



54 | 算法实战（三）：剖析高性能队列Disruptor背后的数据结构和算法

2019-01-30 王争

数据结构与算法之美

进入课程 >



讲述：修阳

时长 11:59 大小 10.99M



Disruptor 你是否听说过呢？它是一种内存消息队列。从功能上讲，它其实有点儿类似 Kafka。不过，和 Kafka 不同的是，Disruptor 是线程之间用于消息传递的队列。它在 Apache Storm、Camel、Log4j 2 等很多知名项目中都有广泛应用。

之所以如此受青睐，主要还是因为它的性能表现非常优秀。它比 Java 中另外一个非常常用的内存消息队列 ArrayBlockingQueue (ABS) 的性能，要高一个数量级，可以算得上是最快的内存消息队列了。它还因此获得过 Oracle 官方的 Duke 大奖。

如此高性能的内存消息队列，在设计和实现上，必然有它独到的地方。今天，我们就来一块儿看下，**Disruptor 是如何做到如此高性能的？其底层依赖了哪些数据结构和算法？**

基于循环队列的“生产者 - 消费者模型”

什么是内存消息队列？对很多业务工程师或者前端工程师来说，可能会比较陌生。不过，如果说“生产者 - 消费者模型”，估计大部分人都知道。在这个模型中，“生产者”生产数据，并且将数据放到一个中心存储容器中。之后，“消费者”从中心存储容器中，取出数据消费。

这个模型非常简单、好理解，那你有没有思考过，这里面存储数据的中心存储容器，是用什么样的数据结构来实现的呢？

实际上，实现中心存储容器最常用的一种数据结构，就是我们在[第 9 节](#)讲的队列。队列支持数据的先进先出。正是这个特性，使得数据被消费的顺序性可以得到保证，也就是说，早被生产的数据就会早被消费。

我们在第 9 节讲过，队列有两种实现思路。一种是基于链表实现的链式队列，另一种是基于数组实现的顺序队列。不同的需求背景下，我们会选择不同的实现方式。

如果我们要实现一个无界队列，也就是说，队列的大小事先不确定，理论上可以支持无限大。这种情况下，我们适合选用链表来实现队列。因为链表支持快速地动态扩容。如果我们要实现一个有界队列，也就是说，队列的大小事先确定，当队列中数据满了之后，生产者就需要等待。直到消费者消费了数据，队列有空闲位置的时候，生产者才能将数据放入。

实际上，相较于无界队列，有界队列的应用场景更加广泛。毕竟，我们的机器内存是有限的。而无界队列占用的内存数量是不可控的。对于实际的软件开发来说，这种不可控的因素，就会有潜在的风险。在某些极端情况下，无界队列就有可能因为内存持续增长，而导致 OOM (Out of Memory) 错误。

在第 9 节中，我们还讲过一种特殊的顺序队列，循环队列。我们讲过，非循环的顺序队列在添加、删除数据的工程中，会涉及数据的搬移操作，导致性能变差。而循环队列正好可以解决这个数据搬移的问题，所以，性能更加好。所以，大部分用到顺序队列的场景中，我们都选择用顺序队列中的循环队列。

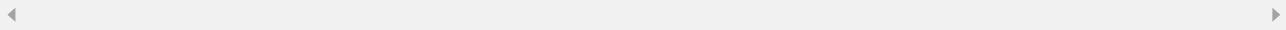
实际上，**循环队列这种数据结构，就是我们今天要讲的内存消息队列的雏形**。我借助循环队列，实现了一个最简单的“生产者 - 消费者模型”。对应的代码我贴到这里，你可以看看。

为了方便你理解，对于生产者和消费者之间操作的同步，我并没有用到线程相关的操作。而是采用了“当队列满了之后，生产者就轮训等待；当队列空了之后，消费者就轮训等待”这样的措施。

 复制代码

```
1 public class Queue {
2     private Long[] data;
3     private int size = 0, head = 0, tail = 0;
4     public Queue(int size) {
5         this.data = new Long[size];
6         this.size = size;
7     }
8
9     public boolean add(Long element) {
10        if ((tail + 1) % size == head) return false;
11        data[tail] = element;
12        tail = (tail + 1) % size;
13        return true;
14    }
15
16    public Long poll() {
17        if (head == tail) return null;
18        long ret = data[head];
19        head = (head + 1) % size;
20        return ret;
21    }
22 }
23
24 public class Producer {
25     private Queue queue;
26     public Producer(Queue queue) {
27         this.queue = queue;
28     }
29
30     public void produce(Long data) throws InterruptedException {
31         while (!queue.add(data)) {
32             Thread.sleep(100);
33         }
34     }
35 }
36
37 public class Consumer {
38     private Queue queue;
39     public Consumer(Queue queue) {
40         this.queue = queue;
41     }
42
43     public void consume() throws InterruptedException {
44         while (true) {
```

```
45     Long data = queue.poll();
46     if (data == null) {
47         Thread.sleep(100);
48     } else {
49         // TODO:... 消费数据的业务逻辑...
50     }
51 }
52 }
53 }
```



基于加锁的并发“生产者 - 消费者模型”

实际上，刚刚的“生产者 - 消费者模型”实现代码，是不完善的。为什么这么说呢？

如果我们只有一个生产者往队列中写数据，一个消费者从队列中读取数据，那上面的代码是没有问题的。但是，如果有多个生产者在并发地往队列中写入数据，或者多个消费者并发地从队列中消费数据，那上面的代码就不能正确工作了。我来给你讲讲为什么。

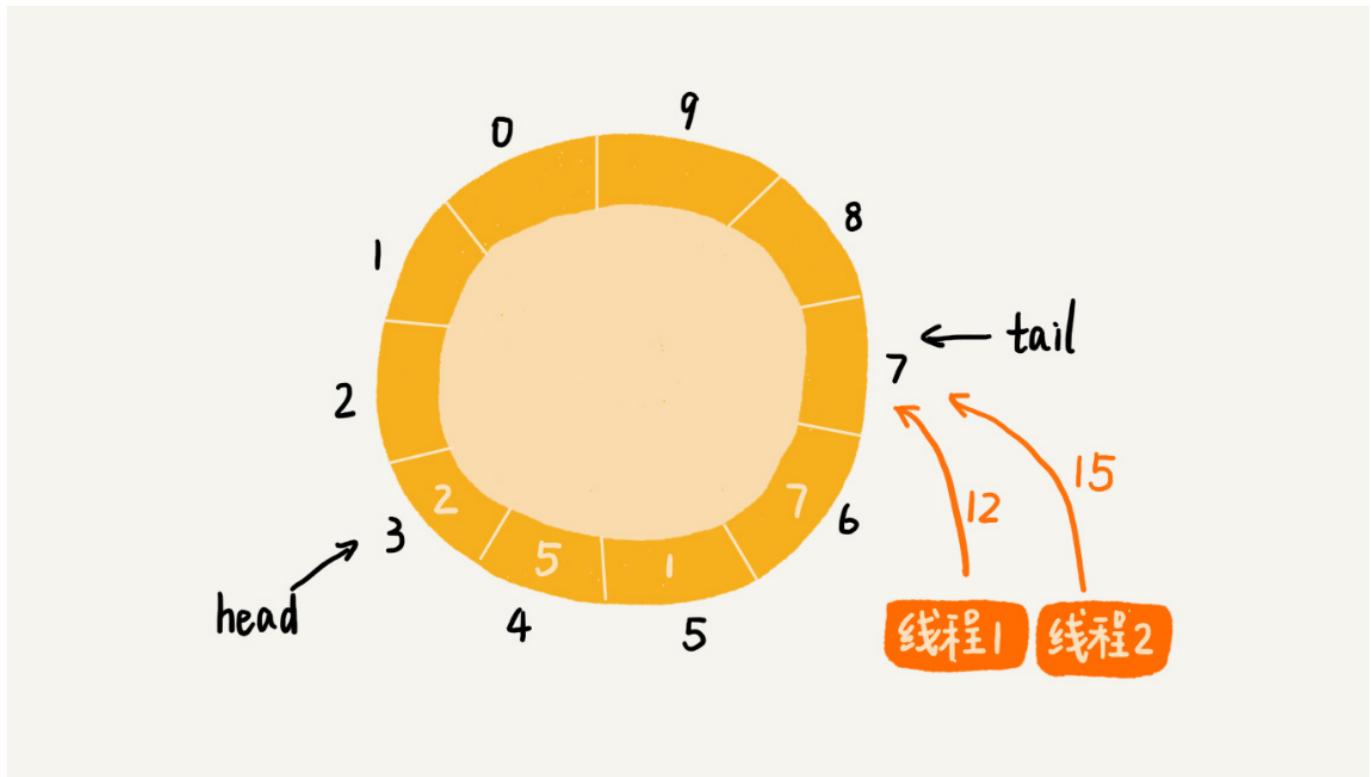
在多个生产者或者多个消费者并发操作队列的情况下，刚刚的代码主要会有下面两个问题：

