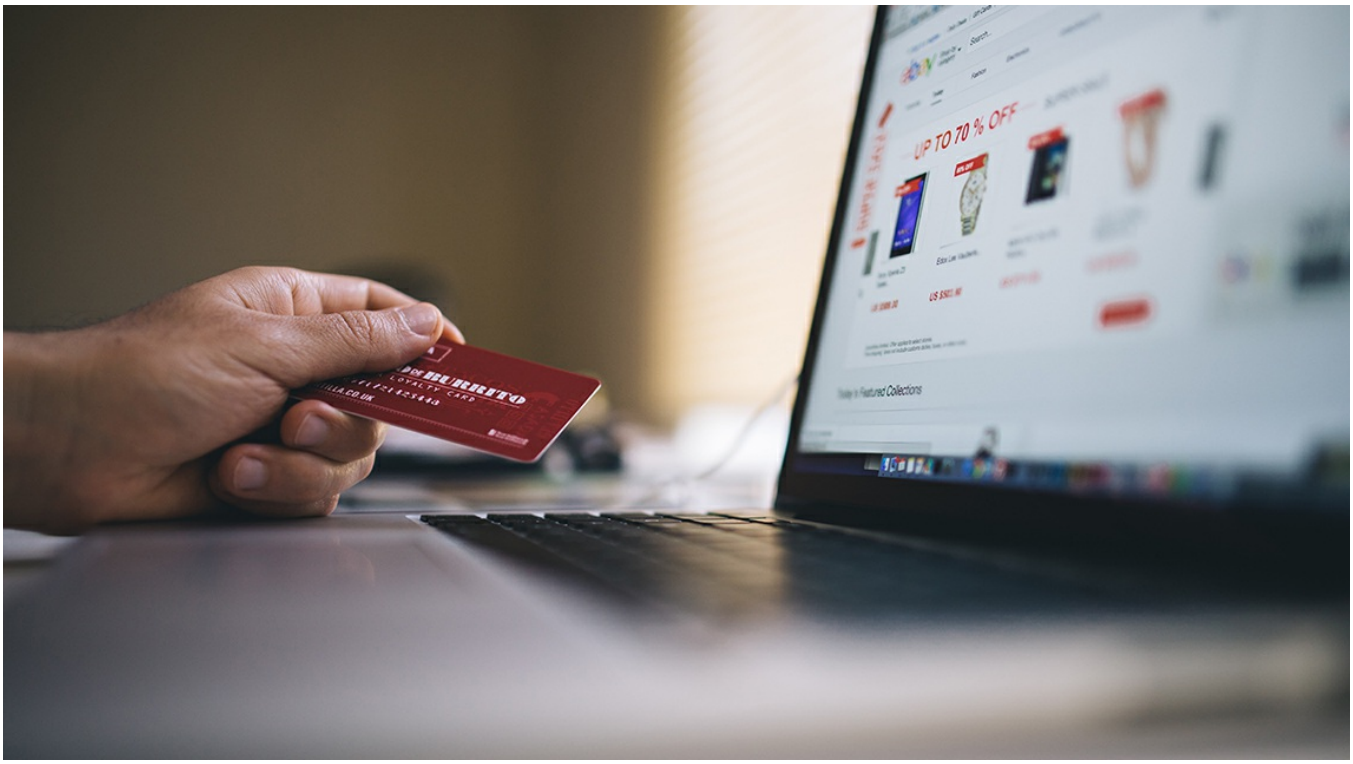
30 | 生产者消费者模式：电商库存设计优化

2019-07-30 刘超



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.size());
    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();// 库存满和空条件
private Condition notFullCondition = productLock.newCondition();// 库存满和空条件

/**
 * 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.incrementAndGet());

  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.decrementAndGet());

  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
```

```
 * @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<>(maxInventory);//缓存队列

 /**
  * 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.size());
  } 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？

