

PROGRAMACION CONCURRENTE Y DISTRIBUIDA

III.4 Concurrencia con Java: Sincronización explícita



J.M. Drake

Notas:

Locks explícitos

- # En Java 5.0, se introdujo un mecanismo de sincronización alternativo al lock intrínseco que se define a través de la clase **ReentrantLock** y cuya funcionalidad se define a través de la interfaz **Lock**.
- # El **ReentrantLock** se introduce por las limitaciones del lock intrínseco:
 - No es posible interrumpir un thread que espera un **wait**.
 - No es posible intentar de forma no bloqueante adquirir un lock sin suspenderse definitivamente en él.
 - Los lock intrínseco deben ser liberados en el mismo bloque de código en el cual se suspendió.
- # Comparación:
 - El **Lock** intrínseco conduce a un estilo de programación sencillo, seguro y compatible con la gestión de excepciones
 - El **ReentrantLock** conduce a estrategias menos seguras, pero más flexibles, proporciona mayor vivacidad y mejores características de respuesta.

Notas:

En las versiones de Java anteriores a 1.5, el único mecanismo de sincronización de thread concurrentes que existían eran los lock implícitos en todos los objetos y clases derivadas de la clase **Object**. En la versión 1.5 se han añadido la clase **ReentrantLock**. Los objetos de esta clase son una alternativa con características avanzadas que se ha desarrollado en función de las características en las que el lock intrínseco presentaba limitaciones:

- No es posible interrumpir un thread que se encuentra suspendido en un **wait** sobre un lock.
- Las operaciones de acceso y liberación de un lock tiene que estar definido en un mismo bloque de código.
- No es posible comprobar si un recurso está libre sin previamente suspenderse en él.
- Un bloqueo es fatalmente irre recuperable, la única salida es reiniciar la aplicación.

La funcionalidad de los objetos **ReentrantLock** viene definida por la interfaz **Lock**. Ofrece las mismas capacidades para de gestión de memoria y recursos compartidos que el lock intrínseco

Interface java.util.concurrent.locks

```
public interface Lock{
    //Provide more extensive locking operations than can be obtained using synchronized
    // methods and statements.
    void lock();
        // Acquires the lock
    void lockInterruptibly() throws InterruptedException;
        // Acquires the lock unless the current thread is interrupted.
    boolean tryLock();
        // Acquires the lock only if it is free at the time of invocation.
    boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException;
        // Acquires the lock if it is free within the given waiting time and the current
        // thread has not been interrupted.
    void unlock();
        // Releases the lock.
    Condition newCondition();
        //Returns a new condition instance that is bound to this Lock instance.
}
```

Notas:

La interfaz Lock define un conjunto de operaciones abstractas de toma y liberación de un lock. A diferencia del lock intrínseco, la interface Lock ofrece diferentes formas de toma de un lock: incondicional, no bloqueante, temporizado o interrumpible. Además todas las operaciones de suspensión y liberación de un lock son explícitas.

Región protegida basada en un Lock explícito

```
Lock control = new ReentrantLock();
...
control.lock();
try{
    // Actualiza al objeto protegido por el lock
    // Atiende excepciones y restaura invariantes si es necesario
}finally{
    control.unlock();
}
```

Notas:

En la imagen se muestra un bloque protegidos típico implementado con un Lock. La liberación debe hacerse en una sentencia finally, ya que hay que prever la posibilidad de una excepción, y en este caso el lock debe de liberarse explícitamente también.

Toma interrumpible de un lock

```
public boolean transferMoney( Account fromAcct, Account toAcct, EuroAmount amount,
    long timeout, TimeUnit unit) throws NoFundsException, InterruptedException{
    long fixedDelay=getFixedDelayComponentNanos(timeout,unit);
    long randMod=getRandomDelayModulusNanos(timeout,unit);
    long stopTime=SystemNanoTime()+unit.toNanos(timeout);
    while (true){
        if (fromAcct.lock.tryLock()){
            try{ if (toAcct.lock.tryLock()){
                try {if (fromAcct.getBalance().compareTo(amount)<0)
                    throws new NoFundsException();
                else {fromAcct.debit(amount); toAcct.credit(amount); return true;}
                }finally {toAcct.lock.unlock();}
            }
            }finally {fromAcct.lock.unlock();}
        } if (System.nanoTime()<stopTime) return false;
        NANOSECONDS.sleep(fixedDelay+rnd.nextLong()%randMod);
    }
}
```

Notas:

La toma condicional y temporizado se realiza a través de la operación tryLock.

Cuando se utilizan las suspensiones condicionadas y temporizadas, si todos los recursos que se requieren no son todos tomados, cabe programar liberar todos los recursos tomados e intentarlo de nuevo más adelante. Esto es lo que se muestra en el ejemplo de la transferencia bancaria. En un bucle cerrado (while(true)) primero se intenta tomar el lock de la cuenta de origen (fromAcct), luego se trata tomar el lock de la cuenta destino (toAcct) y por último se comprueba si existe saldo suficiente en origen. Si todas condiciones se satisfacen la transferencia se realiza y se retorna true. Si falla cualquiera de las condiciones, se sale por el correspondiente finally y se ejecuta la operación de liberación del lock o lock ya tomados, y se intenta mas adelante. Cuando ha transcurrido el tiempo de timeout sin éxito, se retorna false y la transferencia no se realiza.

