Java Concurrency Tutorial – Blocking Queues

javacodegeeks.com Byron Kiourtzoglou September 18th, 2011 view original

As discussed in Part 3, the thread pools introduced in Java 1.5 provided core support that was quickly a favourite of many java developers.
Internally, the implementations make smart use of another concurrency feature introduced in java 1.5 – Blocking Queues.

Queue

First, a brief review of what a standard queue is. In computer science, a queue is simply a collection that always adds elements to the end and always takes elements from the beginning. The expression First-In-First-Out (FIFO) is commonly used to describe a standard queue. Introduced in java 1.6 is **Deque** or double-ended queue and this interface is now implemented on **LinkedList**. Some queues in java allow for alternate ordering, such as using a **Comparator** or even writing your own ordering implementation. While that extended functionality is nice, what we're focusing on today is how **BlockingQueues** really shine in concurrent development.

Blocking Queue

Blocking queues are queues that also expose functionality for blocking on requests to retrieve an element when no element is available with the additional option to limit the amount of time spent waiting. On a constrained size queue, this same blocking functionality is available when trying to add. Let's dive right in and consider an example of **BlockingQueue** usage.

Let's assume a simple scenario. You have a processing thread whose function is simply to execute commands.

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

private BlockingQueue<Command> workQueue = new
LinkedBlockingQueue<Command>();

public void addCommand(Command command) {
    workQueue.offer(command);
}

public Object call() throws Exception {
    try {
        Command command = workQueue.take();
        command.execute();
    } catch (InterruptedException e) {
        throw new WorkException(e);
    }
}
```

Granted, that is a very simple example, but it shows you the basics of using a **BlockingQueue** for multiple threads. Let's try something a little more involved. In this example, we have a need to create a connection pool with limit. It should only create connections as needed. No client will wait longer than 5 seconds for an available connection.

```
private BlockingQueue<Connection> pool = new
ArrayBlockingQueue<Connection>(10);
private AtomicInteger connCount = new AtomicInteger();

public Connection getConnection() {
    Connection conn = pool.poll(5, TimeUnit.SECONDS);
    if (conn == null) {
        synchronized (connCount) {
            if (connCount.get() < 10) {
```

```
                    conn = getNewConnection();
                    pool.offer(conn);
                    connCount.incrementAndGet();
                }
            }
            if (conn == null) {
                throw new ConnUnavailException();
            } else {
                return conn;
            }
        }
    }
```

Finally let's consider a sample usage of an interesting implementation, the **SynchronousQueue**.
In this example, similar to our first, we want to execute a **Command** but need to know when it is done, waiting at most 2 minutes.

```
    private BlockingQueue workQueue = new
    LinkedBlockingQueue();
    private Map commandQueueMap = new ConcurrentHashMap();

    public SynchronousQueue addCommand(Command command) {
        SynchronousQueue queue = new SynchronousQueue();
        commandQueueMap.put(command, queue);
        workQueue.offer(command);
        return queue;
    }

    public Object call() throws Exception {
        try {
            Command command = workQueue.take();
            Result result = command.execute();
            SynchronousQueue queue =
```

```
    commandQueueMap.get(command);
            queue.offer(result);
            return null;
        } catch (InterruptedException e) {
            throw new WorkException(e);
        }
    }
```

Now the consumer can safely poll with timeout on its request to have its Command executed.

```
    Command command;
    SynchronousQueue queue =
    commandRunner.addCommand(command);
    Result result = queue.poll(2, TimeUnit.MINUTES);
    if (result == null) {
            throw new CommandTooLongException(command);
    } else {
            return result;
    }
```

As you're starting to see, the **BlockingQueues** in java provide a lot of flexibility and give you relatively easy structures to serve many, if not all, of your needs in a multi-threaded application. There are some really neat **BlockingQueues** that we haven't even reviewed such as **PriorityBlockingQueue** and **DelayQueue**. Take a look at them and get in touch. We love talking shop with fellow developers.

Reference: Java Concurrency Part 5 – Blocking Queues from our JCG partners at the Carfey Software blog.