多个生产者写入的数据可能会互相覆盖；

多个消费者可能会读取重复的数据。

因为第一个问题和第二个问题产生的原理是类似的。所以，我着重讲解第一个问题是如何产生的以及该如何解决。对于第二个问题，你可以类比我对第一个问题的解决思路自己来想一想。

两个线程同时往队列中添加数据，也就相当于两个线程同时执行类 Queue 中的 add() 函数。我们假设队列的大小 size 是 10，当前的 tail 指向下标 7，head 指向下标 3，也就是说，队列中还有空闲空间。这个时候，线程 1 调用 add() 函数，往队列中添加一个值为 12 的数据；线程 2 调用 add() 函数，往队列中添加一个值为 15 的数据。在极端情况下，本来是往队列中添加了两个数据（12 和 15），最终可能只有一个数据添加成功，另一个数据会被覆盖。这是为什么呢？



为了方便你查看队列 Queue 中的 add() 函数，我把它从上面的代码中摘录出来，贴在这里。

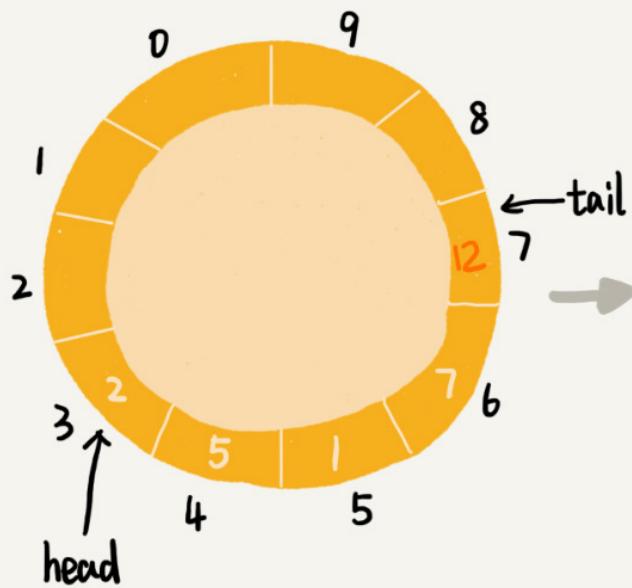
复制代码

```

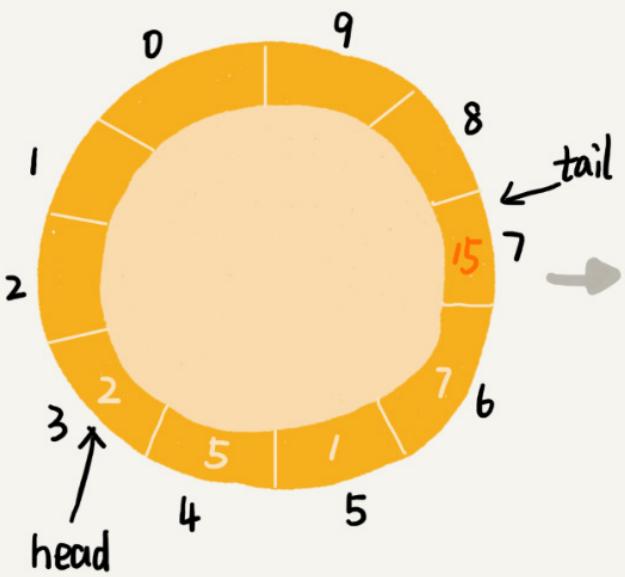
1 public boolean add(Long element) {
2     if ((tail + 1) % size == head) return false;
3     data[tail] = element;
4     tail = (tail + 1) % size;
5     return true;
6 }
```

从这段代码中，我们可以看到，第 3 行给 `data[tail]` 赋值，然后第 4 行才给 `tail` 的值加一。赋值和 `tail` 加一两个操作，并非原子操作。这就会导致这样的情况发生：当线程 1 和线程 2 同时执行 `add()` 函数的时候，线程 1 先执行完了第 3 行语句，将 `data[7]` (`tail` 等于 7) 的值设置为 12。在线程 1 还未执行到第 4 行语句之前，也就是还未将 `tail` 加一之前，线程 2 执行了第 3 行语句，又将 `data[7]` 的值设置为 15，也就是说，那线程 2 插入的数据覆盖了线程 1 插入的数据。原本应该插入两个数据 (12 和 15) 的，现在只插入了一个数据 (15)。

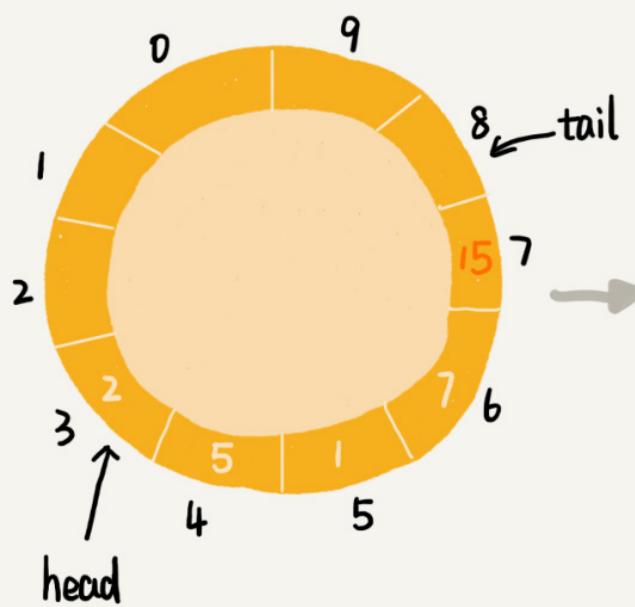
线程1执行第3行代码



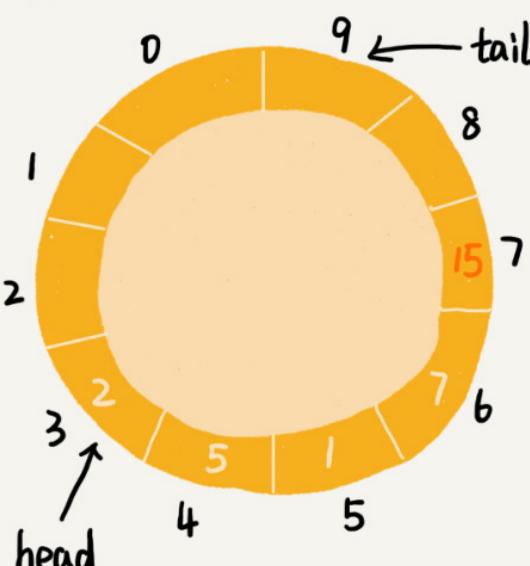
线程2执行第3行代码



线程1执行第4行代码



线程2执行第4行代码



那如何解决这种线程并发往队列中添加数据时，导致的数据覆盖、运行不正确问题呢？

最简单的处理方法就是给这段代码加锁，同一时间只允许一个线程执行 add() 函数。这相当于将这段代码的执行，由并行改成了串行，也就不存在我们刚刚说的问题了。

不过，天下没有免费的午餐，加锁将并行改成串行，必然导致多个生产者同时生产数据的时候，执行效率的下降。当然，我们可以继续优化代码，用 [CAS](#) (compare and swap，比

较并交换) 操作等减少加锁的粒度 , 但是 , 这不是我们这节的重点。我们直接看 Disruptor 的处理方法。

基于无锁的并发“生产者 - 消费者模型”