期待在留言区看到你的见解。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。

精选留言

杨俊

5

我的理解是库存放缓存，用户提交订单在缓存扣减库存，用户端能够快速返回显示订单提交成

功并支付，然后只有支付成功之后才会利用队列实际的扣减数据库库存是吗？要是不支付会在缓存补回库存吧，应该会限时支付吧

2019-07-30

作者回复

对的

2019-07-31

QQ怪

1

在网关层中把请求放入到mq中，后端服务从消费队列中消费消息并处理；或者用有固定容量的消费队列的令牌桶，令牌发生器预估预计的处理能力，匀速生产放入令牌队列中，满了直接丢弃，网关收到请求之后消费一个令牌，获得令牌的请求才能进行后端秒杀请求，反之直接返回秒杀失败

2019-07-30

作者回复

大家都一致想到了限流，限流是非常必要的，无论我们的程序优化的如何，还是有上限的，限流则是一种兜底策略。

除了这个，我们还可以使用协程来优化线程由于阻塞等待带来的上下文切换性能问题，可以回顾第19讲，我们也用协程实现过生产者消费者模式。

2019-07-31

撒旦的堕落

1

网关与服务之间增加令牌桶 或者mq 以保护秒杀服务不会被大的流量压垮 可以么

2019-07-30

作者回复

可以的，很通用的一种解决方案

2019-07-31

2102

0

增加消费者

2019-10-19

作者回复

增加消费者是一种方式

2019-10-19

哲锄

0

LinkedList 的 add 和 removeLast 方法都有可能操作 first 引用，存在线程安全问题吧？

2019-09-23

作者回复

LinkedList是非线程安全容器，存在线程安全问题的

2019-10-02

风轻扬

0

lockInterruptibly()。老师，为啥要用这个API，不用lock。我查了一下，两者的区别是：前者侧重于中断，后者侧重于获取锁。这个地方，您是怎么考虑的呢?

2019-09-15

godtrue

0

课后思考及问题
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？
1：减少生产者的流量压力—限流
2：视业务场景而定判断是否可以拒绝部分多余流量
3：使用工业级消息队列中间件
4：加缓存
5：加机器

2019-09-12

Geek_844248

0

老师，优化ReentrantLock那里是不是有问题呢，product的修改放在两个不同的锁下，就是说可能会同时有两个线程会修改product这个list，这样是否违反了有序性。
而且我尝试无限循环运行生产者消费者线程，发现运行久了会出错的，希望老师讲解一下。

2019-08-12

K

0

老师好，我有个问题，就是实际的inventory会不会超过maxInventory啊？

```
productLock.lock();
try {
while (inventory.get() == maxInventory){ //3
notFullCondition.await();
}
product.add(e); //1

//producer被唤醒了以后，执行完1，还没执行2，这个时候时间片用完了，所以先停止了。
//然后另外的线程被唤醒了，在3处的判断逻辑，（上一个线程并没有inventory.incr()，所以while条件不满足，不循环）
//线程2号执行完代码1，2。
//当之前一个线程1号醒过来，他也会继续执行代码2。这不是相当于，实际的inventory 肯定超过了maxInventory吗？

System.out.println(" 放入一个商品库存，总库存为：" + inventory.incrementAndGet()); //2

//后面的逻辑
...
```

2019-08-11

作者回复

这有一个lock锁，不会同时进来两个线程。

2019-08-12

怎咖叻咖 0

老师，我看到你上面说用协程来优化！我们这边有个服务属于业务网关，要聚合多个下有的数据，涉及大量的网络io，之前是使用多线程并行调用多个下有，现在发现线程越来越多，遇到了瓶颈！希望用协程来改进方案～但是我在网上找到的一些java协程开源组件，文档和生态都不是很健全，希望老师能给出一些建议～ 非常感谢

2019-08-09

作者回复

建议再等等官方的协成组件，或改用go实现，目前Java的一些第三方开源组件的生产环境的实践以及性能验证有待考验，如果不介意当小白鼠，也可以试试这些第三方协成组件。

2019-09-08

Aaron 0

商品从数据库压入redis
缓存。
同时库存压入redis，用商品ID作为key，用list模拟队列【1001，1001，1001】用商品咖做队列元素，100件库存，那就存100个咖在队列中，扣库存的时候，判断队列大小，可以防止超卖。所有都是靠redis的单线程原子操作保证，可行不

2019-08-03

晓杰 0

请问老师在分布式架构中，使用lock和blockqueue实现生产者消费者是不是不适用了

2019-07-31

作者回复

是的，可以基于消息队列或redis缓存来实现。

2019-08-02

JasonK 0

你好，刘老师，最近生产上一个服务，老是半夜cpu飙升，导致服务死掉，排查问题看了下，都是GC task thread#15 (ParallelGC） 线程占用CPU资源，这是为什么？而且同样的服务我布了两台机器，一台服务会死掉，一台不会。请老师解惑。

2019-07-31

作者回复

导致CPU飙升只是一个性能的直接表现，是不是有对象一直在创建，所以导致一直在GC。建议打开dump日志查看具体的内存使用情况以及对象的创建分布情况。

2019-08-06

Jxin 0

1.生产消费模式用信号量也能玩。

2.生产者这边的优化思路应该是提高响应速度和增加资源。提高响应速度就是尽量降低生产逻辑的耗时，增加资源就是根据业务量为该生产者单独线程池并调整线程数。至于限流和令牌桶感觉都是降级处理，属于规避阻塞场景而非解决阻塞场景，应该不在答案范围内吧。
3.对于进程内生产消费模式，大规模，量大的数据本身就不适合，毕竟内存空间有限，消息堆积有限，所以量级达到一定指标就采用跨进程方式，比如kafka和rocketmq。同时，进程内生产消费模式，异常要处理好，不然可能会出现消息堆积和脏数据，毕竟mq的消费确认和重试机制都是开箱即用，而我们得自己实现和把关。

2019-07-31

作者回复

看来有实战经验

2019-08-02

我已经设置了昵称 0

kafka也有事务消息

2019-07-31

作者回复

是的

2019-08-02

nightmare 0

可以在网关通过令牌桶算法限流，真正执行的生产者一方使用线程池来优化

2019-07-30

作者回复

限流是一种方式，线程池其实也是一种限流手段。我们在之前协程这一讲中，其实也用协程代替线程实现了生产者消费者模式，这也不乏是一种优化方式。

2019-07-31

明翼 0

老师，生产者和消费者的锁分开没问题吗？都是用的同一个队列？

2019-07-30

作者回复

这里同步更新下，新增了以下代码作为实时库存：

```
private AtomicInteger inventory = new AtomicInteger(0);
```

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，而消费则是操作队列的尾部，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，即可以保证原子性，也可以保证消费者和生产者之间是可见的。

2019-07-31

正在减肥的胖籽。 0

1.如果把库存放到缓存中，下订单去缓存扣减库存数量，如何是保证数据一致性？希望老师能详细讲解下？

2019-07-30

作者回复

比较常用的手段就是使用分布式锁来实现，在40讲中我们会详细介绍。

2019-07-31

代码搬运工 0

对于文中的案例，可以用tryLock方法，给定一个超时时间，超过时间还未获取到锁，就返回一个错误提示信息。

2019-07-30

作者回复

嗯，可以通过超时来避免长时间阻塞等待。

2019-07-31

-W.LI- 0

课后习题:生产者也用MQ缓冲，在接入层做限流控制流量。客户端增加验证码等操作，防刷。服务器的计算能力，最大吞吐量是有限的。真要一直那么大量就只能加服务器了，只是瞬时就用MQ做流量削峰，或者提高用户门槛客户端限制减少无效请求(各种纬度进行控制)。只能想到这么点了

生产者消费者模型作用:

1.生产者和消费者解耦

2.通过缓冲，削峰，

3.生产者和消费吞吐量分别调控(生产者少只加生产者就好)

4....

2019-07-30

作者回复

思考的很全面

2019-07-31