Concurrencia

# Temas

* Usar las colecciones java.util.concurrent
* Aplicar variables atómicas y bloqueos
* Usar Executors y ThreadPools
* Usar el framework paralelo Fork/Join

Desde el principio, java soporta concurrencia en la forma de manejo de bajo nivel de threads, bloqueos, sincronización y APIs para concurrencia. Cubrimos estas en el capítulo anterior en nuestra discusión sobre la clase Thread, la interface Runnable y la palabra clave synchronized.

Desde java 5.0 también soporta concurrencia de alto nivel en el paquete java.util.concurrent. En este capítulo nos enfocaremos en esta API de programación concurrente. Esta aplicación de alto nivel explota hoy en día hardware multi-core, en donde un único procesador puede tener varios núcleos. Esta API también son útiles para explotar concurrencia en máquinas que soportan varios procesadores.

Muchas de las utilidades de java para concurrencia son provistas en el paquete java.util.concurrent. Las clases para actualizar de manera eficiente variables compartidas sin usar bloqueos son provistas en el paquete java.util.concurrent.atomic. La interface Lock y sus clases derivadas son provistas en el paquete java.util.concurrent.locks.

# Usar las colecciones java.util.concurrent

Hay muchas clases en el paquete java.util.concurrent que proveen APIs para programación concurrente. En esta sección vamos a revisar demasiado las clases synchronizer provistas en el paquete. Seguido a esto, cubriremos clases importantes de la colección concurrente provistas en el paquete java.util.concurrent.

Ahora entiendes la concurrencia en bajo nivel (como el uso de la palabra clave synchronized, la interface Runnable y la clase Thread para crear Threads) del capítulo anterior. En el caso de recursos compartidos que necesitan ser utilizados por múltiples threads, las modificaciones deben ser protegidas. Cuando usas la palabra synchronized empleas exclusión mutua para sincronizar threads para un acceso seguro. Los threads también necesitan coordinar sus ejecuciones para completar una tarea más grande y de alto nivel. El patrón wait/notify discutido en el último capítulo, es una forma de coordinar la ejecución de múltiples threads.

Usando APIs para adquirir y liberar bloqueos (usando exclusión mutua) o invocar los métodos wait/notify en bloqueos son tareas de bajo nivel. Es posible construir abstracciones de más alto nivel para sincronización de threads. Esta abstracción es conocida como synchronizers, los cuales internamente hacen uso de las APIs de bajo nivel para la coordinación de threads.

Los synchronizers provistos en la librería java.util.concurrent y sus usos están listados aquí.

1. **Semaphore** controla el acceso a uno o más recursos compartidos.
2. **Phaser** es usado para soportar una barrera de sincronización.
3. **CountDownLatch** permitea los threads esperar a que una cuenta regresiva se complete.
4. **Exchanger** soporta intercambio de datos entre dos threads.
5. **CyclidBarrier** habilita a los threads a esperar en un punto de ejecución específico.

## Semaphore

Un semáforo controla el acceso a recursos compartidos. Un semáforo mantiene un contador que especifica el número de recursos que el semáforo controla. Acceder a los recursos está permitido si el contador es mayor a cero, mientras que cero indica que no hay recursos disponibles en ese momento, por lo que el acceso es denegado.

Los métodos adquire() y relase() son para adquirir y liberar recursos de un semáforo. Si un thread llama a adquire() y el contador es cero (recursos no disponibles), el thread espera hasta que el contador no sea cero y obtenga el recurso para usar. Ya que el thread termina de usar el recurso, llama a relase() para incrementar el contador de disponibilidad de recursos.

Nota que si el número de recursos es 1, entonces, solo un thread a la vez puede acceder al recurso; en este caso, usar el semáforo es similar a usar un bloqueo. La siguiente tabla muestra importantes métodos en la clase Semaphore.

|  |  |
| --- | --- |
| Método | Descripción |
| Semaphore(int permits) | Constructor para crear un objeto Semaphore con un número dado de permisiones (el número de threads que pueden acceder al recurso al mismo tiempo). Si el valor de permits es negativo, da el número de llamadas a relase() que deben pasar antes de que adquire() sea exitoso. |
| Semaphore(int permits, boolean fair) | Igual que el previo, pero con una opción fair que indica si los permits pueden ser ordenados en la base primero en entrar-primero en servir. |
| void acquire()  void acquire(int permits) | Adquiere un permit si está disponible; de otra manera se bloquea mientras el permit se vuelva disponible. Puede lanzar una InterruptedException si algún otro thread interrumpe mientras espera para adquirir el permit. Las versiones sobrecargadas toman el número de permisos como argumento. |
| void acquireUninterruptibly() | Igual que el anterior, pero este thread no puede ser interrumpido mientras espera para adquirir el permit. |
| boolean tryAcquire()  boolean tryAcquire(long, TimeUnit) | Adquiere un permit del semáforo si está disponible en el momento de la llamada y retorna true; si no está disponible retorna false inmediatamente (sin bloquear). El sobrecargado tryAcquire() adicionalmente toma un argumento timeout para bloquearse por un tiempo mientras adquiere el bloqueo, hasta que el tiempo se acaba. |
| void relase()  void relase(int permits) | Libera permits del semáforo. La versión sobrecargada especifica el número de permits a liberar. |

Vamos a asumir que hay dos cajeros automáticos disponibles en un cuarto. Por lo tanto, solo dos personas están permitidas al mismo tiempo en el cuarto. Aquí hay cinco personas esperando afuera para usar los cajeros. La situación puede ser simulada por el siguiente código, en donde cada cajero es tratado como un recurso controlado por el semáforo.

[ATMRoom.java](src/java/oca2/cap14/ATMRoom.java)

Ahora vamos a analizar cómo trabaja este programa. La gente espera a acceder al cajero que está simulada creando una clase Persona que extiende a Thread. El método run() en la clase Thread adquiere un semáforo, simula el tiempo de estancia en el cajero y libera el semáforo.

El método main() simula el cuarto ATM con dos cajeros creando un objeto Semaphore con dos permits. La gente espera en la cola para acceder al cajero está implementado solo añadiendo a ellas a un objeto semáforo.

Como puedes ver de la salida del programa, el semáforo permito solo dos threads a la vez y los otros threads esperan. Cuando un thread libera el semáforo, otro thread lo adquiere. Cool ¿no crees?

## CountDownLatch

Este sincronizador permite a uno o más threads esperar un tiempo específico. Esta cuenta regresiva puede ser para un conjunto de eventos que pases o mientras un conjunto de operaciones es ejecutado en orden de que los threads se completen. La siguiente tabla muestra métodos importantes en esta clase.

|  |  |
| --- | --- |
| CountDownLatch(int count) | Crea una instancia de CountDownLatch con el número de tiempos que el método countDown debe ser llamado antes de que los threads que esperan con await() puedan continuar su ejecución. |
| void await() | Si la cuenta regresiva es cero inmediatamente retorna; de otra manera el thread se bloquea mientras la cuenta regresiva alcanza cero. Puede lanzar una InterruptedException. |
| boolean await(long timeout , TimeUnit unit) | Igual que el método previo, pero toma un argumento time-out adicional. Si el thread retorna exitosamente después de que el count alcance cero, este método retorna true; si el thread retorna por el time-out, este retorna false. |
| void countDown() | Reduce el número de cuentas por uno en el este objeto. Si la cuenta alcanza cero, todos los threads que están esperando son liberados. Si el contero ya es cero, no pasa nada. |
| long getCount() | Retorna las cuentas pendientes del objeto CountDownLatch |

Cuando creas un objeto CountDownLatch, lo inicializas con un entero, que representa un contero. Los thread deben esperar (llamando al método wait()) para que esta cuenta llegue a cero. Ya que cero es alcanzado, todos los threads esperando se liberan; cualquier llamada a await() cuando el contero es cero retornarán inmediatamente. El valor del contero puede decrementar en uno llamando al método countDown(). Puedes obtener el valor actual del contero usando el método getCoutn(). Observa el siguiente ejemplo.

[RunningRaceStarter](src/java/oca2/cap14/RunningRaceStarter.java).java

Vamos a analizar cómo trabaja el programa. La clase Runner simula un corredor en una carrera esperando empezar a correr llamando al método await() en el objeto CountDownLatch pasado a través del constructor.

La clase RunningRaceStarter crea un objeto CountDownLatch. Este contero es inicializado con el valor de 5, que significa que la cuenta regresiva será de 5 a 0. En el método main() creas objetos Runner; estos tres threads esperan en el objeto counter. Por cada segundo, llamas al método countDown() que decrementa la cuenta por 1. Ya que esta llega a cero, los tres threads esperando se liberan y automáticamente continúan su ejecución.

**NOTA**: en este programa, la secuencia en la que Joe, Carl o Jack es impresa no puede ser predicha, ya que esto depende de la programación de threads. Entonces, si corres este programa, seguirás obteniendo estos tres nombres en diferente orden.

## Exchanger

La clase Exchanger está diseñada para intercambiar datos entre dos threads. Lo que Exchanger hace de esto tan sencillo: este espera hasta que ambos threads llamen al método Exchange(). Cuando hagan eso, el objeto Exchanger intercambia los datos compartidos por los threads con cada uno. Esta clase es útil cuando dos threads necesitan sincronizarse entre ellos e intercambiar datos continuamente.

Esta clase es pequeña, ya que solo contiene un método: exchange(). Nota que este método tiene un sobrecargado que toma un timeout como argumento.

El siguiente programa simula una conversación entre la mascota de java Duke y una tienda de café. Los dos threads DukeThread y CoffeShop corren independientemente. Sin embargo, para que pase un chat, necesitan escuchar lo que el otro está diciendo. Un objeto Exchange provee maneras de hacer esto.

[KnokKnok.java](src/java/oca2/cap14/KnokKnok.java)

El comentario dentro del programa explica cómo el programa trabaja. El principal concepto para entender es que el Exchanger ayuda a coordinar (o sincronizar) mensajes entre dos threads. Ambos de los trheads esperan a que el otro use el método exchange() para intercambiar datos.

## CyclicBarrier

Hay muchas situaciones en la programación concurrente donde los threads pueden necesitar por esperar un punto de ejecución predefinida hasta que los otros threads la alcancen. CyclicBarrier ayuda proveyendo un punto de sincronización; observa la siguiente tabla para analizar los métodos importantes en esta clase.

|  |  |
| --- | --- |
| Método | Descripción |
| CyclicBarrier(int numThreads) | Crea un objeto CyclidBarrier con el número de threads esperando.  Lanza una excepción IllegalArgumentException si el número de threads es negativo o cero. |
| CyclicBarrier(int parties, Runnable barrierAction) | Igual que el anterior pero adicional toma el thread a llamar cuando la barrera sea alcanzada. |
| int await() | Bloquea mientras el número especificado de threads han llamado a await() en esta barrera. El método retorna el índice que viene en este thread. Este método puede lanzar una InterruptedException si el thread es interrumpido mientras espera a los otros threads; BrokenBarrierExcepton si la barrera se rompió por alguna razón (por ejemplo, otro thread agotó su tiempo o ha sido interrumpido). El método sobrecargado toma un límite de tiempo como parámetro adicional; esta versión sobrecargada lanza un TimeoutExcepton si todos los otros threads no han alcanzado con timeout. |
| boolean isBroken() | Retorna true si la barrera está rota. Una barrera se rompe si al menos un thread en esta barrera es interrumpid o su tiempo se agotó, o si la acción de la barrera falló lanzando una excepción. |
| void reset(){ | Resetea la barrera al estado inicial. Si hay threads esperando, lanzará una BrokenBarrierException. |

El siguiente código es un ejemplo que hace uso de la clase CyclicBarrier.

[CyclicBarrierTest.java](src/java/oca2/cap14/CyclicBarrierTest.java)

Ahora veamos cómo funciona este programa. En el método main() creas un objeto CyclicBarrier. El constructor toma dos argumentos: el número de threads a esperar y el thread que se invocará cuando todos los threads alcancen la barrera. En este caso, tienes cuatro jugadores esperando, creas cuatro threads por cada jugador. El segundo argumento del constructor de CyclicBarrier es un objeto MixedDoubleTennisGame, ya que este objeto representa el juego, el cual empezará cuando los cuatro jugadores estén listos.

Dentro del método run() de cada thread Player, llamas al método await() en el CyclicBarrier. Ya que el número de threads esperando alcanzan la cantidad de cuatro, el método run() en MixedDoubleTennisGame es llamado.

## Phaser

Un Phaser es una característica muy útil cuando threads independientes tienen que trabajar en fases para completar una tarea. Entonces, un punto de sincronización es necesario para los threads para trabajar en una parte de una tarea, espera a los otros para completar la otra parte de la tarea y hace un sync-up antes de avanzar a la siguiente parte de la tarea. La siguiente tabla lista importantes métodos en esta clase.

|  |  |
| --- | --- |
| Método | Descripción |
| Phaser() | Crea un objeto Phaser sin registrar parties ni padres. La fase inicial es cero. |
| Phaser(int numThreads) | Crea un objeto Phaser con un número dado de threads (parties) a llegar para avanzar al siguiente estado; la fase inicial es cero. |
| int register() | Agrega un nuevo thread (party) a este Phaser. Retorna la fase en donde se encuentre. Lanza una IllegalStateException si el máximo número de parties ya está registrado. |
| int bulkRegister(int numThreads) | Agrega un número de threads de parties que no han llegado a este Phaser. Retorna la fase actual. Lanza la misma excepción anterior. |
| int arrive() | Llega a esta fase sin esperar a los otros threads a que lleguen. Retorna la fase a la que llega. Puede lanzar IllegalStateException. |
| int arriveAndDeregister() | Igual que el anterior pero también se des-registra del objeto Phaser. |
| int arriveAndWaitAdvance() | Llega a esta fase y espera (se bloquea) mientras llegan los otros threads. |
| int awaitAdvance(int phase) | Espera (Se bloquea) mientras este objeto Phaser avanza al valor de fase dado como argumento. |
| int getRegisteredParties() | Retorna el número de threads (parties) registrados con este objeto Phaser. |
| int getArrivedParties() | Retorna el número de threads (parties) que ya llegaron a la fase actual del objeto Phaser |
| int getUnarrivedParties() | Retorna el número de threads (parties) que no han llegado a la fase actual del objeto Phaser. |

Considera el ejemplo de procesar una orden enviada en un pequeño establecimiento de café. Asume que aquí hay solo tres trabajadores: el concinero, el ayudante y el que atiende. Para simplificar la lógica del programa, asume que cada orden enviada consiste en tres ítems de comida. Completar una orden enviada cosiste en preparar las tres órdenes una después de la otra. Para completar la preparación de un ítem de comida, los tres trabajadores deben de hacer su parte del trabajo. El código siguiente muestra cómo esta situación puede ser implementada usando la clase Phaser.

[ProcessOrder.java](src/java/oca2/cap14/ProcessOrder.java) \*revisar este ejercicio después. No queda claro el funcionamiento.

En este programa creas un objeto Phaser para habilitar la sincronización de los tres trabajadores. Puedes crear un objeto Phaser llamando al constructor por default. Cuando el trabajador es creado, se registra el mismo en el Phaser, Alternativamente, puedes llamar a:

Phaser deliveryOrder = new Phaser(3); // para tres parties (threads)

En este caso no necesitas llamar al método register() en el objeto Phaser en el constructor del thread Worker.

En este programa, has asumido que la orden consiste en procesar tres ítems de comida, entonces el for corre tres veces. Para cada iteración, llamas al método arriveAndAwaitAdvance(). Para que esta sentencia proceda, las tres parties (threads) tienen que completar su parte del trabajo para preparar su ítem de comida. Tú simulas preparar comida llamando al método sleep() en el método run() para estos threads Worker. Estos threads llaman a arriveAndAwaitAdvance() para preparar cada ítem. Ya que cada ítem es preparado (cada fase es completada), el trabajo continúa a la siguiente fase. Ya que las tres fases son completadas, el envío de la órden procede y el programa termina.

## Colecciones concurrentes

El paquete java.util.concurrent provee un número de clases que son thread-safe equivalentes a las provistas en el framework collections, en el paquete java.util. Por ejemplo, ConcurrentHashMap es un Map equivalente a HashMap. La principal diferencia entre estos dos contenedores es que tú necesitas explícitamente sincronizar inserciones y eliminaciones con HashMap, mientras que la sincronización está integrada en ConcurrentHashMap. Su tu sabes cómo usar un HashMap, sabes cómo usar un ConcurrentHashMap implícitamente. Para el exame solo necesitas tener una sobre-vista del entendimiento de estas clases concurrentes, por lo que no entraremos en detalles de cómo usarlas. Observa la siguiente tabla que contiene las clases más importantes.

|  |  |
| --- | --- |
| Clase/Interface | Descripción corta |
| BlockingQueue | Esta interface extiende a la interface Queue. En BlockingQueue, la cola está vacía, espera (se bloquea) a que un elemento sea insertado, y si la cola está llena, espera a que un elemento sea removido de la cola. |
| ArrayBlockingQueue | Esta clase provee un array de tamaño fijo, basado en la implementación de la interface BlockingQueue |
| LinkedBlockingQueue | Esta clase provee una lista enlazada basada en la implementación de la interface BlockingQueue. |
| DelayQueue | Esta clase implementa BlockingQueue y consiste de elementos que son de tipo Delayed. Un elemento puede ser retirado de su cola solo después de un periodo de delay. |
| PriorityBlockingQueue | Equivalente a PriorityQueue, pero implementando la interface BlockingQueue. |
| SynchronousQueue | Esta clase implementa a BlockingQueue. Es este contenedor, cada insert() por un thread espera (se bloquea) a un remove() correspondiente y viceversa. |
| LinkedBlockingDeque | Esta clase implementa a BlockingDeque que inserta y remueve operaciones como bloque; usa una implementación de lista enlazada. |
| ConcurrentHashMap | Analogo a Hashtable, pero con acceso concurrente seguro y actualizaciones. |
| ConcurrentSkipListMap | Análogo a TreeMap, pero provee un acceso y actualización concurrente seguro. |
| ConcurrentSkipListSet | Análogo a TreeSet, pero provee un acceso y actualización concurrente seguro. |
| CopyOnWriteArrayList | Similar a ArrayList, pero provee un acceso concurrente seguro. Cuando una ArrayList es actualizada, crea una copia fresca del array subyacente. |
| CopyOnWriteArraySet | Una implementación de un Set, pero provee un acceso concurrente seguro y su implementación usa a la clase anterior. Cuando el contenedor es actualizado, este crea una copia fresca del array subyacente. |

Los siguientes dos programas muestran las diferencias de las clases concurrentes con las clases no concurrentes. Asume que tienes un objeto PriorityQueue compartido por dos threads. Asume que un thread inserta un elemento en el PriorityQueue y el otro thread remueve el elemento. Si los threads son programados para que la inserción sea antes de la eliminación, no hay ningún problema. Sin embargo, si el primer thread intenta remover un elemento antes de que sea insertado, te meterás en un problema.

[PriorityQueueExample.java](src/java/oca2/cap14/PriorityQueueExample.java)

La excepción que lanza indica que el primer thread intenta remover un elemento de una cola vacía, y por lo tanto resulta en una excepción NoSuchElementException.

Sin embargo, considera una pequeña modificación en este programa como sigue.

[PriorityBlockingQueueExample.java](src/java/oca2/cap14/PriorityBlockingQueueExample.java)

Este programa no resultará en una falla como en el caso previo. Esto es porque el método take() se bloqueará mientras un elemento sea insertado por otro thread; ya que está insertado el método take() retornará ese valor. En otras palabras, si estás usando un objeto PriorityQueue, necesitas sincrinizar los threads para que se inserten elementos antes de intentar removerlos. Sin embargo, en PriorityBlockingQueue, el orden no importa, y no importa cual operación sea invocada primero, el programa correrá correctamente. En esta forma, las colecciones concurrentes proveen soporte para un uso seguro de las colecciones en el contexto de multi-threading, sin la necesidad de ejecutar sincronización explícita.

# Aplicar variables atómicas y candados

El paquete java.util.concurrent tiene dos subpaquetes: atomic y locks. En esta sección vamos a discutir estos dos subpaquetes. A diferencia del resto de este capítulo, donde se discute concurrencia de alto nivel de abstracción, ambos, tanto atomic como lock son de APIs de bajo nivel. Sin embargo, proveen un control más fino cuando quieres escribir código multi-thread.

## Variables Atomic

¿Has visto código que adquiere y libera bloqueos para implementar operaciones primititvas o simples, como incrementar una variable, decremetarla y así? Adquirir y liberar bloqueos para operaciones primitivas no es eficiente. En estos casos, java provee una alternativa eficiente en la forma de variables atómicas (Atomic).

Aquí hay una lista de algunas clases en este paquete y una descripción corta.

* **AtomicBoolean**: valor booleano atómicamente actualizable.
* **AtomicInteger**: valor entero atómicamente actualizable. Hereda de la clase Number.
* **AtomicIntegerArray**: Un array de enteros donde los elementos se actualizan de manera atómica.
* **AtomicLong**: valor long atómicamente actualizable; hereda de la clase Number.
* **AtomicLongArray**: Un array long donde sus elementos son actualizados de manera atómica.
* **AtomicReference<V>**: Una referencia a un objeto atómicamente actualizable, de tipo V.
* **AtomicReferenceArray<E>**: Un array de tipo E, donde sus valores son actualizados de manera atómica (E se refiere al elemento de tipo base).

NOTA: Solo AtomicInteger y AtomicLong extienden de Number pero no AtomicBoolean. Todas las demás clases heredan directamente de Object.

De las clases en el paquete java.util.concurrency.atomic, AtomicInteger y AtomicLong son las más importantes. La siguiente tabla lista importantes métodos en la clase AtomicInteger. Los métodos en AtomicLong son análogos a estos.

|  |  |
| --- | --- |
| Método | Descripción |
| AtomicInteger() | Crea una instancia de AtomicInteger con un valor inicial a 0. |
| AtomicInteger(int initVal) | Crea una instancia de AtomicInteger con un valor inicial “initVal”. |
| int get() | Retorna el valor que almacena este objeto. |
| void set(int newVal) | Resetea el valor de este objeto al valor newVal. |
| int getAndSet(int newValue) | Retorna el valor actual contenido en el objeto y setea un nuevo valor newValue. |
| boolean compareAndSet(int expect, int update) | Compara el valor entero de este objeto con el valor esperado y si son iguales, setea el valor entero de este objeto al valor update. |
| int getAndIncrement() | Retorna el valor actual del objeto y lo incrementa. Comportamiento similar de i++ cuando i es un entero. |
| int getAndDecrement() | Retorna el valor actual del objeto y lo decrementa. Comportamiento similar a i—cuando i es un entero. |
| int gedAndAdd(int delta) | Retorna el valor contenido en este objeto y le suma el valor delta dado. |
| int incrementAndGet() | Incrementa el valor contenido en el objeto y retorna este valor. Comportamiento similar a ++i cuando i es un entero. |
| int decrementAndGet() | Decrementa el valor contenido en el objeto y lo retorna. Comportamiento similar a --i cuando i es un entero. |
| int addAndGet(int delta) | Suma el valor delta al valor contenido en el objeto y retorna el resultado. |
| int intValue()  long longValue()  float floatValue()  double doubleValue() | Castea el valor entero y lo retorna según el método |

Vamos a intentar un ejemplo para entender cómo usar AtomicInteger o AtomicLong. Asume que tienes un valor contador que es público y accesible por todos los threads. ¿Cómo puedes actualizar o acceder a este contador común de manera segura, sin caer en problemas de data-race? Oviamente, puedes usar la palabra synchronized para asegurar que la sección crítica (el código que modifica el valor contador) sea accedida por solo un thread a la vez. La sección crítica sería muy pequeña:

public void run() { synchronized(SharedCounter.class) { SharedCounter.cout++; }}

Sin embargo, este código es ineficiente ya que adquiere y libera un bloqueo siempre que vaya a ser incrementado el valor del contador. De manera alternativa, si tú declaras a count como AtomicInteger o AtomicLong (cualquiera es útil), entonces no hay necesidad de usar el bloqueo con la palabra synchronized. El siguiente programa muestra cómo usar AtomicLong en la práctica.

[AtomicVariableTest.java](src/java/oca2/cap14/AtomicVariableTest.java)

En la salida del programa te podrás haber dado cuenta que incrementar el valor Integer resulta en data-race y en el AtomicInteger no.

Vamos a analizar el programa. La clase AtomicVariableTest tiene dos datos miembro, uno de tipo Integer y otro de tipo AtomicInteger, con el mismo valor inicial.

Hay dos clases Thread. Una clase incrementa el valor Integer en su método run() y la otra incrementa el valor AtomicInteger en su método run(). El método main() creas cinco threads de estos dos tipos de Threads. La salida muestra que incrementar el valor Integer está propenso a data-race cuando está sin bloqueo, mientras que es seguro incrementar AtomicInteger sin bloqueo.

## Candados (Locks)

En el capítulo anterior (13), discutimos la palabra synchronized y cómo se asegura de que solo un thread se ejecute en la sección crítica a la vez. El paquete java.util.concurrent.Lock provee facilidades que son más sofisticadas. En esta sección discutiremos la interface Lock.

Usar un objeto Lock es similar a obtener locks implícitos usando la palabra clave synchronized. El objetivo de ambos constructos es el mismo: para asegurar que solo un thread acceda al recurso compartido a la vez. Sin embargo, a diferencia de synchronized, Locks también soporta el mecanismo wait/notify a lo largo con sus objetos Condition.

NOTA: Puedes pensar que usar synchronized para bloquear implícitamente y usar Lock para bloquear explícitamente.

La ventaja de usar synchronized (bloqueo implícito) es que tú no tienes que recordar liberar el lock en un bloque finally, ya que al final del bloque sincronizado (o método), automáticamente será generado código para liberar el bloqueo. Aunque esta es una característica útil, hay situaciones donde puedes necesitar controlar la liberación del bloqueo de manera manual (digamos, para liberarlo en otro lado que no sea el final del bloque o método), y los objetos Lock proveen esta flexibilidad. Sin embargo es tu responsabilidad asegurarte que liberaste el bloqueo en un bloque finally mientras usas objetos Lock. El siguiente pedazo de código describe el uso de un Lock.

[UsoLock.txt](src/java/oca2/cap14/UsoLock.txt)

Otra diferencia entre bloqueos implícitos y objetos Lock explícitos es que puedes hacer un “intento no bloqueado” para adquirir bloqueos con Locks. Bueno, ¿Qué significa “intento no bloqueado”? Obtienes un bloqueo si está disponible para bloquear, o puedes regresar fuera de la solicitud del bloqueo usando el método tryLock() en un objeto Lock. ¿No es interesante? Si tu adquieres el bloqueo satisfactoriamente, entonces puedes hacer la tarea en la sección crítica; de otra manera puedes ejecutar una acción alternativa. Es notorio que una versión sobrecargada del método tryLock toma un valor timeOut como argumento, entonces puedes esperar para adquirir el bloqueo por un tiempo específico.

tryLock(long time, TimeUnit unit)

Con tryLock(), el idioma para usar este objeto se encuentra en el siguiente pedazo de código.

[UsoTryLock.txt](src/java/oca2/cap14/UsoTryLock.txt)

Usando el método tryLock() evitas algunos problemas relacionados con la sincronización de threads discutidos en el capítulo anterior, como deadlocks y livelocks. La siguiente tabla muestra importantes métodos en la clase Lock.

|  |  |
| --- | --- |
| Método | Descripción |
| void lock() | Adquiere el bloqueo. |
| boolean tryLock() | Adquiere el bloqueo y retorna true si el bloqueo está disponible; si el bloqueo no está disponible, entonces no adquiere el lock y retorna false. |
| boolean tryLock(long time, TimeUnit unit) | Igual que el anterior pero espera el tiempo dado antes de renunciar a adquirir el bloqueo y retornar false. |
| void lockInterruptibly() | Adquiere un bloqueo; durante el proceso de adquirir el bloqueo, si otro thread lo interrumpe, este método lanza InterruptedException. |
| Condition newCondition() | Retorna un objeto Condition asociado con este objeto Lock. |
| void unlock() | Libera el bloqueo. |

Vamos a ver un ejemplo de un objeto Lock. En este ejemplo, usas un objeto Lock y lo pasas a los threads para sincronizarlos a este objeto Lock. Este programa es una variación simple del programa Semaphores dado anteriormente. Simulas el acceso a un cajero automático, que es el recurso compartido. Por supuesto, solo una persona puede usar el cajero a la vez, por lo tanto, el código para acceder al cajero es la sección crítica.

[ATMRoom2.java](src/java/oca2/cap14/ATMRoom2.java)

Como puedes observar de la salida, el cajero es accedido por una persona a la vez, entonces los otros tienen que esperar para acceder. En este programa, la clase ATMMachine crea un objeto Lock representando un cajero. Hay cinco personas esperando para acceder al cajero, que se simula creando cinco instancias de la clase Person. La clase Person extiende a la clase Thread y recuerda el objeto Lock donde adquiere y libera el bloqueo.

El método run() simplemente adquiere el bloqueo, accede al recurso compartido y después libera el bloqueo en el bloque finally. El objeto Lock (variable machine) asegura que solo un thread acceda a la vez. Los otros threads se bloquean mientras un thread tenga tomado el bloqueo.

Nota que puedes obtener diferente orden de personas accediendo al cajero si intentas correr este programa. Esto es porque el orden de acceso depende del programador de tareas dentro de la JVM.

NOTA: La interface ReadWriteLock (Que extiende a Lock) especifica un bloqueo que provee bloqueos separados para acceso de solo lectura y acceso de escritura. Puedes usar los métodos readLock() y writeLock() para obtener instancias de lectura y escritura, respectivamente. La clase ReentrantReadWriteLock implementa esta interface.

## Condiciones (Conditions)

Una condición soporta el mecanismo de notificación a threads. Cuando cierta condición es no satisfecha, un thread puede esperar por otro thread para satisfacer esta condición; que otro thread puede notificar una vez que la condición es lograda. Una condición es delimitada a un bloqueo. Un objeto Condition ofrece tres métodos para soportar el patrón wait/notify: await(), signal() y signalAll(). Estos tres métodos son análogos a wait(), notify() y notifyAll() soportados por la clase Object.

Un thread puede esperar a que una condición sea cierta usando el método await(), que es un llamada a un bloqueo interrumpible. Si quieres una espera no-interrumpible, puedes llamar al método awaitUninterruptibly(). Puedes también especificar un tiempo para la espera usado los métodos sobrecargados:

* long awaitNanos(long nanosTimeout)
* boolean await(long time, TimeUnit unit)
* boolean awaitUntil(Date deadline)

Ahora veamos un ejemplo que hace uso de objetos Condition. Asume que estás esperando a una persona llamado Joe que viene en un tren IC1122 de Madrid a París. Cuando el tren de Joe llega a la estación, se te informa; recoges a Joe y lo llevas a casa.

Asume que hay múltiples trenes que pueden arribar a la estación, tú necesitas esperar a un tren específico. Ya que el tren que te interesa arriba, recibes una notificación o señal de ese tren. Este escenario es un buen candidato para usar el patrón wait/notify. Aquí hay dos formas de implementar este patrón. La primera opción es usar locks implícitos y hacer uso de los métodos wait() y notifyAll() de la clase Object. La segunda opción es usar locks explícitos y objetos Condition y usar los métodos await() y signalAll() de la clase Condition.

[RailwayStation.java](src/java/oca2/cap14/RailwayStation.java)

Vamos a analizar cómo trabaja este programa. En la clase RailwayStation tienes un objeto Lock común llamado station. Para este objeto station obtienes una condición (recuerda que una condición siempre está asociada a un objeto Lock) llamada joeArrival. Usas el método newCondition(), entonces el objeto Condition resultante es una condición interrumpible; no especificaste ningún tiempo de espera, por lo que el thread esperará para siempre mientras obtenga la señal.

La clase Train es un Thread que simula el arribo de trenes en la estación. El método run() en Train primero obtiene el lock antes de anunciar que ha arribado y lo libera antes de que salga del método. Nota que se tú llamas al método await() en el objeto Condition sin adquirir el bloqueo, obtienes una excepción IllegalMonitorStateException. En el método run(), si el nombre del tren inicia con IC1122, enviará la señal de que Joe ha llegado llamando a joeArrival.signalAll().

Tu espera en la estación por Joe es simulada por el thread WaitForJoe. En el método run() adquieres el lock y esperas a la condición joeArrival (la señal de otro thread). Ya que has sido notificado que Joe ha arribado a la estación, lo recoges y lo llevas a casa.

NOTA: en multi-threading, una necesidad común es esperar a que una condición sea satisfecha por un thread antes de que otro proceda. Usando polling (checar repetidamente por la condición usando un loop while) es una mala solución porque esta solución desperdicia ciclos del CPU; por lo tanto también es susceptible de data-races. Usa bloques protegidos con wait/notify o await/signal en su lugar.

# Condiciones múltiples de un Lock

Para el examen es importante entender Locks y Conditions. Entonces vamos a discutir un ejemplo más detallado que hace uso de esto. En este programa te mostraremos cómo puedes obtener múltiples objetos Condition de un objeto Lock.

Asume que estás implementando una cola de tamaño fijo, con el tamaño determinado al tiempo de la creación del Thread. En una cola típica, si no hay elementos en la cola y es llamado el método remove(), lanzará una NoSuchElementException (como viste en un código anterior). Sin embargo, en este caso quieres que el thread se bloquee mientras algún otro thread inserta un elemento. Similarmente, si intentas insertar más elementos, el thread lanzará IllegalStateException para indicar que no es posible insertar más elementos, el thread debe bloquearse mientras un elemento es removido. En otras palabras, necesitas implementar una blocking queue simple.

Observa el siguiente código.

[BlockerQueue.java](src/java/oca2/cap14/BlockerQueue.java)

Como puedes ver en la salida, el método remove() es llamado primero, el cual espera a insert(). Ya que insert esá completado, el método remove() remueve satisfactoriamente el elemento de la cola. Ahora vamos a intentar otro test para probar si el bloqueo en el método insert() funciona:

[BlockerQueueTest2.java](src/java/oca2/cap14/BlockerQueueTest2.java)

Como puedes ver de la salida, cuando un thread invoca al método insert() en una cola llena, el thread se bloquea. Ya que otro thread nofifica que removió un elemento de la cola, el thread continúa e inserta de manera exitosa el elemento en la cola.

# Usar Executors y ThreadPools

Puedes directamente crear y manejar threads en una aplicación con la creación de objetos Thread. Sin embargo, si quieres abstraerte de los detalles de bajo nivel de la programación multi-threaded, puedes hacer uso de la interface Executor.

La siguiente figura muestra las clases importantes e interfaces en la jerarquía de Executor. En esta sección nos enfocaremos en usar la interface Executor, ExecutorService y ThreadPools, ya que cubriremos ForkJoinPool en la siguiente sección, usando el framework paralelo Fork/Join.



## Executor

Executor es una interface que declara solo un método: void execute(Runnable). Esta puede verse como una no-gran interface por sí misma, pero sus clases derivadas (o interfaces), como ExecutorService, ThreadPoolExecutor y ForkJoinPool, soportan funcionalidad útil. Discutiremos las clases derivadas de Executor en más detalle, en el resto de esta sección. Por ahora, observa el código siguiente, que es un ejemplo simple de la interface Executor para entender cómo usarla.

[ExecutorTest.java](src/java/oca2/cap14/ExecutorTest.java)

En este programa tienes una clase Task que implementa la interface Runnable proveyendo la implementación del método run(). La clase RepeatedExecutor implementa la interface Executor y provee la implementación del método execute(Runnable).

Ambos Runnable y Executor son similares en esencia ya que proveen un único método a implementar. En esta definición te habrás dado cuenta que Executor no es un thread por sí mismo y que tienes que crear un objeto Thread para ejecutar el objeto Runnable pasado en el método execute(). Sin embargo, la principal diferencia entre Runnable y Executor es que Executor está diseñado para abstraer cómo el thread es ejecutado. Por ejemplo, dependiendo de la implementación de Executor, Executor puede programar un thread para correr en cierto momento, o ejecutar el thread después de cierto periodo de retraso.