尽管 Disruptor 的源码读起来很复杂 , 但是基本思想其实非常简单。实际上 , 它是换了一种队列和 “ 生产者 - 消费者模型 ” 的实现思路。

之前的实现思路中 , 队列只支持两个操作 , 添加数据和读取并移除数据 , 分别对应代码中的 add() 函数和 poll() 函数 , 而 Disruptor 采用了另一种实现思路。

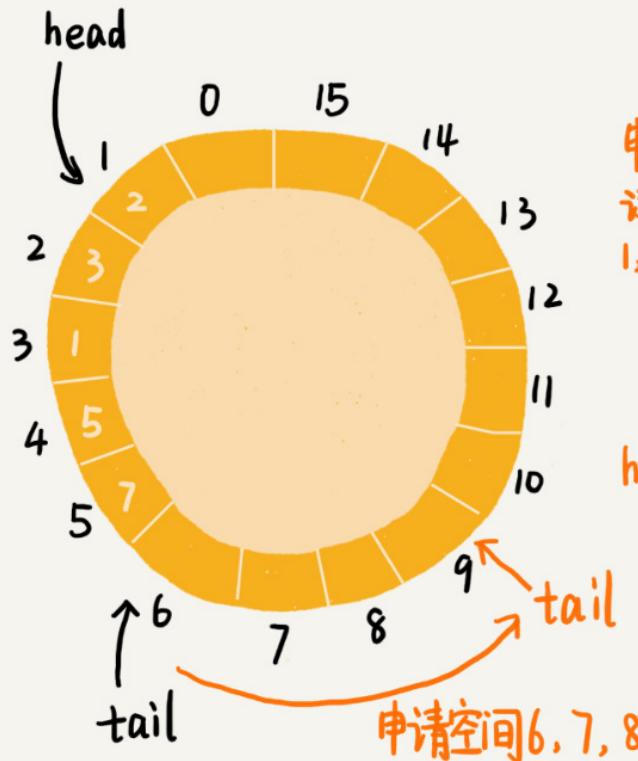
对于生产者来说 , 它往队列中添加数据之前 , 先申请可用空闲存储单元 , 并且是批量地申请连续的 n 个 ($n \geq 1$) 存储单元。当申请到这组连续的存储单元之后 , 后续往队列中添加元素 , 就可以不用加锁了 , 因为这组存储单元是这个线程独享的。不过 , 从刚刚的描述中 , 我们可以看出 , 申请存储单元的过程是需要加锁的。

对于消费者来说 , 处理的过程跟生产者是类似的。它先去申请一批连续可读的存储单元 (这个申请的过程也是需要加锁的) , 当申请到这批存储单元之后 , 后续的读取操作就可以不用加锁了。

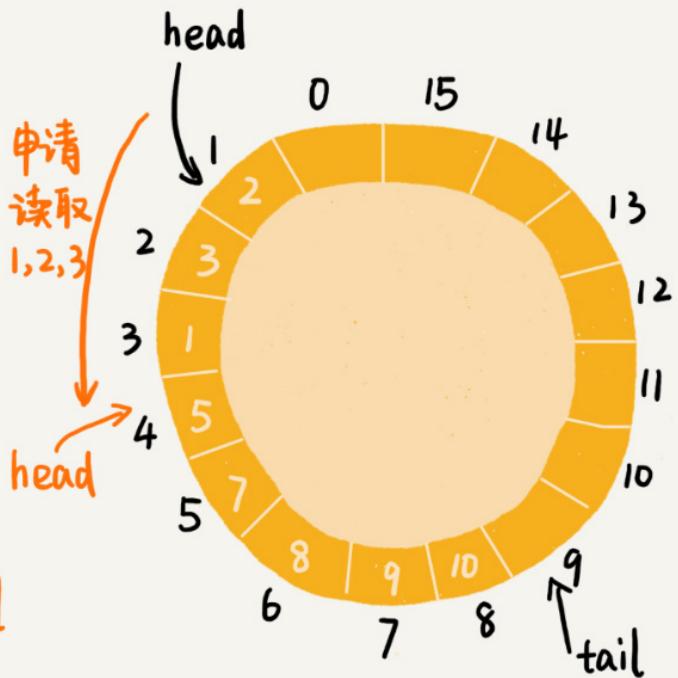
不过 , 还有一个需要特别注意的地方 , 那就是 , 如果生产者 A 申请到了一组连续的存储单元 , 假设是下标为 3 到 6 的存储单元 , 生产者 B 紧跟着申请到了下标是 7 到 9 的存储单元 , 那在 3 到 6 没有完全写入数据之前 , 7 到 9 的数据是无法读取的。这个也是 Disruptor 实现思路的一个弊端。

文字描述不好理解 , 我画了一个图 , 给你展示一下这个操作过程。

生产者申请空间



消费者申请可读取空间



实际上，Disruptor 采用的是 RingBuffer 和 AvailableBuffer 这两个结构，来实现我刚刚讲的功能。不过，因为我们主要聚焦在数据结构和算法上，所以我对这两种结构做了简化，但是基本思想是一致的。如果你对 Disruptor 感兴趣，可以去阅读一下它的[源码](#)。

总结引申

今天，我讲了如何实现一个高性能的并发队列。这里的“并发”两个字，实际上就是多线程安全的意思。

常见的内存队列往往采用循环队列来实现。这种实现方法，对于只有一个生产者和一个消费者的场景，已经足够了。但是，当存在多个生产者或者多个消费者的时候，单纯的循环队列的实现方式，就无法正确工作了。

这主要是因为，多个生产者在同时往队列中写入数据的时候，在某些情况下，会存在数据覆盖的问题。而多个消费者同时消费数据，在某些情况下，会存在消费重复数据的问题。

针对这个问题，最简单、暴力的解决方法就是，对写入和读取过程加锁。这种处理方法，相当于将原来可以并行执行的操作，强制串行执行，相应地就会导致操作性能的下降。

为了在保证逻辑正确的前提下，尽可能地提高队列在并发情况下的性能，Disruptor 采用了“两阶段写入”的方法。在写入数据之前，先加锁申请批量的空闲存储单元，之后往队列中写入数据的操作就不需要加锁了，写入的性能因此就提高了。Disruptor 对消费过程的改造，跟对生产过程的改造是类似的。它先加锁申请批量的可读取的存储单元，之后从队列中读取数据的操作也就不需要加锁了，读取的性能因此也就提高了。

你可能会觉得这个优化思路非常简单。实际上，不管架构设计还是产品设计，往往越简单的设计思路，越能更好地解决问题。正所谓“大道至简”，就是这个意思。

课后思考

为了提高存储性能，我们往往通过分库分表的方式设计数据库表。假设我们有 8 张表用来存储用户信息。这个时候，每张用户表中的 ID 字段就不能通过自增的方式来产生了。因为这样的话，就会导致不同表之间的用户 ID 值重复。

为了解决这个问题，我们需要实现一个 ID 生成器，可以为所有的用户表生成唯一的 ID 号。那现在问题是，如何设计一个高性能、支持并发的、能够生成全局唯一 ID 的 ID 生成器呢？

欢迎留言和我分享，也欢迎点击“[请朋友读](#)”，把今天的内容分享给你的好友，和他一起讨论、学习。



数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金奖励**。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 53 | 算法实战（二）：剖析搜索引擎背后的经典数据结构和算法

下一篇 55 | 算法实战（四）：剖析微服务接口鉴权限流背后的数据结构和算法

精选留言 (34)

写留言



Smallfly

2019-01-30

11

没有读过 Disruptor 的源码，从老师的文章理解，一个线程申请了一组存储空间，如果这组空间还没有被完全填满之前，另一个线程又进来，在这组空间之后申请空间并添加数据，之后第一组空间又继续填充数据，那在消费时如何保证队列是按照添加顺序读取的呢？

...

展开 ▼



想当上帝的...

2019-01-30

6

为什么3到6没有完全写入前，7到9无法读取，不是两个写入操作吗



老杨同志

2019-01-30

5

尝试回答老师的思考题

1) 分库分表也可以使用自增主键，可以设置增加的步长。8台机器分别从1、2、3。。。开始，步长8.

从1开始的下一个id是9，与其他的不重复就可以了。...

展开▼



Keep-Movi...

2019-01-30

5

思考题：加锁批量生成ID，使用时就不用加锁了

展开▼



且听疯吟

2019-02-23

4

disruptor使用环的数据结构，内存连续，初始化时就申请并设置对象，将原本队列的头尾节点锁的争用转化为cas操作，并利用Java对象填充，解决cache line伪共享问题



futute

2019-03-20

2

_sync_fetch_and_add操作即可实现原子自增的操作。

展开▼



belongcai...

2019-01-30

1

弱弱地问一下，后来老师补充过这节课的内容吗？我看后，跟以前留言的同学有相同的感觉。



belongcai...

2019-01-30

1

看完还是有很多困惑（可能跟我不了解线程安全有关）

一是申请一段连续存储空间，怎么成为线程独享的呢？生产者ab分别申请后，消费者为啥无法跨区域读取呢

二是这种方法应该是有实验证明效率高的，私下去了解下。

展开▼

作者回复: 这节课我抽空再补充下。不好意思。



神盾局闹别...

2019-01-30

冂 1

有个问题，按照老师所说，a线程只写1-3区块，b线程只写4-6块，假设a线程先写了块1，切换到线程b，b写块4，然后再切回线程a，a写块2。虽然没有数据覆盖问题，但是最终块1-6的顺序不是按写入先后顺序排布的，读取不是乱套了吗，怎么解决这个问题？求老师能说的详细点。

展开▼



www.xnsms...

2019-05-15

冂

感觉这个限流用的环形队列，又回到了上一节讲disrupter队列的问题一样，接口肯定是多线程，多个线程同时在往队列生产和消费，会出现生产和消费冲突，还是要解决这个问题.....哈哈



Geek_c33c8...

2019-04-13

冂

老师，线程池的实现是单个线程往队列插入任务，单个线程去队列去任务吗，所以不会处理并发控制吗

作者回复: 这里讲的时候没涉及并发问题，实际上会有的



xuery

2019-04-11

冂

项目中有使用过分布式id生成器，但是不知道具体是怎么实现的，参考今天的Disruptor的思路：

1. id生成器相当于一个全局的生产者，可以提前生成一批id

2. 对于每一张表，可以类似于Disruptor的消费者思路，从id生成器中申请一批id，用作当前表的id使用，当然申请一批id对于id生成器来说是需要加锁操作的

展开 ▼



www.xnsms...

2019-03-22



学storm的时候讲过里面的线程有个环形队列，以为这个课程会讲代码的实现，有点失望啊，是不是实现起来太麻烦了

展开 ▼



zj

2019-03-20



循环队列，一个生产者和一个消费者也会有线程安全问题啊，在判断是否队列满了的时候，得去tail和head的值。而没有锁就不能保证取到新的值

展开 ▼



QQ怪

2019-03-08



雪花算法可以根据不同机子生成不同的id

展开 ▼



cn

2019-02-15



申请了一段空间之后，只有写完才能够读取这段空间和后续空间，如果生产者挂了，是不是读取就阻塞了？



漫漫越

2019-02-14



尝试回答老师的问题

1.最简单的办法就是加锁，但这样就不支持并发。

2.将所有生成唯一Id的请求放入队列中，队列每次取出数据来产生Id，优点是不需要加锁。
缺点是不支持并发。

老师有啥方法吗？求赐教

展开 ▼



左瞳

2019-02-13



两阶段写入没有代码还是看不懂

展开 ▼



您的好...

2019-02-11



不同步长的自增主键还是会重复的，应该是使用唯一ID生成器将生成好的ID放到Disruptor中，之后8个以加锁申请之后读取的方式获取相应的ID。

展开 ▼



郭小菜

2019-02-09



一直追老师的课程，每期的质量都很高，收获很多。但是这期确实有点虎头蛇尾，没有把disruptor的实现原理讲得特别明白。不是讲得不好，只是觉得结尾有点突兀，信息量少了。但是瑕不掩瑜，老师的课程总体质量绝对是杠杠的！

展开 ▼

作者回复: 你说的没错。这节课有点不详细，我抽空重新补充下。抱歉。

