** ExceptionHandler<? **super** T> exceptionHandler = **new** FatalExceptionHandler();
8. //数据提供者。(RingBuffer)
9. **private** **final** DataProvider<T> dataProvider;
10. //序列栅栏
11. **private** **final** SequenceBarrier sequenceBarrier;
12. //正真处理事件的回调接口。
13. **private** **final** EventHandler<? **super** T> eventHandler;
14. //事件处理器使用的序列。
15. **private** **final** Sequence sequence = **new** Sequence(Sequencer.INITIAL\_CURSOR\_VALUE);
16. //超时处理器。
17. **private** **final** TimeoutHandler timeoutHandler;
19. **public** BatchEventProcessor(**final** DataProvider<T> dataProvider,
20. **final** SequenceBarrier sequenceBarrier,
21. **final** EventHandler<? **super** T> eventHandler){
22. **this**.dataProvider = dataProvider;
23. **this**.sequenceBarrier = sequenceBarrier;
24. **this**.eventHandler = eventHandler;
25. **if** (eventHandler **instanceof** SequenceReportingEventHandler){
26. ((SequenceReportingEventHandler<?>)eventHandler).setSequenceCallback(sequence);
27. }
28. timeoutHandler = (eventHandler **instanceof** TimeoutHandler) ? (TimeoutHandler) eventHandler : **null**;
29. }

       大体对内部结构有个印象，然后看一下功能方法的实现：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** Sequence getSequence(){
3. **return** sequence;
4. }
5. @Override
6. **public** **void** halt(){
7. //设置运行状态为false。
8. running.set(**false**);
9. //通知序列栅栏。
10. sequenceBarrier.alert();
11. }
12. @Override
13. **public** **boolean** isRunning(){
14. **return** running.get();
15. }

       这几个方法都比较容易理解，重点看下run方法：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **void** run(){
3. //状态设置与检测。
4. **if** (!running.compareAndSet(**false**, **true**)){
5. **throw** **new** IllegalStateException("Thread is already running");
6. }
7. //先清除序列栅栏的通知状态。
8. sequenceBarrier.clearAlert();
9. //如果eventHandler实现了LifecycleAware，这里会对其进行一个启动通知。
10. notifyStart();
11. T event = **null**;
12. //获取要申请的序列值。
13. **long** nextSequence = sequence.get() + 1L;
14. **try**{
15. **while** (**true**){
16. **try**{
17. //通过序列栅栏来等待可用的序列值。
18. **final** **long** availableSequence = sequenceBarrier.waitFor(nextSequence);
19. //得到可用的序列值后，批量处理nextSequence到availableSequence之间的事件。
20. **while** (nextSequence <= availableSequence){
21. //获取事件。
22. event = dataProvider.get(nextSequence);
23. //将事件交给eventHandler处理。
24. eventHandler.onEvent(event, nextSequence, nextSequence == availableSequence);
25. nextSequence++;
26. }
27. //处理完毕后，设置当前处理完成的最后序列值。
28. sequence.set(availableSequence);
29. //继续循环。
30. }
31. **catch** (**final** TimeoutException e){
32. //如果发生超时，通知一下超时处理器(如果eventHandler同时实现了timeoutHandler，会将其设置为当前的超时处理器)
33. notifyTimeout(sequence.get());
34. }**catch** (**final** AlertException ex){
35. //如果捕获了序列栅栏变更通知，并且当前事件处理器停止了，那么退出主循环。
36. **if** (!running.get()){
37. **break**;
38. }
39. }**catch** (**final** Throwable ex){
40. //其他的异常都交给异常处理器进行处理。
41. exceptionHandler.handleEventException(ex, nextSequence, event);
42. //处理异常后仍然会设置当前处理的最后的序列值，然后继续处理其他事件。
43. sequence.set(nextSequence);
44. nextSequence++;
45. }
46. }
47. }
48. **finally**{
49. //主循环退出后，如果eventHandler实现了LifecycleAware，这里会对其进行一个停止通知。
50. notifyShutdown();
51. //设置事件处理器运行状态为停止。
52. running.set(**false**);
53. }
54. }

       贴上其他方法：

**Java代码  [藏代码](javascript:void())**

1. **public** **void** setExceptionHandler(**final** ExceptionHandler<? **super** T> exceptionHandler){
2. **if** (**null** == exceptionHandler){
3. **throw** **new** NullPointerException();
4. }
5. **this**.exceptionHandler = exceptionHandler;
6. }
8. **private** **void** notifyTimeout(**final** **long** availableSequence){
9. **try**{
10. **if** (timeoutHandler != **null**){
11. timeoutHandler.onTimeout(availableSequence);
12. }
13. }
14. **catch** (Throwable e){
15. exceptionHandler.handleEventException(e, availableSequence, **null**);
16. }
17. }
19. **private** **void** notifyStart(){
20. **if** (eventHandler **instanceof** LifecycleAware){
21. **try**{
22. ((LifecycleAware)eventHandler).onStart();
23. }**catch** (**final** Throwable ex){
24. exceptionHandler.handleOnStartException(ex);
25. }
26. }
27. }
29. **private** **void** notifyShutdown(){
30. **if** (eventHandler **instanceof** LifecycleAware){
31. **try**{
32. ((LifecycleAware)eventHandler).onShutdown();
33. }**catch** (**final** Throwable ex){
34. exceptionHandler.handleOnShutdownException(ex);
35. }
36. }
37. }

       总结一下BatchEventProcessor：

              1. BatchEventProcessor内部会记录自己的序列、运行状态。

              2. BatchEventProcessor需要外部提供数据提供者(其实就是队列-RingBuffer)、序列栅栏、异常处理器。

              3. BatchEventProcessor其实是将事件委托给内部的EventHandler来处理的。

       知道了这些，再结合前几篇的内容，来写个生产者/消费者模式的代码爽一下吧！

       首先，要定义具体处理事件的EventHandler，先来看下这个接口：

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** EventHandler<T>{
2. /\*\*
3. \* Called when a publisher has published an event to the {@link RingBuffer}
4. \*
5. \* @param event published to the {@link RingBuffer}
6. \* @param sequence of the event being processed
7. \* @param endOfBatch flag to indicate if this is the last event in a batch from the {@link RingBuffer}
8. \* @throws Exception if the EventHandler would like the exception handled further up the chain.
9. \*/
10. **void** onEvent(T event, **long** sequence, **boolean** endOfBatch) **throws** Exception;
11. }

       接口很明确，没什么要解释的，然后定义我们的具体处理方式：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** MyDataEventHandler **implements** EventHandler<MyDataEvent>{
2. @Override
3. **public** **void** onEvent(MyDataEvent event, **long** sequence, **boolean** endOfBatch)
4. **throws** Exception {
5. //注意这里小睡眠了一下!!
6. TimeUnit.SECONDS.sleep(3);
7. System.out.println("handle event's data:" + event.getData() +"isEndOfBatch:"+endOfBatch);
8. }
9. }

       然后是主程序：

**Java代码  [藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. //创建一个RingBuffer，注意容量是4。
3. RingBuffer<MyDataEvent> ringBuffer =
4. RingBuffer.createSingleProducer(**new** MyDataEventFactory(), 4);
5. //创建一个事件处理器。
6. BatchEventProcessor<MyDataEvent> batchEventProcessor =
7. /\*
8. \* 注意参数：数据提供者就是RingBuffer、序列栅栏也来自RingBuffer
9. \* EventHandler使用自定义的。
10. \*/
11. **new** BatchEventProcessor<MyDataEvent>(ringBuffer,
12. ringBuffer.newBarrier(), **new** MyDataEventHandler());
13. //将事件处理器本身的序列设置为ringBuffer的追踪序列。
14. ringBuffer.addGatingSequences(batchEventProcessor.getSequence());
15. //启动事件处理器。
16. **new** Thread(batchEventProcessor).start();
17. //往RingBuffer上发布事件
18. **for**(**int** i=0;i<10;i++){
19. ringBuffer.publishEvent(**new** MyDataEventTranslatorWithIdAndValue(), i, i+"s");
20. System.out.println("发布事件["+i+"]");
21. }
23. **try** {
24. System.in.read();
25. } **catch** (IOException e) {
26. e.printStackTrace();
27. }
28. }

       注意到当前RingBuffer只有4个空间，然后发了10个事件，而且消费者内部会sleep一下，所以会不会出现事件覆盖的情况呢(生产者在RingBuffer上绕了一圈从后面追上了消费者)。

       看下输出：

**Java代码  [藏代码](javascript:void())**

1. 发布事件[0]
2. 发布事件[1]
3. 发布事件[2]
4. 发布事件[3]
5. handle event's data:MyData [id=0, value=0s]isEndOfBatch:**true**
6. 发布事件[4]
7. handle event's data:MyData [id=1, value=1s]isEndOfBatch:**false**
8. handle event's data:MyData [id=2, value=2s]isEndOfBatch:**false**
9. handle event's data:MyData [id=3, value=3s]isEndOfBatch:**true**
10. 发布事件[5]
11. 发布事件[6]
12. 发布事件[7]
13. handle event's data:MyData [id=4, value=4s]isEndOfBatch:**true**
14. 发布事件[8]
15. handle event's data:MyData [id=5, value=5s]isEndOfBatch:**false**
16. handle event's data:MyData [id=6, value=6s]isEndOfBatch:**false**
17. handle event's data:MyData [id=7, value=7s]isEndOfBatch:**true**
18. 发布事件[9]
19. handle event's data:MyData [id=8, value=8s]isEndOfBatch:**true**
20. handle event's data:MyData [id=9, value=9s]isEndOfBatch:**true**

       发现其实并没有覆盖事件的情况，而是发布程序会等待事件处理程序处理完毕，有可发布的空间，再进行发布。还记得前面文章分析的序列的next()方法，里面会通过观察追踪序列的情况来决定是否等待，也就是说，RingBuffer需要追踪事件处理器的序列，这个关系怎么确立的呢？就是上面代码中的这句话：

**Java代码  [藏代码](javascript:void())**

1. //将事件处理器本身的序列设置为ringBuffer的追踪序列。
2. ringBuffer.addGatingSequences(batchEventProcessor.getSequence());

       接下来再改一下程序，将MyDataEventHandler中的睡眠代码去掉，然后主程序中发布事件后加上睡眠代码：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** MyDataEventHandler **implements** EventHandler<MyDataEvent>{
2. @Override
3. **public** **void** onEvent(MyDataEvent event, **long** sequence, **boolean** endOfBatch)
4. **throws** Exception {
5. //TimeUnit.SECONDS.sleep(3);
6. System.out.println("handle event's data:" + event.getData() +"isEndOfBatch:"+endOfBatch);
7. }
8. }
9. //主程序
10. **public** **static** **void** main(String[] args) {
11. //创建一个RingBuffer，注意容量是4。
12. RingBuffer<MyDataEvent> ringBuffer =
13. RingBuffer.createSingleProducer(**new** MyDataEventFactory(), 4);
14. //创建一个事件处理器。
15. BatchEventProcessor<MyDataEvent> batchEventProcessor =
16. /\*
17. \* 注意参数：数据提供者就是RingBuffer、序列栅栏也来自RingBuffer
18. \* EventHandler使用自定义的。
19. \*/
20. **new** BatchEventProcessor<MyDataEvent>(ringBuffer,
21. ringBuffer.newBarrier(), **new** MyDataEventHandler());
22. //将事件处理器本身的序列设置为ringBuffer的追踪序列。
23. ringBuffer.addGatingSequences(batchEventProcessor.getSequence());
24. //启动事件处理器。
25. **new** Thread(batchEventProcessor).start();
26. //往RingBuffer上发布事件
27. **for**(**int** i=0;i<10;i++){
28. ringBuffer.publishEvent(**new** MyDataEventTranslatorWithIdAndValue(), i, i+"s");
29. System.out.println("发布事件["+i+"]");
30. **try** {
31. TimeUnit.SECONDS.sleep(3);//睡眠！！！
32. } **catch** (InterruptedException e) {
33. e.printStackTrace();
34. }
35. }
37. **try** {
38. System.in.read();
39. } **catch** (IOException e) {
40. e.printStackTrace();
41. }
42. }

       这种情况下，事件处理很快，但是发布事件很慢，事件处理器会等待事件的发布么？

       看下输出：

**Java代码  [藏代码](javascript:void())**

1. 发布事件[0]
2. handle event's data:MyData [id=0, value=0s]isEndOfBatch:**true**
3. 发布事件[1]
4. handle event's data:MyData [id=1, value=1s]isEndOfBatch:**true**
5. 发布事件[2]
6. handle event's data:MyData [id=2, value=2s]isEndOfBatch:**true**
7. 发布事件[3]
8. handle event's data:MyData [id=3, value=3s]isEndOfBatch:**true**
9. 发布事件[4]
10. handle event's data:MyData [id=4, value=4s]isEndOfBatch:**true**
11. 发布事件[5]
12. handle event's data:MyData [id=5, value=5s]isEndOfBatch:**true**
13. 发布事件[6]
14. handle event's data:MyData [id=6, value=6s]isEndOfBatch:**true**
15. 发布事件[7]
16. handle event's data:MyData [id=7, value=7s]isEndOfBatch:**true**
17. 发布事件[8]
18. handle event's data:MyData [id=8, value=8s]isEndOfBatch:**true**
19. 发布事件[9]
20. handle event's data:MyData [id=9, value=9s]isEndOfBatch:**true**

       可见是会等待的，关键就是，事件处理器用的是RingBuffer的序列栅栏，会在栅栏上waitFor事件发布者：

**Java代码  [藏代码](javascript:void())**

1. **new** BatchEventProcessor<MyDataEvent>(ringBuffer,/\*注意这里\*/ringBuffer.newBarrier(), **new** MyDataEventHandler());

       接下来，我们在改变一下姿势，再加一个事件处理器，看看会发生什么：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** KickAssEventHandler **implements** EventHandler<MyDataEvent>{
2. @Override
3. **public** **void** onEvent(MyDataEvent event, **long** sequence, **boolean** endOfBatch)
4. **throws** Exception {
5. System.out.println("kick your ass "+sequence+" times!!!!");
6. }
7. }

       主程序如下：

**Java代码  [藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. //创建一个RingBuffer，注意容量是4。
3. RingBuffer<MyDataEvent> ringBuffer =
4. RingBuffer.createSingleProducer(**new** MyDataEventFactory(), 4);
5. //创建一个事件处理器。
6. BatchEventProcessor<MyDataEvent> batchEventProcessor =
7. /\*
8. \* 注意参数：数据提供者就是RingBuffer、序列栅栏也来自RingBuffer
9. \* EventHandler使用自定义的。
10. \*/
11. **new** BatchEventProcessor<MyDataEvent>(ringBuffer,
12. ringBuffer.newBarrier(), **new** MyDataEventHandler());
13. //创建一个事件处理器。
14. BatchEventProcessor<MyDataEvent> batchEventProcessor2 =
15. /\*
16. \* 注意参数：数据提供者就是RingBuffer、序列栅栏也来自RingBuffer
17. \* EventHandler使用自定义的。
18. \*/
19. **new** BatchEventProcessor<MyDataEvent>(ringBuffer,
20. ringBuffer.newBarrier(), **new** KickAssEventHandler());
21. //将事件处理器本身的序列设置为ringBuffer的追踪序列。
22. ringBuffer.addGatingSequences(batchEventProcessor.getSequence());
23. ringBuffer.addGatingSequences(batchEventProcessor2.getSequence());
24. //启动事件处理器。
25. **new** Thread(batchEventProcessor).start();
26. **new** Thread(batchEventProcessor2).start();
27. //往RingBuffer上发布事件
28. **for**(**int** i=0;i<10;i++){
29. ringBuffer.publishEvent(**new** MyDataEventTranslatorWithIdAndValue(), i, i+"s");
30. System.out.println("发布事件["+i+"]");
31. **try** {
32. TimeUnit.SECONDS.sleep(3);
33. } **catch** (InterruptedException e) {
34. e.printStackTrace();
35. }
36. }
37. **try** {
38. System.in.read();
39. } **catch** (IOException e) {
40. e.printStackTrace();
41. }
42. }

       看下输出：

**Java代码  [藏代码](javascript:void())**

1. 发布事件[0]
2. kick your ass 0 times!!!!
3. handle event's data:MyData [id=0, value=0s]isEndOfBatch:**true**
4. 发布事件[1]
5. kick your ass 1 times!!!!
6. handle event's data:MyData [id=1, value=1s]isEndOfBatch:**true**
7. 发布事件[2]
8. handle event's data:MyData [id=2, value=2s]isEndOfBatch:**true**
9. kick your ass 2 times!!!!
10. 发布事件[3]
11. handle event's data:MyData [id=3, value=3s]isEndOfBatch:**true**
12. kick your ass 3 times!!!!
13. 发布事件[4]
14. kick your ass 4 times!!!!
15. handle event's data:MyData [id=4, value=4s]isEndOfBatch:**true**
16. 发布事件[5]
17. kick your ass 5 times!!!!
18. handle event's data:MyData [id=5, value=5s]isEndOfBatch:**true**
19. 发布事件[6]
20. handle event's data:MyData [id=6, value=6s]isEndOfBatch:**true**
21. kick your ass 6 times!!!!
22. 发布事件[7]
23. kick your ass 7 times!!!!
24. handle event's data:MyData [id=7, value=7s]isEndOfBatch:**true**
25. 发布事件[8]
26. kick your ass 8 times!!!!
27. handle event's data:MyData [id=8, value=8s]isEndOfBatch:**true**
28. 发布事件[9]
29. kick your ass 9 times!!!!
30. handle event's data:MyData [id=9, value=9s]isEndOfBatch:**true**

       相当于有两个消费者，消费相同的数据，又有点像发布/订阅模式了，呵呵。

       当然，按照上面源码的分析，我们还可以让我们的Eventhandler同时实现TimeoutHandler和LifecycleAware来做更多的事；还可以定制一个ExceptionHandler来处理异常情况(框架中也提供了IgnoreExceptionHandler和FatalExceptionHandler两种ExceptionHandler实现，它们在异常处理时会记录不用级别的日志，后者还会抛出运行时异常)。

        Disruptor还提供了一个队列接口-EventPoller。通过这个接口也可以用RingBuffer中获取事件并处理，而且这个获取方式是无等待的，如果当前没有可处理的事件，会返回相应的状态-PollState。但注释说明这还是个实验性质的类，这里就不分析了。

* **Work消费模式：**

       看完上面的内容，可能会有这样的疑惑：实际用的时候可能不会只用单线程来消费吧，能不能使用多个线程来消费同一批事件，每个线程消费这批事件中的一部分呢？

       disruptor对这种情况也进行了支持，如果上面的叫Event模式，那么这个就叫Work模式吧(不要在意这些名词，知道是怎么回事儿就行了...)

       首先要看下WorkProcessor，还是先看结构：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** WorkProcessor<T>
2. **implements** EventProcessor{
4. **private** **final** AtomicBoolean running = **new** AtomicBoolean(**false**);
5. **private** **final** Sequence sequence = **new** Sequence(Sequencer.INITIAL\_CURSOR\_VALUE);
6. **private** **final** RingBuffer<T> ringBuffer;
7. **private** **final** SequenceBarrier sequenceBarrier;
8. **private** **final** WorkHandler<? **super** T> workHandler;
9. **private** **final** ExceptionHandler<? **super** T> exceptionHandler;
10. **private** **final** Sequence workSequence;
11. **private** **final** EventReleaser eventReleaser = **new** EventReleaser(){
12. @Override
13. **public** **void** release(){
14. sequence.set(Long.MAX\_VALUE);
15. }
16. };
18. **public** WorkProcessor(**final** RingBuffer<T> ringBuffer,
19. **final** SequenceBarrier sequenceBarrier,
20. **final** WorkHandler<? **super** T> workHandler,
21. **final** ExceptionHandler<? **super** T> exceptionHandler,
22. **final** Sequence workSequence){
23. **this**.ringBuffer = ringBuffer;
24. **this**.sequenceBarrier = sequenceBarrier;
25. **this**.workHandler = workHandler;
26. **this**.exceptionHandler = exceptionHandler;
27. **this**.workSequence = workSequence;
28. **if** (**this**.workHandler **instanceof** EventReleaseAware){
29. ((EventReleaseAware)**this**.workHandler).setEventReleaser(eventReleaser);
30. }
31. }

       看起来结构和BatchEventProcessor类似，区别是EventHandler变成了WorkHandler，还有了额外的workSequence和一个eventReleaser。

       看下功能方法实现：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** Sequence getSequence(){
3. **return** sequence;
4. }
5. @Override
6. **public** **void** halt(){
7. running.set(**false**);
8. sequenceBarrier.alert();
9. }
10. @Override
11. **public** **boolean** isRunning(){
12. **return** running.get();
13. }

       这些方法和BatchEventProcessor的一样，不说了。看下主逻辑：

**Java代码  [藏代码](javascript:void())**

1. @Override
2. **public** **void** run(){
3. //状态设置与检测。
4. **if** (!running.compareAndSet(**false**, **true**)){
5. **throw** **new** IllegalStateException("Thread is already running");
6. }
7. //先清除序列栅栏的通知状态。
8. sequenceBarrier.clearAlert();
9. //如果workHandler实现了LifecycleAware，这里会对其进行一个启动通知。
10. notifyStart();
11. **boolean** processedSequence = **true**;
12. **long** cachedAvailableSequence = Long.MIN\_VALUE;
13. **long** nextSequence = sequence.get();
14. T event = **null**;
15. **while** (**true**){
16. **try**{
17. //判断上一个事件是否已经处理完毕。
18. **if** (processedSequence){
19. //如果处理完毕，重置标识。
20. processedSequence = **false**;
21. //原子的获取下一要处理事件的序列值。
22. **do**{
23. nextSequence = workSequence.get() + 1L;
24. sequence.set(nextSequence - 1L);
25. }**while** (!workSequence.compareAndSet(nextSequence - 1L, nextSequence));
26. }
27. //检查序列值是否需要申请。这一步是为了防止和事件生产者冲突。
28. **if** (cachedAvailableSequence >= nextSequence){
29. //从RingBuffer上获取事件。
30. event = ringBuffer.get(nextSequence);
31. //委托给workHandler处理事件。
32. workHandler.onEvent(event);
33. //设置事件处理完成标识。
34. processedSequence = **true**;
35. }**else**{
36. //如果需要申请，通过序列栅栏来申请可用的序列。
37. cachedAvailableSequence = sequenceBarrier.waitFor(nextSequence);
38. }
39. }**catch** (**final** AlertException ex){
40. //处理通知。
41. **if** (!running.get()){
42. //如果当前处理器被停止，那么退出主循环。
43. **break**;
44. }
45. }**catch** (**final** Throwable ex){
46. //处理异常
47. exceptionHandler.handleEventException(ex, nextSequence, event);
48. //如果异常处理器不抛出异常的话，就认为事件处理完毕，设置事件处理完成标识。
49. processedSequence = **true**;
50. }
51. }
52. //退出主循环后，如果workHandler实现了LifecycleAware，这里会对其进行一个关闭通知。
53. notifyShutdown();
54. //设置当前处理器状态为停止。
55. running.set(**false**);
56. }

       说明一下WorkProcessor的主逻辑中几个重点：

              1.首先，由于是Work模式，必然是多个事件处理者(WorkProcessor)处理同一批事件，那么肯定会存在多个处理者对同一个要处理事件的竞争，所以出现了一个workSequence，所有的处理者都使用这一个workSequence，大家通过对workSequence的原子操作来保证不会处理相同的事件。

              2.其次，多个事件处理者和事件发布者之间也需要协调，需要等待事件发布者发布完事件之后才能对其进行处理，这里还是使用序列栅栏来协调(sequenceBarrier.waitFor)。

       接下来再看一下Work模式下另一个重要的类-WorkerPool：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** WorkerPool<T>{
3. //运行状态标识。
4. **private** **final** AtomicBoolean started = **new** AtomicBoolean(**false**);
5. //工作序列。
6. **private** **final** Sequence workSequence = **new** Sequence(Sequencer.INITIAL\_CURSOR\_VALUE);
7. //事件队列。
8. **private** **final** RingBuffer<T> ringBuffer;
9. //事件处理器数组。
10. **private** **final** WorkProcessor<?>[] workProcessors;
12. **public** WorkerPool(**final** RingBuffer<T> ringBuffer,
13. **final** SequenceBarrier sequenceBarrier,
14. **final** ExceptionHandler<? **super** T> exceptionHandler,
15. **final** WorkHandler<? **super** T>... workHandlers){
16. **this**.ringBuffer = ringBuffer;
17. **final** **int** numWorkers = workHandlers.length;
18. workProcessors = **new** WorkProcessor[numWorkers];
19. **for** (**int** i = 0; i < numWorkers; i++){
20. workProcessors[i] = **new** WorkProcessor<T>(ringBuffer,
21. sequenceBarrier,
22. workHandlers[i],
23. exceptionHandler,
24. workSequence);
25. }
26. }
28. **public** WorkerPool(**final** EventFactory<T> eventFactory,
29. **final** ExceptionHandler<? **super** T> exceptionHandler,
30. **final** WorkHandler<? **super** T>... workHandlers){
31. ringBuffer = RingBuffer.createMultiProducer(eventFactory, 1024, **new** BlockingWaitStrategy());
32. **final** SequenceBarrier barrier = ringBuffer.newBarrier();
33. **final** **int** numWorkers = workHandlers.length;
34. workProcessors = **new** WorkProcessor[numWorkers];
35. **for** (**int** i = 0; i < numWorkers; i++){
36. workProcessors[i] = **new** WorkProcessor<T>(ringBuffer,
37. barrier,
38. workHandlers[i],
39. exceptionHandler,
40. workSequence);
41. }
42. ringBuffer.addGatingSequences(getWorkerSequences());
43. }

       WorkerPool是Work模式下的事件处理器池，它起到了维护事件处理器生命周期、关联事件处理器与事件队列(RingBuffer)、提供工作序列(上面分析WorkProcessor时看到多个处理器需要使用统一的workSequence)的作用。

       看一下WorkerPool的方法：

**Java代码  [藏代码](javascript:void())**

1. **public** Sequence[] getWorkerSequences(){
2. **final** Sequence[] sequences = **new** Sequence[workProcessors.length + 1];
3. **for** (**int** i = 0, size = workProcessors.length; i < size; i++){
4. sequences[i] = workProcessors[i].getSequence();
5. }
6. sequences[sequences.length - 1] = workSequence;
7. **return** sequences;
8. }

       通过WorkerPool是可以获取内部事件处理器各自的序列和当前的WorkSequence，用于观察事件处理进度。

**Java代码  [藏代码](javascript:void())**

1. **public** RingBuffer<T> start(**final** Executor executor){
2. **if** (!started.compareAndSet(**false**, **true**)){
3. **throw** **new** IllegalStateException("WorkerPool has already been started and cannot be restarted until halted.");
4. }
5. **final** **long** cursor = ringBuffer.getCursor();
6. workSequence.set(cursor);
7. **for** (WorkProcessor<?> processor : workProcessors){
8. processor.getSequence().set(cursor);
9. executor.execute(processor);
10. }
11. **return** ringBuffer;
12. }

       start方法里面会初始化工作序列，然后使用一个给定的执行器(线程池)来执行内部的事件处理器。

**Java代码  [藏代码](javascript:void())**

1. **public** **void** drainAndHalt(){
2. Sequence[] workerSequences = getWorkerSequences();
3. **while** (ringBuffer.getCursor() > Util.getMinimumSequence(workerSequences)){
4. Thread.yield();
5. }
6. **for** (WorkProcessor<?> processor : workProcessors){
7. processor.halt();
8. }
9. started.set(**false**);
10. }

       drainAndHalt方法会将RingBuffer中所有的事件取出，执行完毕后，然后停止当前WorkerPool。

**Java代码  [藏代码](javascript:void())**

1. **public** **void** halt(){
2. **for** (WorkProcessor<?> processor : workProcessors){
3. processor.halt();
4. }
5. started.set(**false**);
6. }

       马上停止当前WorkerPool。

**Java代码  [藏代码](javascript:void())**

1. **public** **boolean** isRunning(){
2. **return** started.get();
3. }

       获取当前WorkerPool运行状态。

       好了，有了WorkProcessor和WorkerPool，我们又可以嗨皮的写一下多线程消费的生产者/消费者模式了！

       还记得WorkProcessor由内部的WorkHandler来具体处理事件吧，所以先写一个WorkHandler：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** MyDataWorkHandler **implements** WorkHandler<MyDataEvent>{
3. **private** String name;
5. **public** MyDataWorkHandler(String name) {
6. **this**.name = name;
7. }
8. @Override
9. **public** **void** onEvent(MyDataEvent event) **throws** Exception {
10. System.out.println("WorkHandler["+name+"]处理事件"+event.getData());
11. }
12. }

       然后是主程序：(其他的类源码见上篇)

**Java代码  [藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. //创建一个RingBuffer，注意容量是4。
3. RingBuffer<MyDataEvent> ringBuffer =
4. RingBuffer.createSingleProducer(**new** MyDataEventFactory(), 4);
5. //创建3个WorkHandler
6. MyDataWorkHandler handler1 = **new** MyDataWorkHandler("1");
7. MyDataWorkHandler handler2 = **new** MyDataWorkHandler("2");
8. MyDataWorkHandler handler3 = **new** MyDataWorkHandler("3");
9. WorkerPool<MyDataEvent> workerPool =
10. **new** WorkerPool<MyDataEvent>(ringBuffer, ringBuffer.newBarrier(),
11. **new** IgnoreExceptionHandler(),
12. handler1,handler2,handler3);
13. //将WorkPool的工作序列集设置为ringBuffer的追踪序列。
14. ringBuffer.addGatingSequences(workerPool.getWorkerSequences());
15. //创建一个线程池用于执行Workhandler。
16. Executor executor = Executors.newFixedThreadPool(4);
17. //启动WorkPool。
18. workerPool.start(executor);
19. //往RingBuffer上发布事件
20. **for**(**int** i=0;i<10;i++){
21. ringBuffer.publishEvent(**new** MyDataEventTranslatorWithIdAndValue(), i, i+"s");
22. System.out.println("发布事件["+i+"]");
23. }
24. **try** {
25. System.in.read();
26. } **catch** (IOException e) {
27. e.printStackTrace();
28. }
29. }

       输出如下：

**Java代码  [藏代码](javascript:void())**

1. 发布事件[0]
2. WorkHandler[1]处理事件MyData [id=0, value=0s]
3. 发布事件[1]
4. 发布事件[2]
5. WorkHandler[1]处理事件MyData [id=3, value=3s]
6. 发布事件[3]
7. WorkHandler[2]处理事件MyData [id=2, value=2s]
8. WorkHandler[3]处理事件MyData [id=1, value=1s]
9. 发布事件[4]
10. WorkHandler[1]处理事件MyData [id=4, value=4s]
11. 发布事件[5]
12. WorkHandler[2]处理事件MyData [id=5, value=5s]
13. WorkHandler[3]处理事件MyData [id=6, value=6s]
14. 发布事件[6]
15. 发布事件[7]
16. WorkHandler[1]处理事件MyData [id=7, value=7s]
17. 发布事件[8]
18. 发布事件[9]
19. WorkHandler[2]处理事件MyData [id=8, value=8s]
20. WorkHandler[3]处理事件MyData [id=9, value=9s]

       正是我们想要的效果，多个处理器处理同一批事件！

* **事件处理者如何等待事件发布者发布事件：**

       上面的分析中已经提到了事件发布者会使用序列栅栏来等待(waitFor)事件发布者发布事件，但具体是怎么等待呢？是阻塞还是自旋？接下来仔细看下这部分，Disruptor框架对这部分提供了很多策略。

       首先我们看下框架中提供的SequenceBarrier的实现-ProcessingSequenceBarrier类的waitFor细节：

**Java代码  [藏代码](javascript:void())**

1. **public** ProcessingSequenceBarrier(**final** Sequencer sequencer,
2. **final** WaitStrategy waitStrategy,
3. **final** Sequence cursorSequence,
4. **final** Sequence[] dependentSequences){
5. **this**.sequencer = sequencer;
6. **this**.waitStrategy = waitStrategy;
7. **this**.cursorSequence = cursorSequence;
8. **if** (0 == dependentSequences.length){
9. dependentSequence = cursorSequence;
10. }
11. **else**{
12. dependentSequence = **new** FixedSequenceGroup(dependentSequences);
13. }
14. }
15. @Override
16. **public** **long** waitFor(**final** **long** sequence)
17. **throws** AlertException, InterruptedException, TimeoutException{
18. checkAlert();
19. **long** availableSequence = waitStrategy.waitFor(sequence, cursorSequence, dependentSequence, **this**);
20. **if** (availableSequence < sequence){
21. **return** availableSequence;
22. }
23. **return** sequencer.getHighestPublishedSequence(sequence, availableSequence);
24. }

       我们注意到waitFor方法中其实是调用了WaitStrategy的waitFor方法来实现等待，WaitStrategy接口是框架中定义的等待策略接口(其实在RingBuffer构造的可以传入一个等待策略，没显式传入的话，会使用默认的等待策略，可以查看下RingBuffer的构造方法)，先看下这个接口：

**Java代码  [藏代码](javascript:void())**

1. **public** **interface** WaitStrategy{
2. /\*\*
3. \* 等待给定的sequence变为可用。
4. \*
5. \* @param sequence 等待(申请)的序列值。
6. \* @param cursor ringBuffer中的主序列，也可以认为是事件发布者使用的序列。
7. \* @param dependentSequence 事件处理者使用的序列。
8. \* @param barrier 序列栅栏。
9. \* @return 对事件处理者来说可用的序列值，可能会比申请的序列值大。
10. \* @throws AlertException if the status of the Disruptor has changed.
11. \* @throws InterruptedException if the thread is interrupted.
12. \* @throws TimeoutException
13. \*/
14. **long** waitFor(**long** sequence, Sequence cursor, Sequence dependentSequence, SequenceBarrier barrier)
15. **throws** AlertException, InterruptedException, TimeoutException;
16. /\*\*
17. \* 当发布事件成功后会调用这个方法来通知等待的事件处理者序列可用了。
18. \*/
19. **void** signalAllWhenBlocking();
20. }

**Disruptor框架中提供了如下几种等待策略：**

*BlockingWaitStrategy*

*BusySpinWaitStrategy*

*LiteBlockingWaitStrategy*

*SleepingWaitStrategy*

*TimeoutBlockingWaitStrategy*

*YieldingWaitStrategy*

*PhasedBackoffWaitStrategy*

       首先看下BlockingWaitStrategy，也是默认的等待序列：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** BlockingWaitStrategy **implements** WaitStrategy{
2. **private** **final** Lock lock = **new** ReentrantLock();
3. **private** **final** Condition processorNotifyCondition = lock.newCondition();
4. @Override
5. **public** **long** waitFor(**long** sequence, Sequence cursorSequence, Sequence dependentSequence, SequenceBarrier barrier) **throws** AlertException, InterruptedException{
7. **long** availableSequence;
8. **if** ((availableSequence = cursorSequence.get()) < sequence){
9. //如果RingBuffer上当前可用的序列值小于要申请的序列值。
10. lock.lock();
11. **try**{
12. //再次检测
13. **while** ((availableSequence = cursorSequence.get()) < sequence){
14. //检查序列栅栏状态(事件处理器是否被关闭)
15. barrier.checkAlert();
16. //当前线程在processorNotifyCondition条件上等待。
17. processorNotifyCondition.await();
18. }
19. }**finally**{
20. lock.unlock();
21. }
22. }
23. //再次检测，避免事件处理器关闭的情况。
24. **while** ((availableSequence = dependentSequence.get()) < sequence){
25. barrier.checkAlert();
26. }
27. **return** availableSequence;
28. }
29. @Override
30. **public** **void** signalAllWhenBlocking(){
31. lock.lock();
32. **try**{
33. //唤醒在processorNotifyCondition条件上等待的处理事件线程。
34. processorNotifyCondition.signalAll();
35. }**finally**{
36. lock.unlock();
37. }
38. }
39. }

       可见BlockingWaitStrategy的实现方法是阻塞等待。当要求节省CPU资源，而不要求高吞吐量和低延迟的时候使用这个策略。

       再看下BusySpinWaitStrategy：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** BusySpinWaitStrategy **implements** WaitStrategy{
2. @Override
3. **public** **long** waitFor(**final** **long** sequence, Sequence cursor, **final** Sequence dependentSequence, **final** SequenceBarrier barrier)**throws** AlertException, InterruptedException{
5. **long** availableSequence;
6. **while** ((availableSequence = dependentSequence.get()) < sequence){
7. barrier.checkAlert();
8. //自旋
9. }
10. **return** availableSequence;
11. }
12. @Override
13. **public** **void** signalAllWhenBlocking(){
14. }
15. }

       BusySpinWaitStrategy的实现方法是自旋等待。这种策略会利用CPU资源来避免系统调用带来的延迟抖动，当线程可以绑定到指定CPU(核)的时候可以使用这个策略。

       再看下LiteBlockingWaitStrategy：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** LiteBlockingWaitStrategy **implements** WaitStrategy{
2. **private** **final** Lock lock = **new** ReentrantLock();
3. **private** **final** Condition processorNotifyCondition = lock.newCondition();
4. **private** **final** AtomicBoolean signalNeeded = **new** AtomicBoolean(**false**);
5. @Override
6. **public** **long** waitFor(**long** sequence, Sequence cursorSequence, Sequence dependentSequence, SequenceBarrier barrier) **throws** AlertException, InterruptedException{
7. **long** availableSequence;
8. **if** ((availableSequence = cursorSequence.get()) < sequence) {
9. lock.lock();
10. **try**{
11. **do**{
12. signalNeeded.getAndSet(**true**);
13. **if** ((availableSequence = cursorSequence.get()) >= sequence) {
14. **break**;
15. }
16. barrier.checkAlert();
17. processorNotifyCondition.await();
18. } **while** ((availableSequence = cursorSequence.get()) < sequence);
19. }**finally**{
20. lock.unlock();
21. }
22. }
23. **while** ((availableSequence = dependentSequence.get()) < sequence){
24. barrier.checkAlert();
25. }
26. **return** availableSequence;
27. }
28. @Override
29. **public** **void** signalAllWhenBlocking(){
30. **if** (signalNeeded.getAndSet(**false**)){
31. lock.lock();
32. **try**{
33. processorNotifyCondition.signalAll();
34. }**finally**{
35. lock.unlock();
36. }
37. }
38. }
39. }

       相比BlockingWaitStrategy，LiteBlockingWaitStrategy的实现方法也是阻塞等待，但它会减少一些不必要的唤醒。从源码的注释上看，这个策略在基准性能测试上是会表现出一些性能提升，但是作者还不能完全证明程序的正确性。

       再看下SleepingWaitStrategy：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** SleepingWaitStrategy **implements** WaitStrategy{
3. **private** **static** **final** **int** DEFAULT\_RETRIES = 200;
4. **private** **final** **int** retries;
5. **public** SleepingWaitStrategy(){
6. **this**(DEFAULT\_RETRIES);
7. }
8. **public** SleepingWaitStrategy(**int** retries){
9. **this**.retries = retries;
10. }
11. @Override
12. **public** **long** waitFor(**final** **long** sequence, Sequence cursor, **final** Sequence dependentSequence, **final** SequenceBarrier barrier) **throws** AlertException, InterruptedException{
14. **long** availableSequence;
15. **int** counter = retries;
16. **while** ((availableSequence = dependentSequence.get()) < sequence){
17. counter = applyWaitMethod(barrier, counter);
18. }
19. **return** availableSequence;
20. }
21. @Override
22. **public** **void** signalAllWhenBlocking(){
23. }
24. **private** **int** applyWaitMethod(**final** SequenceBarrier barrier, **int** counter)
25. **throws** AlertException{
26. barrier.checkAlert();
27. //从指定的重试次数(默认是200)重试到剩下100次，这个过程是自旋。
28. **if** (counter > 100){
29. --counter;
30. }
31. //然后尝试100次让出处理器动作。
32. **else** **if** (counter > 0){
33. --counter;
34. Thread.yield();
35. }
36. //然后尝试阻塞1纳秒。
37. **else**{
38. LockSupport.parkNanos(1L);
39. }
40. **return** counter;
41. }
42. }

       SleepingWaitStrategy的实现方法是先自旋，不行再临时让出调度(yield)，不行再短暂的阻塞等待。对于既想取得高性能，由不想太浪费CPU资源的场景，这个策略是一种比较好的折中方案。使用这个方案可能会出现延迟波动。

       再看下TimeoutBlockingWaitStrategy：

**Java代码  [藏代码](javascript:void())**

1. **public** **class** TimeoutBlockingWaitStrategy **implements** WaitStrategy{
2. **private** **final** Lock lock = **new** ReentrantLock();
3. **private** **final** Condition processorNotifyCondition = lock.newCondition();
4. **private** **final** **long** timeoutInNanos;
5. **public** TimeoutBlockingWaitStrategy(**final** **long** timeout, **final** TimeUnit units){
6. timeoutInNanos = units.toNanos(timeout);
7. }
8. @Override
9. **public** **long** waitFor(**final** **long** sequence,
10. **final** Sequence cursorSequence,
11. **final** Sequence dependentSequence,
12. **final** SequenceBarrier barrier)
13. **throws** AlertException, InterruptedException, TimeoutException{
14. **long** nanos = timeoutInNanos;
15. **long** availableSequence;
16. **if** ((availableSequence = cursorSequence.get()) < sequence){
17. lock.lock();
18. **try**{
19. **while** ((availableSequence = cursorSequence.get()) < sequence){
20. barrier.checkAlert();
21. nanos = processorNotifyCondition.awaitNanos(nanos);
22. **if** (nanos <= 0){
23. **throw** TimeoutException.INSTANCE;
24. }
25. }
26. }**finally**{
27. lock.unlock();
28. }
29. }
30. **while** ((availableSequence = dependentSequence.get()) < sequence){
31. barrier.checkAlert();
32. }
33. **return** availableSequence;
34. }
35. @Override
36. **public** **void** signalAllWhenBlocking(){
37. lock.lock();
38. **try**{
39. processorNotifyCondition.signalAll();
40. }**finally**{
41. lock.unlock();
42. }
43. }
44. }

       TimeoutBlockingWaitStrategy的实现方法是阻塞给定的时间，超过时间的话会抛出超时异常。

       再看下YieldingWaitStrategy：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** YieldingWaitStrategy **implements** WaitStrategy{
2. **private** **static** **final** **int** SPIN\_TRIES = 100;
3. @Override
4. **public** **long** waitFor(**final** **long** sequence, Sequence cursor, **final** Sequence dependentSequence, **final** SequenceBarrier barrier) **throws** AlertException, InterruptedException{
5. **long** availableSequence;
6. **int** counter = SPIN\_TRIES;
7. **while** ((availableSequence = dependentSequence.get()) < sequence){
8. counter = applyWaitMethod(barrier, counter);
9. }
10. **return** availableSequence;
11. }
12. @Override
13. **public** **void** signalAllWhenBlocking(){
14. }
15. **private** **int** applyWaitMethod(**final** SequenceBarrier barrier, **int** counter)
16. **throws** AlertException{
17. barrier.checkAlert();
18. **if** (0 == counter){
19. Thread.yield();
20. }**else**{
21. --counter;
22. }
23. **return** counter;
24. }
25. }

       SleepingWaitStrategy的实现方法是先自旋(100次)，不行再临时让出调度(yield)。和SleepingWaitStrategy一样也是一种高性能与CPU资源之间取舍的折中方案，但这个策略不会带来显著的延迟抖动。

       最后看下PhasedBackoffWaitStrategy：

**Java代码  [藏代码](javascript:void())**

1. **public** **final** **class** PhasedBackoffWaitStrategy **implements** WaitStrategy{
2. **private** **static** **final** **int** SPIN\_TRIES = 10000;
3. **private** **final** **long** spinTimeoutNanos;
4. **private** **final** **long** yieldTimeoutNanos;
5. **private** **final** WaitStrategy fallbackStrategy;
6. **public** PhasedBackoffWaitStrategy(**long** spinTimeout,
7. **long** yieldTimeout,
8. TimeUnit units,
9. WaitStrategy fallbackStrategy){
10. **this**.spinTimeoutNanos = units.toNanos(spinTimeout);
11. **this**.yieldTimeoutNanos = spinTimeoutNanos + units.toNanos(yieldTimeout);
12. **this**.fallbackStrategy = fallbackStrategy;
13. }
15. **public** **static** PhasedBackoffWaitStrategy withLock(**long** spinTimeout,
16. **long** yieldTimeout,
17. TimeUnit units){
18. **return** **new** PhasedBackoffWaitStrategy(spinTimeout, yieldTimeout,
19. units, **new** BlockingWaitStrategy());
20. }
22. **public** **static** PhasedBackoffWaitStrategy withLiteLock(**long** spinTimeout,
23. **long** yieldTimeout,
24. TimeUnit units){
25. **return** **new** PhasedBackoffWaitStrategy(spinTimeout, yieldTimeout,
26. units, **new** LiteBlockingWaitStrategy());
27. }
29. **public** **static** PhasedBackoffWaitStrategy withSleep(**long** spinTimeout,
30. **long** yieldTimeout,
31. TimeUnit units){
32. **return** **new** PhasedBackoffWaitStrategy(spinTimeout, yieldTimeout,
33. units, **new** SleepingWaitStrategy(0));
34. }
35. @Override
36. **public** **long** waitFor(**long** sequence, Sequence cursor, Sequence dependentSequence, SequenceBarrier barrier) **throws** AlertException, InterruptedException, TimeoutException{
37. **long** availableSequence;
38. **long** startTime = 0;
39. **int** counter = SPIN\_TRIES;
40. **do**{
41. **if** ((availableSequence = dependentSequence.get()) >= sequence){
42. **return** availableSequence;
43. }
44. **if** (0 == --counter){
45. **if** (0 == startTime){
46. startTime = System.nanoTime();
47. }**else**{
48. **long** timeDelta = System.nanoTime() - startTime;
49. **if** (timeDelta > yieldTimeoutNanos){
50. **return** fallbackStrategy.waitFor(sequence, cursor, dependentSequence, barrier);
51. }**else** **if** (timeDelta > spinTimeoutNanos){
52. Thread.yield();
53. }
54. }
55. counter = SPIN\_TRIES;
56. }
57. }**while** (**true**);
58. }
59. @Override
60. **public** **void** signalAllWhenBlocking(){
61. fallbackStrategy.signalAllWhenBlocking();
62. }
63. }

       PhasedBackoffWaitStrategy的实现方法是先自旋(10000次)，不行再临时让出调度(yield)，不行再使用其他的策略进行等待。可以根据具体场景自行设置自旋时间、yield时间和备用等待策略。

* **最后总结：**

**1.事件处理者可以通过Event模式或者Work模式来处理事件。**

**2.事件处理者可以使用多种等待策略来等待事件发布者发布事件，可按照具体场景选择合适的等待策略。**

### 3.5 disruptor-3.3.2源码解析(5)-框架支持

* **更方便的使用Disruptor：**

       前面几篇看了Disruptor中的一些重要组件和组件的运行方式，也通过手动组合这些组件的方式给出了一些基本的用例。框架也提供了一个DSL-style API，来帮助我们更容易的使用框架，屏蔽掉一些细节(比如怎么构建RingBuffer、怎么关联追踪序列等)，相当于Builder模式。

       在看Disruptor之前，先看一些辅助类，首先看下ConsumerRepository：

**Java代码  [藏代码](javascript:void())**

1. **class** ConsumerRepository<T> **implements** Iterable<ConsumerInfo>{
3. **private** **final** Map<EventHandler<?>, EventProcessorInfo<T>> eventProcessorInfoByEventHandler = **new** IdentityHashMap<EventHandler<?>, EventProcessorInfo<T>>();
4. **private** **final** Map<Sequence, ConsumerInfo> eventProcessorInfoBySequence = **new** IdentityHashMap<Sequence, ConsumerInfo>();
5. **private** **final** Collection<ConsumerInfo> consumerInfos = **new** ArrayList<ConsumerInfo>();

       可见ConsumerRepository内部存储着事件处理者(消费者)的信息，相当于事件处理者的仓库。

       看一下里面的方法：

**Java代码  [藏代码](javascript:void())**

1. **public** **void** add(**final** EventProcessor eventprocessor,
2. **final** EventHandler<? **super** T> handler,
3. **final** SequenceBarrier barrier){
4. **final** EventProcessorInfo<T> consumerInfo = **new** EventProcessorInfo<T>(eventprocessor, handler, barrier);
5. eventProcessorInfoByEventHandler.put(handler, consumerInfo);
6. eventProcessorInfoBySequence.put(eventprocessor.getSequence(), consumerInfo);
7. consumerInfos.add(consumerInfo);
8. }

       添加事件处理者(Event模式)、事件处理器和序列栅栏到仓库中。

**Java代码  [藏代码](javascript:void())**

1. **public** **void** add(**final** EventProcessor processor){
2. **final** EventProcessorInfo<T> consumerInfo = **new** EventProcessorInfo<T>(processor, **null**, **null**);
3. eventProcessorInfoBySequence.put(processor.getSequence(), consumerInfo);
4. consumerInfos.add(consumerInfo);
5. }

       添加事件处理者(Event模式)到仓库中。

**Java代码  [藏代码](javascript:void())**

1. **public** **void** add(**final** WorkerPool<T> workerPool, **final** SequenceBarrier sequenceBarrier){
2. **final** WorkerPoolInfo<T> workerPoolInfo = **new** WorkerPoolInfo<T>(workerPool, sequenceBarrier);
3. consumerInfos.add(workerPoolInfo);
4. **for** (Sequence sequence : workerPool.getWorkerSequences()){
5. eventProcessorInfoBySequence.put(sequence, workerPoolInfo);
6. }
7. }

       添加事件处理者(Work模式)和序列栅栏到仓库中。

**Java代码  [藏代码](javascript:void())**

1. **public** Sequence[] getLastSequenceInChain(**boolean** includeStopped){
2. List<Sequence> lastSequence = **new** ArrayList<Sequence>();
3. **for** (ConsumerInfo consumerInfo : consumerInfos){
4. **if** ((includeStopped || consumerInfo.isRunning()) && consumerInfo.isEndOfChain()){
5. **final** Sequence[] sequences = consumerInfo.getSequences();
6. Collections.addAll(lastSequence, sequences);
7. }
8. }
9. **return** lastSequence.toArray(**new** Sequence[lastSequence.size()]);
10. }

       获取当前已经消费到RingBuffer上事件队列末尾的事件处理者的序列，可通过参数指定是否要包含已经停止的事件处理者。

**Java代码  [藏代码](javascript:void())**

1. **public** **void** unMarkEventProcessorsAsEndOfChain(**final** Sequence... barrierEventProcessors){
2. **for** (Sequence barrierEventProcessor : barrierEventProcessors){
3. getEventProcessorInfo(barrierEventProcessor).markAsUsedInBarrier();
4. }
5. }

       重置已经处理到事件队列末尾的事件处理者的状态。

       其他方法就不看了。

       上面代码中出现的ConsumerInfo就相当于事件处理者信息和序列栅栏的包装类，ConsumerInfo本身是一个接口，针对Event模式和Work模式提供了两种实现：EventProcessorInfo和WorkerPoolInfo，代码都很容易理解，这里就不贴了。

**现在来看下Disruptor类，先看结构：**

**Java代码  [藏代码](javascript:void())**

1. **public** **class** Disruptor<T>{
3. //事件队列。
4. **private** **final** RingBuffer<T> ringBuffer;
5. //用于执行事件处理的执行器。
6. **private** **final** Executor executor;
7. //事件处理信息仓库。
8. **private** **final** ConsumerRepository<T> consumerRepository = **new** ConsumerRepository<T>();
9. //运行状态。
10. **private** **final** AtomicBoolean started = **new** AtomicBoolean(**false**);
11. //异常处理器。
12. **private** ExceptionHandler<? **super** T> exceptionHandler;

       可见，Disruptor内部包含了我们之前写用例使用到的所有组件。

**再看下构造方法：**

**Java代码  [藏代码](javascript:void())**

1. **public** Disruptor(**final** EventFactory<T> eventFactory, **final** **int** ringBufferSize, **final** Executor executor){
2. **this**(RingBuffer.createMultiProducer(eventFactory, ringBufferSize), executor);
3. }
5. **public** Disruptor(**final** EventFactory<T> eventFactory,
6. **final** **int** ringBufferSize,
7. **final** Executor executor,
8. **final** ProducerType producerType,
9. **final** WaitStrategy waitStrategy){
10. **this**(RingBuffer.create(producerType, eventFactory, ringBufferSize, waitStrategy),
11. executor);
12. }
14. **private** Disruptor(**final** RingBuffer<T> ringBuffer, **final** Executor executor)
15. {
16. **this**.ringBuffer = ringBuffer;
17. **this**.executor = executor;
18. }

       可见，通过构造方法，可以对内部的RingBuffer和执行器进行初始化。

**有了事件队列(RingBuffer)，接下来看看怎么构建事件处理者：**

**Java代码  [藏代码](javascript:void())**

1. @SuppressWarnings("varargs")
2. **public** EventHandlerGroup<T> handleEventsWith(**final** EventHandler<? **super** T>... handlers){
3. **return** createEventProcessors(**new** Sequence[0], handlers);
4. }
5. EventHandlerGroup<T> createEventProcessors(**final** Sequence[] barrierSequences,
6. **final** EventHandler<? **super** T>[] eventHandlers){
7. checkNotStarted();
8. **final** Sequence[] processorSequences = **new** Sequence[eventHandlers.length];
9. **final** SequenceBarrier barrier = ringBuffer.newBarrier(barrierSequences);
10. **for** (**int** i = 0, eventHandlersLength = eventHandlers.length; i < eventHandlersLength; i++){
11. **final** EventHandler<? **super** T> eventHandler = eventHandlers[i];
12. **final** BatchEventProcessor<T> batchEventProcessor = **new** BatchEventProcessor<T>(ringBuffer, barrier, eventHandler);
13. **if** (exceptionHandler != **null**){
14. batchEventProcessor.setExceptionHandler(exceptionHandler);
15. }
16. consumerRepository.add(batchEventProcessor, eventHandler, barrier);
17. processorSequences[i] = batchEventProcessor.getSequence();
18. }
19. **if** (processorSequences.length > 0){
20. consumerRepository.unMarkEventProcessorsAsEndOfChain(barrierSequences);
21. }
22. **return** **new** EventHandlerGroup<T>(**this**, consumerRepository, processorSequences);
23. }

       可见，handleEventsWith方法内部会创建BatchEventProcessor。

       当然，对于Event模式，还有一些玩法，其实之前几篇就看到过，我们可以设置两个EventHandler，然后事件会依次被这两个handler处理。Disruptor类中提供了更明确的定义(事实是结合了EventHandlerGroup的一些方法)，比如我想让事件先被处理器a处理，然后在被处理器b处理，就可以这么写：

**Java代码  [藏代码](javascript:void())**

1. EventHandler<MyEvent> a = **new** EventHandler<MyEvent>() { ... };
2. EventHandler<MyEvent> b = **new** EventHandler<MyEvent>() { ... };
3. disruptor.handleEventsWith(a); //语句1
4. disruptor.after(a).handleEventsWith(b);

       注意上面必须先写语句1，然后才能针对a调用after，否则after找不到处理器a，会报错。

       上面的例子也可以这么写：

**Java代码  [藏代码](javascript:void())**

1. EventHandler<MyEvent> a = **new** EventHandler<MyEvent>() { ... };
2. EventHandler<MyEvent> b = **new** EventHandler<MyEvent>() { ... };
3. disruptor.handleEventsWith(a).then(b);

       效果是一样的。

       Disruptor还允许我们定制事件处理者：

**Java代码  [藏代码](javascript:void())**

1. **public** EventHandlerGroup<T> handleEventsWith(**final** EventProcessorFactory<T>... eventProcessorFactories){
2. **final** Sequence[] barrierSequences = **new** Sequence[0];
3. **return** createEventProcessors(barrierSequences, eventProcessorFactories);
4. }
5. **public** **interface** EventProcessorFactory<T>{
6. EventProcessor createEventProcessor(RingBuffer<T> ringBuffer, Sequence[] barrierSequences);
7. }

       handleEventsWith方法内部创建的Event模式的事件处理者，有没有Work模式的呢？

**Java代码  [藏代码](javascript:void())**

1. @SuppressWarnings("varargs")
2. **public** EventHandlerGroup<T> handleEventsWithWorkerPool(**final** WorkHandler<T>... workHandlers){
3. **return** createWorkerPool(**new** Sequence[0], workHandlers);
4. }
5. EventHandlerGroup<T> createWorkerPool(**final** Sequence[] barrierSequences, **final** WorkHandler<? **super** T>[] workHandlers){
6. **final** SequenceBarrier sequenceBarrier = ringBuffer.newBarrier(barrierSequences);
7. **final** WorkerPool<T> workerPool = **new** WorkerPool<T>(ringBuffer, sequenceBarrier, exceptionHandler, workHandlers);
8. consumerRepository.add(workerPool, sequenceBarrier);
9. **return** **new** EventHandlerGroup<T>(**this**, consumerRepository, workerPool.getWorkerSequences());
10. }

        handleEventsWithWorkerPool内部会创建WorkerPool。

**事件处理者也构建好了，接下来看看怎么启动它们：**

**Java代码  [藏代码](javascript:void())**

1. **public** RingBuffer<T> start(){
2. **final** Sequence[] gatingSequences = consumerRepository.getLastSequenceInChain(**true**);
3. ringBuffer.addGatingSequences(gatingSequences);
4. checkOnlyStartedOnce();
5. **for** (**final** ConsumerInfo consumerInfo : consumerRepository){
6. consumerInfo.start(executor);
7. }
8. **return** ringBuffer;
9. }