En este programa, sobrecargas el método execute con un argumento adicional para crear y ejecutar el thread cierto número de veces. En el método main() primero creas el objeto Thread y lo programas para ejecutarse. Después de esto creas una instancia de RepeatedExecutor para ejecutar el thread tres veces.

## Callable, Executors, ExecutorService, ThreadPool y Future

Callable es una interface que declara solo un método: call(). Su signatura completa es V call() throws Exception. Esta representa una tarea que necesita ser completada por un thread. Ya quee la tarea es completada, esta retorna un valor. Por alguna razón, si el método call() no se ejecuta o falla, lanza una excepción.

Para ejecutar una tarea usando el objeto Callable, primero debes crear un pool de threads. Un pool de threads es una colección de threads que pueden ejecutar tareas. Creas un pool de threads usando la clase utilitaria Executors. Esta clase provee métodos para obtener instancias de pools de threads, fábricas de threads, etc.

La interface ExecutorService implementa la interface Executor y provee servicios como terminación de threads y producción de objetos Future. Algunas tareas pueden tomar un tiempo considerable para completar su ejecución. Por lo tanto, cuando envías una tarea al servicio executor, obtienes un objeto Future.

Future representan objetos que contienen un valor que es retornado por un thread en el futuro (ejemplo: retorna el valor ya que el thread termina en el futuro). Puedes usar el método isDone() en la clase Future para checar si la tarea está completada, y usar el método get() para obtener los resultados de la tarea. Si llamas al método get() directamente mientras la tarea no está completada, el método se bloquea mientras la tarea se completa y retorna el valor ya que está disponible.

Suficiente de hablar, vamos a intentar un ejemplo simple para ver cómo estas clases trabajan juntas.

[CallableTest.java](src/java/oca2/cap14/CallableTest.java)

En este programa tienes una clase Factorial que implementa la interface Callable. Ya que la tarea es calcular el número factorial de n, la tarea necesita retornar un resultado. Usas el tipo Long para el valor de factorial, por lo que implementas a Callable<Long>. Dentro de la clase Factorial, defines el método call() que es el que ejecuta la tarea (la tarea aquí es calcular el factorial de un número dado). Si el valor dado de n es negativo o cero, no ejecutas la tarea y lanzas una excepción al método llamador. De otra manera, iteras de 1 a n para encontrar el factorial.

En la clase CallableTest primero obtienes una instancia de la clase Factorial. Entonces después necesitas ejecutar esta tarea. Por el bien de la simplicidad, obtienes un Executor singled-threaded llamando llamando al método newSingleThreadExecutor() en la clase Executors. Nota que puedes usar otros métodos como newFixedThreadPool(nThreads) para crear un thread pool con múltiples threads dependiendo del nivel de paralelismo que necesites.

Ya que obtienes un ExecutorService, envías la tarea para su ejecución. ExecutorService abstrae los detalles como cuándo la tarea es ejecutada, cómo la tarea es asignada a los threads, etc. Obtienes una referencia a Future<Long> cuando llamas al método submit(task). Para esta referencia Future, llamas al método get() para obtener el resultado después de completar la tarea. Si la tarea está ejecutándose todavía, cuando llamas a get(), este método se bloqueará hasta que la tarea se complete. Ya que la tarea se completó, necesitas manualmente liberar al ExecutorService llamando al método shutdown();

Ahora que estás familiarizado con el mecanismo básico de cómo ejecutar tareas, aquí veremos un ejemplo más complejo. Asume que tu tarea es encontrar la suma de números de 1 a n donde n es un número largo (un millón en nuestro caso). Por supuesto, puedes usar la fórmila [(n \* (n + 1))/2] para encontrar la suma. Sí, harás uso de esta fórmula para checar si la suma de 1...n es correcta o no. Sin embargo, solo para ilustración, dividirás el rango de 1 a 1 millon en n sub-rangos y cada rango lo asignarás en n threads para sumar los números en este sub-rango. Observa el siguiente programa.

[SumOfN.java](src/java/oca2/cap14/SumOfN.java)

Vamos a analizar cómo este programa trabaja, necesitas encontrar la suma de 1 a N donde N es un millón (un número largo). La clase SumCalc implementa Callable<Long> para sumar los valores en el rango ‘from - to’. El método call() ejecuta el cálculo actual de la suma iterando de from a to. El método call() ejecuta el cálculo actual y retorna una suma parcial como un valor Long.

En este programa, divides la tarea de suma en múltiples threads. Puedes determinar el número de threads pasado el número de núcleos disponibles en tu procesador; sin embargo, por simplificar el entendimiento del programa usamos diez threads.

En el método main() creas un ThreadPool con diez Threads. Crearas diez tareas de suma, por lo que necesitarás un contenedor para almacenar las referencias a esas tareas. Usas un ArrayList para almacenar los objetos Future<Long>.

En el primer loop for en main(), creas diez tareas y las envías al ExecutorService. Cuando envías una tarea, obtienes una referencia Future<Long> y la agregas al ArrayList.

Ya que creaste las diez tareas, recorres el ArrayList en el siguiente loop for para obtener los resultados de las tareas. Sumas los resultados parciales de cada tarea individual para completar la suma final.

Ya que sumas los valores de uno a un millón, usas la fórmula para encontrar la suma. De la salida, puedes ver que la suma y la fórmula son iguales, entonces acertaste en la lógica de dividir la tarea y combinar los resultados, correctamente.

Ahora, antes de que nos movamos a discutir el framework fork/join, vamos a discutir rápidamente unas cuantas clases que son útiles para la programación concurrente.

## ThreadFactory

ThreadFactory es una interface que sirve para crear threads en lugar de explícitamente crearlos llamando a new Thread(). Por ejemplo, asume que has creado threads de alta prioridad. Puedes crear una MaxPriorityThreadFactory para fijar la prioridad por default de los threads creados por esta fábrica a la máxima prioridad. Observa el siguiente código.

[TestThreadFactory.java](src/java/oca2/cap14/TestThreadFactory.java)

Con el uso de ThreadFactory puedes reducir código repetitivo para fijar la prioridad, el nombre, thread-pool, etc.

## La clase ThreadLocalRandom

Cuando programas de manera concurrente, encontrarás a menudo que necesitas generar números aleatorios. Usando la clase Math.random() no es eficiente para programación concurrente. Por esta razón, el paquete java.util.concurrent introdujo la clase ThreadLocalRandom que es perfecta para el uso con programación concurrente. Puedes usar el método current() de ThreadLocalRandom y después llamar a los métodos como nextInt() y nextFloat() para generar los números aleatorios.

## Enumeración TimeUnit

Has visto ya que algunos métodos al principio de este capítulo toman un TimeUnit como argumento. TimeUnit es un enumeration que es usado para especificar la resolución del tiempo. La unidad de tiempo en TimeUnit puede ser:

* DAYS
* MINUTES
* SECONDS
* MICROSECONDS
* MILLISECONDS
* NANOSECONDS

La enumeration también tiene métodos útiles para convertir entre estas unidades de tiempo. Por ejemplo.

System.out.printf(“Un día tiene %d horas, %d minutos, %d segundos”, TimeUnit.DAYS.toHours(1), TimeUnit.DAYS.toMunutes(1), TimeUnit.DAYS.toSeconds(1));

Algunos de los métodos en la API de java usan periodos específicos. Por ejemplo, el método sleep() toma el tiempo para dormir en milisegundos. Por lo tanto, si quieres especificar el tiempo para dormir un thread en alguna otra unidad de tiempo, digamos segundos o días, TimeUnit hace esta tarea útil. Observa el siguiente ejemplo.

