

Programación Multihilo

Programación de Servicios y Procesos

2 DAM

Manuel Montoro

Índice

- Concepto de Hilo (Thread)
- Hilos en Java
- Hilos en Java – Clase Thread
- Hilos en Java – Interfaz Runnable
- Hilos en Java – Fin de ejecución de un hilo
- Hilos en Java – Trabajo con hilos
- Datos compartidos - Problemas
- Solución – Los Monitores

Concepto de Hilo (Thread)

- Como vimos en el tema de la programación multiproceso, los sistemas operativos modernos son multiproceso y multitarea, esto es, pueden ejecutar varios procesos al mismo tiempo (multiproceso) e incluso, si es posible, de forma paralela (multitarea)
- El problema es que los procesos son pesados y costosos de mantener por parte del S.O.
- Cada proceso mantiene una gran cantidad de información sobre sí mismo en la memoria del S.O. y son lentos, comparativamente hablando, de crear, mantener y cerrar.
- Además, la comunicación entre procesos es complicada porque tiene que saltar las barreras de seguridad impuestas por los S.O para evitar precisamente dicha comunicación y aislar lo más posible unos procesos de otros

Concepto de Hilo (Thread) (II)

- Por ello los sistemas operativos modernos comenzaron a proporcionar un nuevo mecanismo para:
 - Proporcionar un mecanismo más simple para la multitarea que la creación y destrucción de procesos
 - Proporcionar un mecanismo más ágil para la comunicación entre las distintas tareas.
- El mecanismo desarrollado se llama el hilo o thread.
- Un hilo es una tarea en ejecución dentro de un proceso. Tiene su propia cuota de CPU y su propia pila de llamadas.
- Pueden existir más de un hilo dentro de un sólo proceso. Al mínimo hay un sólo hilo, lo que se correspondería con un proceso clásico.
- Los hilos, al vivir dentro de un mismo proceso, comparten el programa y los datos, es decir, ejecutan el mismo programa (aunque, por supuesto, en cada momento cada hilo estará ejecutando una parte distinta del programa) y comparten los mismos datos del programa

Concepto de Hilo (Thread) (III)

- El compartir los datos hace que la comunicación entre los distintos hilos sea más simple:
- Dos hilos se pueden comunicar simplemente mediante una variable. Lo que uno escriba en una variable, otro puede leerlo. Ya no son necesarios archivos ni líneas de comando para intercambiar información. Por ejemplo, si dos hilos trabajan sobre una lista de números que van a procesar entre los dos, pueden compartirla simplemente pasando una referencia a la lista a los dos hilos. Éstos accederían directamente a la lista leyendo o escribiendo usando las mismas operaciones que se utilizarían en caso de un proceso clásico (los métodos que ofrece la lista)

Hilos en Java

- En Java toda aplicación consta inicialmente de un sólo hilo, que comienza su ejecución por la función `main` de la clase elegida como clase de arranque o inicio de la aplicación.
- Si se quieren crear más hilos hay que hacerlo de forma explícita mediante desde el programa.
- Hay dos formas de crear un nuevo hilo:
 - Mediante la clase `java.lang.Thread`
 - Mediante el interface `java.lang.Runnable`
- Los dos realmente involucran a la clase `Thread`. La necesidad de los dos métodos viene dada por cuestiones de herencia.

Hilos en Java – Clase Thread

- Si se desea crear uno o más hilos mediante la clase Thread hay que....
 - Crear una nueva clase que herede (extends) la clase Thread. Por ejemplo:

```
class MiHilo extends Thread
```
 - Redefinir en dicha clase el método run(), heredado de Thread
- Haciendo esto ya se tiene disponible la infraestructura necesaria para crear y ejecutar un nuevo hilo. Cuando se cree el nuevo hilo la ejecución comenzará de éste invocando el método run() de la clase MiHilo que acabamos de crear que en este caso sería equivalente al método main() del programa pero para un hilo en concreto.

Hilos en Java – Clase Thread (II)

- Para crear e iniciar un hilo usando la clase que hemos creado debemos....
 - Crear un objeto de la clase (la cual, recordemos, hereda de Thread). Por ejemplo:
`MiHilo hilo = new MiHilo()`
 - Invocar al métodos `start()` del nuevo objeto. En este momento es cuando se crea el nuevo hilo y se da inicio de la ejecución invocando al método `run()` del objeto. OJO: NO HAY QUE INVOCAR DIRECTAMENTE AL MÉTODO `run()` PORQUE ENTONCES NO SE CREA HILO SINO QUE SE PORTA COMO UNA LLAMADA A MÉTODO NORMAL. Por ejemplo:
`hilo.start()`
- Si se desean crear más hilos del mismo tipo se puede volver a repetir este proceso cuantas veces se quiera (y la máquina lo soporte)

Hilos en Java – Clase Thread (III)

- Como se puede ver, todos los hilos que se creen mediante una misma clase van a iniciar su ejecución por el mismo método `run()` por lo que se supone que realizarán una tarea más o menos similar.
- Si se desean hacer hilos que realicen tareas distintas se pueden declarar más clases que hereden de `Thread`, cada una con su método `run()` personalizado a la tarea en cuestión.

Hilos en Java – Interfaz Runnable

- El problema con el método que se acaba de ver es que la clase que actúa como punto de inicio del hilo debe heredar de Thread.
- Ésto puede ser un problema si también queremos que dicha clase herede de otra distinta a Thread, ya que Java no permite la herencia (`extends`) de más de una clase.
- En este caso hay un mecanismo más indirecto para crear un hilo haciendo uso de la interfaz Runnable.
- Esta interfaz define un único método: `run()`, igual al de Thread.

Hilos en Java – Interfaz Runnable (II)

- El procedimiento para crear un hilo mediante Runnable es:
 - Se crea una clase que implemente (implements) el interfaz Runnable, por ejemplo:

```
class MiHilo2 extends JFrame implements Runnable
```
 - Se define el método run() en esa clase
- Con esto ya tenemos la infraestructura, como en el caso de Thread. Ahora para crear el hilo....
 - Se crea un nuevo objeto de la clase que implementa Runnable. Por ejemplo:

```
MiHilo2 hilo2 = new MiHilo2();
```
 - Se crea un nuevo objeto de la clase Thread y mediante el constructor se le pasa el objeto creado en el paso anterior. Por ejemplo:

```
Thread t = new Thread(hilo2);
```
 - Se invoca al método start() del objeto de la clase Thread recién creado. Por ejemplo:

```
t.start();
```
- Esto tiene el mismo resultado que en el procedimiento original. La diferencia es que la clase en la que va el método run() puede heredar de otra clase si quiere. Además, dado que Java SI permite implementar múltiples interfaces, el implementar Runnable no evita que la clase pueda implementar más interfaces, si lo desea.

Hilos en Java – Fin de ejecución de un hilo

- ¿Cuando finaliza un hilo en Java?
- El hilo principal finaliza cuando se termina de ejecutar la función `main()`.
- Los hilos secundarios finalizan cuando terminan de ejecutar sus métodos `run()` respectivos.
- Un hilo puede esperar a que termine otro usando el método `join()` de la clase `Thread`. Este método puede llevar un tiempo en milisegundos opcional. En este caso sólo se espera el tiempo indicado como máximo.
- Habitualmente hay dos tipos de hilos:
 - Hilos de un sólo uso. Son hilos que se crean para una tarea concreta. Estos hilos se inician, realizan su tarea y terminan solos.
 - Hilos de servicio. Estos hilos corren continuamente esperando eventos y realizan alguna tarea cuando estos ocurren. Para finalizar estos hilos hay que añadir siempre un evento de finalización que haga que el hilo termine su ejecución. Esto se hace normalmente con una variable booleana.

Hilos en Java – Trabajo con hilos

- Hay que tener en cuenta las siguientes consideraciones cuando se trabaja con hilos:
 - Todos los hilos que están en funcionamiento se ejecutan de forma simultánea. Es como si varias cpus (una por hilo) estuvieran leyendo y ejecutando el mismo programa al mismo tiempo.
 - Los distintos hilos pasan del estado de ejecución (están haciendo algo) al de espera (están en pausa) controlados por un controlador interno de la máquina virtual Java que decide, en función de varios factores: qué hilos están en ejecución en cada momento, cuáles en pausa y cómo se pausan o reinician los hilos. El programador tiene poco control sobre este funcionamiento, siempre de forma indirecta y en muchos casos sin garantías.
 - Los objetos de la clase Thread y descendientes no son reclamados por el colector de basura si el hilo al que corresponden no ha terminado aún.

Hilos en Java – Trabajo con hilos (II)

- Algunos métodos interesantes con Threads:

- `currentThread()`. Devuelve un objeto de la clase Thread correspondiente al hilo que ha ejecutado el método. Se puede usar en código compartido entre varios hilos para saber cual de ellos es el que está ejecutando un bloque de código en un momento dado.
- `setName(String)` / `getName()`. Nos permite asignar un nombre a un hilo y obtenerlo. Esto puede ser útil para identificar los distintos hilos.
- `getId()`. Obtiene un ID numérico único para cada hilo. Este ID lo genera la máquina virtual cada vez que se crea un hilo y se garantiza que es único.
- `setPriority(int)` / `getPriority()`. Estos métodos permiten asignar una prioridad a un hilo y obtener la prioridad actual del mismo. Cuanto mayor sea el valor numérico de la prioridad de un hilo mayor será la preferencia que tenga el controlador de hilos para darle tiempo de ejecución sobre otros hilos de menor valor de prioridad). Hay que tener cuidado con esto porque puede ocurrir que algunos hilos no se ejecuten nunca porque su prioridad es demasiado baja. En su creación un hilo tiene prioridad `Thread.NORM_PRIORITY`.
- `sleep(long)` / `sleep(long, long)`. Suspende el hilo que ejecuta el método durante al menos el tiempo especificado. La versión con un sólo número admite tiempo en milisegundos mientras que la de dos admite tiempo en milisegundos y nanosegundos. Usar `sleep()` es una manera de que un hilo puede ceder tiempo de ejecución a otros hilos del mismo proceso. Otra forma es usando el método `yield()`.

Datos compartidos - Problemas

- Cuando un dato es modificado de forma exclusiva por un sólo hilo no hay ningún problema.
- Estos se presentan cuando hay varios hilos que escriben o modifican los mismos datos. En este caso se pueden presentar *carreras*.
- Una carrera se produce cuando más de un hilo modifica un mismo dato. Dependiendo del orden en que ocurran las operaciones o de si se interrumpe o no un hilo en medio de una de ellas se pueden producir efectos no deseados e inconsistencias.

Datos compartidos – Problemas (II)

- Un posible escenario de este tipo con dos hilos podría ser el siguiente.
- Imaginemos dos hilos que usan un contador compartido. Cada uno de ellos lo usa para llevar una cuenta global o para determinar qué recurso se va a procesar (el que tenga el valor del contador como ID, por ejemplo).
 - El hilo 1 accede al contador y lee su valor, por ejemplo 100
 - El hilo 2 accede al contador y lee su valor, leyendo también 100.
 - El hilo 1 incrementa el valor en 1, obteniendo 101
 - El hilo 1 guarda el valor en el contador, pasando éste a valer 101
 - El hilo 2 incrementa el valor en 1, obteniendo 101 también
 - El hilo 2 guarda el valor en el contador, pasando éste a valer 101
- Como se ve se ha producido un error y el contador, en lugar de valer 102, como debía de ser, vale 101 porque las instrucciones de ambos hilos se han mezclado en una secuencia desafortunada.
- Lo peor de estos errores es que se presentan de forma aparentemente aleatoria y son difíciles de reproducir.

Solución – Los Monitores

- Para solucionar este problema, Java emplea monitores.
- Un monitor es un bloque de código al cual sólo puede acceder un hilo en un momento determinado.
- Si un hilo llega a un monitor y no hay ningún otro dentro de él o esperando, lo bloquea y entra. Cuando sale lo desbloquea.
- Si un hilo llega a un monitor y está bloqueado, se pone en cola en la entrada y se suspende. Cuando se desbloquee el monitor despierta a alguno de los hilos que está en la cola y le deja entrar.

Solución – Los Monitores (II)

- Por lo tanto, cuando una operación pueda ser realizada por más de un hilo y exista un problema de concurrencia se puede introducir dentro de un monitor de forma que el problema se solucione.
- Aunque puedan aparecer otros:
 - Menor eficiencia. Los monitores introducen esperas ya que sólo un hilo puede estar dentro en un momento dado. Esto provoca esperas sobre todo si es un fragmento de código que se usa mucho. En este caso habría que reconsiderar el diseño del programa para intentar eliminar el cuello de botella.
 - Interbloqueos. Los interbloqueos ocurren cuando dos o más procesos están esperandose entre si, de forma que cada uno bloquea el progreso de los demás. Esta situación no es tan corriente como las anteriores pero puede ocurrir. Son problemas difíciles de detectar y de solucionar.

Solución – Los Monitores (III)

- En Java un monitor se implementa mediante la palabra clave `synchronized`.
- Si una función lleva el modificador `synchronized`, se comporta automáticamente como un monitor.
- También se puede hacer, aunque es menos corriente, usando `synchronized` con una referencia a un objeto
`synchronized (objeto) { instrucciones dentro del monitor }`

FIN