Theory: Concurrent queues

9 minutes 0 / 3 problems solved

One of the most popular kinds of concurrent collections is a **concurrent queue**. It is often used to organize some kind of communication between multiple threads within an application by exchanging some data (messages, tasks, unit of works, or something else). To achieve it, several threads should have a reference to a common queue and invoke its methods.

You already know that a queue is a collection that works according to the **first-in-first-out principle** (FIFO): the first element added to the queue will be the first one to be removed.

§1. Thread-safety of ConcurrentLinkedQueue

The simplest type of concurrent queue is `ConcurrentLinkedQueue` that is very similar to a standard queue but it is also **thread-safe**. It has two methods called `add` and `offer` to insert an element to the tail of a queue.

The following example demonstrates the thread-safety of this concurrent queue. The program adds new elements using two threads and then prints the total number of elements in this queue:

```
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

public class ConcurrentQueueDemo {

    public static void main(String[] args) throws InterruptedException {
        // assigning thread-safe implementation
        Queue<Integer> numbers = new ConcurrentLinkedQueue<>();


        Thread writer = new Thread(() -> addNumbers(numbers));

        writer.start();



        addNumbers(numbers); // add number from the main thread



        writer.join(); // wait for writer thread



        System.out.println(numbers.size()); // it prints 200000

    }



    private static void addNumbers(Queue<Integer> target) {

        for (int i = 0; i < 100_000; i++) {

            target.add(i);

        }

    }

}
```

It is not surprising, that this program always prints 200000 as expected, no element lost. You may start this program as many time as you need. So, `ConcurrentLinkedQueue` is really thread-safe. There is also no

Current topic:

Concurrent queues

Topic depends on:

- ✕ Queue and Stack
- ✕ Collections and thread-safety

Table of contents:

↑ Concurrent queues

§1. Thread-safety of ConcurrentLinkedQueue

§2. Communication between threads

§3. Composite operations

Feedback & Comments

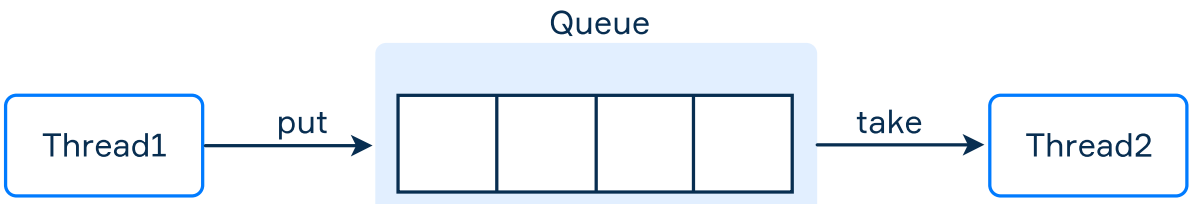
`ConcurrentModificationException` if we would like to iterate through this queue.

Note, that any single operation provided by this queue is thread-safe. However, if we group such operations together in a single method or a sequence of statements, the whole group of operations will not be thread-safe.

Moreover bulk operations of `ConcurrentLinkedQueue` that add, remove, or examine multiple elements, such as `addAll`, `removeIf`, `forEach` methods are *not* guaranteed to be performed atomically

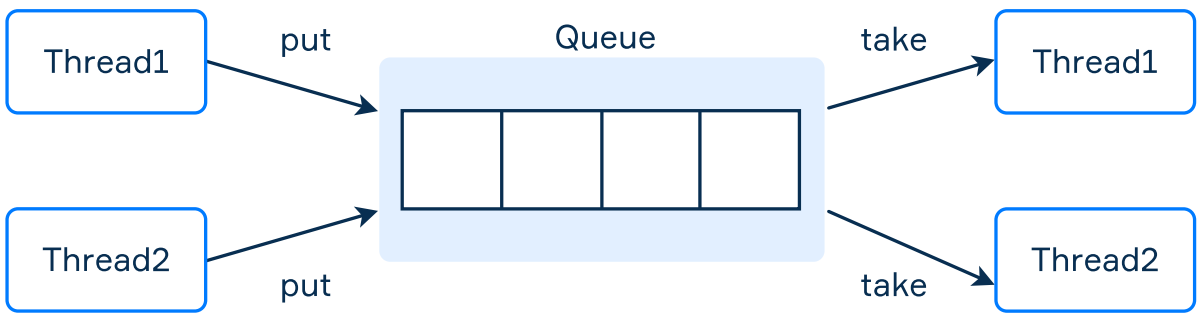
§2. Communication between threads

The following picture demonstrates how to organize communication between threads using a queue. One thread puts elements at the head of a queue, while another thread takes elements from the tail of the same queue.



We suppose that `Queue` is thread-safe, otherwise, it will not work correctly.

It is also possible when more than two threads are interacting through a queue.



The number of threads can be different.

Suppose we want to exchange data between two threads using a concurrent queue. One thread will generate three numbers while another thread will accept these numbers and print them. There is a method called `poll` used for getting the current first element of a concurrent queue. It returns an element or `null` if the queue is empty.

Here is a snippet of code with additional `sleep` invocations to make the output more predictable. The `generator` and `poller` interact using a concurrent queue and no data is lost because the queue is fully thread-safe.

```
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.TimeUnit;

public class GeneratorDemo {

    public static void main(String[] args) {
        Queue<Integer> queue = new ConcurrentLinkedQueue<>();


        Thread generator = new Thread(() -> {

            try {

                queue.add(10);

                TimeUnit.MILLISECONDS.sleep(10);

                queue.add(20);

                TimeUnit.MILLISECONDS.sleep(10);

                queue.add(30);

            } catch (Exception e) {

                e.printStackTrace();

            }

        });


        Thread poller = new Thread(() -> {

            int countRead = 0;

            while (countRead != 3) {

                Integer next = queue.poll();

                if (next != null) {

                    countRead++;

                }

                System.out.println(next);

                try {

                    TimeUnit.MILLISECONDS.sleep(10);

                } catch (Exception e) {

                    e.printStackTrace();

                }

            }

        });


        generator.start();

        poller.start();

    }

}
```

Here is an example of an output:

```
null
10
20
null
30
```

It may be slightly different but all numbers should be printed.

§3. Composite operations

Every standard method of a concurrent queue provides thread-safety. However, if you want to compose several methods together, there are no such guarantees.

Suppose, you want to add two elements in a concurrent queue so that they follow each other in this queue. Here is a method:

```
public static void addTwoElements(ConcurrentLinkedQueue<Integer> queue, int e1, in
t e2) {
        queue.add(e1); // (1)
        queue.add(e2); // (2)
    }
```

The method will add two elements one after the other only in case of one writing thread. If there are more writing threads, one thread may perform (1), and then another thread may intervene and do the same. Only after it, the first thread may perform (2). Thus, the order can be broken in some cases. This problem appears because the method is not **atomic**.

As mentioned above bulk methods such as `addAll` are also not atomic and don't help to avoid this problem

```
queue.addAll(List.of(e1, e2));
```

The problem can be solved only by external synchronization, e.g.

```
public static synchronized void addTwoElements(ConcurrentLinkedQueue<Integer> queu
e, int e1, int e2) {
        queue.add(e1); // (1)
        queue.add(e2); // (2)
    }
```

In that case, you need to be sure that all operations which update the queue should be synchronized, not only the method `addTwoElements`

Report a typo

46 users liked this theory. **1** didn't like it. **What about you?**

Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion