链滴

生产者消费者模式的四种实现方式

作者：valarchie

原文链接：https://ld246.com/article/1568942246113

来源网站：链滴

简述

生产者消费者模式简而言之就是两种不同的线程分别扮演生产者和消费者，通过一个商品容器来生产品和消费商品。生产者和消费者模式是学习多线程的好例子，下文就以四种不同实现的消费者生产者式来理解多线程的编程。

以下的例子都共用消费者和生产者对象，而将商品容器（Stock）按照四种形式进行实现。

生产者：

生产者持有商品容器，并实现了Runnable接口，在run方法中无限循环地往商品容器stock中放入商。

```
public class Producer implements Runnable{
    // 商品容器
    private Stock stock;

    public Producer(Stock stock) {
        this.stock = stock;
    }

    @Override
    public void run() {

        while (true) {
            // 随机生成商品 放入商品容器 stock中
            String product = "商品" + System.currentTimeMillis() % 100;
            System.out.println("生产了" + product);
            stock.put(product);
            // 休眠0.5秒
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }

    }
}
```

消费者：

消费者持有商品容器，并实现了Runnable接口，无限循环地从商品容器stock中取出商品消费。

```
public class Consumer implements Runnable {
    // 商品容器
    private Stock stock;

    public Consumer(Stock stock) {
        this.stock = stock;
    }
```

```
    @Override
    public void run() {

        while (true) {
            // 从商品容器中取出商品消费
            Object take = stock.take();
            System.out.println("消费了" + take);

            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }


        }

    }
}
```

商品容器Stock接口：

该接口主要定义了取出商品和放入商品两个方法供消费者和生产者使用，具体实现由不同子类提供。

```
public interface Stock {
    // 定义了容器的最大容量
    public static final int MAX = 10;
    // 取出商品
    String take();
    // 放入商品
    void put(String good);
}
```

Synchronized实现

该实现主要由synchronized、await、notify配合使用。

synchronized的语义大家应该都知道，当两个并发线程访问同一个对象object中的这个加锁同步代码时，一个时间内只能有一个线程得到执行。即同一时间内要么只有消费者执行take()方法，要么只有产者执行put()方法。

只有synchronized保证只有一个线程执行方法还不够，我们需要在容器空的时候，需要调用await()出锁进行等待，将执行权交给生产者生产商品，生产者生产完商品后再调用notify()方法通知消费者程消费商品（有可能唤醒的还是生产者，如果唤醒的是还是生产者就继续生产商品直到容器满，让出进行等待。）。反之亦然。

```
public class SynchronizedStock implements Stock {
    // 使用链表放置商品
    private LinkedList<String> productList = new LinkedList();

    public synchronized String take() {
```

```
        // 进入方法前先判断数组是否为空，为空的话释放锁进入阻塞状态
        while (productList.isEmpty()) {
            try {
                System.out.println("商品空了");
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // 取出商品
        String product = productList.pop();
        // 通知其他线程，有可能不是唤醒生产者线程
        notifyAll();

        return product;
    }

    public synchronized void put(String good) {
        // 进入方法前先判断数组是否已满，满的话释放锁进入阻塞状态
        while (productList.size() == MAX) {

            try {
                System.out.println("商品满了");
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
        // 放入商品
        productList.push(good);
        // 通知所有线程（生产者和消费者都有可能）
        notifyAll();
    }

}
```

解析：

有的朋友可能会疑惑为什么要使用while循环判断容器空或者满呢？笔者举个例子，假设我们用if判断组为空的话？消费者线程A先判断if条件为空，并进入了if代码块内进行了等待。接下来消费者线程B判断if条件为空，也进入到if代码块内进行了等待。这时候生产者线程C生产了一个商品，先唤醒了消者线程A，A唤醒后从if代码块内恢复执行，然后直接消费一个商品（此时容器空）。接下来可能唤醒消费者线程B，由于消费者线程B刚才也进入到了if代码块中（不会再判断一次if容器为空），此时直从代码块中恢复执行，消费商品时，发现容器中根本没有商品可以消费。所以如果条件用while进行断的话，在唤醒线程时，依然会判断容器是否为空。才能防止出错。

要点：

1. 在放入商品或取出商品时进行while条件判断，条件满足的话，进行等待。
2. 取出商品或者放入商品时通知其他线程。

ReentrantLock实现

该实现主要由ReentrantLock、以及notEmpty、notFull两个Condition来一起实现。Condition一样是用来阻塞等待线程。那为什么需要两个Condition呢？读者可以看看刚才的例子，使用notify的时可能会唤醒生产者和消费者。而两个Condition的话，我们可以在精准的控制唤醒，在消费者中唤醒产者，在生产者中唤醒消费者。

```
public class ReentrantLockStock implements Stock {
    // 使用链表来存放商品
    private LinkedList<String> productList = new LinkedList();
    // 执行take()和put()时需要的锁
    private Lock lock = new ReentrantLock();
    // 当调用notEmpty.signal()时，告诉生产者容器没空可以取商品
    private Condition notEmpty = lock.newCondition();
    // 当调用notFull.signal()时，告诉消费者者容器没满可以放入商品
    private Condition notFull = lock.newCondition();


    @Override
    public String take() {

        String good = null;

        try {
            // 获取锁才可以执行接下来的方法。
            lock.lock();
            // 当商品容器空时，notEmpty调用wait阻塞当前线程，表示现在容器空。
            while (productList.isEmpty()) {
                System.out.println("商品空了");
                notEmpty.await();
            }
            // 结束等待 获取商品
            good = productList.pop();
            // 通知生产者可以继续生产商品
            notFull.signalAll();

        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally {
            lock.unlock();
        }
        // 返回商品
        return good;

    }

    @Override
    public void put(String good) {

        try {
            // 获取锁才可以执行接下来的方法。
            lock.lock();
            // 当商品容器满时，notFull调用wait阻塞当前线程，表示现在容器满
            while (productList.size() == MAX) {
                System.out.println("商品满了");
                notFull.await();
```

```
            }
            // 结束等待时，放入商品
            productList.push(good);
            // 通知消费者可以继续消费
            notEmpty.signalAll();

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

    }

}
```

解析：

许多人会将lock放在try catch块外面，这样很容易出现死锁。因为lock锁和synchronized锁不一样。synchronized锁会自动释放锁。而lock不会自动释放锁，必须手工释放锁。如果lock放在try catch块之的话，持有锁后却发生了异常，此时并不会释放锁。其他线程就永远得不到这个锁了。

Semaphore实现

Semaphore是信号量的意思，信号量代表一张票，拥有了这张票你才能进行相应的操作。Semaphor的acquire()方法是阻塞获取信号量的方法。release()方法是添加信号量。我们使用只有唯一信号量的lck变量来模拟加锁解锁。用10个信号量的notFull变量模拟只可往容器里添加10个商品。当添加完一商品后，增加一个notEmpty的信号量，notEmpy有信号量之后才可以消费商品。

```
public class SemaphoreStock implements Stock {
    // 使用链表存放商品
    private LinkedList<String> goodList = new LinkedList();
    // 使用一个信号量模拟锁（只有一个线程可以使用容器）
    private Semaphore lock = new Semaphore(1);
    // 使用10个信号量模拟商品容器的最大容量
    private Semaphore notFull = new Semaphore(10);
    private Semaphore notEmpty = new Semaphore(0);

    @Override
    public String take() {

        String good = null;

        try {
            // 当notEmpty还有信号量的话 代表容器内有商品
            notEmpty.acquire();
            // 利用唯一的信号量模拟加锁
            lock.acquire();
            // 获取商品
            good = goodList.pop();

        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        } finally {
            // 释放唯一的信号量
            lock.release();
            // 消费完一个商品，往notFull中添加一个信号量
            notFull.release();

        }

        return good;
    }

    @Override
    public void put(String good) {

        try {
            // 当notFull还有信号量的话 代表容器还未满，可以放入商品
            notFull.acquire();
            // 利用唯一的信号量模拟加锁
            lock.acquire();
            // 放入商品
            goodList.push(good);

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 释放唯一的信号量
            lock.release();
            // 生产完一个商品，往notEmpty中添加一个信号量
            notEmpty.release();

        }

    }

}
```

解析：

使用信号量控制消费者和生产者协调时，不能先lock.acquire()，再notNull.acquire()。因为当lock.acquire()先得到信号量时，接着执行notNull.acquire()发现没有信号量，就阻塞等待并且没有释放刚才lock的信号量。导致程序进入死锁。所以一定要先获取生产或者消费的信号量，再使用lock的信号量。

BlockingQueue实现

我们直接使用ArrayBlockingQueue同步队列作为商品容器。该同步队列其实底层也是调用Reentrantock进行实现的。

```
public class BlockingQueueStock implements Stock {
    // 使用固定容量的arrayBlockingQueue同步队列放置商品
    private ArrayBlockingQueue<String> goods = new ArrayBlockingQueue<String>(10);

    @Override
```

```
    public String take() {

        String good = null;
        // 调用take阻塞获取商品
        try {
            good =  goods.take();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return good;
    }

    @Override
    public void put(String good) {

        try {
            goods.put(good);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

}
```

解析：

ArrayBlockingQueue主要有如下方法：

add、offer、put都是放入元素。

remove、poll、take都是移除元素。

element、peek是获取头元素，但不移除。

切记：put和take阻塞。

它们有不同形式

- 抛出异常：add() remove() element()
- 返回一个特殊值（null或false,具体取决于操作）： offer(e) poll() peek()
- 操作成功前，无限期地阻塞：put(e) take()
- 阻塞给定的时间：offer(e,time,unit) poll(time,unit)