二

29 生产者消费者模式：电商库存设计优化

你好，我是刘超。

生产者消费者模式，在之前的一些案例中，我们是有使用过的，相信你有一定的了解。这个模式是一个十分经典的多线程并发协作模式，生产者与消费者是通过一个中间容器来解决强耦合关系，并以此来实现不同的生产与消费速度，从而达到缓冲的效果。

使用生产者消费者模式，可以提高系统的性能和吞吐量，今天我们就来看看该模式的几种实现方式，还有其在电商库存中的应用。

Object 的 wait/notify/notifyAll 实现生产者消费者

在[第 16 讲]中，我就曾介绍过使用 Object 的 wait/notify/notifyAll 实现生产者消费者模式，这种方式是基于 Object 的 wait/notify/notifyAll 与对象监视器（Monitor）实现线程间的等待和通知。

还有，在[第 12 讲]中我也详细讲解过 Monitor 的工作原理，借此我们可以得知，这种方式实现的生产者消费者模式是基于内核来实现的，有可能会导致大量的上下文切换，所以性能并不是最理想的。

Lock 中 Condition 的 await/signal/signalAll 实现生产者消费者

相对 Object 类提供的 wait/notify/notifyAll 方法实现的生产者消费者模式，我更推荐使用 java.util.concurrent 包提供的 Lock && Condition 实现的生产者消费者模式。

在接口 Condition 类中定义了 await/signal/signalAll 方法，其作用与 Object 的 wait/notify/notifyAll 方法类似，该接口类与显示锁 Lock 配合，实现对线程的阻塞和唤醒操作。

我在[第 13 讲]中详细讲到了显示锁，显示锁 ReentrantLock 或 ReentrantReadWriteLock 都是基于 AQS 实现的，而在 AQS 中有一个内部类 ConditionObject 实现了 Condition 接口。

我们知道 AQS 中存在一个同步队列（CLH 队列），当一个线程没有获取到锁时就会进入到同步队列中进行阻塞，如果被唤醒后获取到锁，则移除同步队列。

除此之外，AQS 中还存在一个条件队列，通过 addWaiter 方法，可以将 await() 方法调用的线程放入到条件队列中，线程进入等待状态。当调用 signal 以及 signalAll 方法后，线程将会被唤醒，并从条件队列中删除，之后进入到同步队列中。条件队列是通过一个单向链表实现的，所以 Condition 支持多个等待队列。

由上可知，Lock 中 Condition 的 await/signal/signalAll 实现的生产者消费者模式，是基于 Java 代码层实现的，所以在性能和扩展性方面都更有优势。

下面来看一个案例，我们通过一段代码来实现一个商品库存的生产和消费。

```
public class LockConditionTest {

        private LinkedList<String> product = new LinkedList<String>();

        private int maxInventory = 10; // 最大库存

        private Lock lock = new ReentrantLock();// 资源锁

        private Condition condition = lock.newCondition();// 库存非满和非空条件

        /**
         * 新增商品库存
         * @param e
         */
        public void produce(String e) {
                lock.lock();
                try {
                        while (product.size() == maxInventory) {
                                condition.await();
                        }

                        product.add(e);
                        System.out.println(" 放入一个商品库存，总库存为: " + product.s
                        condition.signalAll();

                } catch (Exception ex) {
                        ex.printStackTrace();
                } finally {
                        lock.unlock();
                }
        }

        /**
         * 消费商品
         * @return
         */
        public String consume() {
                String result = null;
```

```
        lock.lock();
        try {
                while (product.size() == 0) {
                        condition.await();
                }

                result = product.removeLast();
                System.out.println(" 消费一个商品，总库存为：" + product.size
                condition.signalAll();

        } catch (Exception e) {
                e.printStackTrace();
        } finally {
                lock.unlock();
        }

        return result;
}

/**
 * 生产者
 * @author admin
 *
 */
private class Producer implements Runnable {

        public void run() {
                for (int i = 0; i < 20; i++) {
                        produce(" 商品 " + i);
                }
        }

}

/**
 * 消费者
 * @author admin
 *
 */
private class Customer implements Runnable {

        public void run() {
                for (int i = 0; i < 20; i++) {
                        consume();
                }
        }
}

public static void main(String[] args) {

        LockConditionTest lc = new LockConditionTest();
        new Thread(lc.new Producer()).start();
        new Thread(lc.new Customer()).start();
        new Thread(lc.new Producer()).start();
        new Thread(lc.new Customer()).start();
```

```
        }
}
```

