Hebras y Sincronización en Java

Jonathan Makuc – http://jmakuc.ublog.cl

Noviembre 2008

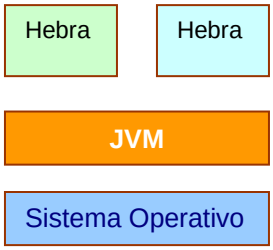
Tópicos:

1. Introducción a hebras en Java

El presente tutorial tiene por objetivo explicar de forma simple y práctica el trabajo básico con hebras en Java, en cuanto a su creación, correcto uso y sincronización. Para un estudio más acabado, refiérase al *Tutorial de concurrencia en Java* de Sun en http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html.

Computacionalmente se define como *hebra* como una unidad de procesamiento de datos, la cual es más liviana que un *proceso*, entendiendo como proceso a un programa en ejecución. Un proceso puede tener bajo su comando a múltiples hebras que realicen trabajos específicos, a veces de forma paralela.

Java proporciona 2 formas para trabajar con hebras: extender un Thread o implementar un Runnable. Ambas resultan en la creación de una hebra de usuario, la cual es invisible al sistema operativo, como se muestra en el diagrama a la derecha, por lo tanto no se puede establecer prioridades de ejecución de hebras Java en el scheduler del sistema operativo.



Las hebras pueden tener un nombre de identificación, prioridad de ejecución (a mayor prioridad, más probabilidad de ejecutarse) y la posibilidad de indicar si son hebras "demonio". La JVM posee en su interior un scheduler que se encarga de calendarizar la ejecución de las hebras, manejo de prioridades, creación y destrucción, etc.

Al partir la JVM (Java Virtual Machine), se crea una hebra inicial la cual ejecuta al método main. La ejecución continúa hasta que se invoque al método *exit* de la clase *Runtime*, o que todas las hebras que *no son "demonios"* han terminado.

2. Hebras a través del clase Thread

La clase Thread es la forma real en la cual se crea una nueva hebra en Java. Sus constructores permite establecer nombre, grupo, Runnable a ejecutar y en algunas plataformas tamaño del snack. Esta clase forma parte del lenguaje base de java y se encuentra en el paquete java.lang.

La forma tradicional de trabajar con la clase Thread, es extenderla. Para hacer esto, se debe implementar el método ***run***, el cual contendrá el código real a ejecutarse en la nueva hebra.

A la derecha vemos el código fuente de una hebra simple de ejemplo la cual recibe como único parámetro de su constructor el nombre que tendrá esta hebra. El constructor es utilizado para inicializar correctamente la hebra pero no inicia su ejecución.

Para iniciar la ejecución de la hebra se debe invocar al método ***start()***.

En el ejemplo se muestra como se declara la clase MiHebra y luego en el método main se instancia el objeto y luego se corre el código que esta contiene.

De esta forma se pueden lanzar múltiples hebras del tipo MiHebra que se ejecuten en paralelo, donde cada una de ellas ejecutará el código contenido en el método run().

```
class MiHebra extends Thread {

  public MiHebra(String nombre) {

    this.setName(nombre);

  }

  public void run() {

      // CODIGO DE LA HEBRA AQUI
  }
}

public  static  void  main(String[]
args) {

  MiHebra hebra1 = new MiHebra("1");
  Hebra1.start();

}
```

3. Hebras a través de la interfaz Runnable

Una forma alternativa de poder construir el código a ejecutarse dentro de una nueva hebra, es a través de la interfaz *Runnable.* Esta segunda opción es útil cuando se tienen clases que ya están heredando y además se requiere que ejecuten en paralelo.

Todo objeto que implemente la interfaz Runnable debe implementar el método ***run()***, que al igual como sucede con la clase Thread, contiene el código real a ejecutar en la hebra.

```
class MiRunnable implements Runnable {

  public void run() {

      // CODIGO DE LA HEBRA AQUI
  }
}

public static void main(String[] args) {

  MiHebra hebra1 = new MiHebra(new MiRunnable(), “trabajador 1”);
  Hebra1.start();

  MiHebra hebra2 = new MiHebra(new MiRunnable(), “trabajador 2”);
  Hebra1.start();

}
```

En este ejemplo tenemos la clase MiRunnable que implementa correctamente a Runnable. Para poder ejecutar el código dentro del método run, se debe instanciar el Runnable y pasarlo como parámetro al constructor de un Thread. Desde aquí el manejo de la hebra es idéntico al descrito anteriormente.

4. Trabajo con hebras y buenas prácticas

4.1. Hebras Demonio

Se denomina demonio a un proceso que corre en segundo plano, encargado de prestar un servicio. Java entrega la posibilidad de indicar que un Thread es demonio. No existe ninguna diferencia "física" entre una hebra demonio y una normal, pero si como estas influyen en el comportamiento del programa principal, puesto que JVM termina cuando todas las hebras *no-demonio* han finalizado, es decir, si existe al menos 1 hebra no-demonio funcionando, el programa seguirá ejecutándose, aunque este programa no realice ninguna acción.

Los demonios son útiles para encapsular código de servicio a otros procesos (como limpieza, programación de tareas, etc.), los cuales no tienen utilidad luego de que el último proceso de primer plano ha terminado. Así, simplificamos la programación de nuestra aplicación dado que no hay que preocuparse por detener las hebras secundarias o de apoyo, puesto que la JVM lo hará por nosotros.

```
class MiDemonio extends Thread {

  public MiDemonio(String nombre) {

    this.setName(nombre);
    this.setDaemon(true);

  }

  public void run() {

      // CODIGO DEL DEMONIO AQUI
  }
}
```

4.2. Iniciar y detener

Ya hemos visto como iniciar una hebra, invocando al método ***start()***, pero detenerla no es tan simple. Si bien la clase Thread provee el método *stop()*, este se encuentra deprecado hace muchas versiones del lenguaje y no se recomiendo utilizarlo bajo ningún punto de vista.

El motivo es que invocar a *stop()* provoca la liberación de todos los monitores que sostiene la hebra, a medida que la excepción *ThreadDeath* se propaga por el stack. Esta liberación anticipada puede poner en funcionamiento a hebras que se encuentran en estados inconsistentes o que están por ingresar a regiones críticas que no han concluido su ejecución; resultando esto un comportamiento impredecible en la computación final. Para más información, puede referirse al artículo de Sun *Java Thread Primitive Deprecation* en http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html.

La forma correcta de programar una hebra, es a través de una variable booleana que indique si continuar la ejecución o no, en conjunto con la reescritura del método stop seteando esta variable en falso.

```
class MiDemonio extends Thread {

  private corriendo = true;

  public MiDemonio(String nombre) {

    this.setName(nombre);
    this.setDaemon(true);

  }

  public void run() {

      while(corriendo) {

            // CODIGO DEL DEMONIO AQUÍ
      }
  }

  public synchronized void stop() {

      this.corriendo = false;
  }
}
```

De esta manera es posible planear el funcionamiento del programa correctamente, evitando problemas producto de la terminación a medias de una hebra. Estos errores suelen ser los más difícil de encontrar en una aplicación y por tanto deben tomarse todas las medidas preventivas posibles.

4.3. Yield y currentThread

La clase Thread provee 2 métodos interesantes que cabe mencionar.

Por una parte se tiene el método estático ***currentThread()***, el cual retorna un puntero a la hebra que esta actualmente ejecutándose en este proceso java.

```
class MiDemonio extends Thread {

  public void run() {

      Thread yo = Thread.currentThread();
  }
}
```

El método ***yield()*** por su parte, cede el tiempo de CPU que tiene esta hebra, a la siguiente. Útil para procesos demonios y otros códigos donde se realiza una acción condicionada a una verificación, de lo contrario no hace nada.

```
class MiDemonio extends Thread {

  private corriendo = true;

  public void run() {

      while(corriendo) {

        if(condicionOK)
            // codigo a ejecutar
        else
            yield();
      }
  }
}
```

4.4. Dormir y esperar

Java también provee mecanismos para poder bloquear la ejecución de una hebra por tiempo determinado o poder esperar al término de otra para poder continuar.

El método ***sleep()*** de Thread permite "dormir" a la hebra por la cantidad de milisegundos indicados en el parámetro. El método sleep puede ser interrumpido, lanzándose una InterruptedException por lo que debe ir en un bloque try-catch. Más información sobre interrupciones en el siguiente punto.

```
class MiDemonio extends Thread {

  private corriendo = true;

  public void run() {

      while(corriendo) {

        // codigo a ejecutar

        try {
            sleep(1 * 1000);
        }
        catch(InterruptedException e) { }

      }
  }
}
```

Por otra parte ***join()*** permite que una hebra espere a que otra termine su ejecución. A través de este comando, la hebra que lo invoca queda bloqueada hasta que la termine el método run de la hebra en cuestión. Al igual que el método sleep(), también debe colocarse dentro de un bloque try-catch dado que la hebra a la cual se espera podría terminar de forma interrumpida.

```
public static void main(String[] args) {

      Hebra hebra1 = new Hebra();
      hebra1.start();

      try {

        hebra1.join();
        System.out.println("hebra1 ha terminada OK");
      }
      catch(InterruptedException e) { }
}
```

También existe un método ***sleep(long milis)*** y ***sleep(long milis, int nanos)*** que permite limitar la cantidad de tiempo de espera. Todas las variaciones solo pueden ser ejecutadas por la hebra para si y no para otras hebras.

