Laboratorio de programación

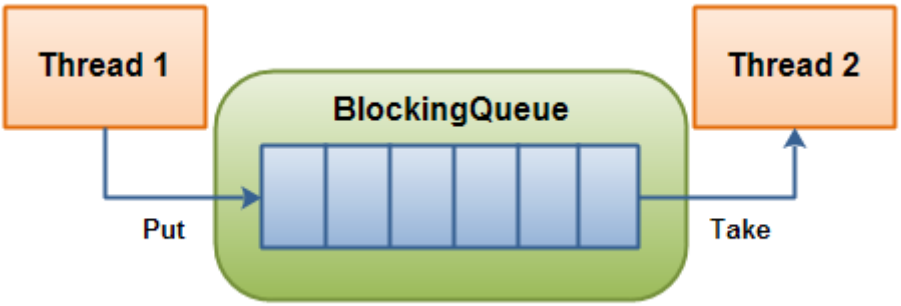
Colecciones Concurrentes

BlockingQueue, TransferQueue, CopyOnWriteArrayList

2015

¿Qué es una BlockingQueue?

BlockingQueue es una cola que es thread safe para insertar o recuperar elementos de la misma. Además, proporciona un mecanismo que bloquea las solicitudes de inserción de nuevos elementos cuando la cola está llena o las solicitudes de eliminación de elementos cuando la cola está vacía, con la opción adicional para dejar de esperar cuando pasa un tiempo específico.



En resumen, BlockingQueue es:

• Es una cola para comunicar procesos.

• Thread-safe.

• Amplia la interfaz Queue con operaciones bloqueantes.

BlockingQueue posee 4 tipos diferentes de métodos para insertar, eliminar y examinar los elementos en la cola. Estos se dividen en cuatro categorías, dependiendo de la manera de manejar las operaciones que no pueden ser atendidas de inmediato. Como por ejemplo, en los casos que el hilo intenta insertar un elemento en una cola llena o eliminar un elemento de una cola vacía. La primera categoría incluye los métodos que lanzan una excepción, la segunda categoría se incluyen los métodos que retornan un valor especial (por ejemplo, nula o falso), la tercera categoría está relacionada con los métodos que bloquean el hilo hasta que la operación puede llevarse a cabo, y por último, la cuarta categoría se incluyen los métodos que bloquean el hilo por un tiempo determinado.

Cada conjunto de métodos se comporta de forma diferente en caso de que la operación solicitada no puede llevarse a cabo inmediatamente. Aquí hay una tabla de los métodos:

	Throws Exception	Special Value	Blocks	Times Out
Insertar	`add()`	`offer()`	`put()`	`offer(timeout, timeunit)`
Eliminar	`remove()`	`poll()`	`take()`	`poll(timeout, timeunit)`
Examinar	`element()`	`peek()`		

No es posible insertar nulo en una BlockingQueue. Si intenta insertar nulo, la BlockingQueue lanzará una NullPointerException.

BlockingQueue es una interfaz, por lo que es necesario utilizar una de sus implementaciones para poder usarla. El paquete java.util.concurrent tiene las siguientes implementaciones de la interfaz BlockingQueue:

- ArrayBlockingQueue
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue
- LinkedBlockingDeque

ArrayBlockingQueue

ArrayBlockingQueue es una cola de bloqueo acotada, que almacena internamente los elementos en un array. Es acotada ya que utiliza un array para guardar los elementos, por lo que su tamaño es fijado al momento de la inicialización, y luego no puede ser cambiado. El ArrayBlockingQueue almacena los elementos internos en orden FIFO (First In, First Out).

Se utiliza en el ejemplo de los productores y consumidores para controlar a los productores. Ya que tiene una capacidad, los productores deberán de esperar en el caso que la cola este llena para continuar insertando elementos en la misma.

Para utilizarla se debe importar el paquete import java.util.concurrent.ArrayBlockingQueue.

Esta es la única que posee constructores que además de definir el tamaño, te da la opción con un boolean de poder mantener el orden de la espera. Es decir mantiene, si es que lo hay, el orden de llagada de los consumidores que están en espera.

LinkedBlockingQueue

LinkedBlocking es una cola que utiliza nodos (como una lista enlazada). Por la tanto no tiene un tamaño fijo. Y los elementos son almacenados en orden FIFO (First In, First Out).

Se utiliza en el ejemplo del productor consumidor para controlar a los consumidores. Ya que si no hay elementos en la cola estos se quedaran a la espera de la llegada de alguno.

Para utilizarla se debe importar el paquete import java.util.concurrent.LinkedBlockingQueue.

Esta posee si así se lo desea un constructor con el cual le podemos dar un tamaño fijo a la cola. Esto puede ser de gran ayuda en el caso que deseemos controlar a los productores ya que pasaría lo mismo que con el ArrayBlockingQueue.

PriorityBlockingQueue

PriorityBlockingQueue es una cola concurrente sin límites. Utiliza las mismas reglas de ordenación como la clase java.util.PriorityQueue. Todos los elementos insertados deben implementar la interfaz java.lang.Comparable. Con esto los elementos se podrán ordenar de acuerdo a su prioridad.

Para utilizarla se debe importar el paquete import java.util.concurrent.PriorityBlockingQueue.

Esta posee un constructor por el cual le podemos dar un tamaño y además también se lo puede crear con un comparador el cual dará el orden interno de la misma.

Ejemplo:

A continuación se demuestra un ejemplo de cómo se debe utilizar estas tres últimas clases

```
class Productor  implements Runnable{
private BlockingQueue cola;
private String nombre;
private int cant;
private int cantMaxima;
private int prioridad;

public void run() {
try {
    boolean band = true;
    while (band) {
       Produc val = new Produc(nombre + "-" + cant, prioridad);
       cant++;
       Thread.sleep(333);
       if (cant > cantMaxima)
             band = false;
       System.out.println("Creado producto N: " + (cant - 1) +" de productor " + nombre);
       cola.put(val); // aca pone un porducto en la BlockingQueue cada // 1.33 segundos
       System.out.println(cola.toString());         }
       } catch (Exception ex) {ex.printStackTrace();}
```

```
class Consumidor implements Runnable{
private BlockingQueue cola;
private String hiloNombre;

public void run(){
try{
   boolean band=true;
   int con=0;
   while(band){
    Thread.sleep(2333);
    consumir(cola.take()); //aca saca de la cola cuando se puede ya q tiene q esperar que
    halla un producto en la cola
    if(con>3)
       band=false;
    con++;
}   }
catch(InterruptedException ex){  ex.printStackTrace();}
}
private void consumir(Object x) throws InterruptedException{
System.out.println("Consumiendo "+hiloNombre+" producto N: "+x.toString());
Thread.sleep(1000); }
```

DelayQueue

DelayQueue utiliza nodos y los elementos que se almacenan en esta son bloqueados hasta que haya transcurrido un cierto tiempo de espera. Los métodos de BlockingQueue son aplicados de tal manera que solo pueden ser sacados de cola aquellos que su espera a caducado. Si el tiempo no ha expirado para cualquier elemento de la cola, entonces el método devolverá nulo. Cuando se devuelve un valor menor que o igual a cero ese elemento se considera que está caducado. Por lo tanto los elementos de DelayQueue solo pueden ser tomados cuando su espera ha expirado.

DelayQueue solo se pueden insertar los elementos que implementen la interfaz Delayed. Esta interfaz te obliga a poner en 2 métodos:

getDelay: este método devuelve la cantidad de tiempo que queda antes de que la espera se complete, este método es importante ya que es el que se utiliza para poder quitar los elemento de la cola. Si devuelve cero, el elemento se debe quitar de la cola.

compareTo: La interfaz Delayed utiliza la interfaz Comparable, este es para especificar cómo deben ser ordenados con respecto a otros objetos.

En resumen:

- ArrayBlockingQueue, con límite de capacidad.
- LinkedBlockingQueue, con límite opcional de capacidad
- DelayQueue, los elementos pueden retirarse cuando su delay ha expirado
- PriorityBlockingQueue, ordenados por prioridad
- SynchronousQueue, de capacidad 0 (cero)

¿Qué es una TransferQueue?

TransferQueue es una interfaz y un tipo de BlockingQueue. Esta se extiende desde la interfaz BlockingQueue añadiendo comportamiento a la misma.

TransferQueue es una BlockingQueue en la que los productores pueden esperar a que los consumidores reciban los elementos. En otras palabras TransferQueue es muy similar a la SynchronousQueue. En el sentido que el productor puede esperar la llegada de un consumidor. Al igual que con otras colas de bloqueo, una TransferQueue se puede limitar su capacidad. Si es así, una operación de poner en cola se puede bloquear a la espera de espacio disponible, y/o bloquear la espera para la recepción de un consumidor.

En otras palabras, cuando se utiliza BlockingQueue, sólo se puede poner elementos en cola (y bloquear si es que la cola está llena). Con TransferQueue además de esto último también se puede bloquear hasta que otro hilo recibe el elemento (para esto se debe utilizar nuevo método de transferencia).

En TransferQueue no hay ningún cambio en la parte del consumidor.

TransferQueue es una interfaz, por lo que es necesario utilizar la única implementaciones que posee.

LinkedTransferQueue:

LinkedTransferQueue es una TransferQueue ilimitada basada en nodos enlazados. Por la tanto no tiene un tamaño fijo. El orden en el que almacena es FIFO(primero en entrar, primero en salir).

Para utilizarla se debe importar el paquete java.util.concurrent.LinkedTransferQueue.

A continuación se detallan los métodos que posee una LinkedTransferQueue. Y no está demás recordar que además posee los métodos correspondientes a una BlockingQueue.

- *Transfer(obj):* Transfiere el elemento a un consumidor, pero si no hay ninguno a la espera el productor esperar a la llegada de uno y realizara la transferencia para luego seguir su camino.
- *tryTransfer(obj):* Transfiere el elemento a un consumidor si es que hay alguno a la espera. Caso contrario de no haber ningún consumidor en la cola, el productor pasara sin dejar nada en cola y seguirá con su recorrido. Retornara true si realizo la transferencia caso contrario devolverá false.
- *tryTransfer(obj, timeout, TimeUnit):* Idéntico al método tryTransfer(obj) pero este espera un tiempo específico, si es que no hay ningún consumidor a la espera, antes de continuar su marcha.
- *hasWaitingConsumer():* Devuelve un boolean si es que hay algún consumidor a la espera
- *getWaitingConsumerCount():* Devuelve un int lo cual es la cantidad de consumidores que están a la espera.

A continuación se demuestra un ejemplo de su utilización en la clase productor, ya que para la clase consumidor es igual a una blockingQueue

```
public class Productor implements Runnable{
	private String nombre;
	private LinkedTransferQueue cola;

	public Productor(String nom, LinkedTransferQueue co){
		nombre=nom;
		cola=co;
	}

	public void run(){
		String itemNom = "";
		int itemId = 0;
		try{
		    for (int x = 1; x < 8; x++){
			itemNom = "Item" + x;
			itemId = x;

			cola.transfer(new Item(itemNom, itemId));
			//cola.tryTransfer(new Item(itemName, itemId));

			System.out.println("creado: " + itemNom);
			Thread.sleep(250);
			if (cola.hasWaitingConsumer() == true) {//Devuelve true
si hay al menos un consumidor de espera para recibir un elemento
				System.out.println("mas rapido");
		}   }   }
		catch (InterruptedException ex) { ex.printStackTrace(); }
	  }
```

CopyOnWriteArrayList:

Esta es una clase q se extiende de object. Para utilizarla debemos importar la librería java.util.concurrent.CopyOnWriteArrayList. También utiliza interfaz lista, random access, serializable, clonable e iterator.

CopyOnWriteArrayList proporciona un acceso seguro y no es necesario ningún tipo de sincronización, y además mantiene las mismas operaciones que una lista común.

A su vez es muy poco eficiente en operaciones de escritura, pero muy rápida en operaciones de lectura. Ya que hacer una copia de un arraylist en java es muy costoso computacionalmente (desde el punto de vista del tiempo y de la memoria), por lo tanto es aconsejable usarla cuando el número de lecturas concurrentes sea mucho mayor que al número de escrituras.

CopyOnWriteArrayList utiliza una técnica la cual consta de hacer una copia de páginas de memoria cada vez que se realiza una modificación. Lo que garantiza que mientras nadie quiera modificar el estado de la estructura, la memoria será compartida, pero una vez que exista una modificación, los datos serán copiados de manera que se obtiene un iterador sobre dicha

estructura que es un snapshot (o muestra) de lo que tenía la estructura en el momento que se creo el iterador. Por lo tanto, los cambios que hagamos cuando recorremos con dicho iterador no serán reflejados en el iterador.

El metodo iterador utiliza una referencia al estado de la lista en el punto que se creo el iterador. Esta lista no cambia durante la vida útil del iterador, por lo que la interferencia es imposible y el iterador garantiza no largar la excepcion ConcurrentModificationException. El iterador no reflejará los cambios a la lista desde la creación del iterador.

•Demostración ejemplo

```
public class Lector implements Runnable {
    private String nombre;
    private CopyOnWriteArrayList lista;

    public Lector(String n, CopyOnWriteArrayList l) {
        this.lista = l;
        this.nombre = n;
    }
    public void run() {
        Iterator iterador;
            iterador = this.lista.iterator();
            System.out.println(nombre + " capturo el iterador ");
        while (iterador.hasNext()) {
            System.out.println(nombre + " esta leyendo el elemento: " +
iterador.next());
            try {
                Thread.sleep((long) (Math.random() * 1000));
            } catch (InterruptedException e) {         }
       }
        System.out.println(nombre + " termino de recorrer la lista ");    }
public class Modifica implements Runnable {
    private String nombre;
    private CopyOnWriteArrayList lista;
    private Object elem;
    private String tipoDeModificacion;

    public Modifica(String n, CopyOnWriteArrayList l, Object e, String t) {
        this.lista = l;
        this.nombre = n;
        this.elem = e;
        this.tipoDeModificacion = t;
    }

    public void run() {
        if (this.tipoDeModificacion.compareTo("agregar") == 0) {
            synchronized (this.lista) {
                this.lista.add(elem);
                System.out.println(nombre + " inserto al final de la lista
ele elemento: " + elem);   }
        } else {
            synchronized (this.lista) {
                if (!this.lista.isEmpty()) {//elimina por posicion
                    this.lista.remove(0);
                    System.out.println(nombre + " elimino del final de
lista");
            }    }
```

Bibliografía:

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TransferQueue.html

http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html

http://www.concretepage.com/java/example_delayqueue_java

Alumnos:

Agüero Jorge, Beroisa Jorge, Duran Alejandro