SpringBoot + Disruptor 实现特快高并发处理，支撑每秒 600 万订单无压力！

程序员追风 2023-10-21 22:00 发表于湖南

收录于合集
#Spring Boot
49个

上方蓝色“程序员追风”，选择“设为星标”
回复“资料”获取整理好的面试资料

原文：

blog.csdn.net/buertianci/article/details/105327031

1、背景

工作中遇到项目使用Disruptor做消息队列，对你没看错，不是Kafka也不是rabbitmq。Disruptor有个最大的优点就是快，还有一点它是开源的哦，下面做个简单的记录。

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。

Disruptor是一个开源的Java框架，它被设计用于在生产者—消费者（producer-consumer problem，简称PCP）问题上获得尽量高的吞吐量（TPS）和尽量低的延迟。

从功能上来看，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。

Disruptor是LMAX在线交易平台的关键组成部分，LMAX平台使用该框架对订单处理速度能达到600万TPS，除金融领域之外，其他一般的应用中都可以用到Disruptor，它可以带来显著的性能提升。

其实Disruptor与其说是一个框架，不如说是一种设计思路，这个设计思路对于存在“并发、缓冲区、生产者—消费者模型、事务处理”这些元素的程序来说，Disruptor提出了一种大幅提升性能（TPS）的方案。

Disruptor的github主页：

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

3、Disruptor 的核心概念

先从了解 Disruptor 的核心概念开始，来了解它是如何运作的。下面介绍的概念模型，既是领域对象，也是映射到代码实现上的核心对象。

1. Ring Buffer

如其名，环形的缓冲区。曾经 RingBuffer 是 Disruptor 中的最主要的对象，但从3.0版本开始，其职责被简化为仅仅负责对通过 Disruptor 进行交换的数据（事件）进行存储和更新。在一些更高级的应用场景中，Ring Buffer 可以由用户的自定义实现来完全替代。

2. Sequence 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。一个 Sequence 用于跟踪标识某个特定的事件处理者(RingBuffer/Consumer)的处理进度。

虽然一个 AtomicLong 也可以用于标识进度，但定义 Sequence 来负责该问题还有另一个目的，那就是防止不同的 Sequence 之间的CPU缓存伪共享(Flase Sharing)问题。

> 注：这是 Disruptor 实现高性能的关键点之一，网上关于伪共享问题的介绍已经汗牛充栋，在此不再赘述。

3. Sequencer

Sequencer 是 Disruptor 的真正核心。此接口有两个实现类 SingleProducerSequencer、MultiProducerSequencer，它们定义在生产者和消费者之间快速、正确地传递数据的并发算法。

4. Sequence Barrier

用于保持对RingBuffer的 main published Sequence 和Consumer依赖的其它Consumer的 Sequence 的引用。Sequence Barrier 还定义了决定 Consumer 是否还有可处理的事件的逻辑。

5. Wait Strategy

定义 Consumer 如何进行等待下一个事件的策略。（注：Disruptor 定义了多种不同的策略，针对不同的场景，提供了不一样的性能表现）

6. Event

在 Disruptor 的语义中，生产者和消费者之间进行交换的数据被称为事件(Event)。它不是一个被 Disruptor 定义的特定类型，而是由 Disruptor 的使用者定义并指定。

7. EventProcessor

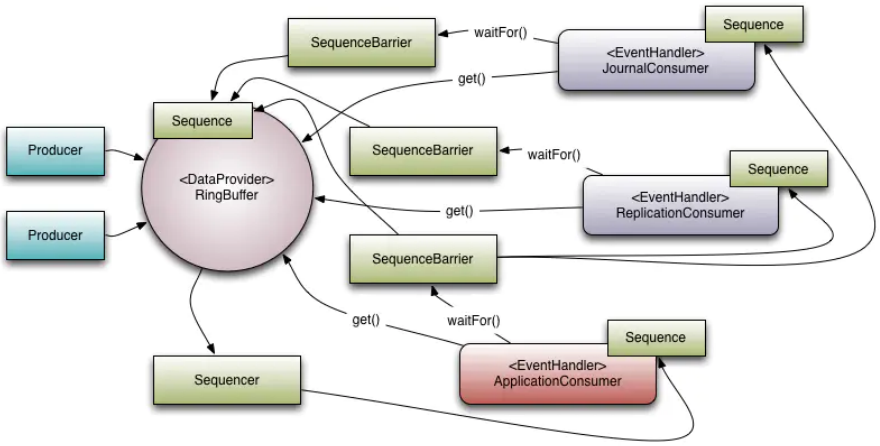
EventProcessor 持有特定消费者(Consumer)的 Sequence，并提供用于调用事件处理实现的事件循环(Event Loop)。

8. EventHandler

Disruptor 定义的事件处理接口，由用户实现，用于处理事件，是 Consumer 的真正实现。

9. Producer

即生产者，只是泛指调用 Disruptor 发布事件的用户代码，Disruptor 没有定义特定接口或类型。



https://blog.csdn.net/buertianci

4、案例-demo

通过下面8个步骤，你就能将Disruptor Get回家啦：

1、添加pom.xml依赖

```
<dependency>
    <groupId>com.lmax</groupId>
    <artifactId>disruptor</artifactId>
    <version>3.4.4</version>
</dependency>
```

2、消息体Model

```
/**
 * 消息体
 */
@Data
public class MessageModel {
    private String message;
}
```

3、构造EventFactory

```
public class HelloEventFactory implements EventFactory<MessageModel> {
    @Override
    public MessageModel newInstance() {
        return new MessageModel();
    }
}
```

4、构造EventHandler-消费者

```
@Slf4j
public class HelloEventHandler implements EventHandler<MessageModel> {
    @Override
    public void onEvent(MessageModel event, long sequence, boolean endOfBatch) {
        try {
            //这里停止1000ms是为了确定消费消息是异步的
            Thread.sleep(1000);
            log.info("消费者处理消息开始");
            if (event != null) {
                log.info("消费者消费的信息是: {}",event);
            }
        } catch (Exception e) {
            log.info("消费者处理消息失败");
        }
        log.info("消费者处理消息结束");
    }
}
```

5、构造BeanManager

```
/**
 * 获取实例化对象
 */
@Component
public class BeanManager implements ApplicationContextAware {

    private static ApplicationContext applicationContext = null;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws Be
        this.applicationContext = applicationContext;
    }

    public static ApplicationContext getApplicationContext() { return applicationConte

    public static Object getBean(String name) {
        return applicationContext.getBean(name);
    }

    public static <T> T getBean(Class<T> clazz) {
        return applicationContext.getBean(clazz);
    }
}
```

6、构造MQManager

```
@Configuration
public class MQManager {

    @Bean("messageModel")
    public RingBuffer<MessageModel> messageModelRingBuffer() {
        //定义用于事件处理的线程池， Disruptor通过java.util.concurrent.ExecutorSerivce提供的
        ExecutorService executor = Executors.newFixedThreadPool(2);
```

```
        //指定事件工厂
        HelloEventFactory factory = new HelloEventFactory();

        //指定ringbuffer字节大小，必须为2的N次方（能将求模运算转为位运算提高效率），否则将影响效率
        int bufferSize = 1024 * 256;

        //单线程模式，获取额外的性能
        Disruptor<MessageModel> disruptor = new Disruptor<>(factory, bufferSize, execu
                ProducerType.SINGLE, new BlockingWaitStrategy());

        //设置事件业务处理器---消费者
        disruptor.handleEventsWith(new HelloEventHandler());

        // 启动disruptor线程
        disruptor.start();

        //获取ringbuffer环，用于接取生产者生产的事件
        RingBuffer<MessageModel> ringBuffer = disruptor.getRingBuffer();

        return ringBuffer;
    }
}
```

7、构造Mqservice和实现类-生产者

```
public interface DisruptorMqService {

    /**
     * 消息
     * @param message
     */
    void sayHelloMq(String message);
}

@Slf4j
@Component
@Service
public class DisruptorMqServiceImpl implements DisruptorMqService {

    @Autowired
    private RingBuffer<MessageModel> messageModelRingBuffer;


    @Override
    public void sayHelloMq(String message) {
        log.info("record the message: {}",message);
        //获取下一个Event槽的下标
        long sequence = messageModelRingBuffer.next();
        try {
            //给Event填充数据
            MessageModel event = messageModelRingBuffer.get(sequence);
            event.setMessage(message);
            log.info("往消息队列中添加消息：{}", event);
        } catch (Exception e) {
            log.error("failed to add event to messageModelRingBuffer for : e = {},{}",
```

```
        } finally {
            //发布Event，激活观察者去消费，将sequence传递给改消费者
            //注意最后的publish方法必须放在finally中以确保必须得到调用；如果某个请求的sequence未被
            messageModelRingBuffer.publish(sequence);
        }
    }
}
```

8、构造测试类及方法

```
@Slf4j
@RunWith(SpringRunner.class)
@SpringBootTest(classes = DemoApplication.class)
public class DemoApplicationTests {

    @Autowired
    private DisruptorMqService disruptorMqService;
    /**
     * 项目内部使用Disruptor做消息队列
     * @throws Exception
     */
    @Test
    public void sayHelloMqTest() throws Exception{
        disruptorMqService.sayHelloMq("消息到了，Hello world!");
        log.info("消息队列已发送完毕");
        //这里停止2000ms是为了确定是处理消息是异步的
        Thread.sleep(2000);
    }
}
```

测试运行结果

```
2020-04-05 14:31:18.543  INFO 7274 --- [           main] c.e.u.d.d.s.Impl.DisruptorMqS
2020-04-05 14:31:18.545  INFO 7274 --- [           main] c.e.u.d.d.s.Impl.DisruptorMqS
2020-04-05 14:31:18.545  INFO 7274 --- [           main] c.e.utils.demo.DemoApplicatio
2020-04-05 14:31:19.547  INFO 7274 --- [pool-1-thread-1] c.e.u.d.disrupMq.mq.HelloEver
2020-04-05 14:31:19.547  INFO 7274 --- [pool-1-thread-1] c.e.u.d.disrupMq.mq.HelloEver
2020-04-05 14:31:19.547  INFO 7274 --- [pool-1-thread-1] c.e.u.d.disrupMq.mq.HelloEver
```

5、总结

其实 生成者 -> 消费者 模式是很常见的，通过一些消息队列也可以轻松做到上述的效果。不同的地方在于，Disruptor 是在内存中以队列的方式去实现的，而且是无锁的。这也是 Disruptor 为什么高效的原因。

JAVA
程序员追风
专注于分享Java各类学习笔记、面试题以及IT类资讯。
公众号

你在看吗

收录于合集 #Spring Boot 49

上一篇 · 拒绝繁琐，SpringBoot 拦截器与统一功能处理

喜欢此内容的人还喜欢

国产chatgpt这次彻底火了！免费使用！
程序员追风

60 个 Vue 常见问题汇总及解决方案
前端技术江湖

敏感数据的保护伞——SpringBoot集成jasypt
Java架构栈