        可见，启动过程中会将事件处理者的序列设置为RingBuffer的追踪序列。最后会启动事件处理者，并利用执行器来执行事件处理线程。

**事件处理者构建好了，也启动了，看看怎么发布事件：**

**Java代码  [藏代码](javascript:void())**

1. **public** **void** publishEvent(**final** EventTranslator<T> eventTranslator){
2. ringBuffer.publishEvent(eventTranslator);
3. }

        很简单，里面就是直接调用了RingBuffer来发布事件，之前几篇都分析过了。

**再看看其他的方法：**

**Java代码  [藏代码](javascript:void())**

1. **public** **void** halt(){
2. **for** (**final** ConsumerInfo consumerInfo : consumerRepository){
3. consumerInfo.halt();
4. }
5. }

        停止事件处理者。

**Java代码  [藏代码](javascript:void())**

1. **public** **void** shutdown(){
2. **try**{
3. shutdown(-1, TimeUnit.MILLISECONDS);
4. }**catch** (**final** TimeoutException e){
5. exceptionHandler.handleOnShutdownException(e);
6. }
7. }
8. **public** **void** shutdown(**final** **long** timeout, **final** TimeUnit timeUnit) **throws** TimeoutException{
9. **final** **long** timeOutAt = System.currentTimeMillis() + timeUnit.toMillis(timeout);
10. **while** (hasBacklog()){
11. **if** (timeout >= 0 && System.currentTimeMillis() > timeOutAt){
12. **throw** TimeoutException.INSTANCE;
13. }
14. // Busy spin
15. }
16. halt();
17. }
18. **private** **boolean** hasBacklog(){
19. **final** **long** cursor = ringBuffer.getCursor();
20. **for** (**final** Sequence consumer : consumerRepository.getLastSequenceInChain(**false**)){
21. **if** (cursor > consumer.get()){
22. **return** **true**;
23. }
24. }
25. **return** **false**;
26. }

        等待所有能处理的事件都处理完了，再定制事件处理者，有超时选项。

**好了，其他方法都比较简单，不看了。最后来使用Disruptor写一个生产者消费者模式吧：**

**Java代码  [藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. //创建一个执行器(线程池)。
3. Executor executor = Executors.newFixedThreadPool(4);
4. //创建一个Disruptor。
5. Disruptor<MyDataEvent> disruptor =
6. **new** Disruptor<MyDataEvent>(**new** MyDataEventFactory(), 4, executor);
7. //创建两个事件处理器。
8. MyDataEventHandler handler1 = **new** MyDataEventHandler();
9. KickAssEventHandler handler2 = **new** KickAssEventHandler();
10. //同一个事件，先用handler1处理再用handler2处理。
11. disruptor.handleEventsWith(handler1).then(handler2);
12. //启动Disruptor。
13. disruptor.start();
14. //发布10个事件。
15. **for**(**int** i=0;i<10;i++){
16. disruptor.publishEvent(**new** MyDataEventTranslator());
17. System.out.println("发布事件["+i+"]");
18. **try** {
19. TimeUnit.SECONDS.sleep(3);
20. } **catch** (InterruptedException e) {
21. e.printStackTrace();
22. }
23. }
24. **try** {
25. System.in.read();
26. } **catch** (IOException e) {
27. e.printStackTrace();
28. }
29. }

        看下输出：

**Java代码  [藏代码](javascript:void())**

1. 发布事件[0]
2. handle event's data:MyData [id=0, value=holy shit!]isEndOfBatch:**true**
3. kick your ass 0 times!!!!
4. 发布事件[1]
5. handle event's data:MyData [id=1, value=holy shit!]isEndOfBatch:**true**
6. kick your ass 1 times!!!!
7. 发布事件[2]
8. handle event's data:MyData [id=2, value=holy shit!]isEndOfBatch:**true**
9. kick your ass 2 times!!!!
10. 发布事件[3]
11. handle event's data:MyData [id=3, value=holy shit!]isEndOfBatch:**true**
12. kick your ass 3 times!!!!
13. 发布事件[4]
14. handle event's data:MyData [id=4, value=holy shit!]isEndOfBatch:**true**
15. kick your ass 4 times!!!!
16. 发布事件[5]
17. handle event's data:MyData [id=5, value=holy shit!]isEndOfBatch:**true**
18. kick your ass 5 times!!!!
19. 发布事件[6]
20. handle event's data:MyData [id=6, value=holy shit!]isEndOfBatch:**true**
21. kick your ass 6 times!!!!
22. 发布事件[7]
23. handle event's data:MyData [id=7, value=holy shit!]isEndOfBatch:**true**
24. kick your ass 7 times!!!!
25. 发布事件[8]
26. handle event's data:MyData [id=8, value=holy shit!]isEndOfBatch:**true**
27. kick your ass 8 times!!!!
28. 发布事件[9]
29. handle event's data:MyData [id=9, value=holy shit!]isEndOfBatch:**true**
30. kick your ass 9 times!!!!

* **最后总结：**

**1.使用时可以直接使用Disruptor这个类来更方便的完成代码编写，注意灵活使用。**

**2.最后别忘了单线程/多线程生产者、Event/Work处理模式等等。**

# CPU高速缓存行

CPU 为了更快的执行代码。于是当从内存中读取数据时，并不是只读自己想要的部分。而是读取足够的字节来填入高速缓存行。根据不同的 CPU ，高速缓存行大小不同。如 X86 是 32BYTES ，而 ALPHA 是 64BYTES 。并且始终在第 32 个字节或第 64 个字节处对齐。这样，当 CPU 访问相邻的数据时，就不必每次都从内存中读取，提高了速度。 因为访问内存要比访问高速缓存用的时间多得多。

但是，多核发达的年代。情况就不能那么简单了。试想下面这样一个情况。

1、 CPU1 读取了一个字节，以及它和它相邻的字节被读入 CPU1 的高速缓存。

2、 CPU2 做了上面同样的工作。这样 CPU1 ， CPU2 的高速缓存拥有同样的数据。

3、 CPU1 修改了那个字节，被修改后，那个字节被放回 CPU1 的高速缓存行。但是该信息并没有被写入 RAM 。

4、  CPU2 访问该字节，但由于 CPU1 并未将数据写入 RAM ，导致了数据不同步。

为了解决这个问题，芯片设计者制定了一个规则。当一个 CPU 修改高速缓存行中的字节时，计算机中的其它 CPU 会被通知，它们的高速缓存将视为无效。于是，在上面的情况下， CPU2 发现自己的高速缓存中数据已无效， CPU1 将立即把自己的数据写回 RAM ，然后 CPU2 重新读取该数据。 可以看出，高速缓存行在多处理器上会导致一些不利。

从上面的情况可以看出，在设计数据结构的时候，应该尽量将只读数据与读写数据分开，并具尽量将同一时间访问的数据组合在一起。这样 CPU 能一次将需要的数据读入。

如：

Struct \_\_a

{

   Int id; // 不易变

   Int factor;// 易变

   Char name[64];// 不易变

   Int value;// 易变

} ；

这样的数据结构就很不利。

在 X86 下，可以试着修改和调整它

Struct \_\_a

{

   Int id; // 不易变

Char name[64];// 不易变

Char \_\_Align[32 – sizeof(int)+sizeof(name)\*sizeof(name[0])%32]

   Int factor;// 易变

   Int value;// 易变

Char \_\_Align2[32 –2\* sizeof(int)%32]

} ；

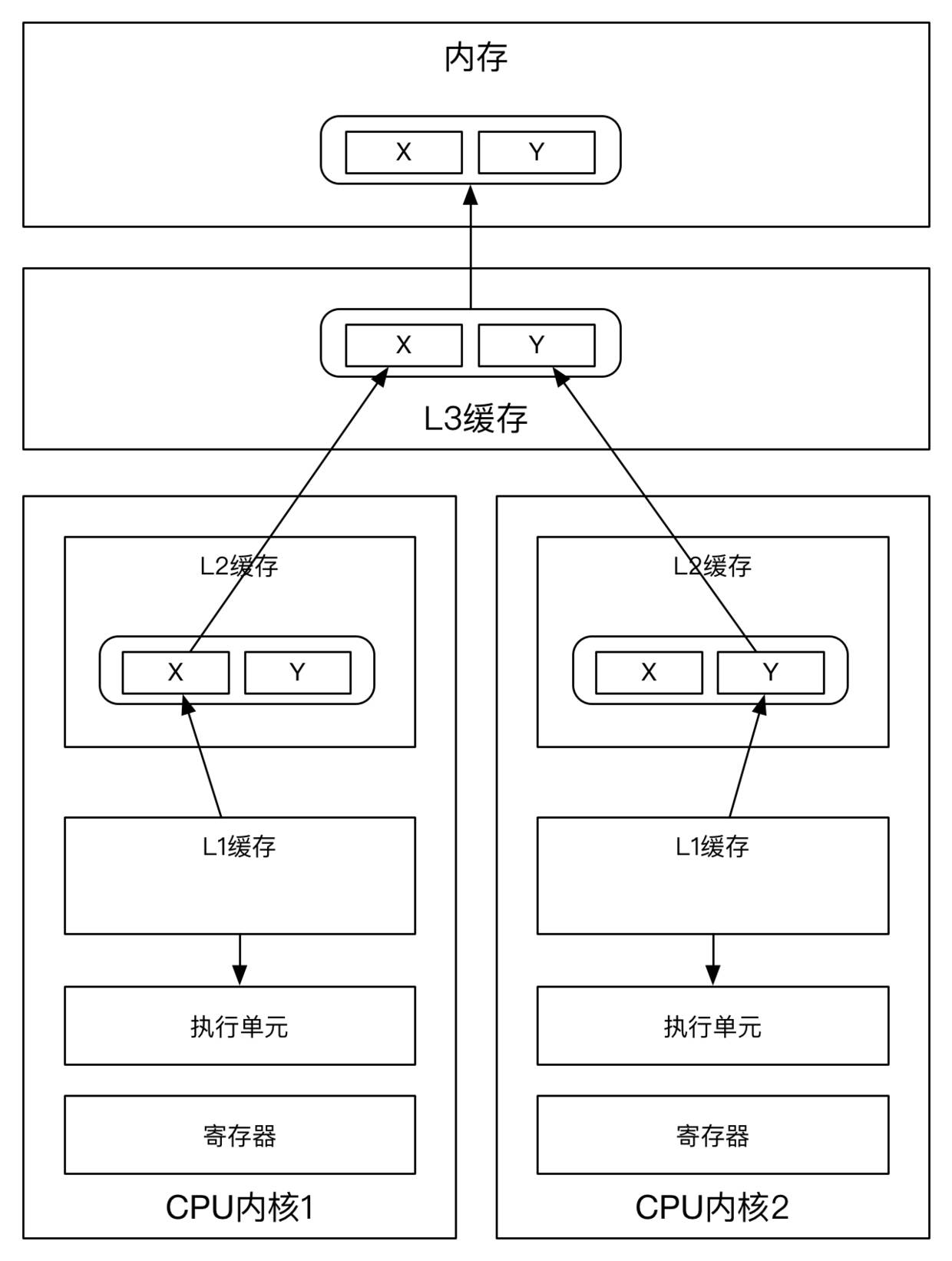
32 – sizeof(int)+sizeof(name)\*sizeof(name[0])%32

看起来很不和谐， 32 表示 X86 中，高速缓存行为 32BYTES 大小。 \_\_Align 用于显式对齐。

# 伪共享和缓存行

在计算机系统中，内存是以缓存行为单位存储的，一个缓存行存储字节的数量为2的倍数，在不同的机器上，缓存行大小为32字节到256字节不等，通常来说为64字节。伪共享指的是在多个线程同时读写同一个缓存行的不同变量的时候，尽管这些变量之间没有任何关系，但是在多个线程之间仍然需要同步，从而导致性能下降的情况。在对称多处理器结构的系统中，伪共享是影响性能的主要因素之一，由于很难通过走查代码的方式定位伪共享的问题，因此，大家把伪共享称为“性能杀手”。

为了通过增加线程数来达到计算能力的水平扩展，我们必须确保多个线程不能同时对一个变量或者缓存行进行读写。我们可以通过代码走查的方式，定位多个线程读写一个变量的情况，但是，要想知道多个线程读写同一个缓存行的情况，我们必须先了解系统内存的组织形式，如下图所示。



系统的缓存结构

# [Java并发编程：volatile关键字解析](https://www.cnblogs.com/dolphin0520/p/3920373.html)

## 5.1 内存模型的相关概念

大家都知道，计算机在执行程序时，每条指令都是在CPU中执行的，而执行指令过程中，势必涉及到数据的读取和写入。由于程序运行过程中的临时数据是存放在主存（物理内存）当中的，这时就存在一个问题，由于CPU执行速度很快，而从内存读取数据和向内存写入数据的过程跟CPU执行指令的速度比起来要慢的多，因此如果任何时候对数据的操作都要通过和内存的交互来进行，会大大降低指令执行的速度。因此在CPU里面就有了高速缓存。

　　也就是，当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。举个简单的例子，比如下面的这段代码：

|  |  |
| --- | --- |
| 1 | i = i + 1; |

 　　当线程执行这个语句时，会先从主存当中读取i的值，然后复制一份到高速缓存当中，然后CPU执行指令对i进行加1操作，然后将数据写入高速缓存，最后将高速缓存中i最新的值刷新到主存当中。

　　这个代码在单线程中运行是没有任何问题的，但是在多线程中运行就会有问题了。在多核CPU中，每条线程可能运行于不同的CPU中，因此每个线程运行时有自己的高速缓存（对单核CPU来说，其实也会出现这种问题，只不过是以线程调度的形式来分别执行的）。本文我们以多核CPU为例。

　　比如同时有2个线程执行这段代码，假如初始时i的值为0，那么我们希望两个线程执行完之后i的值变为2。但是事实会是这样吗？

　　可能存在下面一种情况：初始时，两个线程分别读取i的值存入各自所在的CPU的高速缓存当中，然后线程1进行加1操作，然后把i的最新值1写入到内存。此时线程2的高速缓存当中i的值还是0，进行加1操作之后，i的值为1，然后线程2把i的值写入内存。

　　最终结果i的值是1，而不是2。这就是著名的缓存一致性问题。通常称这种被多个线程访问的变量为共享变量。

　　也就是说，如果一个变量在多个CPU中都存在缓存（一般在多线程编程时才会出现），那么就可能存在缓存不一致的问题。

　　为了解决缓存不一致性问题，通常来说有以下2种解决方法：

　　1）通过在总线加LOCK#锁的方式

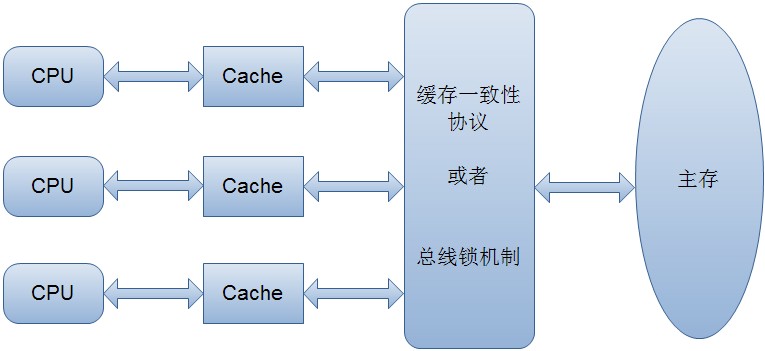
　　2）通过缓存一致性协议

　　这2种方式都是硬件层面上提供的方式。

　　在早期的CPU当中，是通过在总线上加LOCK#锁的形式来解决缓存不一致的问题。因为CPU和其他部件进行通信都是通过总线来进行的，如果对总线加LOCK#锁的话，也就是说阻塞了其他CPU对其他部件访问（如内存），从而使得只能有一个CPU能使用这个变量的内存。比如上面例子中 如果一个线程在执行 i = i +1，如果在执行这段代码的过程中，在总线上发出了LCOK#锁的信号，那么只有等待这段代码完全执行完毕之后，其他CPU才能从变量i所在的内存读取变量，然后进行相应的操作。这样就解决了缓存不一致的问题。

　　但是上面的方式会有一个问题，由于在锁住总线期间，其他CPU无法访问内存，导致效率低下。

　　所以就出现了缓存一致性协议。最出名的就是Intel 的MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。



## 5.2 并发编程中的三个概念

在并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题。我们先看具体看一下这三个概念：

**1.原子性**

　　原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

　　一个很经典的例子就是银行账户转账问题：

　　比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。

　　试想一下，如果这2个操作不具备原子性，会造成什么样的后果。假如从账户A减去1000元之后，操作突然中止。然后又从B取出了500元，取出500元之后，再执行 往账户B加上1000元 的操作。这样就会导致账户A虽然减去了1000元，但是账户B没有收到这个转过来的1000元。

　　所以这2个操作必须要具备原子性才能保证不出现一些意外的问题。

　　同样地反映到并发编程中会出现什么结果呢？

　　举个最简单的例子，大家想一下假如为一个32位的变量赋值过程不具备原子性的话，会发生什么后果？

|  |  |
| --- | --- |
| 1 | i = 9; |

 　　假若一个线程执行到这个语句时，我暂且假设为一个32位的变量赋值包括两个过程：为低16位赋值，为高16位赋值。

　　那么就可能发生一种情况：当将低16位数值写入之后，突然被中断，而此时又有一个线程去读取i的值，那么读取到的就是错误的数据。

**2.可见性**

　　可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

　　举个简单的例子，看下面这段代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //线程1执行的代码  int i = 0;  i = 10;    //线程2执行的代码  j = i; |

 　　假若执行线程1的是CPU1，执行线程2的是CPU2。由上面的分析可知，当线程1执行 i =10这句时，会先把i的初始值加载到CPU1的高速缓存中，然后赋值为10，那么在CPU1的高速缓存当中i的值变为10了，却没有立即写入到主存当中。

　　此时线程2执行 j = i，它会先去主存读取i的值并加载到CPU2的缓存当中，注意此时内存当中i的值还是0，那么就会使得j的值为0，而不是10.

　　这就是可见性问题，线程1对变量i修改了之后，线程2没有立即看到线程1修改的值。

**3.有序性**

　　有序性：即程序执行的顺序按照代码的先后顺序执行。举个简单的例子，看下面这段代码：

|  |  |
| --- | --- |
| 1  2  3  4 | int i = 0;  boolean flag = false;  i = 1;                //语句1  flag = true;          //语句2 |

 　　上面代码定义了一个int型变量，定义了一个boolean类型变量，然后分别对两个变量进行赋值操作。从代码顺序上看，语句1是在语句2前面的，那么JVM在真正执行这段代码的时候会保证语句1一定会在语句2前面执行吗？不一定，为什么呢？这里可能会发生指令重排序（Instruction Reorder）。

　　下面解释一下什么是指令重排序，一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

　　比如上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。

　　但是要注意，虽然处理器会对指令进行重排序，但是它会保证程序最终结果会和代码顺序执行结果相同，那么它靠什么保证的呢？再看下面一个例子：

|  |  |
| --- | --- |
| 1  2  3  4 | int a = 10;    //语句1  int r = 2;    //语句2  a = a + 3;    //语句3  r = a\*a;     //语句4 |

 　　这段代码有4个语句，那么可能的一个执行顺序是：

https://images0.cnblogs.com/blog/288799/201408/212305263939989.jpg

　　那么可不可能是这个执行顺序呢： 语句2   语句1    语句4   语句3

　　不可能，因为处理器在进行重排序时是会考虑指令之间的数据依赖性，如果一个指令Instruction 2必须用到Instruction 1的结果，那么处理器会保证Instruction 1会在Instruction 2之前执行。

　　虽然重排序不会影响单个线程内程序执行的结果，但是多线程呢？下面看一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //线程1:  context = loadContext();   //语句1  inited = true;             //语句2    //线程2:  while(!inited ){    sleep()  }  doSomethingwithconfig(context); |

 　　上面代码中，由于语句1和语句2没有数据依赖性，因此可能会被重排序。假如发生了重排序，在线程1执行过程中先执行语句2，而此是线程2会以为初始化工作已经完成，那么就会跳出while循环，去执行doSomethingwithconfig(context)方法，而此时context并没有被初始化，就会导致程序出错。

 　　从上面可以看出，指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。

　　也就是说，要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。

## 5.3 Java内存模型

在前面谈到了一些关于内存模型以及并发编程中可能会出现的一些问题。下面我们来看一下Java内存模型，研究一下Java内存模型为我们提供了哪些保证以及在java中提供了哪些方法和机制来让我们在进行多线程编程时能够保证程序执行的正确性。

　　在Java虚拟机规范中试图定义一种Java内存模型（Java Memory Model，JMM）来屏蔽各个硬件平台和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。那么Java内存模型规定了哪些东西呢，它定义了程序中变量的访问规则，往大一点说是定义了程序执行的次序。注意，为了获得较好的执行性能，Java内存模型并没有限制执行引擎使用处理器的寄存器或者高速缓存来提升指令执行速度，也没有限制编译器对指令进行重排序。也就是说，在java内存模型中，也会存在缓存一致性问题和指令重排序的问题。

　　Java内存模型规定所有的变量都是存在主存当中（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程不能访问其他线程的工作内存。

　　举个简单的例子：在java中，执行下面这个语句：

|  |  |
| --- | --- |
| 1 | i  = 10; |

 　　执行线程必须先在自己的工作线程中对变量i所在的缓存行进行赋值操作，然后再写入主存当中。而不是直接将数值10写入主存当中。

　　那么Java语言 本身对 原子性、可见性以及有序性提供了哪些保证呢？

**1.原子性**

　　在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

　　上面一句话虽然看起来简单，但是理解起来并不是那么容易。看下面一个例子i：

　　请分析以下哪些操作是原子性操作：

|  |  |
| --- | --- |
| 1  2  3  4 | x = 10;         //语句1  y = x;         //语句2  x++;           //语句3  x = x + 1;     //语句4 |

 　　咋一看，有些朋友可能会说上面的4个语句中的操作都是原子性操作。其实只有语句1是原子性操作，其他三个语句都不是原子性操作。

　　语句1是直接将数值10赋值给x，也就是说线程执行这个语句的会直接将数值10写入到工作内存中。

　　语句2实际上包含2个操作，它先要去读取x的值，再将x的值写入工作内存，虽然读取x的值以及 将x的值写入工作内存 这2个操作都是原子性操作，但是合起来就不是原子性操作了。

　　同样的，x++和 x = x+1包括3个操作：读取x的值，进行加1操作，写入新的值。

 　　所以上面4个语句只有语句1的操作具备原子性。

　　也就是说，只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作。

　　不过这里有一点需要注意：在32位平台下，对64位数据的读取和赋值是需要通过两个操作来完成的，不能保证其原子性。但是好像在最新的JDK中，JVM已经保证对64位数据的读取和赋值也是原子性操作了。

　　从上面可以看出，Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性。

**2.可见性**

　　对于可见性，Java提供了volatile关键字来保证可见性。

　　当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

　　而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

　　另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

3.有序性

　　在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

　　在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

　　另外，Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 happens-before 原则。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

　　下面就来具体介绍下happens-before原则（先行发生原则）：

程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作

锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作

volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作

传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作

线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

　　这8条原则摘自《深入理解Java虚拟机》。

　　这8条规则中，前4条规则是比较重要的，后4条规则都是显而易见的。

　　下面我们来解释一下前4条规则：

　　对于程序次序规则来说，我的理解就是一段程序代码的执行在单个线程中看起来是有序的。注意，虽然这条规则中提到“书写在前面的操作先行发生于书写在后面的操作”，这个应该是程序看起来执行的顺序是按照代码顺序执行的，因为虚拟机可能会对程序代码进行指令重排序。虽然进行重排序，但是最终执行的结果是与程序顺序执行的结果一致的，它只会对不存在数据依赖性的指令进行重排序。因此，在单个线程中，程序执行看起来是有序执行的，这一点要注意理解。事实上，这个规则是用来保证程序在单线程中执行结果的正确性，但无法保证程序在多线程中执行的正确性。

　　第二条规则也比较容易理解，也就是说无论在单线程中还是多线程中，同一个锁如果出于被锁定的状态，那么必须先对锁进行了释放操作，后面才能继续进行lock操作。

　　第三条规则是一条比较重要的规则，也是后文将要重点讲述的内容。直观地解释就是，如果一个线程先去写一个变量，然后一个线程去进行读取，那么写入操作肯定会先行发生于读操作。

　　第四条规则实际上就是体现happens-before原则具备传递性。

## 5.4 深入剖析volatile关键字

在前面讲述了很多东西，其实都是为讲述volatile关键字作铺垫，那么接下来我们就进入主题。

**1.volatile关键字的两层语义**

　　一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

　　1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

　　2）禁止进行指令重排序。

　　先看一段代码，假如线程1先执行，线程2后执行：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //线程1  boolean stop = false;  while(!stop){      doSomething();  }    //线程2  stop = true; |

 　　这段代码是很典型的一段代码，很多人在中断线程时可能都会采用这种标记办法。但是事实上，这段代码会完全运行正确么？即一定会将线程中断么？不一定，也许在大多数时候，这个代码能够把线程中断，但是也有可能会导致无法中断线程（虽然这个可能性很小，但是只要一旦发生这种情况就会造成死循环了）。

　　下面解释一下这段代码为何有可能导致无法中断线程。在前面已经解释过，每个线程在运行过程中都有自己的工作内存，那么线程1在运行的时候，会将stop变量的值拷贝一份放在自己的工作内存当中。

　　那么当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

　　但是用volatile修饰之后就变得不一样了：

　　第一：使用volatile关键字会强制将修改的值立即写入主存；

　　第二：使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

　　第三：由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

　　那么在线程2修改stop值时（当然这里包括2个操作，修改线程2工作内存中的值，然后将修改后的值写入内存），会使得线程1的工作内存中缓存变量stop的缓存行无效，然后线程1读取时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

　　那么线程1读取到的就是最新的正确的值。

**2.volatile保证原子性吗？**

　　从上面知道volatile关键字保证了操作的可见性，但是volatile能保证对变量的操作是原子性吗？

　　下面看一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class Test {      public volatile int inc = 0;        public void increase() {          inc++;      }        public static void main(String[] args) {          final Test test = new Test();          for(int i=0;i<10;i++){              new Thread(){                  public void run() {                      for(int j=0;j<1000;j++)                          test.increase();                  };              }.start();          }            while(Thread.activeCount()>1)  //保证前面的线程都执行完              Thread.yield();          System.out.println(test.inc);      }  } |

 　　大家想一下这段程序的输出结果是多少？也许有些朋友认为是10000。但是事实上运行它会发现每次运行结果都不一致，都是一个小于10000的数字。

　　可能有的朋友就会有疑问，不对啊，上面是对变量inc进行自增操作，由于volatile保证了可见性，那么在每个线程中对inc自增完之后，在其他线程中都能看到修改后的值啊，所以有10个线程分别进行了1000次操作，那么最终inc的值应该是1000\*10=10000。

　　这里面就有一个误区了，volatile关键字能保证可见性没有错，但是上面的程序错在没能保证原子性。可见性只能保证每次读取的是最新的值，但是volatile没办法保证对变量的操作的原子性。

　　在前面已经提到过，自增操作是不具备原子性的，它包括读取变量的原始值、进行加1操作、写入工作内存。那么就是说自增操作的三个子操作可能会分割开执行，就有可能导致下面这种情况出现：

　　假如某个时刻变量inc的值为10，

　　线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；

　　然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的缓存行无效，所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。

　　然后线程1接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。

　　那么两个线程分别进行了一次自增操作后，inc只增加了1。

　　解释到这里，可能有朋友会有疑问，不对啊，前面不是保证一个变量在修改volatile变量时，会让缓存行无效吗？然后其他线程去读就会读到新的值，对，这个没错。这个就是上面的happens-before规则中的volatile变量规则，但是要注意，线程1对变量进行读取操作之后，被阻塞了的话，并没有对inc值进行修改。然后虽然volatile能保证线程2对变量inc的值读取是从内存中读取的，但是线程1没有进行修改，所以线程2根本就不会看到修改的值。

　　根源就在这里，自增操作不是原子性操作，而且volatile也无法保证对变量的任何操作都是原子性的。

　　把上面的代码改成以下任何一种都可以达到效果：

　　采用synchronized：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[制代码](javascript:void(0);)

public class Test {

public int inc = 0;

public synchronized void increase() {

inc++;

