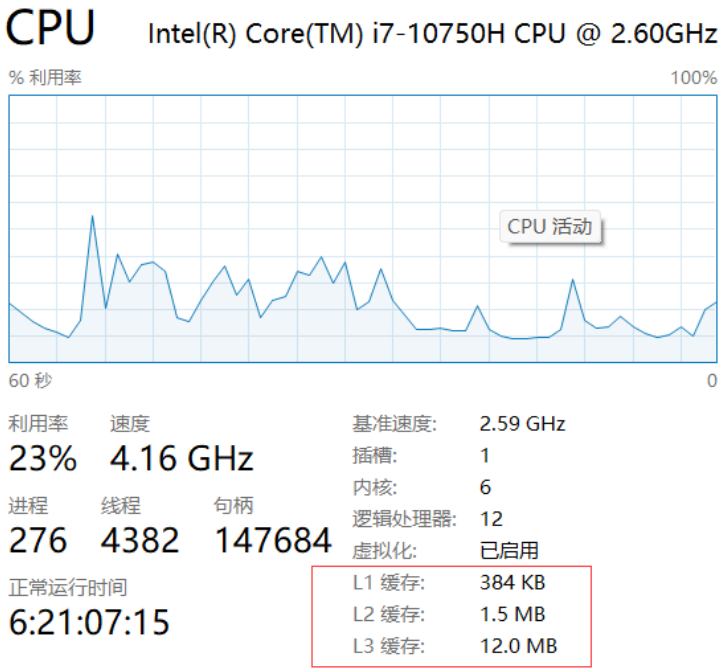
主讲老师：Fox老师

> 有道笔记地址：https://note.youdao.com/s/ALicPbhV
> 计算机基础系列-计算机组成原理：https://vip.tulingxueyuan.cn/p/t_pc/course_pc_detail/column/p_61766cd6e4b03b86217cc82b
> 计算机基础系列-操作系统：https://vip.tulingxueyuan.cn/p/t_pc/course_pc_detail/column/p_61766c11e4b0740575ad6d15

1.CPU缓存架构详解

1.1 CPU高速缓存概念

CPU缓存即高速缓冲存储器，是位于CPU与主内存间的一种容量较小但速度很高的存储器。CPU高速缓存可以分为一级缓存，二级缓存，部分高端CPU还具有三级缓存，每一级缓存中所储存的全部数据都是下一级缓存的一部分，这三种缓存的技术难度和制造成本是相对递减的，所以其容量也是相对递增的。



由于CPU的速度远高于主内存，CPU直接从内存中存取数据要等待一定时间周期，Cache中保存着CPU刚用过或循环使用的一部分数据，当CPU再次使用该部分数据时可从Cache中直接调用,减少CPU的等待时间，提高了系统的效率。

> 在CPU访问存储设备时，无论是存取数据抑或存取指令，都趋于聚集在一片连续的区域中，这就是**局部性原理。**
>
> **时间局部性（Temporal Locality）**：如果一个信息项正在被访问，那么在近期它很可能还会被再次访问。
> 比如循环、递归、方法的反复调用等。
> **空间局部性（Spatial Locality）**：如果一个存储器的位置被引用，那么将来他附近的位置也会被引用。
> 比如顺序执行的代码、连续创建的两个对象、数组等。

1.2 CPU多核缓存架构

现代CPU为了提升执行效率，减少CPU与内存的交互，一般在CPU上集成了多级缓存架构，常见的为三级缓存结构。如下图：

> CPU寄存器是位于CPU内部的存储器，数据的读写速度非常快，比缓存和主存更快。CPU会把常用的数据放到寄存器中进行处理，而不是直接从主存中读取，这样可以加速数据的访问和处理。但是寄存器的容量非常有限，一般只有几十个或者几百个字节，因此只能存储少量的数据。

当CPU读取一个地址中的数据时，会先在 L1 Cache 中查找。如果数据在 L1 Cache 中找到，CPU 会直接从 L1 Cache 中读取数据。如果没有找到，则会将这个请求发送给 L2 Cache，然后在 L2 Cache 中查找，如果 L2 Cache 中也没有找到，则会继续将请求发送到 L3 Cache 中。如果在 L3 Cache 中还是没有找到数据，则最后会从主内存中读取数据并将其存储到 CPU 的缓存中。
当CPU写入一个地址中的数据时，同样会先将数据写入 L1 Cache 中，然后再根据缓存一致性协议将数据写入 L2 Cache、L3 Cache 以及主内存中。具体写入过程与缓存一致性协议相关，有可能只写入 L1 Cache 中，也有可能需要将数据写入 L2 Cache、L3 Cache 以及主内存中。写入过程中也可能会使用缓存行失效、写回等技术来提高效率。

思考：这种缓存架构在多线程访问的时候存在什么问题？

CPU多核缓存架构缓存一致性问题分析

场景一

场景二

在CPU多核缓存架构中，每个处理器都有一个单独的缓存，共享数据可能有多个副本：一个副本在主内存中，一个副本在请求它的每个处理器的本地缓存中。当数据的一个副本发生更改时，其他副本必须反映该更改。也就是说，CPU多核缓存架构要保证缓存一致性。

1.3 CPU缓存架构缓存一致性的解决方案

《64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf》中有如下描述：

> The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag. The processor uses three interdependent mechanisms for carrying out locked atomic operations:
> • Guaranteed atomic operations
> • Bus locking, using the LOCK# signal and the LOCK instruction prefix
> • Cache coherency protocols that ensure that atomic operations can be carried out on cached data structures (cache lock); this mechanism is present in the Pentium 4, Intel Xeon, and P6 family processors

32位的IA-32处理器支持对系统内存中的位置进行锁定的原子操作。这些操作通常用于管理共享的数据结构(如信号量、段描述符、系统段或页表)，在这些结构中，两个或多个处理器可能同时试图修改相同的字段或标志。处理器使用三种相互依赖的机制来执行锁定的原子操作:

- 有保证的原子操作

> 处理器提供一些特殊的指令或者机制，可以保证在多个处理器同时执行原子操作时，它们不会相互干扰，从而保证原子性。这些指令或者机制的实现通常需要硬件支持。例如x86架构中提供了一系列的原子操作指令，如XADD、XCHG、CMPXCHG等，可以保证在多个处理器同时执行这些指令时，它们不会相互干扰，从而保证原子性。

- 总线锁定，使用LOCK#信号和LOCK指令前缀

> 总线锁定是一种用于确保原子操作的机制，通常会在LOCK指令前缀和一些特殊指令中使用。在执行LOCK指令前缀时，处理器会将LOCK#信号拉低，这个信号会通知其他处理器当前总线上的数据已经被锁定，从而确保原子性。

- 缓存一致性协议，确保原子操作可以在缓存的数据结构上执行(缓存锁定);这种机制出现在Pentium 4、Intel Xeon和P6系列处理器中

缓存一致性协议是一种用于确保处理器缓存中的数据和主存中的数据一致的机制。缓存一致性协议会通过处理器之间的通信，确保在一个处理器修改了某个数据后，其他处理器缓存中的该数据会被更新或者失效，从而保证在多个处理器同时对同一个数据进行操作时，它们所看到的数据始终是一致的。缓存锁定则是在缓存一致性协议的基础上实现原子操作的机制，它会利用缓存一致性协议来确保在多个处理器同时修改同一个缓存行中的数据时，只有一个处理器能够获得锁定，从而保证原子性。缓存锁定的实现也需要硬件的支持，而且不同的处理器架构可能会有不同的实现方式。

缓存一致性协议不能使用的特殊情况:

- 当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行时，则处理器会调用总线锁定。

> 不能被缓存在处理器内部的数据通常指的是不可缓存的设备内存（如显存、网络接口卡的缓存等），这些设备内存一般不受缓存一致性协议的管辖，处理器无法将其缓存到自己的缓存行中。

- 有些处理器不支持缓存锁定。早期的Pentium系列处理器并不支持缓存锁定机制。在这些处理器上，只能使用总线锁定来实现原子操作。

但现代的处理器通常都支持缓存锁定机制，因此应该尽量使用缓存锁定来实现原子操作，以获得更好的性能。

1.4 缓存一致性协议实现原理

总线窥探

总线窥探(Bus snooping)是缓存中的一致性控制器(snoopy cache)监视或窥探总线事务的一种方案，其目标是在分布式共享内存系统中维护缓存一致性。包含一致性控制器(snooper)的缓存称为snoopy缓存。该方案由Ravishankar和Goodman于1983年提出。

