HILOS

Objetivo

El objetivo de la guía es el de afianzar el conocimiento acerca de los hilos para resolver problemas reales con el uso del lenguaje de programación Java. Además le presentará las diferentes variantes de dicho mecanismo y los pros y contras de cada una.

Competencias

- Entender el concepto de Hilo
- Utilizar el lenguaje Java para implementar Hilos
- Resolver problemas reales mediante el uso de Hilos

Fundamentos

Qué es un Hilo

Conceptualmente, la noción de un hilo no es difícil de entender: es una vía de ejecución independiente a través del código de un programa. Cuando se ejecutan varios subprocesos, la ruta de un subproceso a través del mismo código normalmente difiere de las demás. Por ejemplo, supongamos que un subproceso ejecuta el equivalente de la parte if de un bloque if-else, mientras que otro subproceso ejecuta el equivalente de la parte else. ¿Cómo realiza la JVM el seguimiento de la ejecución de cada hilo? La JVM da a cada hilo su propio método de pila de llamadas. Además de seguir la instrucción actual, la pila de llamadas controla las variables locales, los parámetros que la JVM pasa a un método y el valor de retorno del método.

Cuando varios subprocesos ejecutan secuencias de instrucciones de bytecode en el mismo programa, esa acción se conoce como multithreading. Multithreading beneficia a un programa de varias maneras:

- Los programas basados en GUI (interfaz gráfica de usuario) multithreaded siguen respondiendo a los usuarios mientras realizan otras tareas, como repaginar o imprimir un documento.
- Los programas con hilos típicamente terminan más rápido que sus homólogos sin hilos. Esto es especialmente cierto en los hilos que se ejecutan en una máquina multiprocesador, donde cada hilo tiene su propio procesador.

Java realiza el multithreading a través de su clase java.lang.Thread. Cada objeto Thread describe un solo hilo de ejecución. Esta ejecución se produce en el método run() de Thread.

Debido a que el método run() por defecto no hace nada, debe crearse una subclase de Thread y sobrecargar el método run() para realizar un trabajo útil. Para una muestra de subprocesos y multihilo en el contexto de Thread, examine el Listado 1:

Listado 1. ThreadDemo.java

```
// ThreadDemo.java
class ThreadDemo
{
   public static void main (String [] args)
   {
      MyThread mt = new MyThread ();
      mt.start ();
      for (int i = 0; i < 50; i++)
           System.out.println ("i = " + i + ", i * i = " + i * i);
   }
}
class MyThread extends Thread
{
   public void run ()
   {
      for (int count = 1, row = 1; row < 20; row++, count++)
      {
           for (int i = 0; i < count; i++)
                System.out.print ('*');
           System.out.print ('\n');
      }
   }
}
```

El Listado 1 presenta el código fuente de una aplicación que consta de las clases ThreadDemo y MyThread. La Clase ThreadDemo ejecuta la aplicación creando un objeto MyThread, iniciando un subproceso que se asocia con ese objeto y ejecutando algún código para imprimir una tabla de cuadrados. Por el contrario, MyThread sobrecarga el método run() de Thread para imprimir (en el flujo de salida estándar) un triángulo de ángulo recto compuesto de caracteres de asterisco.

Programación de subprocesos y la JVM

La mayoría (si no todas) las implementaciones JVM utilizan las capacidades de subprocesamiento de la plataforma subyacente. Debido a que estas capacidades son específicas de la plataforma, el orden de salida de los programas multiproceso puede diferir del orden de la salida de otra persona.

Cuando escribe java ThreadDemo para ejecutar la aplicación, la JVM crea un subproceso de ejecución que ejecuta el método main(). Al ejecutar mt.start () ;, el subproceso de inicio le dice a la JVM que cree un segundo subproceso de ejecución que ejecute las instrucciones de BYTECODE que comprenden el método run() del objeto MyThread. Cuando el método start() retorna, el subproceso de inicio ejecuta su bucle for para imprimir una tabla de cuadrados, mientras que el subproceso nuevo ejecuta el método run() para imprimir el triángulo de ángulo recto.

¿Cómo se ve la salida? Ejecute ThreadDemo para averiguarlo. Notará que la salida de cada hilo tiende a intercalar con la salida del otro. Esto se debe a que ambos hilos envían su salida a la misma corriente de salida estándar

La clase Thread

Para ser eficiente en la escritura de código multiproceso, primero debe comprender los diversos métodos que componen la clase Thread. Esta sección explora muchos de estos métodos. Específicamente, aprenderá acerca de métodos para iniciar subprocesos, nombrar subprocesos, poner los hilos en suspensión, determinar si un subproceso está vivo, unir un subproceso a otro subproceso y enumerar todos los subprocesos activos en el subgrupo de subprocesos del subproceso actual. También se presentan las ayudas de depuración de Thread y de los subprocesos de usuario frente a los subprocesos de daemon.

Métodos obsoletos
Sun ha descontinuado una variedad de métodos Thread, como suspend() y resume(), porque pueden bloquear sus programas o dañar objetos. Como resultado, no debe llamarlos en su código. Consulte la documentación del SDK para ver las soluciones a esos métodos.

Construyendo hilos

El hilo tiene ocho constructores. Los más simples son:

- Thread (), que crea un objeto Thread con un nombre predeterminado
- Thread (String name), que crea un objeto Thread con un nombre que especifica el argumento name

Los siguientes constructores más simples son Thread (Runnable target) y Thread (Runnable target, String name). Aparte de los parámetros Runnable, aquellos constructores son idénticos a los constructores mencionados anteriormente. La diferencia: Los parámetros Runnable identifican objetos fuera de Thread que proporcionan los métodos run (). Los últimos cuatro constructores se asemejan a Thread (String name), Thread (Runnable target) y Thread (Runnable target, String name); Sin embargo, los constructores finales también incluyen un argumento ThreadGroup con fines organizativos.

Uno de los cuatro constructores finales, Thread(group ThreadGroup, target Runnable, name String, stackSize long), es interesante ya que le permite especificar el tamaño deseado de la pila del hilo. Ser capaz de especificar el tamaño resulta útil en los programas con métodos que utilizan la recursión -una técnica de ejecución mediante la cual un método se llama repetidamente- para resolver elegantemente ciertos problemas. Al establecer explícitamente el tamaño de pila, a veces puede evitar StackOverflowErrors. Sin embargo, un tamaño demasiado grande puede resultar en OutOfMemoryErrors. Además, Sun considera el tamaño de la pila de llamadas como dependiente de la plataforma. Dependiendo de la plataforma, el tamaño de la pila de llamadas podría cambiar. Por lo

tanto, piense cuidadosamente acerca de las ramificaciones de su programa antes de escribir el código que llama Thread(group ThreadGroup, target Runnable, name String, stackSize long).

Inicie sus vehículos

Los hilos se parecen a los vehículos: mueven los programas de principio a fin. Los objetos de las subclases de Thread y Thread no son subprocesos. En su lugar, describen los atributos de un subproceso, como su nombre, y contienen código (mediante un método run()) que ejecuta el subproceso. Cuando llega el momento de que un nuevo subproceso ejecute run(), otro subproceso llama al método start() del Thread o de una subclase de Thread. Por ejemplo, para iniciar un segundo subproceso, el subproceso de inicio de la aplicación, que ejecuta main(), llama a start(). En respuesta, el código de gestión de subprocesos de la JVM funciona con la plataforma para garantizar que el subproceso se inicializa correctamente y llama al método run() de un subproceso Thread o de una subclase de Thread.

Una vez que start() finaliza, se ejecutan múltiples threads. Debido a que tendemos a pensar de manera lineal, a menudo nos resulta difícil entender la actividad concurrente (simultánea) que se produce cuando dos o más subprocesos se están ejecutando. Por lo tanto, debe examinar un gráfico que muestra dónde se está ejecutando un subproceso (su posición) en función del tiempo. La siguiente figura presenta un gráfico de este tipo.

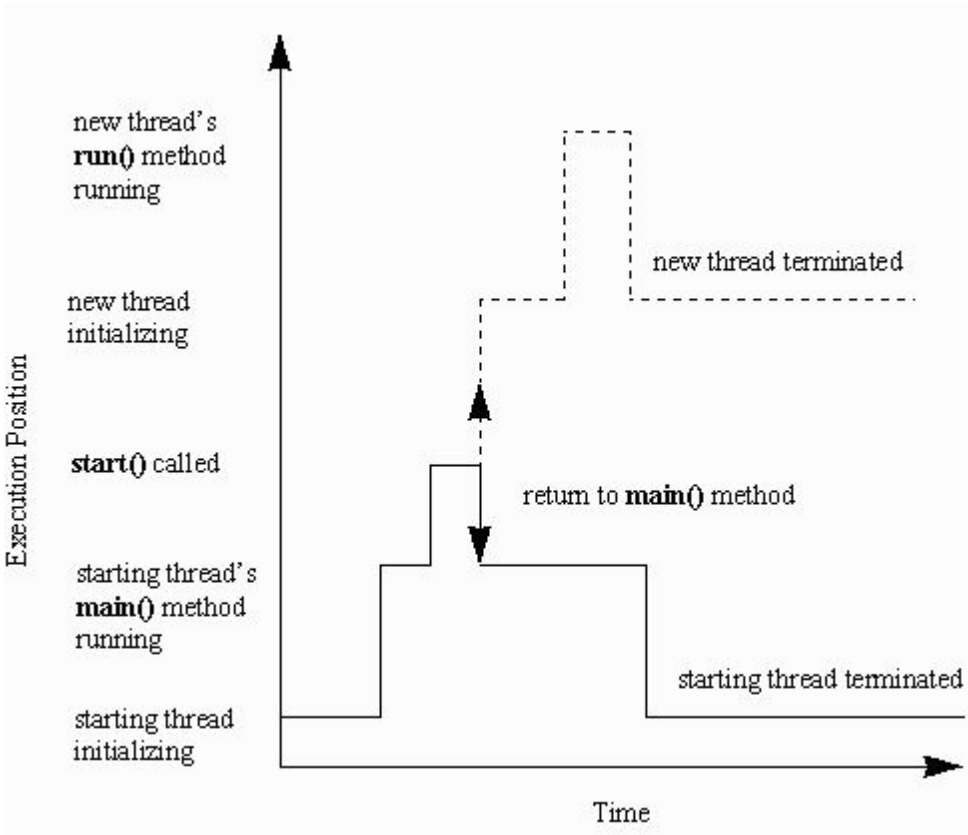

Figure 1. El comportamiento de hilo pincipal y un nuevo hilo credo, mostrando su poición de ejecución versus el tiempo.

El gráfico muestra varios períodos de tiempo significativos:

- La inicialización del hilo inicial
- El momento en que el hilo comienza a ejecutar main ()
- El momento en que el hilo comienza a ejecutar start ()
- El momento start () crea un nuevo hilo y vuelve a main ()
- La inicialización del nuevo hilo
- En el momento en que el nuevo subproceso comienza a ejecutar ejecutar ()
- Los diferentes momentos en que cada hilo termina

Tenga en cuenta que la inicialización del nuevo subproceso, la ejecución de run() y su terminación ocurren simultáneamente con la ejecución del hilo principal. También tenga en cuenta que después de que un hilo llama a start(), las llamadas posteriores a ese método antes del método run() producen la excepción java.lang.IllegalThreadStateException

¿Qué hay en un nombre?

Durante una sesión de depuración, distinguir un hilo de otro de una manera fácil resulta útil. Para diferenciar entre los subprocesos, Java asocia un nombre a un subproceso. El nombre predeterminado es Thread, un carácter de guión y un número entero que inicia desde cero. Puede aceptar los nombres de subproceso predeterminados de Java o puede elegir los propios. Para acomodar nombres personalizados, Thread proporciona constructores que toman argumentos de nombre y un método setName(String name). Thread también proporciona un método getName() que devuelve el nombre actual. El Listado 2 muestra cómo establecer un nombre personalizado a través del constructor Thread(String name) y recuperar el nombre actual en el método run() llamando a getName():

Listing 2. NameThatThread.java (implementar este ejemplo)

```
// NameThatThread.java

public class NameThatThread
{
   public static void main (String [] args)
   {
      MyThread mt;
      if (args.length == 0)
          mt = new MyThread ();
      else
          mt = new MyThread (args [0]);
      mt.start ();
   }
}

class MyThread extends Thread
```

```
{
   MyThread ()
   {
      // The compiler creates the byte code equivalent of super ();
   }

   MyThread(String name)
   {
      super(name); // Pass name to Thread superclass
   }
   public void run()
   {
      System.out.println("My name is: " + getName ());
   }
}
```

Puede pasar un argumento opcional a MyThread en la línea de comandos, el nombre. Por ejemplo, `java NameThatThread X` establece X como el nombre del hilo. Si no especifica un nombre, verá la siguiente salida:

Mi nombre es: Thread-1

Si lo prefiere, puede cambiar el `super(nombre);` Llamar en el constructor MyThread(String name) a setName(String name) - como setName (name) ;. Este último método de llamada alcanza el mismo objetivo-establecer el nombre del hilo-como super(nombre) ;. Dejo eso como un ejercicio para ti.

El nombre del hilo main

Java asigna el nombre main al subproceso que ejecuta el método main(), el subproceso de inicio. Normalmente, usted ve ese nombre en el mensaje Exception in thread "main" que el manejador de excepciones predeterminado de JVM imprime cuando el subproceso de inicio lanza una excepción.

Dormir o no dormir

Más adelante en esta columna, os presentaré una animación, dibujando repetidamente en una superficie imágenes que difieren ligeramente unas de otras para lograr una ilusión de movimiento. Para realizar la animación, un hilo debe pausar durante la visualización de dos imágenes consecutivas. Llamar el método de sueño estático de Thread (long millis) obliga a un hilo a pausar durante algunos milisegundos. Otro hilo podría interrumpir el hilo que duerme. Si eso sucede, el hilo durmiente se despierta y lanza una excepción InterruptedException desde el método sleep(long millis). Como resultado, el código que llama a sleep(long millis) debe aparecer dentro de un bloque try, o el el método debe incluir InterruptedException en su cláusula throws.