}

public static void main(String[] args) {

final Test test = new Test();

for(int i=0;i<10;i++){

new Thread(){

public void run() {

for(int j=0;j<1000;j++)

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

[制代码](javascript:void(0);)

　　采用Lock：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[制代码](javascript:void(0);)

public class Test {

public int inc = 0;

Lock lock = new ReentrantLock();

public void increase() {

lock.lock();

try {

inc++;

} finally{

lock.unlock();

}

}

public static void main(String[] args) {

final Test test = new Test();

for(int i=0;i<10;i++){

new Thread(){

public void run() {

for(int j=0;j<1000;j++)

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

[制代码](javascript:void(0);)

　　采用AtomicInteger：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[制代码](javascript:void(0);)

public class Test {

public AtomicInteger inc = new AtomicInteger();

public void increase() {

inc.getAndIncrement();

}

public static void main(String[] args) {

final Test test = new Test();

for(int i=0;i<10;i++){

new Thread(){

public void run() {

for(int j=0;j<1000;j++)

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

[制代码](javascript:void(0);)

　　在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 自增（加1操作），自减（减1操作）、以及加法操作（加一个数），减法操作（减一个数）进行了封装，保证这些操作是原子性操作。atomic是利用CAS来实现原子性操作的（Compare And Swap），CAS实际上是利用处理器提供的CMPXCHG指令实现的，而处理器执行CMPXCHG指令是一个原子性操作。

**3.volatile能保证有序性吗？**

　　在前面提到volatile关键字能禁止指令重排序，所以volatile能在一定程度上保证有序性。

　　volatile关键字禁止指令重排序有两层意思：

　　1）当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

　　2）在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

　　可能上面说的比较绕，举个简单的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //x、y为非volatile变量  //flag为volatile变量    x = 2;        //语句1  y = 0;        //语句2  flag = true;  //语句3  x = 4;         //语句4  y = -1;       //语句5 |

 　　由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

　　并且volatile关键字能保证，执行到语句3时，语句1和语句2必定是执行完毕了的，且语句1和语句2的执行结果对语句3、语句4、语句5是可见的。

　　那么我们回到前面举的一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //线程1:  context = loadContext();   //语句1  inited = true;             //语句2    //线程2:  while(!inited ){    sleep()  }  doSomethingwithconfig(context); |

 　　前面举这个例子的时候，提到有可能语句2会在语句1之前执行，那么久可能导致context还没被初始化，而线程2中就使用未初始化的context去进行操作，导致程序出错。

　　这里如果用volatile关键字对inited变量进行修饰，就不会出现这种问题了，因为当执行到语句2时，必定能保证context已经初始化完毕。

**4.volatile的原理和实现机制**

　　前面讲述了源于volatile关键字的一些使用，下面我们来探讨一下volatile到底如何保证可见性和禁止指令重排序的。

　　下面这段话摘自《深入理解Java虚拟机》：

　　“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”

　　lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

　　1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

　　2）它会强制将对缓存的修改操作立即写入主存；

　　3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

## 5.5 使用volatile关键字的场景

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

　　1）对变量的写操作不依赖于当前值

　　2）该变量没有包含在具有其他变量的不变式中

　　实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

　　事实上，我的理解就是上面的2个条件需要保证操作是原子性操作，才能保证使用volatile关键字的程序在并发时能够正确执行。

　　下面列举几个Java中使用volatile的几个场景。

**1.状态标记量**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | volatile boolean flag = false;    while(!flag){      doSomething();  }    public void setFlag() {      flag = true;  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | volatile boolean inited = false;  //线程1:  context = loadContext();  inited = true;    //线程2:  while(!inited ){  sleep()  }  doSomethingwithconfig(context); |

**2.double check**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Singleton{      private volatile static Singleton instance = null;        private Singleton() {        }        public static Singleton getInstance() {          if(instance==null) {              synchronized (Singleton.class) {                  if(instance==null)                      instance = new Singleton();              }          }          return instance;      }  } |

**至于为何需要这么写请参考：**

**《Java 中的双重检查（Double-Check）》**[**http://blog.csdn.net/dl88250/article/details/5439024**](http://blog.csdn.net/dl88250/article/details/5439024)

**和**[**http://www.iteye.com/topic/652440**](http://www.iteye.com/topic/652440)

**参考资料：**

**《Java编程思想》**

**《深入理解Java虚拟机》**

[**http://jiangzhengjun.iteye.com/blog/652532**](http://jiangzhengjun.iteye.com/blog/652532)

[**http://blog.sina.com.cn/s/blog\_7bee8dd50101fu8n.html**](http://blog.sina.com.cn/s/blog_7bee8dd50101fu8n.html)

[**http://ifeve.com/volatile/**](http://ifeve.com/volatile/)

[**http://blog.csdn.net/ccit0519/article/details/11241403**](http://blog.csdn.net/ccit0519/article/details/11241403)

[**http://blog.csdn.net/ns\_code/article/details/17101369**](http://blog.csdn.net/ns_code/article/details/17101369)

[**http://www.cnblogs.com/kevinwu/archive/2012/05/02/2479464.html**](http://www.cnblogs.com/kevinwu/archive/2012/05/02/2479464.html)

[**http://www.cppblog.com/elva/archive/2011/01/21/139019.html**](http://www.cppblog.com/elva/archive/2011/01/21/139019.html)

[**http://ifeve.com/volatile-array-visiblity/**](http://ifeve.com/volatile-array-visiblity/)

[**http://www.bdqn.cn/news/201312/12579.shtml**](http://www.bdqn.cn/news/201312/12579.shtml)

[**http://exploer.blog.51cto.com/7123589/1193399**](http://exploer.blog.51cto.com/7123589/1193399)

[**http://www.cnblogs.com/Mainz/p/3556430.html**](http://www.cnblogs.com/Mainz/p/3556430.html)

# Java 中的双重检查（Double-Check）

在 Effecitve Java 一书的第 48 条中提到了双重检查模式，并指出这种模式在 Java 中通常并不适用。该模式的结构如下所示：

**[java]** [view plain](https://blog.csdn.net/dl88250/article/details/5439024) [copy](https://blog.csdn.net/dl88250/article/details/5439024)

1. **public** Resource getResource() {
2. **if** (resource == **null**) {
3. **synchronized**(**this**){
4. **if** (resource==**null**) {
5. resource = **new** Resource();
6. }
7. }
8. }
9. **return** resource;
10. }

该模式是对下面的代码改进：

**[java]** [view plain](https://blog.csdn.net/dl88250/article/details/5439024) [copy](https://blog.csdn.net/dl88250/article/details/5439024)

1. **public** **synchronized** Resource getResource(){
2. **if** (resource == **null**){
3. resource = **new** Resource();
4. }
5. **return** resource;
6. }

这段代码的目的是对 resource 延迟初始化。但是每次访问的时候都需要同步。为了减少同步的开销，于是有了双重检查模式。

在 Java 中双重检查模式无效的原因是在不同步的情况下引用类型不是线程安全的。对于除了 long 和 double 的基本类型，双重检查模式是适用 的。比如下面这段代码就是正确的：

**[java]** [view plain](https://blog.csdn.net/dl88250/article/details/5439024) [copy](https://blog.csdn.net/dl88250/article/details/5439024)

1. **private** **int** count;
2. **public** **int** getCount(){
3. **if** (count == 0){
4. **synchronized**(**this**){
5. **if** (count == 0){
6. count = computeCount();  //一个耗时的计算
7. }
8. }
9. }
10. **return** count;
11. }

上面就是关于java中双重检查模式（double-check idiom）的一般结论。但是事情还没有结束，因为java的内存模式也在改进中。Doug Lea 在他的文章中写道：“根据最新的 JSR133 的 Java 内存模型，如果将引用类型声明为 volatile，双重检查模式就可以工作了”，参见 <http://gee.cs.oswego.edu/dl/cpj/updates.html> 。

所以以后要在 Java 中使用双重检查模式，可以使用下面的代码：

**[java]** [view plain](https://blog.csdn.net/dl88250/article/details/5439024) [copy](https://blog.csdn.net/dl88250/article/details/5439024)

1. **private** **volatile** Resource resource;
2. **public** Resource getResource(){
3. **if** (resource == **null**){
4. **synchronized**(**this**){
5. **if** (resource==**null**){
6. resource = **new** Resource();
7. }
8. }
9. }
10. **return** resource;
11. }

当然了，得是在遵循 JSR133 规范的 Java 中。

所以，double-check 在 J2SE 1.4 或早期版本在多线程或者 JVM 调优时由于 out-of-order writes，是不可用的。 这个问题在 J2SE 5.0 中已经被修复，可以使用 volatile 关键字来保证多线程下的单例。

**[java]** [view plain](https://blog.csdn.net/dl88250/article/details/5439024) [copy](https://blog.csdn.net/dl88250/article/details/5439024)

1. **public** **class** Singleton {
2. **private** **volatile** Singleton instance = **null**;
3. **public** Singleton getInstance() {
4. **if** (instance == **null**) {
5. **synchronized**(**this**) {
6. **if** (instance == **null**) {
7. instance = **new** Singleton();
8. }
9. }
10. }
11. **return** instance;
12. }
13. }

推荐方法 是Initialization on Demand Holder（IODH），

详见 <http://en.wikipedia.org/wiki/Initialization_on_demand_holder_idiom>

**[java]** [view plain](https://blog.csdn.net/dl88250/article/details/5439024) [copy](https://blog.csdn.net/dl88250/article/details/5439024)

1. **public** **class** Singleton {
2. **static** **class** SingletonHolder {
3. **static** Singleton instance = **new** Singleton();
4. }
6. **public** **static** Singleton getInstance(){
7. **return** SingletonHolder.instance;
8. }
9. }

# 单例模式之Initialization Demand Holder (IoDH)

**一种更好的单例实现方法**

       饿汉式单例类不能实现延迟加载，不管将来用不用始终占据内存；懒汉式单例类线程安全控制烦琐，而且性能受影响。可见，无论是饿汉式单例还是懒汉式单例都存在这样那样的问题，有没有一种方法，能够将两种单例的缺点都克服，而将两者的优点合二为一呢？答案是：Yes！下面我们来学习这种更好的被称之为**Initialization Demand Holder (IoDH)**的技术。

      在IoDH中，我们在单例类中增加一个**静态(static)内部类**，在该内部类中创建单例对象，再将该单例对象通过getInstance()方法返回给外部使用，实现代码如下所示：

**[java]** [view plain](http://blog.csdn.net/lovelion/article/details/7420888) [copy](http://blog.csdn.net/lovelion/article/details/7420888)

1. //Initialization on Demand Holder
2. **class** Singleton {
3. **private** Singleton() {
4. }
6. **private** **static** **class** HolderClass {
7. **private** **final** **static** Singleton instance = **new** Singleton();
8. }
10. **public** **static** Singleton getInstance() {
11. **return** HolderClass.instance;
12. }
14. **public** **static** **void** main(String args[]) {
15. Singleton s1, s2;
16. s1 = Singleton.getInstance();
17. s2 = Singleton.getInstance();
18. System.out.println(s1==s2);
19. }
20. }

       编译并运行上述代码，运行结果为：true，即创建的单例对象s1和s2为同一对象。由于静态单例对象没有作为Singleton的成员变量直接实例化，因此类加载时不会实例化Singleton，第一次调用getInstance()时将加载内部类HolderClass，在该内部类中定义了一个static类型的变量instance，此时会首先初始化这个成员变量，由[Java](http://lib.csdn.net/base/javaee)虚拟机来保证其线程安全性，确保该成员变量只能初始化一次。由于getInstance()方法没有任何线程锁定，因此其性能不会造成任何影响。

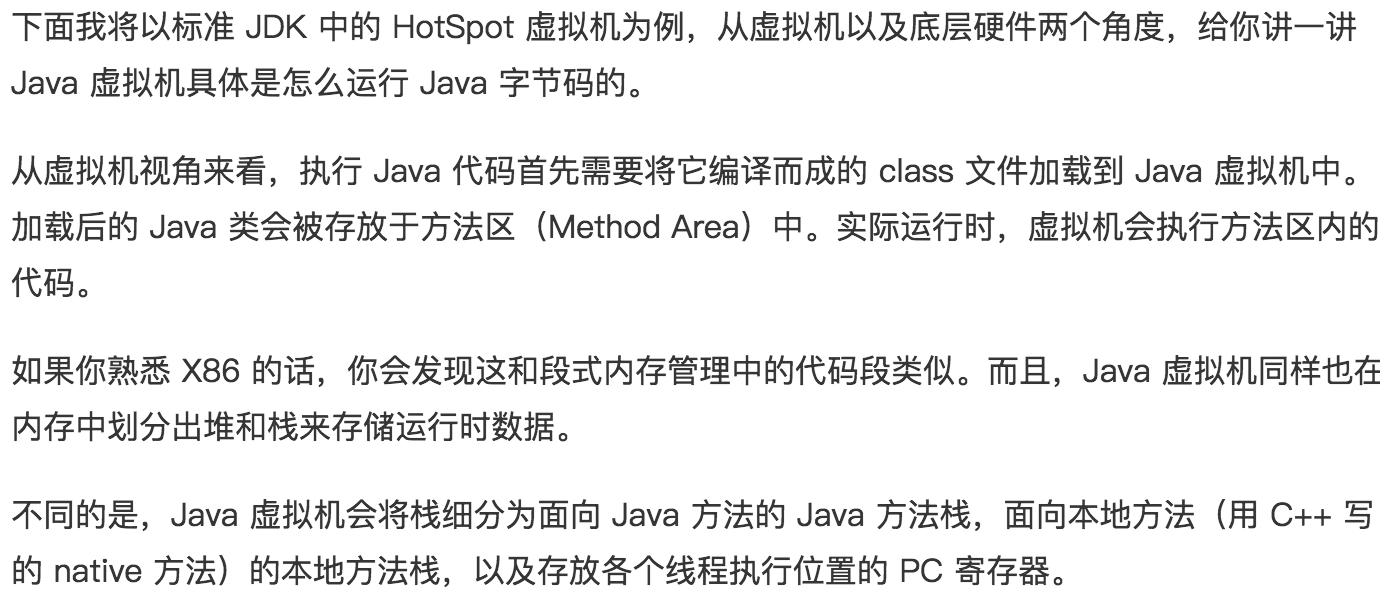
**通过使用IoDH，我们既可以实现延迟加载，又可以保证线程安全，不影响系统性能，不失为一种最好的Java语言单例模式实现方式**（其缺点是与编程语言本身的特性相关，很多面向对象语言不支持IoDH）。

