高性能无锁并发框架Disruptor，太强了！

程序员飞鱼 2024-01-20 12,185 阅读8分钟 专栏：源码相关

智能总结 复制 重新生成

文章介绍了高性能无锁并发框架 Disruptor，包括其产生原因、基本概念、等待策略、核心设计原理、数据结构、写数据流程、使用场景等。它旨在解决高并发下队列锁问题，众多知名项目应用，比 ArrayBlockingQueue 性能优，常用于“生产者-消费者”且要求顺序处理的场景。

关联问题：Disruptor适用多生产者吗 Disruptor怎样优化性能 RingBuffer优势在哪

基于该文章内容继续向AI提问

前言

Disruptor是一个开源框架，研发的初衷是为了解决高并发下队列锁的问题，最早由LMAX提出并使用，能够在无锁的情况下实现队列的并发操作，并号称能够在一个线程里每秒处理6百万笔订单

官网：lmax-exchange.github.io/disruptor/

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

「觉得不错，希望点赞，在看，转发支持一下，谢谢」

「文章内容收录到个人网站，方便阅读」：hardyfish.top/

「资料分享」

Redis设计与实现：

- 资料链接：url81.ctfile.com/f/57345181-...
- 访问密码：3899

Redis开发运维实践指南：

- 资料链接：url81.ctfile.com/f/57345181-...
- 访问密码：3899

Redis开发与运维(完整版)：

- 资料链接：url81.ctfile.com/f/57345181-...
- 访问密码：3899

Redis 深度历险：核心原理与应用实践：

- 资料链接：url81.ctfile.com/f/57345181-...
- 访问密码：3899

为什么会产生Disruptor框架

「「目前Java内置队列保证线程安全的方式：」」

ArrayBlockingQueue：基于数组形式的队列，通过加锁的方式，来保证多线程情况下数据的安全；

LinkedBlockingQueue：基于链表形式的队列，也通过加锁的方式，来保证多线程情况下数据的安全；

ConcurrentLinkedQueue：基于链表形式的队列，通过CAS的方式

我们知道，在编程过程中，加锁通常会严重地影响性能，所以尽量用无锁方式，就产生了Disruptor这种无锁高并发框架

基本概念

参考地址：github.com/LMAX-Exchan...

RingBuffer——Disruptor底层数据结构实现，核心类，是线程间交换数据的中转地；

程序员飞鱼 LV.5
后端工程师 @美团
作者榜No.3 优秀作者
96 文章 294k 阅读 1.5k 粉丝
关注 私信

目录 收起

相关推荐

- 单机最快的队列Disruptor解析和使用 6.5k阅读 · 103点赞
- 开发需了解的知识：Java虚拟线程设计... 1.2k阅读 · 4点赞
- 记一次事务里发普通消息的线上问题排... 2.4k阅读 · 30点赞
- 百度交易中台之系统对账篇 2.1k阅读 · 26点赞
- 库存领域核心能力--库存预占 建设实践 1.5k阅读 · 19点赞

精选内容

- Zookeeper（51）如何优化Zookeeper... Victor356 · 15阅读 · 1点赞
- 2025年02月：一些有趣的强化学习研究... Y11_推特同名 · 30阅读 · 1点赞
- Go语言扩展包x/sync使用指南 名字的问题 · 25阅读 · 0点赞
- 鸿蒙轻内核M核源码分析系列二一 05 文... 别说我什么都不会 · 8阅读 · 0点赞
- OpenHarmony（鸿蒙南向开发）——小... 塞尔维亚大汉 · 18阅读 · 0点赞

Sequencer——序号管理器，生产同步的实现者，负责消费者/生产者各自序号、序号栅栏的管理和协调,Sequencer有单生产者,多生产者两种不同的模式,里面实现了各种同步的算法；

Sequence——序号，声明一个序号，用于跟踪ringbuffer中任务的变化和消费者的消费情况，disruptor里面大部分的并发代码都是通过对Sequence的值同步修改实现的,而非锁,这是disruptor高性能的一个主要原因；

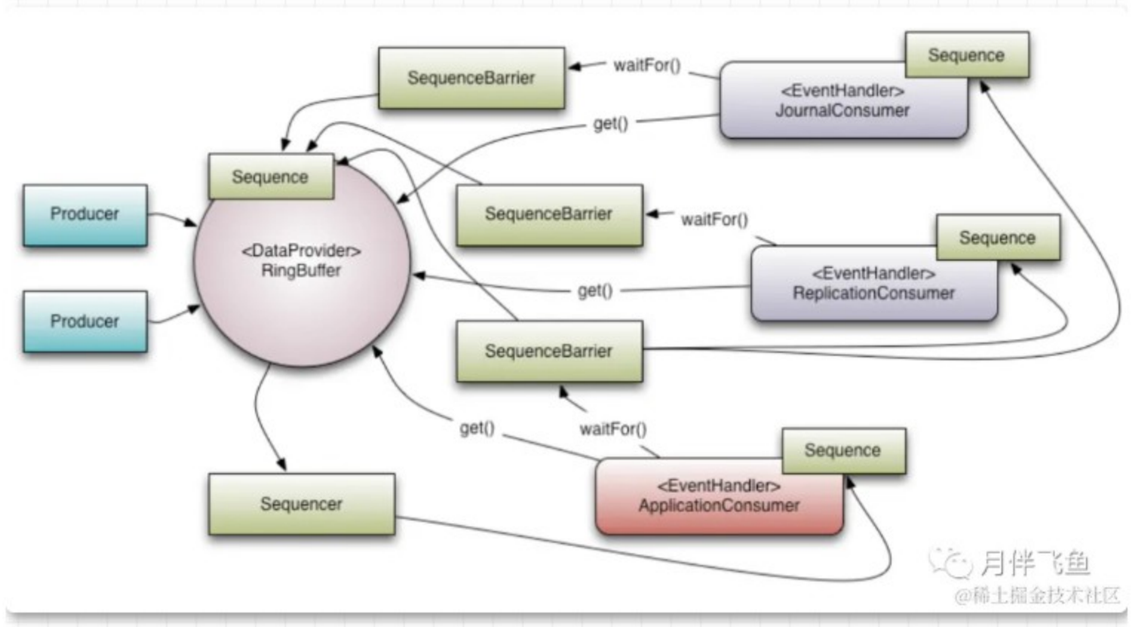
SequenceBarrier——序号栅栏，管理和协调生产者的游标序号和各个消费者的序号，确保生产者不会覆盖消费者未来得及处理的消息，确保存在依赖的消费者之间能够按照正确的顺序处理

EventProcessor——事件处理器，监听RingBuffer的事件，并消费可用事件，从RingBuffer读取的事件会交由实际的生产者实现类来消费；它会一直侦听下一个可用的序号，直到该序号对应的事件已经准备好。

EventHandler——业务处理器，是实际消费者的接口，完成具体的业务逻辑实现，第三方实现该接口；代表着消费者。

Producer——生产者接口，第三方线程充当该角色，producer向RingBuffer写入事件。

Wait Strategy：Wait Strategy决定了一个消费者怎么等待生产者将事件（Event）放入Disruptor中。



等待策略

源码地址：github.com/LMAX-Exchan...

「「BlockingWaitStrategy」」

Disruptor的默认策略是BlockingWaitStrategy。在BlockingWaitStrategy内部是使用锁和condition来控制线程的唤醒。BlockingWaitStrategy是最低效的策略，但其对CPU的消耗最小并且在各种不同部署环境中能提供更加一致的性能表现。

「「SleepingWaitStrategy」」

SleepingWaitStrategy 的性能表现跟 BlockingWaitStrategy 差不多，对 CPU 的消耗也类似，但其对生产者线程的影响最小，通过使用 **LockSupport.parkNanos(1)** 来实现循环等待。

「「YieldingWaitStrategy」」

YieldingWaitStrategy是可以使用在低延迟系统的策略之一。YieldingWaitStrategy将自旋以等待序列增加到适当的值。在循环体内，将调用 **Thread.yield()** 以允许其他排队的线程运行。在要求极高性能且事件处理线数小于 CPU 逻辑核心数的场景中，推荐使用此策略；例如，CPU开启超线程的特性。

「「BusySpinWaitStrategy」」

性能最好，适合用于低延迟的系统。在要求极高性能且事件处理线程数小于CPU逻辑核心数的场景中，推荐使用此策略；例如，CPU开启超线程的特性。

「「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。

使用举例

参考地址：github.com/LMAX-Exchan...

xml 代码解读 复制代码

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

java 代码解读 复制代码

```java
//定义事件event 通过Disruptor 进行交换的数据类型。
public class LongEvent {

    private Long value;

    public Long getValue() {
        return value;
    }
```

```
9
10    public void setValue(Long value) {
11        this.value = value;
12    }
13
14 }
```

java 代码解读 复制代码

```
1  public class LongEventFactory implements EventFactory<LongEvent> {
2      public LongEvent newInstance() {
3          return new LongEvent();
4      }
5  }
```

java 代码解读 复制代码

```
1  //定义事件消费者
2  public class LongEventHandler implements EventHandler<LongEvent>  {
3      public void onEvent(LongEvent event, long sequence, boolean endOfBatch) throws Ex
4          System.out.println("消费者:"+event.getValue());
5      }
6  }
```

java 代码解读 复制代码

```
1  //定义生产者
2  public class LongEventProducer {
3      public final RingBuffer<LongEvent> ringBuffer;
4      public LongEventProducer(RingBuffer<LongEvent> ringBuffer) {
5          this.ringBuffer = ringBuffer;
6      }
7      public void onData(ByteBuffer byteBuffer) {
8          // 1.ringBuffer 事件队列 下一个槽
9          long sequence = ringBuffer.next();
10         Long data = null;
11         try {
12             //2.取出空的事件队列
13             LongEvent longEvent = ringBuffer.get(sequence);
14             data = byteBuffer.getLong(0);
15             //3.获取事件队列传递的数据
16             longEvent.setValue(data);
17             try {
18                 Thread.sleep(10);
19             } catch (InterruptedException e) {
20                 // TODO Auto-generated catch block
21                 e.printStackTrace();
22             }
23         } finally {
24             System.out.println("生产这准备发送数据");
25             //4.发布事件
26             ringBuffer.publish(sequence);
27         }
28     }
29 }
```

java 代码解读 复制代码

```
1  public class DisruptorMain {
2      public static void main(String[] args) {
3          // 1.创建一个可缓存的线程 提供线程来出发Consumer 的事件处理
4          ExecutorService executor = Executors.newCachedThreadPool();
5          // 2.创建工厂
6          EventFactory<LongEvent> eventFactory = new LongEventFactory();
7          // 3.创建ringBuffer 大小
8          int ringBufferSize = 1024 * 1024; // ringBufferSize大小一定要是2的N次方
9          // 4.创建Disruptor
10         Disruptor<LongEvent> disruptor = new Disruptor<LongEvent>(eventFactory, ringBu
11                 ProducerType.SINGLE, new YieldingWaitStrategy());
12         // 5.连接消费端方法
13         disruptor.handleEventsWith(new LongEventHandler());
14         // 6.启动
15         disruptor.start();
16         // 7.创建RingBuffer容器
17         RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();
18         // 8.创建生产者
19         LongEventProducer producer = new LongEventProducer(ringBuffer);
20         // 9.指定缓冲区大小
21         ByteBuffer byteBuffer = ByteBuffer.allocate(8);
22         for (int i = 1; i <= 100; i++) {
23             byteBuffer.putLong(0, i);
24             producer.onData(byteBuffer);
25         }
26         //10.关闭disruptor和executor
27         disruptor.shutdown();
28         executor.shutdown();
29     }
30 }
```

核心设计原理

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

> “
>
> 原因：CPU缓存是由很多个缓存行组成的。每个缓存行通常是64字节，并且它有效地引用主内存中的一块儿地址。一个Java的long类型变量是8字节，因此在一个缓存行中可以存8个long类型的变量。CPU每次从主存中拉取数据时，会把相邻的数据也存入同一个缓存行。在访问一个long数组的时候，如果数组中的一个值被加载到缓存中，它会自动加载另外7个。因此你能非常快的遍历这个数组。
>
> ”

「「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：」」

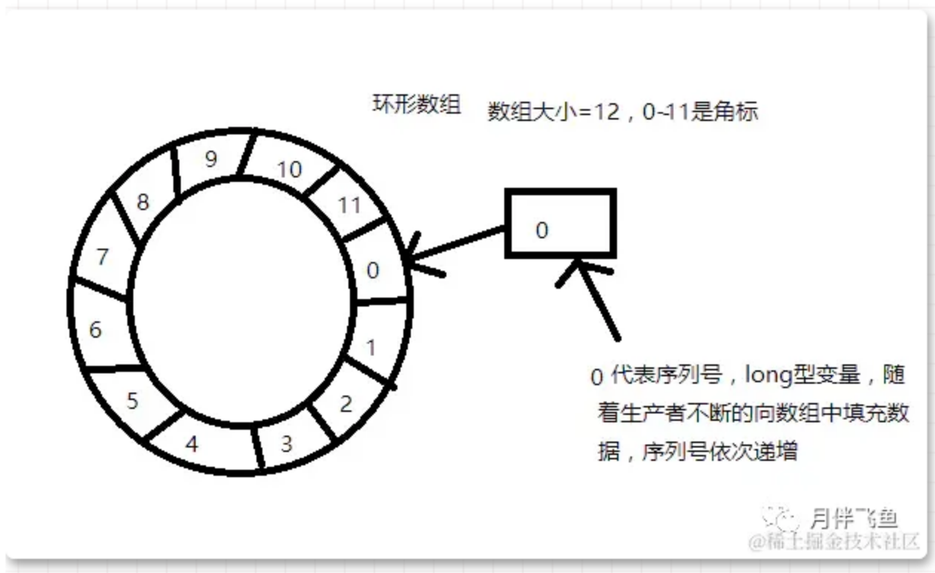
每个生产者或者消费者线程，会先申请可以操作的元素在数组中的位置，申请到之后，直接在该位置写入或者读取数据，整个过程通过原子变量CAS，保证操作的线程安全

数据结构

框架使用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：



Sequence

mark：Disruptor通过顺序递增的序号来编号管理通过其进行交换的数据（事件），对数据(事件)的处理过程总是沿着序号逐个递增处理。

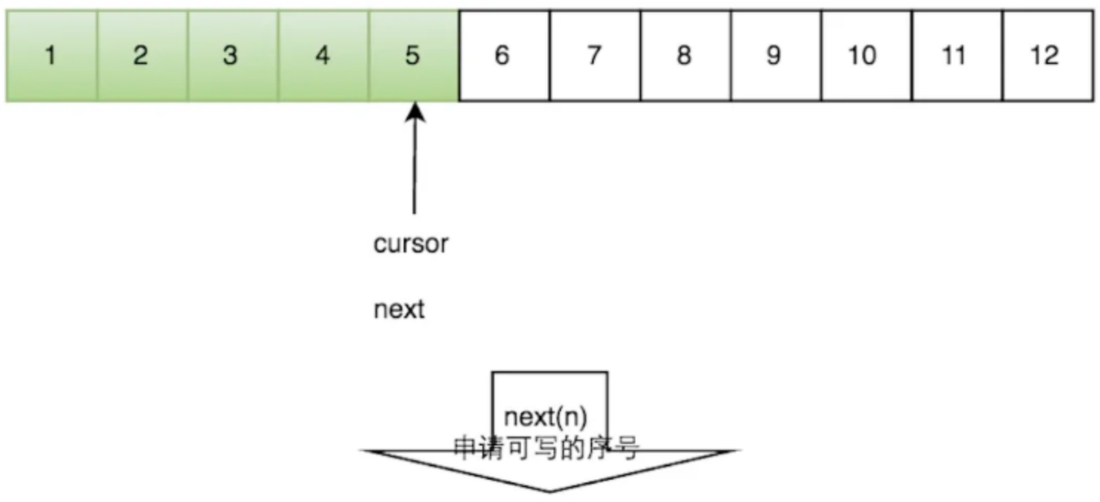
「「数组+序列号设计的优势是什么呢？」」

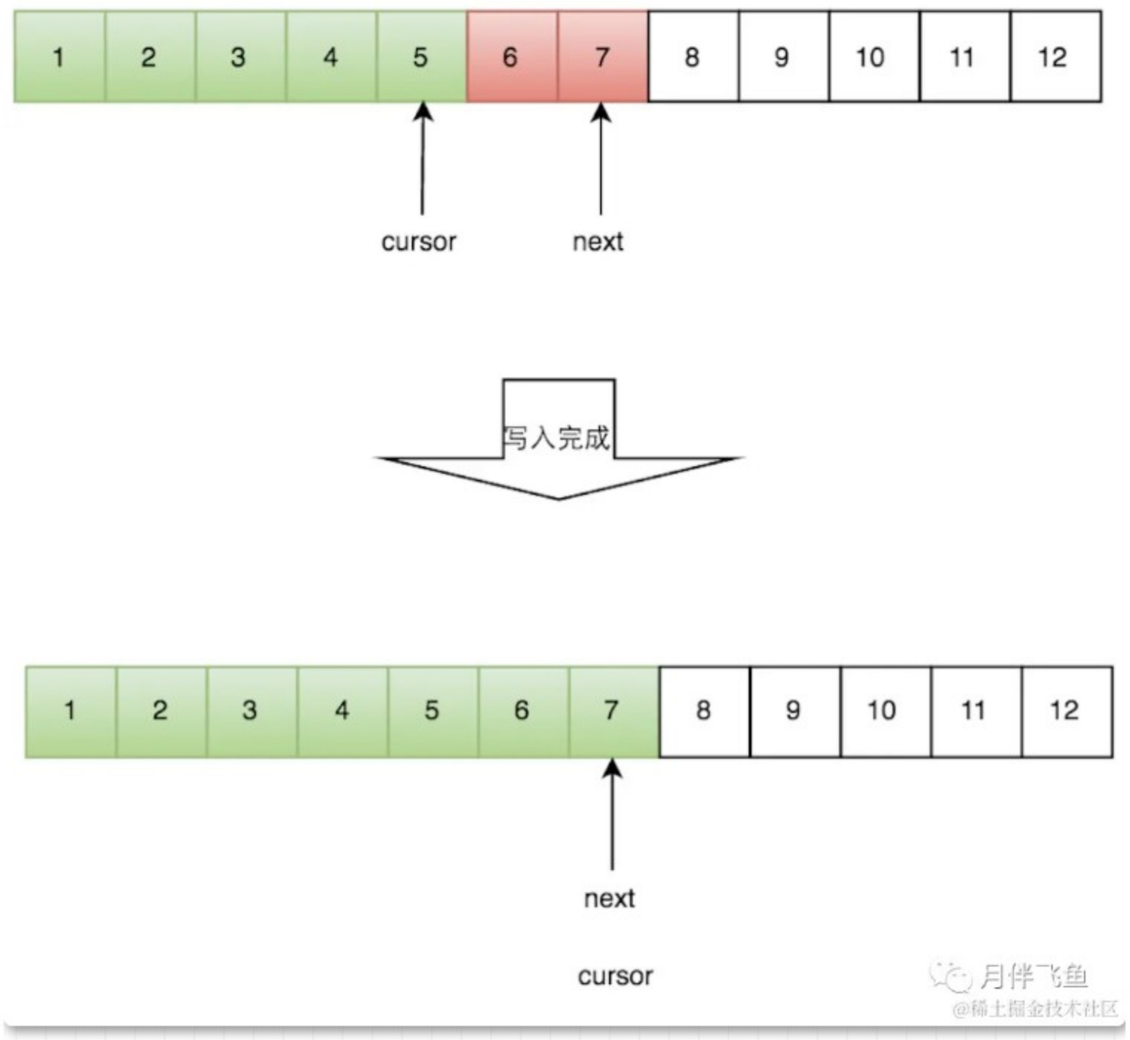
回顾一下HashMap，在知道索引(index)下标的情况下，存与取数组上的元素时间复杂度只有O(1)，而这个index我们可以通过序列号与数组的长度取模来计算得出， **index=sequence % table.length** 。当然也可以用位运算来计算效率更高，此时table.length必须是2的幂次方。

写数据流程

单线程写数据的流程：

- 申请写入m个元素；
- 若是有m个元素可以入，则返回最大的序列号。这儿主要判断是否会覆盖未读的元素；
- 若是返回的正确，则生产者开始写入元素。





使用场景

经过测试，Disruptor的的延时和吞吐量都比ArrayBlockingQueue优秀很多，所以，当你在使用ArrayBlockingQueue出现性能瓶颈的时候，你就可以考虑采用Disruptor的代替。

参考：github.com/LMAX-Exchan…

	Array Blocking Queue	Disruptor
Unicast: 1P – 1C	4,057,453	22,381,378
Pipeline: 1P – 3C	2,006,903	15,857,913
Sequencer: 3P – 1C	2,056,118	14,540,519
Multicast: 1P – 3C	260,733	10,860,121
Diamond: 1P – 3C	2,082,725	15,295,197

	Array Blocking Queue (ns)	Disruptor (ns)
Mean Latency	32,757	52
99% observations less than	2,097,152	128
99.99% observations less than	4,194,304	8,192

当然，Disruptor性能高并不是必然的，所以，是否使用还得经过测试。

Disruptor的最常用的场景就是“生产者-消费者”场景，对场景的就是“一个生产者、多个消费者”的场景，并且要求顺序处理。

举个例子，我们从MySQL的BigLog文件中顺序读取数据，然后写入到ElasticSearch（搜索引擎）中。在这种场景下，BigLog要求一个文件一个生产者，那个是一个生产者。而写入到ElasticSearch，则严格要求顺序，否则会出现问题，所以通常意义上的多消费者线程无法解决该问题，如果通过加锁，则性能大打折扣

「参考：」

tech.meituan.com/2016/11/18/…

github.com/LMAX-Exchan…

标签： 后端 Java 面试

本文收录于以下专栏

源码相关 专栏目录
源码相关知识！
1 订阅 · 12 篇文章
订阅

上一篇 学习集合类源码对我们实际工作的帮...
下一篇 动态代理总结，面试你要知道的都在...

评论 6

登录 / 注册 即可发布评论!

最热 | 最新

用户8814763128181
binlog 多个消费者怎么保证顺序? 😘
3月前 1 评论 ···

汪汪队首席上单
不持久化怎么用吗？队列的数据没处理完肯定也会出关机重启服务器那不就丢失了
12月前 点赞 1 ···

2382546457 : 单机队列，你可以把它看成跟 ArrayBlockingQueue 一个作用的队列
6月前 点赞 回复 ···

豆豆豆豆变 java
厉害大佬
1年前 1 1 ···

程序员飞鱼 作者 : 谢谢!
1年前 点赞 回复 ···

查看全部 6 条评论

为你推荐

Disruptor高性能之道—开篇&介绍
Life_of_Coder | 3年前 | 602 | 3 | 评论 后端

高性能无锁队列 Disruptor 初体验
haifeiWu | 6年前 | 6.8k | 156 | 评论 后端 Netty Apache S...

并发编程 | 并发编程框架 - Disruptor - 深入理解高性能异步处理框架
Kfaino | 1年前 | 174 | 2 | 评论 后端

Disruptor在流程编排中的应用与探索
ZA技术社区 | 1年前 | 750 | 1 | 评论 后端

深入浅出-高性能低延迟消息传递框架-Disruptor
宋小黑 | 11月前 | 1.5k | 10 | 评论 后端 面试 Java

如此狂妄，自称高性能队列的Disruptor有啥来头?
博学谷_狂野架构师 | 2年前 | 1.5k | 6 | 2 消息队列 Java 后端

高性能队列Disruptor的初体验!
加瓦点灯 | 27天前 | 153 | 2 | 评论 后端

Java并发编程高阶技术-高性能并发框架源码解析与实战分享
脑洞大开 | 6年前 | 380 | 1 | 评论 Java

初识Disruptor框架
源码笔记 | 2年前 | 2.9k | 12 | 评论 Java

如此狂妄，自称高性能队列的Disruptor有啥来头?
欧子有话说 | 2年前 | 49 | 点赞 | 评论 Java 程序员 后端

并发编程之Disruptor并发框架
codeobj | 5年前 | 3.6k | 6 | 1 Spring Boot

高性能队列框架-Disruptor使用、Netty结合Disruptor大幅提高数据处理性能
11来了 | 1年前 | 1.2k | 22 | 1 面试 Java

一文读懂锁、CAS、volitale 附赠并发学习神文(英文版)
Java猴子先生 | 5年前 | 968 | 点赞 | 评论 源码

你应该知道的高性能无锁队列Disruptor
咖啡拿铁 | 6年前 | 22k | 184 | 18 后端 Java

Disruptor高性能之道—False Sharing(伪共享)
Life_of_Coder | 3年前 | 1.4k | 2 | 评论 后端