Espera con un plazo previsto

```
public boolean trySendOnSharedLine(String message, long timeout, TimeUnit unit)
    throws InterruptedException{
    long nanosToLock=unit.toNanos(timeout)- estimatedNanosToSend(message);
    if (!lock.tryLock(nanosToLock, NANOSECONDS)) return false;
    try{
        return sendOnsharedLine(message);
    } finally {lock.unlock();}
}
```

Notas:

Las suspensiones temporizadas son útiles para implementar transacciones en las que se dispone de una ventana temporal para ejecutar la actividad. Si el plazo se va a acabar si éxito se realiza otra operación alternativa.

En el ejemplo, se desea transferir un mensaje por un canal compartido con otras tareas. Con la sentencia tryLock temporizada se trata de conseguir acceso al canal dentro del tiempo en el que la operación es aun posible. Si en el acceso se tarda mas del plazo previsible, el intento de acceso al canal se suspende.

Toma interrumpible de un lock

```
public boolean trySendOnSharedLine(String message) throws InterruptedException{
    lock.lockInterruptibly();
    try{
        return cancellableSendOnsharedLine(message);
    } finally {lock.unlock();}
}

private boolean cancellableSendOnSharedLine(String message)
    throws InterruptedException{...}
```

Notas:

Utilizando la operación `lockInterruptible` se puede tratar de acceder al lock dentro de una actividad cancelable.

Objetos Condition

- # Cuando se utiliza un Lock explícito para definir una región asíncrona, dentro de ella se utilizan los objetos *Condition* como mecanismo de sincronización entre threads.
- # Un objeto *Condition* está estructuralmente ligado a un objeto Lock. Sólo puede crearse invocando el método *newCondition()* sobre un objeto Lock.
- # El objeto **Condition** solo puede ser invocado por un thread que previamente haya tomado el Lock al que pertenece.

Notas:

Interface Condition

Es ofrecida por variables de condición asociadas a un reentrantLock :

```
public interface Condition {  
    void await();  
    //Causes the current thread to wait until it is signalled or interrupted.  
    boolean await(long time, TimeUnit unit)  
    // Causes the current thread to wait until it is signalled or interrupted, or the specified  
    // waiting time elapses.  
    long awaitNanos (long nanosTimeout)  
    //Causes the current thread to wait until it is signalled or interrupted, or the specified  
    // waiting time elapses.  
    void awaitUninterruptibly()  
    // Causes the current thread to wait until it is signalled.  
    boolean awaitUntil(Date deadline)  
    // Causes the current thread to wait until it is signalled or interrupted, or the specified  
    // deadline elapses.  
    void signal()  
    // Wakes up one waiting thread.  
    void signalAll()  
    // Wakes up all waiting threads.  
}
```

Notas:

Ejemplo de uso de un objeto Condition

```
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x)
        throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();

            items[putptr] = x;
            putptr=(putptr+1) % items.length;
            count=count+1;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }

    public Object take()
        throws InterruptedException {
        lock.lock();
        try { while (count == 0)
                notEmpty.await();

            Object x = items[takeptr];
            takeptr=(takeptr+1) % items.length;
            count=count-1;
            notFull.signal();
            return x;
        } finally { lock.unlock(); }
    }
}
```

Notas:

Read-write Locks

- ⌘ La interfaz Lock esta concebida para evitar la exclusión mutua que previene la actualización concurrente del bloque que protege. Garantiza la exclusión mutua sin diferenciar las situaciones writer/writer y writer/reader que son incompatibles y las reader/reader que si son compatibles
- ⌘ La interfaz ReadWriteLock diferencia dos tipos de lock uno para writer y otro reader:

```
public interface ReadWriteLock{  
    Lock readLock(); // Returns the lock used for reading.  
    Lock writeLock(); // Returns the lock used for writing.  
}
```

- ⌘ La clase **ReentrantWriteLock** implementa esta interfaz

Notas:

Ejemplo de uso de reader/writer lock

```
public class readWriterMap<K,V>
    private final Map<K,V> map;
    private final ReadWriterLock lock=new ReentrantReadWriteLock();
    private final Lock r= lock.readLock();
    private final Lock w= lock.writeLock();

    public ReadWriter(Map<K,V> map){this.map=map;}
    public V put(K key, V value){
        w.lock();
        try{ return map.put(key,value);} finally {w.unlock();}
    }
    // Lo mismo hay que hacer para remove(),putAll() y Clear()
    public V get(Object key){
        r.lock();
        try { return map.get(key);} finally{ r.unlock();}
    }
    // Lo mismo para los restantes read-only métodos
}
```

Notas: