

二

55 理解Disruptor（下）：不需要换挡和踩刹车的CPU，有多快？

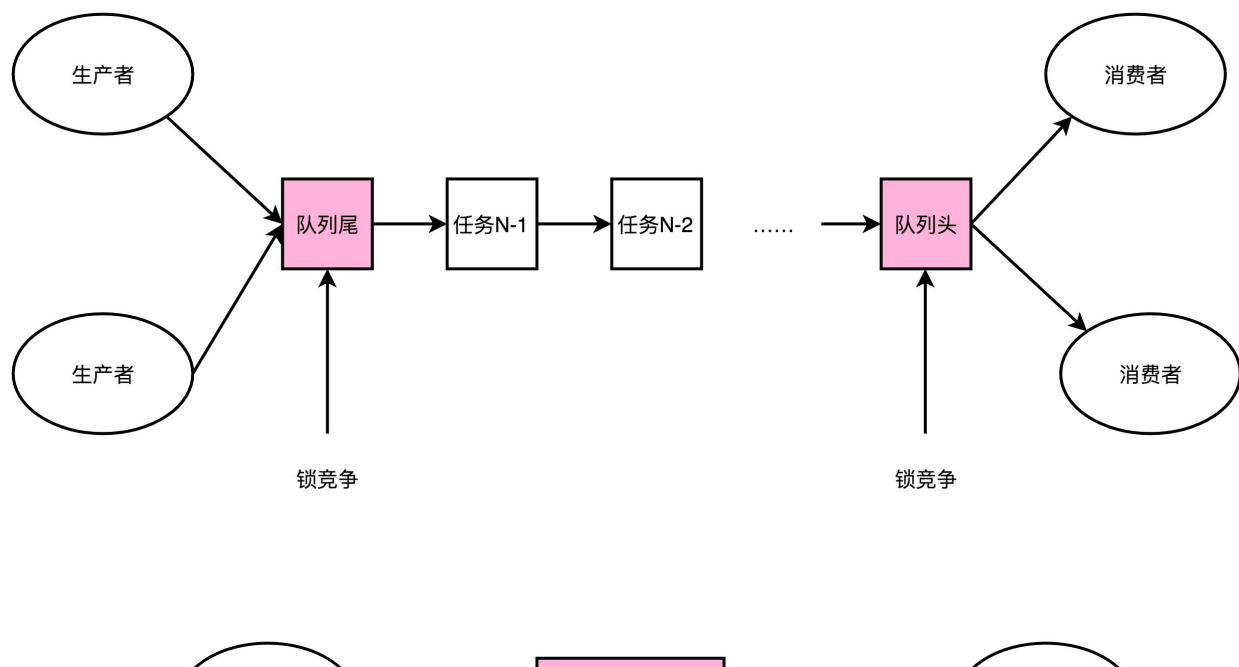
上一讲，我们学习了一个精妙的想法，Disruptor 通过缓存行填充，来利用好 CPU 的高速缓存。不知道你做完课后思考题之后，有没有体会到高速缓存在实践中带来的速度提升呢？

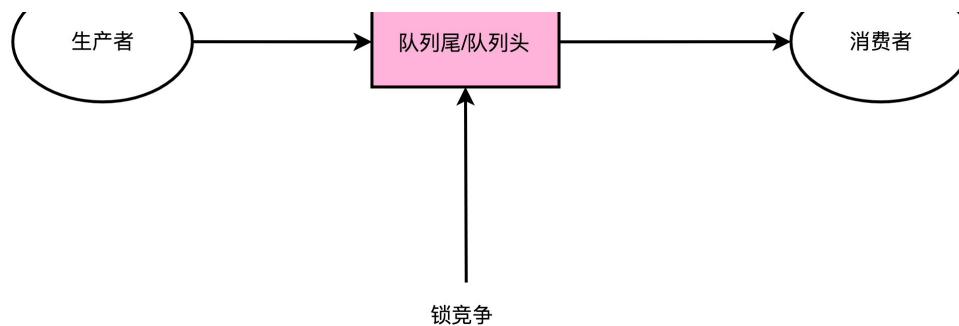
不过，利用 CPU 高速缓存，只是 Disruptor“快”的一个因素，那今天我们就来看一看 Disruptor 快的另一个因素，也就是“无锁”，而尽可能发挥 CPU 本身的高速处理性能。

缓慢的锁

Disruptor 作为一个高性能的生产者 - 消费者队列系统，一个核心的设计就是通过 RingBuffer 实现一个无锁队列。

上一讲里我们讲过，Java 里面的基础库里，就有像 LinkedBlockingQueue 这样的队列库。但是，这个队列库比起 Disruptor 里用的 RingBuffer 要慢上很多。慢的第一个原因我们说过，因为链表的数据在内存里面的布局对于高速缓存并不友好，而 RingBuffer 所使用的数组则不然。





LinkedBlockingQueue 慢，有另外一个重要的因素，那就是它对于锁的依赖。在生产者 - 消费者模式里，我们可能有多个消费者，同样也可能有多个生产者。多个生产者都要往队列的尾指针里面添加新的任务，就会产生多个线程的竞争。于是，在做这个事情的时候，生产者就需要拿到对于队列尾部的锁。同样地，在多个消费者去消费队列头的时候，也就产生竞争。同样消费者也要拿到锁。

那只有一个生产者，或者一个消费者，我们是不是就没有这个锁竞争的问题了呢？很遗憾，答案还是否定的。一般来说，在生产者 - 消费者模式下，消费者要比生产者快。不然的话，队列会产生积压，队列里面的任务会越堆越多。

一方面，你会发现越来越多的任务没有能够及时完成；另一方面，我们的内存也会放不下。虽然生产者 - 消费者模型下，我们都有一个队列来作为缓冲区，但是大部分情况下，这个缓冲区里面是空的。也就是说，即使只有一个生产者和一个消费者，这个生产者指向的队列尾和消费者指向的队列头是同一个节点。于是，这两个生产者和消费者之间一样会产生锁竞争。

在 LinkedBlockingQueue 上，这个锁机制是通过 synchronized 这个 Java 关键字来实现的。一般情况下，这个锁最终会对应到操作系统层面的加锁机制（OS-based Lock），这个锁机制需要由操作系统的内核来进行裁决。这个裁决，也需要通过一次上下文切换（Context Switch），把没有拿到锁的线程挂起等待。

不知道你还记不记得，我们在第 28 讲讲过的异常和中断，这里的上下文切换要做的和异常和中断里的是一样的。上下文切换的过程，需要把当前执行线程的寄存器等等信息，保存到线程栈里面。而这个过程也必然意味着，已经加载到高速缓存里面的指令或者数据，又回到了主内存里面，会进一步拖慢我们的性能。

我们可以按照 Disruptor 介绍资料里提到的 Benchmark，写一段代码来看看，是不是真是这样的。这里我放了一段 Java 代码，代码的逻辑很简单，就是把一个 long 类型的 counter，从 0 自增到 5 亿。一种方式是没有任何锁，另外一个方式是每次自增的时候都要去取一个锁。

你可以在自己的电脑上试试跑一下这个程序。在我这里，两个方式执行所需要的时间分别是 207 毫秒和 9603 毫秒，性能差出了将近 50 倍。

```
package com.xuwenhao.perf.jmm;

import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockBenchmark{

    public static void runIncrement()
    {
        long counter = 0;
        long max = 500000000L;
        long start = System.currentTimeMillis();
        while (counter < max) {
            counter++;
        }
        long end = System.currentTimeMillis();
        System.out.println("Time spent is " + (end-start) + "ms without lock");
    }

    public static void runIncrementWithLock()
    {
        Lock lock = new ReentrantLock();
        long counter = 0;
        long max = 500000000L;
        long start = System.currentTimeMillis();
        while (counter < max) {
            if (lock.tryLock()){
                counter++;
                lock.unlock();
            }
        }
        long end = System.currentTimeMillis();
        System.out.println("Time spent is " + (end-start) + "ms with lock");
    }

    public static void main(String[] args) {
        runIncrement();
        runIncrementWithLock();
    }
}
```