[TimeUnitExample.java](src/java/oca2/cap14/TimeUnitExample.java)

# Usar el Framework Paralelo Fork/Join

Este framework en el paquete java.util.concurrent ayuda a simplificar la escritura de código paralelo. El framework es una implementación de la interface ExecutorService y provee una plataforma concurrente de fácil utilización, en función de explotar múltiples procesadores. Este framework es muy útil para modelar problemas de divide y vencerás. Esta aproximación es apta para tareas que pueden ser divididas recurrentemente y calculadas en una baja escala; los resultados calculados son combinados después. Dividir la tarea en tareas más pequeñas es llamado *forking*, y fusionar los resultados de las tareas pequeñas se llama *joining*.

El framework fork/join usa el algoritmo work-stealing: cuando un thread trabajador completa su trabajo y está libre, este toma (o roba) trabajo de otro thread que está todavía ocupado haciendo algún otro trabajo. Inicialmente parecerá que usar el framework es una tarea complicada. Ya que te familiarices con él, sin embargo, realizarás que esto es conceptualmente fácil y simplificarás tu trabajo significativamente. La clave es de subdividir recursivamente la tarea en pedazos pequeños que pueden ser procesados en threads separados.

Brevemente, el algoritmo fork/join está diseñado como sigue:

[algoritmoFJ.txt](src/java/oca2/cap14/algoritmoFJ.txt)

La siguiente figura visualiza como la tarea es recursivamente subdividida en tareas más pequeñas y cómo los resultados parciales son combinados. Coso se muestra en la figura, una tarea es partida en dos subtareas, y después cada subtarea es nuevamente dividida en dos subtareas, y así recursivamente, hasta que cada tarea dividida sea computable por cada thread. Ya que un thread completa de computar la tarea, retorna el resultado para combinarlo con otros resultados; de esta forma, todos los resultados computados son combinados de regreso.



## Clases útiles en el framework Fork/Join

Las siguientes clases juegan un papel clave en el framework:

* ForkJoinPool
* ForkJoinTask
* RecursiveTask
* RecursiveAction

Vamos a considerar estas clases en más detalle.

ForkJoinPool es la clase más importante en el framework. Es un Thread-pool para correr tareas fork/join, esta ejecuta una instancia de ForkJoinTask. Esta ejecuta la tarea y maneja su ciclo de vida. La siguiente tabla lista importantes métodos que tiene esta clase abstracta.

|  |  |
| --- | --- |
| Método | Descripción |
| void execute(ForkJoinTask<?> task) | Ejecuta la tarea dada asincrónicamente. |
| <T> T invoke(ForkJoinTask<?> task) | Ejecuta la tarea dada y retorna el resultado. |
| <T> List<Future<T>> invokeAll(Collection<?  extends Callable<T>> tasks) | Ejecuta todas las tareas dadas y retorna una List de objetos Future cuando todas las tareas se hayan completado. |
| boolean isTerminated() | Retorna true si todas las tareas se completaron. |
| int getParallelism()  int getPoolSize()  long getStealCount()  int getActiveThreadCount() | Métodos que checan el estatus. |
| <T> ForkJoinTask<T> submit(Callable<T> task)  <T> ForkJoinTask<T> submit(ForkJoinTask<T>  task)  ForkJoinTask<?> submit(Runnable task)  <T> ForkJoinTask<T> submit(Runnable task,  T result) | Ejecuta la tarea enviada. Las versiones sobrecargadas toman diferentes tipos de tareas; retornan un objeto Task o un objeto Future. |

ForkJoinTask<V> es como un thread ligero representando una tarea que define métodos como fork() y join(). La siguiente tabla muestra importantes métodos en esta clase.

|  |  |
| --- | --- |
| Método | Descripcion |
| boolean cancel(boolean mayInterruptIfRunning) | Intenta cancelar la ejecución de la tarea. |
| ForkJoinTask<V> fork() | Ejecuta la tarea asincrónicamente. |
| V join() | Retorna el resultado de la computación cuando esta está terminada. |
| V get() | Retorna el resultado de la computación; espera si esta no está terminada. |
| V invoke() | Inicia la ejecución de las tareas enviadas; espera mientras la computación completa y regresa los resultados. |
| static <T extends ForkJoinTask<?>> Collection<T> invokeAll(Collection<T> tasks) | Igual que el método de arriba. |
| boolean isCanceled() | Retorna true si la tarea es cancelada. |
| boolean isDone() | Retorna true si la tarea está completada. |

RecursiveTask<V> es una tarea que puede correr en un ForkJoinPool; el método compute() retorna un valor de tipo V. Esta hereda de ForkJoinTask.

RecursiveAction es una tarea que puede correr en un ForkJoinPool; su método compute() ejecuta los pasos de computación actual en la tarea. Es similar a RecursiveTask, pero no retorna valor.

## Usar el framework Fork/Join

Vamos a acercarte a cómo utilizar el framework en la solución de un problema. Aquí están los pasos para usar el framework.

* Primero, checa si el problema es apto para el framework o no. Recuerda: el framework Fork/Join no es apto para todos los tipos de tareas. Este framework es apto si tu problema encaja en esta descripción:
  + El problema puede ser designado como una tarea recursiva donde la tarea puede ser subdividida en unidades más pequeñas y los resultados pueden combinarse juntos.
  + Las tareas son independientes y pueden ser computadas por separado sin la necesidad de comunicación entre las tareas cuando la computación está en proceso. (Por supuesto, después de que la computación terminó, necesitas unirlas todas juntas).
* Si el problema que quieres resolver puede ser modelado recursivamente, entonces define una clase Task que extiende a RecursiveTAsk o RecursiveAction. Si la tarea retorna un resultado extiende a RecursiveTask, de otra manera extiende a RecursiveAction.
* Sobre-escribes el método compute() in la nueva clase Task definida. El método compute() de hecho ejecuta la tarea si la tarea es suficientemente pequeña para su ejecución; o la divide en subtareas y las invoca. Las subtareas pueden ser invocadas tanto por el método invoqueAll() o por el método fork() (usa fork() cuando la subtarea retorna un valor). Usa el método join() para obtener los resultados computados (si usas el método fork() antes).
* Combina los resultados, si son computados por las subtareas.
* Después crea una instancia de ForkJoinPool, crea una instancia de la clase task e inicia la ejecución de la tarea usando el método invoke en la instancia ForkJoinPool.
* Esto es todo.

Ahora vamos a intentar resolver el problema de como sumar de 1 a N donde N es un número largo. En el siguiente programa subdivides la tarea de sumar de manera iterativa en diez sub rangos; entonces computas la suma de cada sub rango y después computas la suma de las sumas parciales. Alternativamente, puedes resolver este problema recursivamente usando el framework.

[SumOfUsingForkJoin.java](src/java/oca2/cap14/SumOfNUsingForkJoin.java)

Vamos a analizar cómo funciona este programa. En este programa quieres computar la suma de los valores en el rango de 1 a 1,000,000. Para hacerlo simple, decides usar diez threads para ejecutar la tarea. La clase RecursiveSumOfN extiende a RecursiveTask<Long>.

En RecursiveTask<Long> usas <Long> porque la suma de los números en cada sub-rango es un valor long. En suma, escoges RecursiveTask<Long> en lugar de un plano RecursiveAction porque cada subtarea retorna un valor; si la tarea no devuelve ningún valor, puedes usar RecursiveAction.

En el método compute(), decides si computas el rango de números o lo divides en subtareas usando la siguiente condición:

((to - from) <= N/NUM\_THREADS)

Usas este umbral en esta computación. En otras palabras, si el rango de los valores está dentro del umbral, entonces esta suma puede ser computada por una tarea, entonces realizas la computación; de otra manera, tú, recursivamente divides la tarea en dos partes. Usas un simple loop for para encontrar la suma de los valores en este rango. En el otro caso, divides el rango similarmente a como divides el rango en el algoritmo de búsqueda binaria: para el rango from..to, encuentras el punto medio y creas dos sub-rangos from..mid y mid+1..to. Ya que llamas al método fork, esperas a que la primera tarea complete la computación de la suma y generas otra tarea para la segunda mitad de la computación.

En el método main() creas un ForkJoinPool con un número de threads dado por NUM\_THREADS. Envías la tarea al fork/join pool y obtienes la suma computada para 1 a 1,000,000. Ahora también calculas la suma usando la fórmula de N números continuos.

De la salida del programa, puedes observar cómo la tarea es subdividida en subtareas. Puedes también verificar de la salida que la suma computada y la suma por fórmula es la misma, indicando que la subdivisión de la tarea es correcta.

En este programa, arbitrariamente asumes el número de threads a usar (10). Esto fue para simplificar la lógica del programa. Una mejor aproximación para decidir el umbral, es dividir el tamaño de los datos por el número de procesadores disponibles. En otras palabras:

umbral = (tamaño de los datos) / (número de procesadores disponibles)

¿Cómo puedes obtener el número de procesadores disponibles? Para esto puedes usar el método Runtime.getRuntime().availableProcesors();

En el programa anterior usaste RecursiveTask; sin embargo, si la tarea no retorna un valor, entonces debes usar RecursiveAction. Asume que tienes un gran array (digamos de 10,000 items) y quieres buscar un elemento clave. Puedes usar el fork/join framework para dividir la tarea en muchas subtareas y ejecutarlas en paralelo. El siguiente programa contiene la solución implementada.

[SearchUsingForkJoin.java](src/java/oca2/cap14/SearchUsingForkJoin.java)

La principal diferencia entre el primer programa y el segundo es que usas RecursiveTask y RecursiveAction. Puedes hacer cambios severos para extender la tarea a RecursiveAction. El primer cambio es que el método compute() no retorna nada. Otro cambio es que usas invokeAll() para enviar las subtareas a ejecutarse. Otro cambio obvio es que tu realizas una búsqueda en lugar de una suma en el método compute(). Aparte de estos cambios, los dos programas trabajan de manera parecida.

# Puntos a recordar

Recuerda estos puntos para tu examen:

* Si es posible lograr lo que ofrece el framework fork/join usando concurrencia básica con start() y join(). Sin embargo, el framework fork/join abstrae muchos de los detalles de bajo nivel y entonces es fácil usarlo. En suma, es mucho más eficiente usar el framework fork/join en lugar de manejar los threads a bajo nivel. Además, usando el ForkJoinPool manejas eficientemente los threads y ejecuta mucho mejor que los thread pool convencionales. Por todas estas razones, te animamos a usar el framework fork/join.
* Cada thread trabajador en el framework fork/join tiene una cola de trabajo, que es implementada usando una Deque. Cada vez que una nueva tarea (o subtarea) es creada, esta es colocada en la cabeza de su propia Deque. Cuando una tarea completa una tarea y ejecuta join con otra tarea que no está completada todavía, trabaja de manera inteligente. El thread saca una nueva tarea de la cabeza de su queue y empieza a ejecutar en lugar de dormir (En orden de esperar a que otra tarea termine). De hecho, si la queue de un thread está vacía, entonces el thread saca una tarea de la cola de la queue que tenga otro thread. Esto no es más que el algoritmo de work-stealing.
* Se ve obvio llamar a fork() para ambas tareas (si estás dividiendo en dos tareas) y llamar a join() dos veces. Esto es correcto, pero ineficiente. ¿Por qué? Bueno, básicamente estás creando más tareas paralelas que son útiles. En este caso, el thread original esperará por las otras dos tareas para completarse, lo que es ineficiente considerando el costo de creación de las tareas. Esta es la razón por la cual llamas fork() en una y compute() en la otra.
* El lugar de las llamadas a fork() y join() es muy importante. Por ejemplo, vamos a asumir que pones las llamadas en el siguiente orden:
  + first.fork();
  + resultFirst = first.join();
  + resultSecond = second.compute();

Este uso es la ejecución serial de dos tareas, ya que la segunda tarea inicia a ejecutarse solo después de que la primera se completó. Así que es menos eficiente que su versión secuencial ya que también incluye la creación de la tarea. Observa dónde pones las llamadas a fork() y join()

* El performance no está siempre garantizado mientras uses el framework fork/join. Una de las razones es la posición de fork() y join().

# Resumen

Usar java.util.concurrent Collections

* Un semaphore controla el acceso a recursos compartidos. Un semáforo mantiene un contador para especificar el número de recursos que el semáforo controla.
* CountDownLatch permite que uno o más threads esperar a que se acabe una cuenta regresiva.
* La clase Exchanger está diseñada para intercambiar datos entre dos threads. Esta clase es útil cuando dos threads necesitan sincronizarse entre ellos y continuamente intercambiar datos.
* CiclicBarrier ayuda a proveer un punto de sincronización donde los threads pueden necesitar esperar a un punto de ejecución específico mientras otros threads llegan a este punto.
* Phaser es útil cuando unos pocos threads independientes tienen que trabajar en fases para completar una tarea.

Aplicar variables atómicas y Locks

* Java provee una alternativa eficiente en la forma de variables atómicas, donde necesitas adquirir y liberar candados solo para realizar operaciones primitivas en variables.
* Un candado asegura que solo un thread pueda acceder al recurso compartido a la vez.
* Un Condition soporta el mecanismo de thread notification. Cuando una cierta condición no está satisfecha, un thread puede esperar a que otro thread satisfaga esta condición; este otro thread notifica ya que la condición es lograda.

Usar Executors y ThreadPools

* La jerarquía de Executors abstrae los detalles de bajo nivel de la programación multi-threading y ofrece constructos amigables de alto nivel.
* La interface Callable representa una tarea que necesita ser completada por un thread. Yaque la tarea se completó, el método call() de la implementación Callable retorna un valor.
* Un thread-pool es una colección de threads que pueden ejecutar tareas.
* Future representa objetos que contienen un valor que es retornado por un thread en el futuro.
* ThreadFactory es una interface que está diseñada para crear threads en lugar de explícitamente crear threads con la palabra new.

Usar el framework paralelo fork/join

* El framework fork/join es la manera portable de ejecutar un programa con paralelismo decente.
* El framework es una implementación de la interface ExecutorService y provee una plataforma concurrente fácil de usar, en función de usar múltiples procesadores.
* Este framework es muy útil para modelar problemas de divide y vencerás.
* Este framework usa el algoritmo de work-stealing: cuando un thread trabajador completa su trabajo y está libre, toma (o roba) trabajo de otros threads que están todavía ocupados haciendo algún trabajo.
* La técnica work-stealing resulta en una carga balanceada de manejo de threads con un costo mínimo de sincronización.
* ForkJoinTask<V> es una entidad como thread ligero representando una tarea que define métodos como fork() y join().