> 在计算机中，数据通过总线在处理器和内存之间传递。每次处理器和内存之间的数据传递都是通过一系列步骤来完成的，这一系列步骤称之为总线事务（Bus Transaction）

工作原理

当特定数据被多个缓存共享时，处理器修改了共享数据的值，更改必须传播到所有其他具有该数据副本的缓存中。这种更改传播可以防止系统违反缓存一致性。数据变更的通知可以通过总线窥探来完成。所有的窥探者都在监视总线上的每一个事务。如果一个修改共享缓存块的事务出现在总线上，所有的窥探者都会检查他们的缓存是否有共享块的相同副本。如果缓存中有共享块的副本，则相应的窥探者执行一个动作以确保缓存一致性。这个动作可以是刷新缓存块或使缓存块失效。它还涉及到缓存块状态的改变，这取决于缓存一致性协议（cache coherence protocol）。

窥探协议类型

根据管理写操作的本地副本的方式，有两种窥探协议:

写失效（Write-invalidate）

当处理器写入一个共享缓存块时，其他缓存中的所有共享副本都会通过总线窥探失效。这种方法确保处理器只能读写一个数据的一个副本。其他缓存中的所有其他副本都无效。这是最常用的窥探协议。MSI、MESI、MOSI、MOESI和MESIF协议属于该类型。

写更新（Write-update）

当处理器写入一个共享缓存块时，其他缓存的所有共享副本都会通过总线窥探更新。这个方法将写数据广播到总线上的所有缓存中。它比write-invalidate协议引起更大的总线流量。这就是为什么这种方法不常见。Dragon和firefly协议属于此类别。

缓存一致性协议

缓存一致性协议在多处理器系统中应用于高速缓存一致性。为了保持一致性，人们设计了各种模型和协议，如MSI、MESI(又名Illinois)、MOSI、MOESI、MERSI、MESIF、write-once、Synapse、Berkeley、Firefly和Dragon协议。

MESI协议

MESI协议是一个基于写失效的缓存一致性协议，是支持回写（write-back）缓存的最常用协议。也称作**伊利诺伊协议** (Illinois protocol，因为是在伊利诺伊大学厄巴纳-香槟分校被发明的)。

缓存行有4种不同的状态:

- **已修改Modified (M)**

缓存行是脏的（*dirty*），与主存的值不同。如果别的CPU内核要读主存这块数据，该缓存行必须回写到主存，状态变为共享(S).

- **独占Exclusive (E)**

缓存行只在当前缓存中，但是干净的——缓存数据同于主存数据。当别的缓存读取它时，状态变为共享；当前写数据时，变为已修改状态。

- **共享Shared (S)**

缓存行也存在于其它缓存中且是未修改的。缓存行可以在任意时刻抛弃。

- **无效Invalid (I)**

缓存行是无效的

MESI协议用于确保多个处理器之间共享的内存数据的一致性。当一个处理器需要访问某个内存数据时，它首先会检查自己的缓存中是否有该数据的副本。如果缓存中没有该数据的副本，则会发出一个缓存不命中（miss）请求，从主内存中获取该数据的副本，并将该数据的副本存储到自己的缓存中。

当一个处理器发出一个缓存不命中请求时，如果该数据的副本已经存在于另一个处理器或核心的缓存中（即处于共享状态），则该处理器可以从另一个处理器的缓存中复制该数据的副本。这个过程称为缓存到缓存复制（cache-to-cache transfer）。

缓存到缓存复制可以减少对主内存的访问，从而提高系统的性能。但是，需要确保数据的一致性，否则会出现数据错误或不一致的情况。因此，在进行缓存到缓存复制时，需要使用MESI协议中的其他状态转换来确保数据的一致性。例如，如果两个缓存都处于修改状态，那么必须先将其中一个缓存的数据写回到主内存，然后才能进行缓存到缓存复制。

1.5 伪共享的问题

如果多个核上的线程在操作同一个缓存行中的不同变量数据，那么就会出现频繁的缓存失效，即使在代码层面看这两个线程操作的数据之间完全没有关系。这种不合理的资源竞争情况就是伪共享（False Sharing）。

ArrayBlockingQueue有三个成员变量：

- takeIndex：需要被取走的元素下标
- putIndex：可被元素插入的位置的下标
- count：队列中元素的数量

这三个变量很容易放到一个缓存行中，但是之间修改没有太多的关联。所以每次修改，都会使之前缓存的数据失效，从而不能完全达到共享的效果。

如上图所示，当生产者线程put一个元素到ArrayBlockingQueue时，putIndex会修改，从而导致消费者线程的缓存中的缓存行无效，需要从主存中重新读取。

linux下查看Cache Line大小

Cache Line大小是64Byte

或者执行 cat /proc/cpuinfo 命令

避免伪共享方案

方案1：缓存行填充

```
class Pointer {
    volatile long x;
    //避免伪共享：  缓存行填充
    long p1, p2, p3, p4, p5, p6, p7;
    volatile long y;
}
```

方案2：使用 @sun.misc.Contended 注解（java8）

注意需要配置jvm参数：-XX:-RestrictContended

```java
public class FalseSharingTest {

    public static void main(String[] args) throws InterruptedException {
        testPointer(new Pointer());
    }

    private static void testPointer(Pointer pointer) throws InterruptedException {
        long start = System.currentTimeMillis();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100000000; i++) {
                pointer.x++;
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100000000; i++) {
                pointer.y++;
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println(pointer.x+","+pointer.y);

        System.out.println(System.currentTimeMillis() - start);


    }
}


class Pointer {
    // 避免伪共享:  @Contended +  jvm参数: -XX:-RestrictContended  jdk8支持
    //@Contended
```

```
    volatile long x;
    //避免伪共享： 缓存行填充
    //long p1, p2, p3, p4, p5, p6, p7;
    volatile long y;
}
```

方案3： 使用线程的本地内存，比如ThreadLocal

2.高性能内存队列Disruptor详解

2.1 juc包下阻塞队列的缺陷

1） juc下的队列大部分采用加ReentrantLock锁方式保证线程安全。在稳定性要求特别高的系统中，为了防止生产者速度过快，导致内存溢出，只能选择有界队列。

2）加锁的方式通常会严重影响性能。线程会因为竞争不到锁而被挂起，等待其他线程释放锁而唤醒，这个过程存在很大的开销，而且存在死锁的隐患。

3） 有界队列通常采用数组实现。但是采用数组实现又会引发另外一个问题false sharing(伪共享)。

2.2 Disruptor介绍

Disruptor是英国外汇交易公司LMAX开发的一个高性能队列，研发的初衷是解决内存队列的延迟问题（在性能测试中发现竟然与I/O操作处于同样的数量级）。基于Disruptor开发的系统单线程能支撑每秒600万订单，2010年在QCon演讲后，获得了业界关注。2011年，企业应用软件专家Martin Fowler专门撰写长文介绍。同年它还获得了Oracle官方的Duke大奖。

目前，包括Apache Storm、Camel、Log4j2在内的很多知名项目都应用了Disruptor以获取高性能。

Github：https://github.com/LMAX-Exchange/disruptor

Disruptor实现了队列的功能并且是一个有界队列，可以用于生产者-消费者模型。

2.3 Disruptor的高性能设计方案

Disruptor通过以下设计来解决队列速度慢的问题：

- **环形数组结构**

  为了避免垃圾回收，采用数组而非链表。同时，数组对处理器的缓存机制更加友好（空间局部性原理）。

- 元素位置定位

数组长度2^n，通过位运算，加快定位的速度。下标采取递增的形式。不用担心index溢出的问题。index是long类型，即使100万QPS的处理速度，也需要30万年才能用完。

- **无锁设计**

每个生产者或者消费者线程，会通过先申请可以操作的元素在数组中的位置，申请到之后，直接在该位置写入或者读取数据。整个过程通过原子变量CAS，保证操作的线程安全。

- **利用缓存行填充解决了伪共享的问题**
- 实现了基于事件驱动的生产者消费者模型（观察者模式）

消费者时刻关注着队列里有没有消息，一旦有新消息产生，消费者线程就会立刻把它消费

RingBuffer数据结构

使用RingBuffer来作为队列的数据结构，RingBuffer就是一个可自定义大小的环形数组。除数组外还有一个序列号(sequence)，用以指向下一个可用的元素，供生产者与消费者使用。原理图如下所示：

- Disruptor要求设置数组长度为2的n次幂。在知道索引(index)下标的情况下，存与取数组上的元素时间复杂度只有O(1)，而这个index我们可以通过序列号与数组的长度取模来计算得出，index=sequence % entries.length。也可以用位运算来计算效率更高，此时array.length必须是2的幂次方，index=sequece&(entries.length-1)
- 当所有位置都放满了，再放下一个时，就会把0号位置覆盖掉

思考：覆盖数据是否会导致数据丢失呢?

