译：Java 中生产者与消费者问题的演变

作者：liumapp
原文链接：https://ld246.com/article/1540175336971
来源网站：链滴

Java中生产者与消费者问题的演变

原文链接：https://dzone.com/articles/the-evolution-of-producer-consumer-problem-in-java

作者：Ioan Tinca

译者：liumapp

想要了解更多关于Java生产者消费者问题的演变吗？那就看看这篇文章吧，我们分别用旧方法和新方来处理这个问题。

生产者消费者问题是一个典型的多进程同步问题。

对于大多数人来说，这个问题可能是我们在学校，执行第一次并行算法所遇到的第一个同步问题。

虽然它很简单，但一直是并行计算中的最大挑战 - 多个进程共享一个资源。

问题陈述

生产者和消费者两个程序,共享一个大小有限的公共缓冲区。

假设一个生产者"生产"一份数据并将其存储在缓冲区中，而一个消费者"消费"这份数据，并将这份数从缓冲区中删除。

再假设现在这两个程序在并发地运行，我们需要确保当缓冲区的数据已满时，生产者不会放置新数据来，也要确保当缓冲区的数据为空时，消费者不会试图删除数据缓冲区的数据。

解决方案

为了解决上述的并发问题，生产者和消费者将不得不相互通信。

如果缓冲区已满，生产者将处于睡眠状态，直到有通知信息唤醒。

在消费者将一些数据从缓冲区删除后，消费者将通知生产者，随后生产者将重新开始填充数据到缓冲中。

如果缓冲区内容为空的化，那么情况是一样的，只不过，消费者会先等待生产者的通知。

但如果这种沟通做得不恰当，在进程彼此等待的位置可能导致程序死锁。

经典的方法

首先来看一个典型的Java方案来解决这个问题。

```
package ProducerConsumer;
import java.util.LinkedList;
import java.util.Queue;
public class ClassicProducerConsumerExample {
 public static void main(String[] args) throws InterruptedException { Buffer buffer = new Buffer
2); Thread producerThread = new Thread(new Runnable() { @Override public void run() { try {
buffer.produce(); } catch (InterruptedException e) { e.printStackTrace(); } } }); Thread consumer
hread = new Thread(new Runnable() { @Override public void run() { try { buffer.consume(); } c
```

```
tch (InterruptedException e) { e.printStackTrace(); } } }); producerThread.start(); consumerThre
d.start(); producerThread.join(); consumerThread.join(); } static class Buffer { private Queue list;
private int size; public Buffer(int size) { this.list = new LinkedList<>(); this.size = size; } public v
id produce() throws InterruptedException { int value = 0; while (true) { synchronized (this) { wh
le (list.size() >= size) { // wait for the consumer wait(); } list.add(value); System.out.println("Pro
uced " + value); value++; // notify the consumer notify(); Thread.sleep(1000); } } } public void
onsume() throws InterruptedException { while (true) { synchronized (this) { while (list.size() ==
0) { // wait for the producer wait(); } int value = list.poll(); System.out.println("Consume " + val
e); // notify the producer notify(); Thread.sleep(1000); } } } }}
```

这里我们有生产者和消费者两个线程，它们共享一个公共缓冲区。生产者线程开始产生新的元素并将们存储在缓冲区。如果缓冲区已满，那么生产者线程进入睡眠状态，直到有通知唤醒。否则，生产者程将会在缓冲区创建一个新元素然后通知消费者。就像我之前说的，这个过程也适用于消费者。如果冲区为空，那么消费者将等待生产者的通知。否则，消费者将从缓冲区删除一个元素并通知生产者。

正如你所看到的，在之前的例子中，生产者和消费者的工作都是管理缓冲区的对象。这些线程仅仅调了buffer.produce()和buffer.consume()两个方法就搞定了一切。

对于缓冲区是否应该负责创建或者删除元素，一直都是一个有争议的话题，但在我看来，缓冲区不应做这种事情。当然，这取决于你想要达到的目的，但在这种情况下，缓冲区应该只是负责以线程安全形式存储合并元素，而不是生产新的元素。

所以，让我们把生产和消费的逻辑从缓冲对象中进行解耦。

```
package ProducerConsumer;
import java.util.LinkedList;
import java.util.Queue;
public class ProducerConsumerExample2 {
 public static void main(String[] args) throws InterruptedException { Buffer buffer = new Buffer
2); Thread producerThread = new Thread(() -> { try { int value = 0; while (true) { buffer.add(va
ue); System.out.println("Produced " + value); value ++; Thread.sleep(1000); } } catch (Interrupt
dException e) { e.printStackTrace(); } }); Thread consumerThread = new Thread(() -> { try { whil
 (true) { int value = buffer.poll(); System.out.println("Consume " + value); Thread.sleep(1000); }
 catch (InterruptedException e) { e.printStackTrace(); } }); producerThread.start(); consumerThr
ad.start(); producerThread.join(); consumerThread.join(); } static class Buffer { private Queue lis
; private int size; public Buffer(int size) { this.list = new LinkedList<>(); this.size = size; } public
oid add(int value) throws InterruptedException { synchronized (this) { while (list.size() >= size)
 wait(); } list.add(value); notify(); } } public int poll() throws InterruptedException { synchronized
(this) { while (list.size() == 0) { wait(); } int value = list.poll(); notify(); return value; } } }}
```

这样好多了，至少现在缓冲区仅仅负责以线程安全的形式来存储和删除元素。

队列阻塞(BlockingQueue)

不过，我们还可以进一步改善。

在前面的例子中，我们已经创建了一个缓冲区，每当存储一个元素之前，缓冲区将等待是否有可用的个槽以防止没有足够的存储空间，并且，在合并之前，缓冲区也会等待一个新的元素出现，以确保存和删除的操作是线程安全的。

但是，Java本身的库已经整合了这些操作。它被称之为BlockingQueue，在这里可以查看它的详细文。

BlockingQueue是一个以线程安全的形式存入和取出实例的队列。而这就是我们所需要的。

所以,如果我们在示例中使用BlockingQueue，我们就不需要再去实现等待和通知的机制。

接下来，我们来看看具体的代码。

```
package ProducerConsumer;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingDeque;
public class ProducerConsumerWithBlockingQueue {
 public static void main(String[] args) throws InterruptedException { BlockingQueue blocking
ueue = new LinkedBlockingDeque<>(2); Thread producerThread = new Thread(() -> { try { int
value = 0; while (true) { blockingQueue.put(value); System.out.println("Produced " + value); va
ue++; Thread.sleep(1000); } } catch (InterruptedException e) { e.printStackTrace(); } }); Thread
onsumerThread = new Thread(() -> { try { while (true) { int value = blockingQueue.take(); Syst
m.out.println("Consume " + value); Thread.sleep(1000); } } catch (InterruptedException e) { e.pr
ntStackTrace(); } }); producerThread.start(); consumerThread.start(); producerThread.join(); con
umerThread.join(); }}
```

虽然runnables看起来跟之前一样，他们按照之前的方式生产和消费元素。

唯一的区别在于，这里我们使用blockingQueue代替缓冲区对象。

关于Blocking Queue的更多细节

这儿有很多种类型的BlockingQueue：

- 无界队列
- 有界队列

一个无界队列几乎可以无限地增加元素，任何添加操作将不会被阻止。

你可以以这种方式去创建一个无界队列：

```
BlockingQueue blockingQueue = new LinkedBlockingDeque<>();
```

在这种情况下,由于添加操作不会被阻塞,生产者添加新元素时可以不用等待。每次当生产者想要添加个新元素时，会有一个队列先存储它。但是，这里面也存在一个异常需要捕获。如果消费者删除元素速度比生产者添加新的元素要慢，那么内存将被填满，我们将可能得到一个OutOfMemory异常。

与之相反的则是有界队列，存在一个固定大小。你可以这样去创建它：

```
BlockingQueue blockingQueue = new LinkedBlockingDeque<>(10);
```

两者最主要的区别在于，使用有界队列的情况下，如果队列内存已满，而生产者仍然试图往里面塞元，那么队列将会被阻塞（具体阻塞方式取决于添加元素的方法）直到有足够的空间腾出来。

往blocking queue里面添加元素一共有以下四种方式：

- add() - 如果插入成功返回true，否则抛出IllegalStateException
- put() - 往队列中插入元素，并在有必要的情况下等待一个可用的槽(slot)
- offer() - 如果插入元素成功返回true，否则返回false
- offer(E e, long timeout, TimeUnit unit) – 在队列没有满的情况下，或者为了一个可用的slot而等指定的时间后，往队列中插入一个元素。

所以，如果你使用put()方法插入元素，而队列内存已满的情况下，我们的生产者就必须等待，直到有用的slot出现。

以上就是我们上一个案例的全部，这跟ProducerConsumerExample2的工作原理是一样的。

使用线程池

还有什么地方我们可以优化的？那首先来分析一下我们干了什么，我们实例化了两个线程，一个被叫生产者，专门往队列里面塞元素，另一个被叫做消费者，负责从队列里面删元素。

然而，好的软件技术表明，手动地去创建和销毁线程是不好的做法。首先创建线程是一项昂贵的任务每创建一个线程，意味着要经历一遍下面的步骤：

- 首先要分配内存给一个线程堆栈
- 操作系统要创建一个原生线程对应于Java的线程
- 跟这个线程相关的描述符被添加到JVM内部的数据结构中

首先别误会我，我们的案例中用了几个线程是没有问题的，而那也是并发工作的方式之一。这里的问是，我们是手动地去创建线程，这可以说是一次糟糕的实践。如果我们手动地创建线程，除了创建过中的消耗外，还有另一个问题，就是我们无法控制同时有多少个线程在运行。举个例子，如果同时有百万次请求线上服务，那么每一次请求都会相应的创建一个线程，那么同时会有一百万个线程在后台行，这将会导致thread starvation

所以，我们需要一种全局管理线程的方式，这就用到了线程池。

线程池将基于我们选择的策略来处理线程的生命周期。它拥有有限数量的空闲线程，并在需要解决任时启用它们。通过这种方式,我们不需要为每一个新的请求创建一个新线程，因此，我们可以避免出现程饥饿的问题。

Java线程池的实现包括：

- 一个任务队列
- 一个工作线程的集合
- 一个线程工厂
- 管理线程池状态的元数据

为了同时运行一些任务，你必须把他们先放到任务队列里。然后，当一个线程可用的时候，它将接收个任务并运行它。可用的线程越多，并行执行的任务就越多。

除了管理线程生命周期，使用线程池还有另一个好处，当你计划如何分割任务，以便同时执行时，你想到更多种方式。并行性的单位不再是线程了，而是任务。你设计一些任务来并发执行，而不是让一线程通过共享公共的内存块来并发运行。按照功能需求来思考的方式可以帮助我们避免一些常见的多程问题，如死锁或数据竞争等。没有什么可以阻止我们再次深入这些问题，但是，由于使用了功能范，我们没办法命令式地同步并行计算(锁)。这比直接使用线程和共享内存所能碰到的几率要少的多。我们的例子中，共享一个阻塞队列不是想要的情况，但我就是想强调这个优势。

在这里和这里你可以找到更多有关线程池的内容。

说了那么多，接下来我们看看在案例中如何使用线程池。

```
package ProducerConsumer;
import java.util.concurrent.BlockingQueue;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.LinkedBlockingDeque;
public class ProducerConsumerExecutorService {
 public static void main(String[] args) { BlockingQueue blockingQueue = new LinkedBlocking
eque<>(2); ExecutorService executor = Executors.newFixedThreadPool(2); Runnable producer
ask = () -> { try { int value = 0; while (true) { blockingQueue.put(value); System.out.println("Pr
duced " + value); value++; Thread.sleep(1000); } } catch (InterruptedException e) { e.printStac
Trace(); } }; Runnable consumerTask = () -> { try { while (true) { int value = blockingQueue.take
); System.out.println("Consume " + value); Thread.sleep(1000); } } catch (InterruptedException
) { e.printStackTrace(); } }; executor.execute(producerTask); executor.execute(consumerTask); e
ecutor.shutdown(); }}
```

这里的区别在于，我们不在手动创建或运行消费者和生产者线程。我们建立一个线程池，它将收到两任务，生产者和消费者的任务。生产者和消费者的任务，实际上跟之前例子里面使用的runnable是相的。现在，执行程序(线程池实现)将接收任务，并安排它的工作线程去执行他们。

在我们简单的案例下，一切都跟之前一样运行。就像之前的例子，我们仍然有两个线程，他们仍然要同样的方式生产和消费元素。虽然我们并没有让性能得到提升，但是代码看起来干净多了。我们不再动创建线程，而只是具体说明我们想要什么：我们想要并发执行某些任务。

所以，当你使用一个线程池时。你不需要考虑线程是并发执行的单位，相反的，你把一些任务看作并执行的就好。以上就是你需要知道的，剩下的由执行程序去处理。执行程序会收到一些任务，然后，会分配工作线程去处理它们。

总结

首先，我们看到了一个"传统"的消费者-生产者问题的解决方案。我们尽量避免了重复造没有必要的车，恰恰相反，我们重用了已经测试过的解决方案，因此，我们不是写一个通知等待系统，而是尝试使Java已经提供的blocking queue，因为Java为我们提供了一个非常有效的线程池来管理线程生命周，让我们可以摆脱手动创建线程。通过这些改进，消费者-生产者问题的解决方案看起来更可靠和更理解。