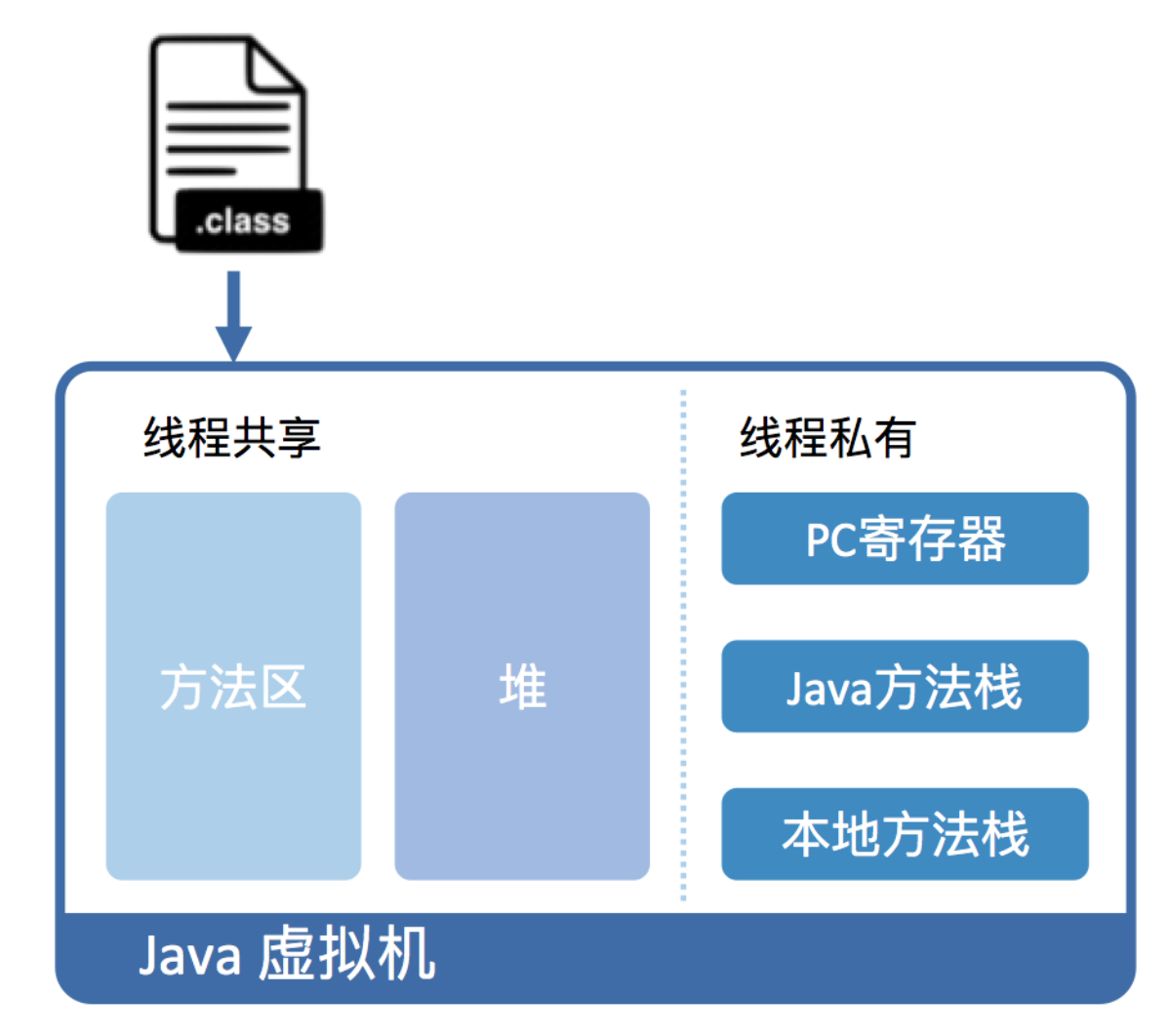
|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **练习**  分别使用饿汉式单例、带双重检查锁定机制的懒汉式单例以及IoDH技术实现负载均衡器LoadBalancer。 | |

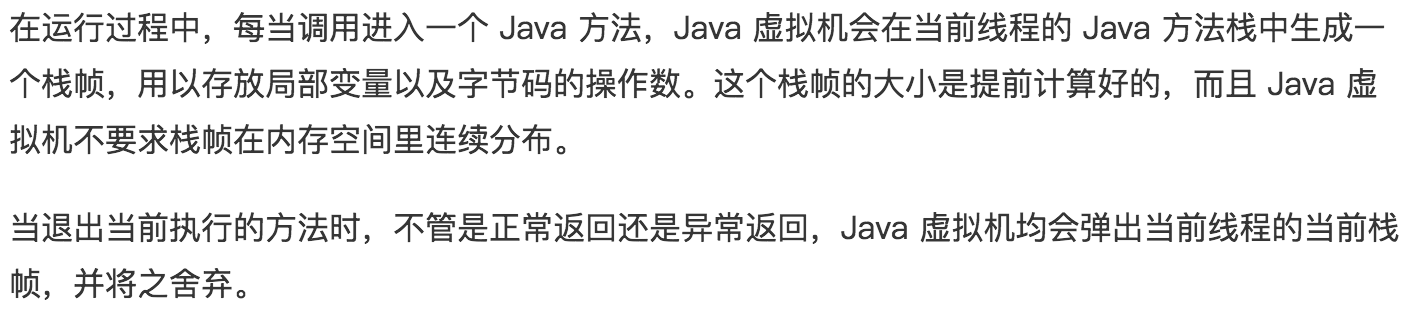
https://blog.csdn.net/guanyue520/article/details/52703964

# 深入拆解Java虚拟机

## Java 虚拟机具体是怎样运行 Java 字节码的？







# Java面试题

## http 默认端口，https 默认端口

http默认端口80，https默认端口443

## DNS的作用

DNS是域名系统（Domain Name System）的缩写，是因特网的一项核心服务，它作为可以将域名和IP地址相互映射的一个分布式数据库，能够使人更方便的访问互联网，而不用去记住能够被机器直接读取的IP数串。

今天我们以白话文的方式给大家更形象的讲解DNS的作用以及为什么一定要设置DNS才能上网？

DNS的作用：在互联网中，其实没有类似于[www.xxx.com](http://www.xxx.com/)这种域名方式，而替代的是以IP地址，如222.222.222.222，那我们在IE地址栏中应当输入222.222.222.222才能打开网站www.xxx.com，但我们细想一下，互联网上的网站成千上万，如果每个网站登陆都需要记住一大串数字，那是不是特别不方便，对于记忆力不强的人，根本无法记住这么烦琐的数字。这个时候DNS就出现了，它的作用就是将222.222.222.222解析为www.xxx.com，那么我们登陆的时候就直接输入域名就可以了。

为什么一定要设置DNS才能上网？有些朋友可能会发现，为什么我可能登陆QQ、MSN，但却打不开网页呢？其实大部分原因都是因为DNS服务器故障造成的，DNS服务器地址是唯一的，是运营商提供给终端用户用来解析IP地址及域名的关系，而如果不设定DNS服务器地址，那么就无法查询地址的去向，自然也就打不开网页，而QQ、MSN等即时聊天软件，采用的是UDP传输协议，即不可靠传输协议，无需提供DNS服务器地址，也同样可以登陆。

## 你们开发用的 ide 是啥？你能说下 idea 的常用几个快捷键吧？

Eclipse；https://www.cnblogs.com/mq0036/p/4995390.html

## 代码版本管理你们用的是啥？

Git

## git rebase 和 merge 有什么区别？

https://blog.csdn.net/liuxiaoheng1992/article/details/79108233

## 微服务的边界 (粒度) 是 "决策", 而不是个 "标准答案"

微服务的边界 (粒度) 是 "决策",而不是个 "标准答案"。

许多人面对微服务时，往往都会纠结着一个问题：微服务太小？太大？

其实，会纠结在这个问题上，最根本的原因便是误解了微服务粒度划分这件事的本质；微服务划分本身是 "架构设计"。也就是说微服务划分本身绝不是一个只讲"太大"或 "太小"标准答案的 "是非题"。而是需综合考量以下的因素，所作出的一个 "架构决策"：

**1. 市场业务的扩展性**

**2. 与已有架构间的冲突**

**3. 开发团队在开发上所可能面临的风险**

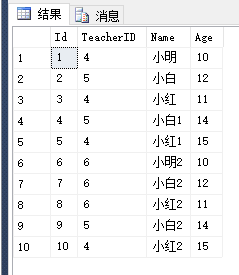
**4. 测试人员测试执行的效率**

所以，请不要再简单粗暴的便脱口而出：你的微服务划得太细、太小...

而是应该将各微服务划分的方式，深度思考，周全的考量各方面的因素下，所作出的一个 ”最适合” 的架构决策，而不是一个人芸亦芸的 ”标准答案”。

## sql having 的使用场景

现在 Student表有 如下数据



现需求如下： 查找每个老师的学生的平均年龄且平均年齿大于12

不使用 HAVING

SELECT \* FROM (SELECT TeacherID, AVG(Age) AS AGE FROM Student

GROUP BY TeacherID) T WHERE T.AGE > 12

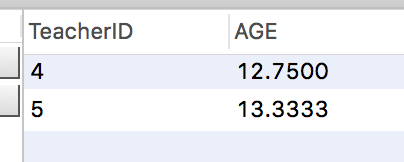
使用HAVING

SELECT TeacherID, AVG(Age) AS AGE FROM Student

GROUP BY TeacherID

HAVING AVG(Age) > 12

结果如下：



## MySQL 分页查询语句

limit 基本实现方式

一般情况下，客户端通过传递 pageNo（页码）、pageSize（每页条数）两个参数去分页查询数据库中的数据，在数据量较小（元组百/千级）时使用 MySQL自带的 limit 来解决这个问题：

收到客户端{pageNo:1,pagesize:10}   
select \* from table limit (pageNo-1)\*pageSize, pageSize;

收到客户端{pageNo:5,pageSize:30}   
select \* from table limit (pageNo-1)\*pageSize,pageSize;

建立主键或者唯一索引

在数据量较小的时候简单的使用 limit 进行数据分页在性能上面不会有明显的缓慢，但是数据量达到了 **万级到百万级** sql语句的性能将会影响数据的返回。这时需要利用主键或者唯一索引进行数据分页；

假设主键或者唯一索引为 good\_id   
收到客户端{pageNo:5,pagesize:10}   
select \* from table where good\_id > (pageNo-1)\*pageSize limit pageSize;   
–返回good\_id为40到50之间的数据

基于数据再排序

当需要返回的信息为顺序或者倒序时，对上面的语句基于数据再排序。order byASC/DESC 顺序或倒序 默认为顺序

select \* from table where good\_id > (pageNo-1)\*pageSize order by good\_id limit pageSize;   
–返回good\_id为40到50之间的数据,数据依据good\_id顺序排列

## lock 和 synchronized 的区别？

参考资料

**[book]Disruptor使用入门**

[**https://edagarli.gitbooks.io/java-route/content/disruptorshi\_yong\_ru\_men.html**](https://edagarli.gitbooks.io/java-route/content/disruptorshi_yong_ru_men.html)

**[book]剖析Disruptor为什么会这么快？**

[**https://lixiangyun.gitbooks.io/disruptor/content/1.1.html**](https://lixiangyun.gitbooks.io/disruptor/content/1.1.html)

Mechanical Sympathy 译文

http://ifeve.com/mechanical-sympathy/

[Disruptor 极速体验](https://www.cnblogs.com/haiq/p/4112689.html)

<https://www.cnblogs.com/haiq/p/4112689.html>

[Disruptor深入解读](https://www.cnblogs.com/miao-rui/p/6379473.html)

http://www.cnblogs.com/miao-rui/p/6379473.html

Disruptor源码介绍（一）-RingBuffer

https://blog.csdn.net/dtlscsl/article/details/79908104

Mechanical Sympathy - Hardware and software working together in harmony

https://mechanical-sympathy.blogspot.com/

**disruptor-3.3.2源码解析汇总**

**http://brokendreams.iteye.com/blog/2255720**

Java专家系列：CPU Cache与高性能编程

https://blog.csdn.net/karamos/article/details/80126704

[mysql中set autocommit=0与start transaction区别](https://www.cnblogs.com/langtianya/p/4777662.html)

https://www.cnblogs.com/langtianya/p/4777662.html

**A port of the LMAX Disruptor to the Go language.**

<https://github.com/smartystreets/go-disruptor>

使用Golang实现的无锁队列，性能与Disruptor相当达到1400万/秒

<https://zhuanlan.zhihu.com/p/24432607>

<https://github.com/yireyun/go-queue>

[**Java并发编程：volatile关键字解析**](https://www.cnblogs.com/dolphin0520/p/3920373.html)

[**https://www.cnblogs.com/dolphin0520/p/3920373.html**](https://www.cnblogs.com/dolphin0520/p/3920373.html)

**CPU缓存刷新的误解**

[**http://ifeve.com/cpu-cache-flushing-fallacy-cn/**](http://ifeve.com/cpu-cache-flushing-fallacy-cn/)

**OpenJDK里的AsmTools简介**

[**https://blog.csdn.net/hengyunabc/article/details/81122760**](https://blog.csdn.net/hengyunabc/article/details/81122760)

**如何设计高可用的微服务架构？**

[**http://36kr.com/p/5093162.html**](http://36kr.com/p/5093162.html)

# 面试总结

## 任何让进程运行在指定的cpu上？

**管理处理器的亲和性（affinity）**

<https://www.ibm.com/developerworks/cn/linux/l-affinity.html>

## 进程间通信概述

<https://www.cnblogs.com/yangykaifa/p/7295863.html>

## JAVA之删除数组中某个元素值

## Kafka实现原理

<https://blog.csdn.net/ychenfeng/article/details/74980531>

<https://kaimingwan.com/post/framworks/kafka/kafkayuan-li-yi-ji-she-ji-shi-xian-si-xiang>

## zookeeper原理

<https://www.cnblogs.com/felixzh/p/5869212.html>

## redis使用bitmap实现网站活跃用户的统计

<http://www.shixinke.com/redis/redis-bitmaps-command>

bitmap的应用场景：需要存储特别大的是否信息

## 查看JVM使用什么垃圾收集器

<https://blog.csdn.net/earthhour/article/details/76468084>

## redis set

http://redisdoc.com/string/set.html