加锁和不加锁自增 counter

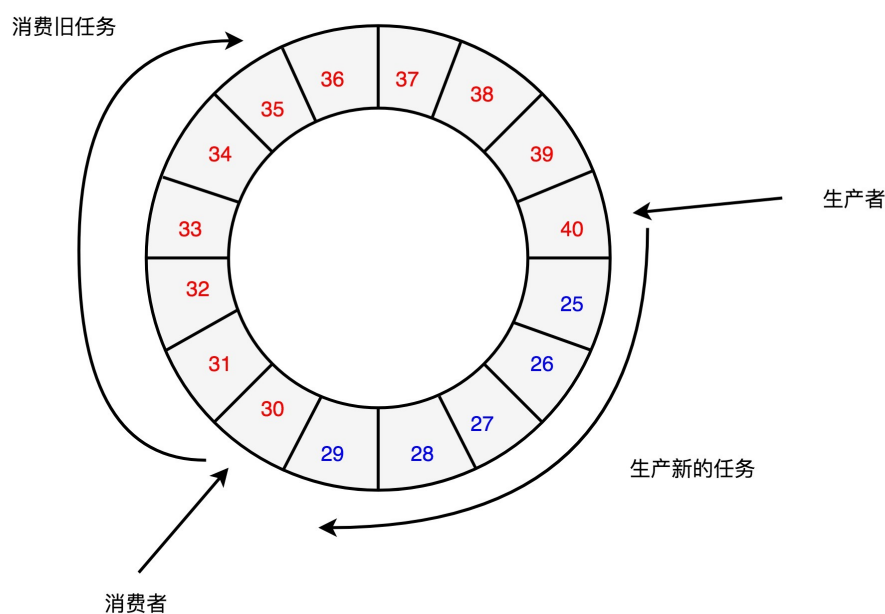
```
Time spent is 207ms without lock
Time spent is 9603ms with lock
```

性能差出将近 10 倍

无锁的 RingBuffer

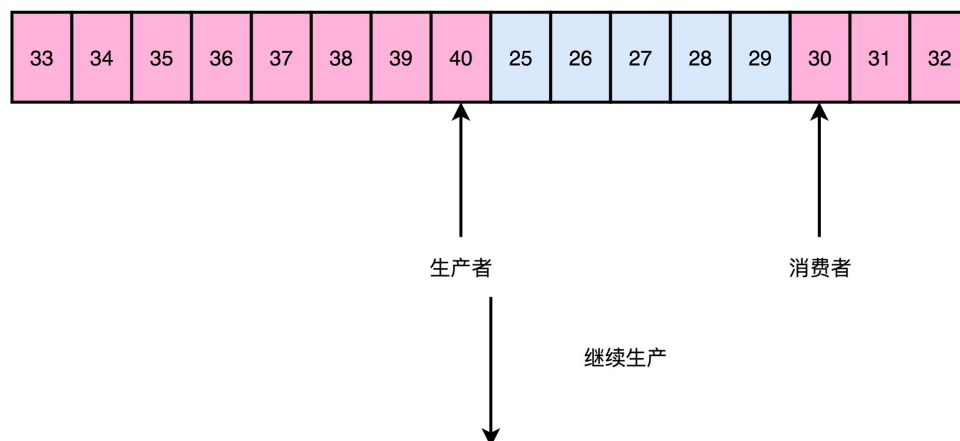
加锁很慢，所以 Disruptor 的解决方案就是“无锁”。这个“无锁”指的是没有操作系统层面的锁。实际上，Disruptor 还是利用了一个 CPU 硬件支持的指令，称之为 CAS（Compare And Swap，比较和交换）。在 Intel CPU 里面，这个对应的指令就是 `cmpxchg`。那么下面，我们就一起从 Disruptor 的源码，到具体的硬件指令来看看这是怎么一回事儿。

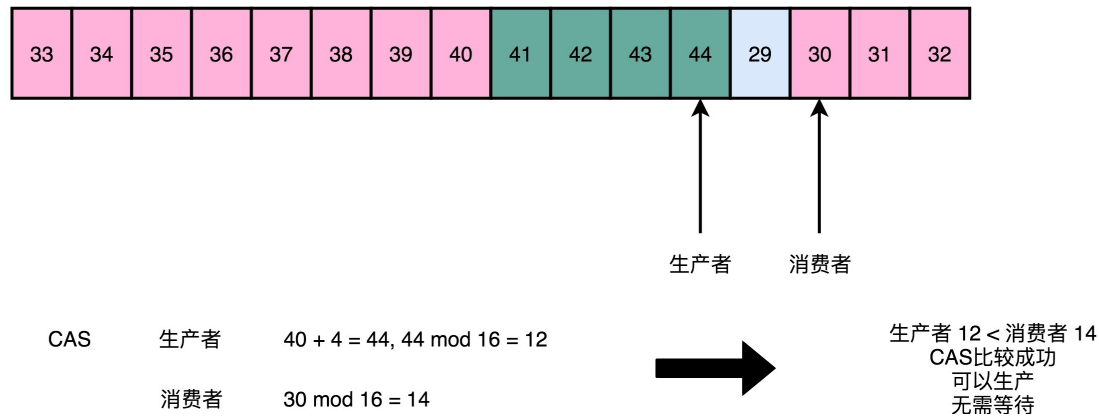
Disruptor 的 RingBuffer 是这么设计的，它和直接在链表的头和尾加锁不同。Disruptor 的 RingBuffer 创建了一个 Sequence 对象，用来指向当前的 RingBuffer 的头和尾。这个头和尾的标识呢，不是通过一个指针来实现的，而是通过一个**序号**。这也是为什么对应源码里面的类名叫 Sequence。



蓝色 生产者可以继续生产的缓冲区

红色 消费者可以继续消费的任务





在这个 RingBuffer 当中，进行生产者和消费者之间的资源协调，采用的是对比序号的方式。当生产者想要往队列里加入新数据的时候，它会把当前的生产者的 Sequence 的序号，加上需要加入的新数据的数量，然后和实际的消费者所在的位置进行对比，看看队列里是不是有足够的空间加入这些数据，而不会覆盖掉消费者还没有处理完的数据。

在 Sequence 的代码里面，就是通过 `compareAndSet` 这个方法，并且最终调用到了 `UNSAFE.compareAndSwapLong`，也就是直接使用了 CAS 指令。

```
public boolean compareAndSet(final long expectedValue, final long newValue)
{
    return UNSAFE.compareAndSwapLong(this, VALUE_OFFSET, expectedValue,
    }

public long addAndGet(final long increment)
{
    long currentValue;
    long newValue;

    do
    {
        currentValue = get();
        newValue = currentValue + increment;
    }
    while (!compareAndSet(currentValue, newValue));

    return newValue;
}
```

Sequence 源码中的 `addAndGet`，如果 CAS 的操作没有成功，它会不断忙等待地重试

这个 CAS 指令，也就是比较和交换的操作，并不是基础库里的一个函数。它也不是操作系统里面实现的一个系统调用，而是一个 **CPU 硬件支持的机器指令**。在我们服务器所使用的 Intel CPU 上，就是 `cmpxchg` 这个指令。

`cmpxchg [ax]`（隐式参数，EAX 累加器），`[bx]`（源操作数地址），`[cx]`（目标操作数地址）
复制代码

`cmpxchg` 指令，一共有三个操作数，第一个操作数不在指令里面出现，是一个隐式的操作数，也就是 EAX 累加寄存器里面的值。第二个操作数就是源操作数，并且指令会对比这个操作数和上面的累加寄存器里面的值。

如果值是相同的，那一方面，CPU 会把 ZF（也就是条件码寄存器里面零标志位的值）设置为 1，然后再把第三个操作数（也就是目标操作数），设置到源操作数的地址上。如果不相等的话，就会把源操作数里面的值，设置到累加器寄存器里面。

我在这里放了这个逻辑对应的伪代码，你可以看一下。如果你对汇编指令、条件码寄存器这些知识点有点儿模糊了，可以回头去看看第 5 讲、第 6 讲关于汇编指令的部分。

```
IF [ax] <== [bx] THEN [ZF] = 1, [bx] = [cx]
                     ELSE [ZF] = 0, [ax] = [bx]
```

单个指令是原子的，这也就意味着在使用 CAS 操作的时候，我们不再需要单独进行加锁，直接调用就可以了。

没有了锁，CPU 这部高速跑车就像在赛道上行驶，不会遇到需要上下文切换这样的红灯而停下来。虽然会遇到像 CAS 这样复杂的机器指令，就好像赛道上会有 U 型弯一样，不过不用完全停下来等待，我们 CPU 运行起来仍然会快很多。

那么，CAS 操作到底会有多快呢？我们还是用一段 Java 代码来看一下。

```
package com.xuwenhao.perf.jmm;

import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockBenchmark {

    public static void runIncrementAtomic()
    {
        AtomicLong counter = new AtomicLong(0);
        long max = 500000000L;
        long start = System.currentTimeMillis();
        while (counter.incrementAndGet() < max) {
        }
        long end = System.currentTimeMillis();
        System.out.println("Time spent is " + (end-start) + "ms with cas");
    }
}
```

```
    }

    public static void main(String[] args) {
        runIncrementAtomic();
    }
}
Time spent is 3867ms with cas
📄复制代码
```

和上面的 counter 自增一样，只不过这一次，自增我们采用了 AtomicLong 这个 Java 类。里面的 incrementAndGet 最终到了 CPU 指令层面，在实现的时候用的就是 CAS 操作。可以看到，它所花费的时间，虽然要比没有任何锁的操作慢上一个数量级，但是比起使用 ReentrantLock 这样的操作系统锁的机制，还是减少了一半以上的时间。

总结延伸

好了，咱们专栏的正文内容到今天就要结束了。今天最后一讲，我带着你一起看了 Disruptor 代码的一个核心设计，也就是它的 RingBuffer 是怎么做到无锁的。

Java 基础库里面的 BlockingQueue，都需要通过显式地加锁来保障生产者之间、消费者之间，乃至生产者和消费者之间，不会发生锁冲突的问题。

但是，加锁会大大拖慢我们的性能。在获取锁过程中，CPU 没有去执行计算的相关指令，而要等待操作系统进行锁竞争的裁决。而那些没有拿到锁而被挂起等待的线程，则需要进行上下文切换。这个上下文切换，会把挂起线程的寄存器里的数据放到线程的程序栈里面去。这也意味着，加载到高速缓存里面的数据也失效了，程序就变得更慢了。

Disruptor 里的 RingBuffer 采用了一个无锁的解决方案，通过 CAS 这样的操作，去进行序号的自增和对比，使得 CPU 不需要获取操作系统的锁。而是能够继续顺序地执行 CPU 指令。没有上下文切换、没有操作系统锁，自然程序就跑得快了。不过因为采用了 CAS 这样的忙等待（Busy-Wait）的方式，会使得我们的 CPU 始终满负荷运转，消耗更多的电，算是一个小小的缺点。

程序里面的 CAS 调用，映射到我们的 CPU 硬件层面，就是一个机器指令，这个指令就是 cmpxchg。可以看到，当想要追求最极致的性能的时候，我们会从应用层、贯穿到操作系统，乃至最后的 CPU 硬件，搞清楚从高级语言到系统调用，乃至最后的汇编指令，这个过程是怎么执行代码的。而这个，也是学习组成原理这门专栏的意义所在。

推荐阅读

不知道上一讲说的 Disruptor 相关材料，你有没有读完呢？如果没有读完的话，我建议你

是先去研读一下。

如果你已经读完了，这里再给你推荐一些额外的阅读材料，那就是著名的[Implement Lock-Free Queues](#)这篇论文。你可以更深入地学习一下，怎么实现一个无锁队列。

[上一页](#)

[下一页](#)