看完案例，请你思考下，我们对此还有优化的空间吗？

从代码中应该不难发现，生产者和消费者都在竞争同一把锁，而实际上两者没有同步关系，由于 Condition 能够支持多个等待队列以及不响应中断， 所以我们可以将生产者和消费者的等待条件和锁资源分离，从而进一步优化系统并发性能，代码如下：

```
private LinkedList<String> product = new LinkedList<String>();
private AtomicInteger inventory = new AtomicInteger(0);// 实时库存

private int maxInventory = 10; // 最大库存

private Lock consumerLock = new ReentrantLock();// 资源锁
private Lock productLock = new ReentrantLock();// 资源锁

private Condition notEmptyCondition = consumerLock.newCondition();// 库存满
private Condition notFullCondition = productLock.newCondition();// 库存满和

/**
 * 新增商品库存
 * @param e
 */
public void produce(String e) {
        productLock.lock();
        try {
                while (inventory.get() == maxInventory) {
                        notFullCondition.await();
                }

                product.add(e);

                System.out.println(" 放入一个商品库存，总库存为：" + inventory

                if(inventory.get()<maxInventory) {
                        notFullCondition.signalAll();
                }

        } catch (Exception ex) {
                ex.printStackTrace();
        } finally {
                productLock.unlock();
        }

        if(inventory.get()>0) {
                try {
                        consumerLock.lockInterruptibly();
                        notEmptyCondition.signalAll();
                } catch (InterruptedException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                }finally {
```

```
                        consumerLock.unlock();
                }
        }

}

/**
 * 消费商品
 * @return
 */
public String consume() {
        String result = null;
        consumerLock.lock();
        try {
                while (inventory.get() == 0) {
                        notEmptyCondition.await();
                }

                result = product.removeLast();
                System.out.println(" 消费一个商品，总库存为: " + inventory.de

                if(inventory.get()>0) {
                        notEmptyCondition.signalAll();
                }
        } catch (Exception e) {
                e.printStackTrace();
        } finally {
                consumerLock.unlock();
        }

        if(inventory.get()<maxInventory) {

                try {
                        productLock.lockInterruptibly();
                        notFullCondition.signalAll();
                } catch (InterruptedException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                }finally {
                        productLock.unlock();
                }
        }
        return result;
}
/**
 * 生产者
 * @author admin
 *
 */
private class Producer implements Runnable {

        public void run() {
                for (int i = 0; i < 20; i++) {
                        produce(" 商品 " + i);
                }
        }
```

```
        }

        /**
         * 消费者
         * @author admin
         *
         */
        private class Customer implements Runnable {

                public void run() {
                        for (int i = 0; i < 20; i++) {
                                consume();
                        }
                }
        }

        public static void main(String[] args) {

                LockConditionTest2 lc = new LockConditionTest2();
                new Thread(lc.new Producer()).start();
                new Thread(lc.new Customer()).start();

        }
}
```

我们分别创建 productLock 以及 consumerLock 两个锁资源，前者控制生产者线程并行操作，后者控制消费者线程并发运行；同时也设置两个条件变量，一个是 notEmptyCondition，负责控制消费者线程状态，一个是 notFullCondition，负责控制生产者线程状态。这样优化后，可以减少消费者与生产者的竞争，实现两者并发执行。

我们这里是基于 LinkedList 来存取库存的，虽然 LinkedList 是非线程安全，但我们新增是操作头部，而消费是操作队列的尾部，理论上来说没有线程安全问题。而库存的实际数量 inventory 是基于 AtomicInteger（CAS 锁）线程安全类实现的，既可以保证原子性，也可以保证消费者和生产者之间是可见的。

