Producer Consumer Problem using BlockingQueue

Neha Gupta
Nov 23, 2017 · 3 min read

In continuation of my previous blog, let's try to solve the Producer Consumer Problem using BlockingQueue. Ummm… Before proceeding with the problem itself, we shall have a look at BlockingQueue first.

BlockingQueue is an interface which extends Queue Interface (as the name suggests). The Producer Consumer design pattern has two major challenges i.e. Producer, producing to the full queue and consumer consuming from an empty queue. In the previous blog we were managing these situations using wait() and notifyAll() methods. Also, we had to manage the synchronize blocks as well. But, using the **Blocking queue** Interface we get rigid of these challenges.

BlockingQueue Interface has 5 implementations:

- ArrayBlockQueue
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue

Out of the above mentioned implementation ArrayBlockQueue, LinkedBlockingQueue and PriorityBlockingQueue require you to specify the fixed capacity while creating/declaring the queue.

So far, as per my experience, we have 4 methods helping best to manage the producer-consumer behavior, i.e. offer(), poll(), put() and take(). Although we've add() and remove() as well but they are of least help. Why!!! add() method returns true if the element/data are added successfully to the queue, throws an Exception only when

there's no space in the queue. remove() method has an optional restriction that it may or may not throw an exception, when the queue is empty (based on the collection).

put() method returns true if the element/data are added successfully to the queue, or it blocks until it is done adding; same applies for take().

The best to use are offer() and poll() but that also depends on what functionality you want to have exactly. offer() and poll() works have acouple of implementations, first one is similar to put() and take() but, another one provides you the time out functionality.

- add(o)/remove()
- put(o)/take()
- offer(o, timeout, timeunit)/poll(timeout, timeunit)

Coming back to our initial problem **producer-consumer** design pattern, mention below is the sample producer and consumer.

```
class Producer implements Runnable{

    private final BlockingQueue<Integer> queue;
    private final int SIZE;

    public Producer(BlockingQueue<Integer> queue, int size) {
        this.queue = queue;
        SIZE=size;
    }

    @Override
    public void run() {
        try{
            for(int i=0;i<SIZE;i++){
                System.out.println("Adding "+i+" object");
                System.out.println("The item is added
"+queue.offer(i));
                System.out.println(queue);
                Thread.sleep(3000);
            }
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
    }


}
```

SIZE and **queue** will be shared amongst both of them (i.e. the producer and the consumer). We'll be adding data to the queue counting the size.

Consumer will some what look like:

```
class Consumer implements Runnable{

    private final BlockingQueue<Integer> queue;
    private final int SIZE;

    public Consumer(BlockingQueue<Integer> queue,int size) {
        this.queue = queue;
        SIZE=size;
    }

    @Override
    public void run() {
        try{
            for (int i=0;i<SIZE;i++){
                System.out.println("Deleting object");
                System.out.println(queue);
                System.out.println("Able to delete "+queue.poll());
                Thread.sleep(2000);
            }
        } catch (Exception e){
            System.out.println(e.getMessage());
        }
    }

}
```

Notice in both the classes the sleep timings, the Producer will be producing little slowly as compared to consumer's consuming speed. This is being done in order to understand the waiting functionality using the BlockingQueue.

Here's the main class to start with the execution of the process:

```
public class ProducerConsumerBlockingQueue {
    public static void main(String[] args) throws
InterruptedException{
        System.out.println("-----------------------------------");
        System.out.println("starting the
factory.................");
        System.out.println("-----------------------------------");

        final int size=50;
        BlockingQueue<Integer> queue=new ArrayBlockingQueue<>(size);
```

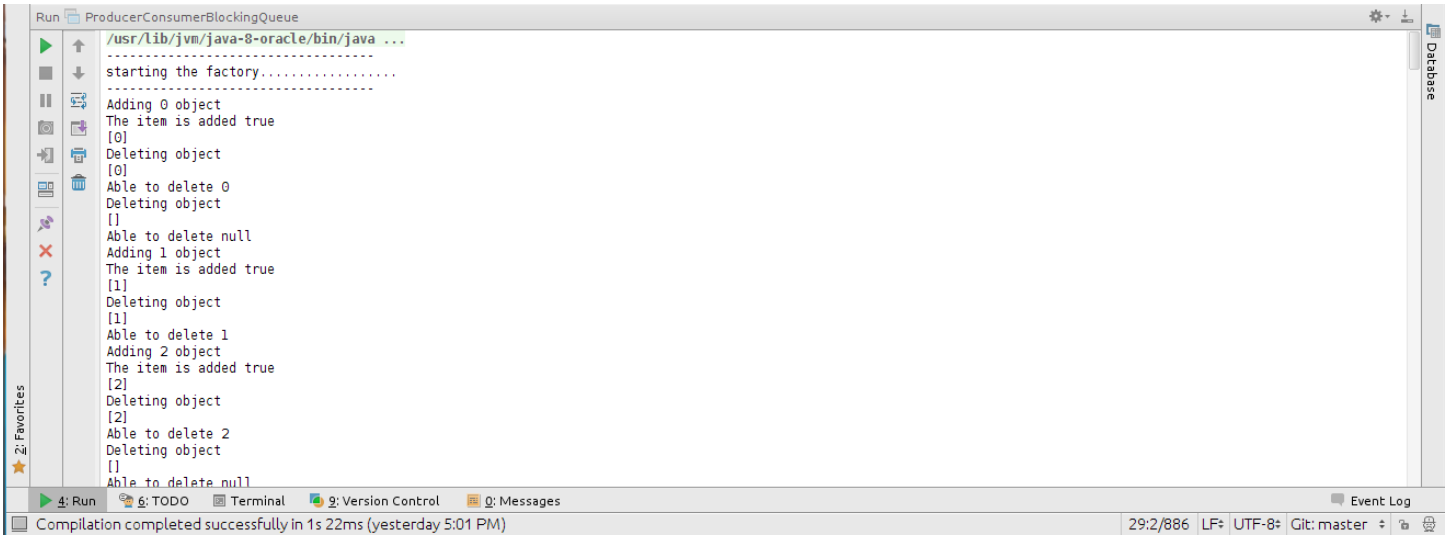
```
            Thread producerThread =new Thread(new Producer(queue,size));
            Thread consumerThread =new Thread(new Consumer(queue,size));

            producerThread.start();
            consumerThread.start();

            producerThread.join();
            consumerThread.join();

            System.out.println("-----------------------------------");
            System.out.println("End of the process/problem.");
            System.out.println("-----------------------------------");
        }
    }
```

This is how the output for this code snippet would look like, although I've just shared partial one as initial output was comparative comprehensive.



If this was of any help do share your comments, and follow me for my technical information. Do drop comments over for queries and improvements.

Cheers!!!

Java Multithreading Blocking Queue Producer Consumer Problem