4.5. Interrupción

Una interrupción es una señal a una hebra para que esta deje de hacer lo que este haciendo y realice alguna otra acción, análogo a las interrupciones de sistema operativo.

Para interrumpir a una hebra, se invoca al método ***interrupt()***. Existen 2 manera de que una hebra haga uso de las interrupciones: manejar adecuadamente las InterruptedExceptions y consultar el método ***interrupted()***.

Para el primer caso, interrumpimos el bloqueo de una hebra ya sea por invocación a sleep o join. Aquí una hebra externa puede reanudar la ejecución de la hebra detenida forzosamente, produciéndose el lanzamiento de la InterruptedException.

HEBRA SLEEP INTERRUMPIDO

```
class Hebra extends Thread {

  public void run() {

    Date inicio = new Date();

    try {

       sleep(10 * 1000);
       System.out.println("Dormido 10 segundos")
    }
    catch(InterruptedException e) {

       long diff = (new Date()).getTime() – inicio.getTime();
       System.out.println("Dormido: " + diff + " milisegundos.");
    }

  }
```

HEBRA INTERRUPTORA SLEEP

```
public static void main(String[] args) {

   Hebra h1 = new Hebra();
   h1.start();
   try { Thread.sleep(1000) } catch(InterruptedException e) { }

   h1.interrupt();

}
```

Nótese como en la hebra main invocamos al método estático sleep, el cual aplica sobre la “hebra actual”, que claramente es aquella que esta ejecutando el comando.

El funcionamiento con join() es análogo. La hebra que esta esperando a que otra termine, ve su bloqueo interrumpido externamente, debiendo manejar correctamente la excepción.

Ahora, para el caso de hebras que no tienen sleep o join en un su código, sino que solo ejecución continua de código, es el programador el encargado de verificar periódicamente si se han recibido interrupciones invocando al método *interrupted()*.

```
class MiDemonio extends Thread {

  private corriendo = true;

  public void run() {

      while(corriendo) {

         if(interrupted()) {

            // manejo de la interrupción
            // possible termino de la ejecución
            break;
         }
         else {

            // codigo ejecución normal
         }

      }
  }
```

5. Elementos de Sincronización

5.1. Generalidades

Desde el momento en que se utiliza programación multiproceso o multihebra (concurrente), afloran una serie de problemas sobre el trabajo de los datos por parte de las múltiples hebras. Algo tan simple como i++ puede ser complicado. Esta operación que al parecer es unitaria, son de verdad 3:

- Consulta del valor de i
- Aumento del valor en 1
- Asignación del nuevo valor a la variable i

Imaginemos que existen 2 procesos que trabajan con la misma variable i. Si ambos ejecutan al mismo tiempo las instrucciones, ambos podrían obtener el valor inicial de i (ej: i = 0), sumar 1 al valor (resultado = 1) y asignar ese valor a i, resultando en un resultado final erróneo dado que seria solo 1 unidad más, en caso que debería ser 2.

Para solucionar este y otros problemas, existen múltiples soluciones implementadas por hardware y software.

Una de las soluciones clásicas son los *semáforos*. Java incluye además *monitores* para otorgar otra herramienta de sincronización. Sobre estos elementos, Java incorpora como parte de su distribución J2SE el paquete *java.util.concurrent* que contiene un conjunto de clases e interfaces listas para su uso en el manejo de la concurrencia.

Proceso 1	***Programa***	***Proceso 2***

i = 0
i = 1
	`class Hebra extends Thread {`
	`  public static int i = 0;`
	`  public void run() {`
	`  System.out.println("i = " + i);`
	`  i++;`
	`  System.out.println("i = " + i);`
	`  }`
	`}`
		i = 0
		i = 1

Ejemplo de ejecución errónea cuando se tienen múltiples hebras ejecutando un mismo código

5.2. Monitores

Los monitores son por definición un método de proporcionar exclusión mutua que esta implementado en el lenguaje de programación. Java implementa monitores como parte de su lenguaje base, entregando una herramienta embebida en cada objeto Java, dado que se encuentran en *Object*.