BlockingQueue 实现生产者消费者

相对前两种实现方式，BlockingQueue 实现是最简单明了的，也是最容易理解的。

因为 BlockingQueue 是线程安全的，且从队列中获取或者移除元素时，如果队列为空，获取或移除操作则需要等待，直到队列不为空；同时，如果向队列中添加元素，假设此时队列无可用空间，添加操作也需要等待。所以 BlockingQueue 非常适合用来实现生产者消费者模式。还是以一个案例来看下它的优化，代码如下：

```
public class BlockingQueueTest {
```

```
private int maxInventory = 10; // 最大库存

private BlockingQueue<String> product = new LinkedBlockingQueue<>(maxInvent

/**
 * 新增商品库存
 * @param e
 */
public void produce(String e) {
        try {
                product.put(e);
                System.out.println(" 放入一个商品库存，总库存为：" + product.s
        } catch (InterruptedException e1) {
                // TODO Auto-generated catch block
                e1.printStackTrace();
        }
}

/**
 * 消费商品
 * @return
 */
public String consume() {
        String result = null;
        try {
                result = product.take();
                System.out.println(" 消费一个商品，总库存为：" + product.size
        } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

        return result;
}

/**
 * 生产者
 * @author admin
 *
 */
private class Producer implements Runnable {

        public void run() {
                for (int i = 0; i < 20; i++) {
                        produce(" 商品 " + i);
                }
        }

}

/**
 * 消费者
 * @author admin
 *
 */
private class Customer implements Runnable {
```

```
                public void run() {
                        for (int i = 0; i < 20; i++) {
                                consume();
                        }
                }
        }

        public static void main(String[] args) {

                BlockingQueueTest lc = new BlockingQueueTest();
                new Thread(lc.new Producer()).start();
                new Thread(lc.new Customer()).start();
                new Thread(lc.new Producer()).start();
                new Thread(lc.new Customer()).start();

        }
}
```

在这个案例中，我们创建了一个 LinkedBlockingQueue，并设置队列大小。之后我们创建一个消费方法 consume()，方法里面调用 LinkedBlockingQueue 中的 take() 方法，消费者通过该方法获取商品，当队列中商品数量为零时，消费者将进入等待状态；我们再创建一个生产方法 produce()，方法里面调用 LinkedBlockingQueue 中的 put() 方法，生产方通过该方法往队列中放商品，如果队列满了，生产者就将进入等待状态。

生产者消费者优化电商库存设计

了解完生产者消费者模式的几种常见实现方式，接下来我们就具体看看该模式是如何优化电商库存设计的。

电商系统中经常会有抢购活动，在这类促销活动中，抢购商品的库存实际是存在库存表中的。为了提高抢购性能，我们通常会将库存存放在缓存中，通过缓存中的库存来实现库存的精确扣减。在提交订单并付款之后，我们还需要再去扣除数据库中的库存。如果遇到瞬时高并发，我们还都去操作数据库的话，那么在单表单库的情况下，数据库就很可能会出现性能瓶颈。

而我们库存表如果要实现分库分表，势必会增加业务的复杂度。试想一个商品的库存分别在不同库的表中，我们在扣除库存时，又该如何判断去哪个库中扣除呢？

如果随意扣除表中库存，那么就会出现有些表已经扣完了，有些表中还有库存的情况，这样的操作显然是不合理的，此时就需要额外增加逻辑判断来解决问题。

在不分库分表的情况下，为了提高订单中扣除库存业务的性能以及吞吐量，我们就可以采用生产者消费者模式来实现系统的性能优化。

创建订单等于生产者，存放订单的队列则是缓冲容器，而从队列中消费订单则是数据库扣除库存操作。其中存放订单的队列可以极大限度地缓冲高并发给数据库带来的压力。

我们还可以基于消息队列来实现生产者消费者模式，如今 RabbitMQ、RocketMQ 都实现了事务，我们只需要将订单通过事务提交到 MQ 中，扣除库存的消费方只需要通过消费 MQ 来逐步操作数据库即可。

总结

使用生产者消费者模式来缓冲高并发数据库扣除库存压力，类似这样的例子其实还有很多。

例如，我们平时使用消息队列来做高并发流量削峰，也是基于这个原理。抢购商品时，如果所有的抢购请求都直接进入判断是否有库存和冻结缓存库存等逻辑业务中，由于这些逻辑业务操作会增加资源消耗，就可能会压垮应用服务。此时，为了保证系统资源使用的合理性，我们可以通过一个消息队列来缓冲瞬时的高并发请求。

生产者消费者模式除了可以做缓冲优化系统性能之外，它还可以应用在处理一些执行任务时间比较长的场景中。

例如导出报表业务，用户在导出一种比较大的报表时，通常需要等待很长时间，这样的用户体验是非常差的。通常我们可以固定一些报表内容，比如用户经常需要在今天导出昨天的销量报表，或者在月初导出上个月的报表，我们就可以提前将报表导出到本地或内存中，这样用户就可以在很短的时间内直接下载报表了。

思考题

我们可以用生产者消费者模式来实现瞬时高并发的流量削峰，然而这样做虽然缓解了消费方的压力，但生产方则会因为瞬时高并发，而发生大量线程阻塞。面对这样的情况，你知道有什么方式可以优化线程阻塞所带来的性能问题吗？

上一页 下一页