Objetos de Concurrencia de Alto Nivel

Blocking Queue

- Interfaz del paquete java.util.concurrent

Es una cola que adicionalmente soporta operaciones que esperan a que la cola deje de estar vacía para recuperar un elemento, y espera a que haya espacio disponible en la cola cuando se ingrese un elemento.

- Una BlockingQueue no acepta elementos nulos. Se arroja un NullPointerException cuando se intenta ingresar un elemento nulo.
- Las BlockingQueue pueden estar limitadas por su capacidad. En todo momento, la cola puede tener un **remainingCapacity** más allá del cual no se pueden agregar elementos.
- Una BlockingQueue sin restricciones de capacidad intrínsica siempre reportará un remainingCapacity igual a Integer.MAX_VALUE.

- Las implementaciones de BlockingQueue son diseñadas para ser principalmente usadas en colas de **productor-consumidor**.

- Es posible eliminar un elemento arbitrario de la cola haciendo uso de `remove( x )`. Sin embargo, esas operaciones no suelen ser realizadas eficientemente, y están pensadas para uso ocasional, e.g. cuando un mensaje de cola es cancelado.

- Todos los métodos de encolado consiguen sus efectos de forma atómica haciendo uso de bloqueos internos o alguna otra forma de control de concurrencia.

- Al igual que otras colecciones concurrentes, las acciones en un hilo antes de colocar un objeto en la BlockingQueue guarda una relación happens-before con las acciones subsecuentes al acceso o retiro de dicho elemento de la BlockingQueue en algún otro hilo.

Métodos de Blocking Queue

Haciendo uso del método int remainCapacity( ) podemos obtener la cantidad de elementos adicionales que la cola puede aceptar sin bloquearse, o Integer.MAX_VALUE si no hay límite intrínseco.

Los métodos de una BlockingQueue vienen de 4 formas, con diferentes formas de manejar las operaciones que no pueden ser realizadas al momento, pero si en un futuro:

- Arrojar una excepción.
  - Insertar: boolean add(E e)
    - Arroja un IllegalStateException cuando la cola se encuentra llena.
  - Remover: E remove( )
    - Arroja un NoSuchElement cuando la cola se encuentra vacía.
  - Examinar: E element( )
    - Arroja un NoSuchElement cuando la cola se encuentra vacía.

- Retornar un valor especial (null o false).
  - Insertar: boolean offer(E e)
    - Retorna true si el elemento fue agregado, o false si no hay espacio en la cola.
  - Remover: E poll( )
    - Retorna la cabeza de la cola, o null si la cola se encuentra vacía.
  - Examinar: E peek( )
    - Retorna la cabeza de la cola, o null si la cola se encuentra vacía.

- Bloquear el hilo de forma indefinida hasta concretar la operación.
  - Insertar: void put(E e)
  - Remover: E take( )

- Bloquear el hilo por un periodo de tiempo dado.
  - Insertar: boolean offer(E e, long timeout, TimeUnit unit)
    - Retorna true si se ingresa el elemento, o false si el tiempo se completa antes de encontrar espacio disponible.
  - Remover: E poll(long timeout, TimeUnit unit)
    - Retorna la cabeza de la cola, o null si el tiempo especificado vence antes de tener un elemento disponible.

Array Blocking Queue

- Clase del paquete java.util.concurrent

Es una Blocking Queue de espacio limitado que se encuentra respaldado por un array.

Este es un clásico “buffer limitado”, en el que un array de tamaño fijo mantiene los elementos insertados por los productores y extraídos por los consumidores.

Una vez creada, la capacidad no puede ser modificada. Cualquier intento de ingresar un elemento en una cola llena o de extraer de la cola vacía resultará en operaciones de bloqueo.

Callable

- Interfaz del paquete java.util.concurrent

La interfaz **Callable** implementa una tarea que retorna un resultado y podría arrojar una excepción.

Callable es similar a la interfaz Runnable, en el hecho que ambos son diseñados para clases cuyas instancias son potencialmente ejecutadas por otro hilo. Sin embargo, Runnable no permite retornar un resultado y no puede arrojar una excepción verificada.

Al igual que Runnable, al hacer uso de la interfaz Callable debemos definir un método que contenga la tarea a realizarse, **call**, el cual deberá tener un valor de retorno definido.

```
public class CallableExample implements Callable<Integer> {

    public Integer call() throws Exception {
        int returnVal;
        […]
        return returnVal;
    }

}
```

- El tipo de valor a retornar debe ser especificado entre “< … >” junto al llamado de la interfaz Callable.

- El método call no puede recibir argumentos.

- Para poder ejecutar una tarea definida por un objeto Callable debemos hacer uso de otras clases o interfaces definidas también en el paquete de concurrencia de java.

Future

➢Interfaz del paquete java.util.concurrent

Un **Future** representa el resultado de una computación asíncrona. Al igual que Callable, se debe especificar el tipo de resultado entre "<…>".

La interfaz proporciona los siguientes métodos:

- boolean cancel(boolean mayInterruptIfRunning)
  - Intenta cancelar la ejecución de esta tarea. Este intento fallará si la tarea ya está completada, ya ha sido cancelada, o no puede ser cancelada por alguna razón.
  - Retorna true en caso de éxito o false en caso contrario.
  - El argumento indica si la tarea debe ser interrumpida en caso esté en ejecución o si las tareas en progreso están permitidas a terminarse.
- V get( )
  - Espera a que se termine la computación de ser necesario, y recupera el resultado.

- V get(long timeout, TimeUnit unit)
  - Espera por a lo mucho el tiempo indicado para que la computación termine, y recupera el resultado si se encuentra disponible.
  - Arroja un TimeoutException si se agota el tiempo de espera.

- boolean isDone( )
  - Retorna true si la tarea ha sido completada, ya sea por terminación normal, una excepción, o cancelación.

- boolean isCancelled( )
  - Retorna true si la tarea fue cancelada antes de que haya sido completada de forma normal.

NOTAS:

1. Las acciones tomadas por la computación asíncrona guardan una relación happens-before con las acciones subsecuentes a la llamada del método get en algún otro hilo.
2. Si la computación arroja una excepción, los métodos arrojaran una **ExecutionException**.

Future Task

- Clase del paquete java.util.concurrent

Un **Future Task** es una computación asíncrona cancelable. La clase proporciona una implementación base de la interfaz Future y permite construirse sobre objetos Callable y Runnable. Además, dado que también implementan la interfaz Runnable, un Future Task puede se enviado a un **Executor** para su ejecución, además de crear un objeto Thread en base a un Future Task.

Para ejecutar un objeto Callable haciendo uso de un Future Task, podemos hacer lo siguiente:

```
CallableExample example  = new CallableExample( );

FutureTask<Integer> task = new FutureTask<Integer>(example);
Thread t = new Thread(task);
t.start( );
```

Executor

- Interfaz del paquete java.util.concurrent

En aplicaciones de gran escala, tiene más sentido separar el manejo y creación de hilos del resto de la aplicación, los **ejecutores** son objetos que encapsulan estas funciones. La interfaz **Executor** simplemente proporciona un objeto capaz de lanzar tareas Runnable.

La interfaz Executor proporciona un único método, **execute**, diseñado para ser un reemplazo directo para una expresión común de creación de hilo. Si r es un objeto Runnable, y e es un Executor, podemos reemplazar

```
new( Thread( r ) ).start( );
```

con

```
e.execute( r );
```

- La expresión de bajo nivel (haciendo uso de objetos Thread) crea un nuevo hilo y lo lanza inmediatamente.

- La interfaz Executor no requiere estrictamente que la ejecución sea asíncrona, el método execute podría simplemente ejecutar el objeto Runnable en el mismo hilo que fue llamado.

- Dependiendo de la implementación, el método execute podría:
  - Crear un nuevo hilo y lanzarlo de forma inmediata.
  - Usar un hilo trabajador existente para ejecutar el objeto Runnable.
  - Colocar el objeto Runnable en una cola en espera de un hilo trabajador disponible.

```
class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}

class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

Executor Service

- Interfaz del paquete java.util.concurrent

La interfaz **ExecutorService** suplementa execute con un método similar pero más versátil. Al igual que executes, **submit** acepta objetos Runnable, pero también acepta objetos Callable, lo cual permite que la tarea retorne un valor.

El método submit retorna un objeto Future, el cual es usado para recuperar el valor de retorno de Callable y administrar el estado de tanto las tareas Callable y Runnable.

Un ExecutorService puede ser apagado, lo que causará que rechaze nuevas tareas. Usando el método **shutdown** se permitirá que las tareas enviadas previamente se ejecuten antes de terminar, mientras que con **shutdownNow** evita que las tareas en espera se ejecuten e intenta detener las tares en ejecución.

Scheduled Executor Service

- Interfaz del paquete java.util.concurrent

La interfaz **ScheduledExecutorService** suplementa los métodos de su padre ExecutorService con **schedule**, el cual ejecuta tareas Runnable y Callable luego de un retardo especificado.

También se definen un par de métodos para ejecutar tareas específicas de forma repetida en intervalos definidos.

- scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)
  - Crea y ejecuta una acción periódica que inicia luego de un retraso inicial (initialDelay) y luego en initialDelay + period, initialDelay + 2*period, y así sucesivamente.
- ScheduleAtFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)
  - Crea y ejecuta una acción periodica que empieza luego de un retraso inicial y luego se ejecuta después de cierto retraso entre la terminación de una ejecución y el inicio de la siguiente.

Thread Factory

- Interfaz del paquete java.util.concurrent

La interfaz **ThreadFactory** proporciona un objeto que crea nuevos hilos en base a la demanda. El uso de fábricas de hilos elimina la necesidad de llamar a “new Thread”, permitiendo que las aplicaciones usen subclases especiales de hilos, prioridades, etc.

La implementación más sencilla de esta interfaz es:

```
class SimpleThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        return new Thread(r);
    }
}
```

Executors

➢Clase del paquete java.util.concurrent

La clase Executors proporciona métodos estáticos de fábrica y utilidad de las siguientes maneras:

- Métodos que crean y retornan un ExecutorService preparado con configuraciones usualmente útiles ya establecidas.
- Métodos que crean y retornan un ScheduledExecutorService preparado con configuraciones usualmente útiles ya establecidas.
- Métodos que crean y retornan un ThreadFactory que fija hilos recientemente creados a un estado conocido.
- Métodos que crean y retornan un Callable en base a acciones especificadas de tal forma que puedan ser usadas en métodos de ejecución.

Thread Pool

- Un Thread Pool representa un **conjunto de hilos trabajadores** que están **esperando** un trabajo y son **reutilizados** muchas veces.

- En el caso de un thread pool se crea un **grupo hilos de tamaño fijo**. Un hilo del pool es sacado y asignado a un trabajo por el proveedor de servicios. Luego de terminar el trabajo, el hilo es retornado al pool.

Ventaja: Mejora el redimiendo. Ahorra tiempo debido a que no es necesario crear un nuevo hilo.

➢ Son usados en los servlets y JSP donde el container debe crear un thread pool para procesar consultas.

La clase **Executors** cuenta con varios métodos para crear Thread Pools, entre los cuales tenemos:

- newFixedThreadPool(int nThreads)
  - Crea un Thread Pool que **reúsa una cantidad fija de hilo**s.
- newCachedThreadPool( )
  - Crea un Thread Pool que agrega hilos de ser necesario, pero reúsa los hilos previos que se encuentren libres.
- newSingleThreadExecutor( )
  - Crea un Executor que utiliza un solo hilo trabajador .

```
public class TestThreadPool {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5)
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.submit(worker);
          }
        executor.shutdown();
        while (!executor.isTerminated()) {   }

        System.out.println("Finished all threads");
    }
}
```

```
class WorkerThread implements Runnable {
    private String message;
    public WorkerThread(String s){
        this.message=s;
    }
     public void run() {
        System.out.println(Thread.currentThread().getName() + " (Start) message =
"+message);
        processmessage();
        System.out.println(Thread.currentThread().getName()+ " (End)");
    }
    private void processmessage() {
        try {  Thread.sleep(2000);  }
        catch (InterruptedException e) { e.printStackTrace(); }
    }

}
```

Nota 1: Todos estos métodos retornan un objeto ExecutorService por lo que debemos usar una instancia de esta clase para almacenar un Thread Pool.

Nota 2: Cada uno de estos métodos también puede recibir un ThreadFactory de forma adicional para generar sus hilos en base dicho objeto.

Nota 3: Hay versiones similares a estos métodos en la clase Executors que están definidos para generar ScheduledExecutorService

Si ninguno de los ejecutores proporcionados por estos métodos satisface nuestas necesidades, existen dos clases con opciones adicionales del paquete java.util.concurrency :

- ThreadPoolExecutor
- ScheduledThreadPoolExecutor

Pero por lo general solo se usa los métodos ya presentados debido a su simplicidad de uso.

Semaphore

➢Clase del paquete java.util.concurrent

Un semáforo mantiene un **conjunto de permisos** y proporciona principalmente dos métodos:

- **acquire**: bloquea de ser necesario hasta que haya un permiso disponible y luego lo toma.
- **release**: agrega un permiso, potencialmente liberando uno adquirido previamente.

Sin embargo, no se requiere hacer uso de algún objeto permiso, el **Semaphore** sólo mantiene un conteo de la cantidad disponible y actúa en base a ello.

Los semáforos son usualmente usados para restringir la cantidad de hilos que pueden acceder a algún recurso.

Hay dos constructores para un Semaphore:

- semaphore(int permits)
- semaphore(int permits, boolean fair)

El entero permints especifica la **cantidad de permisos iniciales** con los que contará (el valor puede ser negativo), y de forma opcional, el booleano fair será true si se quiere que los permisos sean concedidos en first-in-first-out.

NOTA:

Por lo general, los semáforos usados para controlar el acceso a recursos deben ser inicializados deben ser inicializados como justos (fair) para asegurarnos que **ningún hilo sea privado de acceder al recursos** a causa de starvation.

Thread Group

Un **ThreadGroup** representa un conjunto de hilos capaz de contener incluso a otro grupo de hilos. El grupo de hilos crea un árbol en el cual cada grupo de hilos excepto el hilo inicial tiene un padre.

Un hilo tiene permitido acceder a la información de su propio grupo, pero no puede acceder a la información del padre de su grupo o algún otro grupo de hilos.

Constructores:

- ThreadGroup(String name)
- ThreadGroup(ThreadGroup parent, String name)

Al momento de instanciar un grupo, se puede indicar el padre del grupo. Si no se especifica un padre, se toma al grupo del hilo en ejecución que realizó la llamada al constructor como padre.

```
public class ThreadGroupDemo implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        ThreadGroupDemo runnable = new ThreadGroupDemo();
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

        Thread t1 = new Thread(tg1, runnable,"one");
        t1.start();
        Thread t2 = new Thread(tg1, runnable,"two");
        t2.start();
        Thread t3 = new Thread(tg1, runnable,"three");
        t3.start();

        System.out.println("Thread Group Name: "+tg1.getName());
        tg1.list();

    }
}
```