El uso de monitores en Java es muy simple; basta colocar la palabra clave ***synchronized*** antecediendo el nombre del método a sincronizar. Esto protegerá que solo 1 hebra a la vez pueda acceder a los todos métodos que se declaren sincronizados dentro de la misma clase. Estos locks a los métodos son reentrantes, es decir, que cuando una hebra ha adquirido el monitor en un método, puede reentrar al mismo método u otro sincronizado sin necesidad de liberarlo.

```
public class Hebra extends Thread {

  public void run() {

      Hebra.metodo();
  }

  public static synchronized void metodo() {

    System.out.println("Hebra ingresada " + Thread.currentThread().getName());
    try { Thread.sleep(3000); } catch(Exception e) {}
  }

  public static synchronized void metodo2() {

    System.out.println("Ingresada " + Thread.currentThread().getName());
    try { Thread.sleep(3000); } catch(Exception e) {}
  }

}
```

```
public static void main(String[] main) {

  Hebra h = new Hebra();
  h.start();

  Hebra.metodo2();

}
```

Por defecto Java sigue ejecutando la hebra que inicie a otra hasta que ocurra el timeslice de la JVM. Así, primero se imprimirá en pantalla “Ingreada main” y 3 segundos después “Hebra ingresada Thread-0”. El monitor fue adquirido primero por la hebra “main” y una vez que sale del método lo libera permitiendo que la hebra “Thread-0” pueda adquirirlo.

Si bien lo monitores Java entregan una solución para la mayoría de los casos, su capacidad de ser adquiridos por una hebra a la vez es una limitante no menor. Supongamos el caso de un objeto centralizador de recursos con múltiples métodos estáticos para acceder a tales recursos. En este caso cuando 1 hebra adquiere el monitor para utilizar 1 recurso, bloquea el acceso a todo el resto de los recursos aun cuando sean independientes. Para solucionar correctamente este problema, se pueden utilizar semáforos.

5.3. Semáforos

Los semáforos son una de las soluciones clásicas al problema de concurrencia. Concebidos en 1965 por Dijkstra, son implementados en java a través del objeto ***java.util.concurrent.Semaphore***.

Formalmente un semáforo es un elemento que lleva en su interior un contador que indica la cantidad inicial de autorizaciones de paso o "banderitas". Un proceso al querer entrar a una *región crítica*[1] adquiere una banderita del semáforo (formalmente en la literatura métodos llamados p() o down()) y cuando la deja lo libera (formalmente v() o up()). Si existen banderitas disponibles, entonces el proceso continua su ejecución, de lo contrario queda bloqueado a la espera que otro proceso libere alguna banderita y se pueda continuar.

Un ejemplo de utilización de semáforos, puede ser la protección de una variable, de manera que no se produzca el error mencionado en el punto anterior. Un simple semáforo binario (2 estados: con y sin banderita), permite asegurar que solo un proceso a la vez modifique el valor de la variable i. Para esto inicializamos una variable estática (para que sea única entre todas las instancias de la clase) con el semáforo en 1.

```
class MiHebra extends Thread {

  private static Semaphore mutex = new Semaphore(1);
  private static int i = 0;

  public void run() {

    while(running) {

       try {

         mutex.acquire();
         i++;
         mutex.release();
       }
       catch(InterruptedException e) { }
    }
  }
}
```

[1] Región Crítica es aquella porción del código de un programa que al ser ejecutado por más procesos que los intencionados originalmente por el programador, produce un resultado indeterminado en la computación.

Aquí hemos colocado el match a la InterruptedException, dado que durante la espera de adquisición de semáforo, la hebra podría ser interrumpida externamente. Para evitar este comportamiento se tienen 2 alternativas. La primera es usar el método ***acquireUninterrumptibly*** el cual no interrumpe el bloqueo de espera por el semáforo; la segunda es trabajar con el método ***tryAcquire*** el cual no bloquea la ejecución, retornando true cuando se pudo adquirir el semáforo y false cuando no.

5.4. java.util.concurrent

Adicionalmente a los semáforos, Java provee una serie de objetos para el control de la concurrencia listos para su uso. A continuación una breve lista de objetos *thread-safe* :
y su utilidad.

Paquete java.util.concurrent	
ConcurrentHashMap	Hashmap con operaciones atómicas
ConcurrentLinkedQueue	Lista enlazada con operaciones atómicas.
CyclicBarrier	Barreras en Java
Semaphore	Semáforos en Java
Paquete java.util.concurrent.atomic	
AtomicBoolean	Booleano con operaciones atómicas, incluido testAndSet.
AtomicInteger	Entero con operaciones atómicas, incluido testAndSet.
AtomicIntegerArray	Arreglo de enteros con operaciones atómicas, incluido testAndSet.
AtomicLong	Real con operaciones atómicas, incluido testAndSet.
AtomicReference	Referencia a un objeto (cualquiera) con operaciones atómicas.
Paquete java.util.concurrent.lock	
ReentrantLock	Lock reentrante, aquel que permite que una misma hebra lo adquiera múltiples veces luego de adquirirlo la primera vez, bloqueando a otras hebras.

EOF - - -

Jonathan Makuc
http://jmakuc.ublog.cl