等待策略

名称	措施	适用场景
BlockingWaitStrategy	加锁	CPU资源紧缺，吞吐量和延迟并不重要的场景
BusySpinWaitStrategy	自旋	通过不断重试，减少切换线程导致的系统调用，而降低延迟。推荐在线程绑定到固定的CPU的场景下使用
PhasedBackoffWaitStrategy	自旋 + yield + 自定义策略	CPU资源紧缺，吞吐量和延迟并不重要的场景
SleepingWaitStrategy	自旋 + yield + sleep	性能和CPU资源之间有很好的折中。延迟不均匀
TimeoutBlockingWaitStrategy	加锁，有超时限制	CPU资源紧缺，吞吐量和延迟并不重要的场景
YieldingWaitStrategy	自旋 + yield + 自旋	性能和CPU资源之间有很好的折中。延迟比较均匀

Disruptor在日志框架中的应用

Log4j 2相对于Log4j 1最大的优势在于多线程并发场景下性能更优。该特性源自于Log4j 2的异步模式采用了Disruptor来处理。 在Log4j 2的配置文件中可以配置WaitStrategy，默认是Timeout策略。

loggers all async采用的是Disruptor，而Async Appender采用的是ArrayBlockingQueue队列。
由图可见，单线程情况下，loggers all async与Async Appender吞吐量相差不大，但是在64个线程的时候，loggers all async的吞吐量比Async Appender增加了12倍，是Sync模式的68倍。

2.4 Disruptor实战

引入依赖

```
<!-- disruptor -->
<dependency>
    <groupId>com.lmax</groupId>
    <artifactId>disruptor</artifactId>
    <version>3.3.4</version>
</dependency>
```

Disruptor构造器

```
public Disruptor(
        final EventFactory<T> eventFactory,
        final int ringBufferSize,
        final ThreadFactory threadFactory,
        final ProducerType producerType,
        final WaitStrategy waitStrategy)
```

- EventFactory：创建事件（任务）的工厂类。
- ringBufferSize：容器的长度。
- ThreadFactory ： 用于创建执行任务的线程。
- ProductType：生产者类型：单生产者、多生产者。
- WaitStrategy：等待策略。

使用流程：

1）构建消息载体（事件）

2） 构建生产者

3）构建消费者

4） 生产消息，消费消息的测试

单生产者单消费者模式

1）创建Event(消息载体/事件)和EventFactory（事件工厂）

创建 OrderEvent 类，这个类将会被放入环形队列中作为消息内容。创建OrderEventFactory类，用于创建OrderEvent事件

```java
@Data
public class OrderEvent {
    private long value;
    private String name;
}

public class OrderEventFactory implements EventFactory<OrderEvent> {

    @Override
    public OrderEvent newInstance() {
        return new OrderEvent();
    }
}
```

2） 创建消息（事件）生产者

创建 OrderEventProducer 类，它将作为生产者使用

```java
public class OrderEventProducer {
    //事件队列
    private RingBuffer<OrderEvent> ringBuffer;

    public OrderEventProducer(RingBuffer<OrderEvent> ringBuffer) {
        this.ringBuffer = ringBuffer;
    }

    public void onData(long value,String name) {
        // 获取事件队列 的下一个槽
        long sequence = ringBuffer.next();
        try {
            //获取消息（事件）
            OrderEvent orderEvent = ringBuffer.get(sequence);
            // 写入消息数据
            orderEvent.setValue(value);
            orderEvent.setName(name);
        } catch (Exception e) {
            // TODO  异常处理
            e.printStackTrace();
        } finally {
            System.out.println("生产者发送数据value:"+value+",name:"+name);
            //发布事件
            ringBuffer.publish(sequence);
        }
    }
}
```

3）创建消费者

创建 OrderEventHandler 类，并实现 EventHandler<T> ，作为消费者。

```java
public class OrderEventHandler implements EventHandler<OrderEvent> {


    @Override
    public void onEvent(OrderEvent event, long sequence, boolean endOfBatch) throws Exception {
        // TODO 消费逻辑
        System.out.println("消费者获取数据value:"+event.getValue()+",name:"+event.getName());
    }
}
```

4） 测试

```java
public class DisruptorDemo {

    public static void main(String[] args) throws Exception {

        //创建disruptor
        Disruptor<OrderEvent> disruptor = new Disruptor<>(
                new OrderEventFactory(),
                1024 * 1024,
                Executors.defaultThreadFactory(),
                ProducerType.SINGLE, //单生产者
                new YieldingWaitStrategy()  //等待策略
        );
        //设置消费者用于处理RingBuffer的事件
        disruptor.handleEventsWith(new OrderEventHandler());
        disruptor.start();

        //创建ringbuffer容器
        RingBuffer<OrderEvent> ringBuffer = disruptor.getRingBuffer();
        //创建生产者
        OrderEventProducer eventProducer = new OrderEventProducer(ringBuffer);
        //发送消息
        for(int i=0;i<100;i++){
            eventProducer.onData(i,"Fox"+i);
        }

        disruptor.shutdown();
    }
}
```

单生产者多消费者模式

如果消费者是多个，只需要在调用 handleEventsWith 方法时将多个消费者传递进去。

```
//设置多消费者,消息会被重复消费
disruptor.handleEventsWith(new OrderEventHandler(), new OrderEventHandler());
```

上面传入的两个消费者会重复消费每一条消息，如果想实现一条消息在有多个消费者的情况下，只会被一个消费者消费，那么需要调用 handleEventsWithWorkerPool 方法。

```
//设置多消费者,消费者要实现WorkHandler接口，一条消息只会被一个消费者消费
disruptor.handleEventsWithWorkerPool(new OrderEventHandler(), new OrderEventHandler());
```

注意：消费者要实现WorkHandler接口

```
public class OrderEventHandler implements EventHandler<OrderEvent>,
WorkHandler<OrderEvent> {

    @Override
    public void onEvent(OrderEvent event, long sequence, boolean endOfBatch) throws
Exception {
        // TODO 消费逻辑
        System.out.println("消费者"+ Thread.currentThread().getName()
                +"获取数据value:"+ event.getValue()+",name:"+event.getName());
    }

    @Override
    public void onEvent(OrderEvent event) throws Exception {
        // TODO 消费逻辑
        System.out.println("消费者"+ Thread.currentThread().getName()
                +"获取数据value:"+ event.getValue()+",name:"+event.getName());
    }
}
```

多生产者多消费者模式

在实际开发中，多个生产者发送消息，多个消费者处理消息才是常态。

```java
public class DisruptorDemo2 {

    public static void main(String[] args) throws Exception {

        //创建disruptor
        Disruptor<OrderEvent> disruptor = new Disruptor<>(
                new OrderEventFactory(),
                1024 * 1024,
                Executors.defaultThreadFactory(),
                ProducerType.MULTI, //多生产者
                new YieldingWaitStrategy()  //等待策略
        );

        //设置消费者用于处理RingBuffer的事件
        //disruptor.handleEventsWith(new OrderEventHandler());
        //设置多消费者,消息会被重复消费
        //disruptor.handleEventsWith(new OrderEventHandler(),new OrderEventHandler());
        //设置多消费者,消费者要实现WorkHandler接口，一条消息只会被一个消费者消费
        disruptor.handleEventsWithWorkerPool(new OrderEventHandler(), new OrderEventHandler());

        //启动disruptor
        disruptor.start();

        //创建ringbuffer容器
        RingBuffer<OrderEvent> ringBuffer = disruptor.getRingBuffer();

        new Thread(()->{
            //创建生产者
            OrderEventProducer eventProducer = new OrderEventProducer(ringBuffer);
            // 发送消息
            for(int i=0;i<100;i++){
                eventProducer.onData(i,"Fox"+i);
            }
        },"producer1").start();

        new Thread(()->{
            //创建生产者
```

```
38              OrderEventProducer eventProducer = new OrderEventProducer(ringBuffer);
39              // 发送消息
40              for(int i=0;i<100;i++){
41                  eventProducer.onData(i,"monkey"+i);
42              }
43          },"producer2").start();
44
45
46          //disruptor.shutdown();
47
48      }
49  }
```