Para demostrar sleep(long millis), he escrito una aplicación CalcPI1. Esta aplicación inicia un nuevo subproceso que utiliza un algoritmo matemático para calcular el valor de la constante matemática pi. Mientras el nuevo subproceso calcula, el hilo de inicio se detiene durante 10 milisegundos llamando a sleep(long millis). Después de despertar el subproceso de inicio, se imprime el valor pi, que el nuevo hilo almacena en la variable pi. El Listado 3 presenta el código fuente de CalcPI1:

Listado 3. CalcPI1.java (implementar este ejemplo)

```
// CalcPI1.java
public class CalcPI1
{
   public static void main(String[] args)
   {
      MyThread mt = new MyThread();
      mt.start();
      try
      {
          Thread.sleep(10); // Sleep for 10 milliseconds
      }
      catch(InterruptedException e)
      {
      }
      System.out.println ("pi = " + mt.pi);
   }
}

class MyThread extends Thread
{
   boolean negative = true;
   double pi; // Initializes to 0.0, by default

   public void run()
   {
      for (int i = 3; i < 100000; i += 2)
      {
           if (negative)
               pi -= (1.0 / i);
           else
               pi += (1.0 / i);
           negative = !negative;
      }
      pi += 1.0;
      pi *= 4.0;
      System.out.println ("Finished calculating PI");
   }
}
```

Si ejecuta este programa, verá una salida similar (pero probablemente no idéntica) a lo siguiente:

```
pi = -0.2146197014017295
Finished calculating PI
```

¿Por qué la salida es incorrecta? Después de todo, el valor de pi es aproximadamente 3.14159. La respuesta: El hilo de partida se despertó demasiado pronto. Justo cuando el nuevo hilo comenzaba a calcular pi, el hilo inicial se despertó, leyó el valor actual de pi e imprimió ese valor. Podemos compensar aumentando el retardo de 10 milisegundos a un valor más largo. Ese valor más largo, que (por desgracia) depende de la plataforma, dará al nuevo hilo la posibilidad de completar sus cálculos antes de que el hilo de inicio se despierte. (Más adelante, aprenderá sobre una técnica independiente de la plataforma que impide que el hilo de inicio se despierte hasta que el nuevo hilo termine.)

Los hilos durmientes no mienten

El hilo también suministra un método sleep(long millis, int nanos), que pone el hilo en reposo por millis milisegundos y nanos milisegundos nanosegundos. Dado que la mayoría de las plataformas basadas en JVM no admiten resoluciones tan pequeñas como un nanosegundo, el código de gestión de hilos de JVM redondea el número de nanosegundos al número de milisegundos más próximo. Si una plataforma no admite una resolución tan pequeña como un milisegundo, el código de manejo de subprocesos JVM redondea el número de milisegundos al múltiplo más cercano de la resolución más pequeña que admite la plataforma.

¿Está vivo o muerto?

Cuando un programa llama al método start() de Thread, pasa un poco de tiempo (para la inicialización) antes de que el nuevo hilo llame a run(). Después de que run() retorna, pasa un período de tiempo antes de que la JVM limpie el hilo. La JVM considera que el subproceso está vivo inmediatamente antes de la llamada del thread a run(), durante la ejecución del subproceso de run(), e inmediatamente después de que run() retorna. Durante ese intervalo, el método isAlive() de Thread devuelve un valor verdadero booleano. De lo contrario, el método devuelve false.

IsAlive() resulta útil en situaciones en las que un hilo necesita esperar a que otro subproceso termine su método run() antes de que el primer subproceso pueda examinar los resultados del otro hilo. Esencialmente, el subproceso que necesita esperar entra en un bucle while. Mientras que isAlive() devuelve true para el otro hilo, el hilo de espera llama a sleep(long millis) (o sleep(long millis, int nanos)) para dormir periódicamente (y evitar perder muchos ciclos de CPU). Una vez que isAlive() devuelve false, el hilo de espera puede examinar los resultados del otro hilo.

¿Dónde usaría esa técnica? Para empezar, ¿qué tal una versión modificada de CalcPI1,

donde el thread de inicio espera a que el nuevo hilo termine antes de imprimir el valor de pi? El código fuente CalcPI2 del Listado 4 demuestra la técnica:

Listing 4. CalcPI2.java

```
// CalcPI2.java
public class CalcPI2
{
   public static void main(String [] args)
   {
      MyThread mt = new MyThread();
      mt.start();
      while (mt.isAlive())
        try
        {
            Thread.sleep(10); // Sleep for 10 milliseconds
        }
        catch(InterruptedException e)
        {
        }
      System.out.println("pi = " + mt.pi);
   }
}

class MyThread extends Thread
{
   boolean negative = true;
   double pi; // Initializes to 0.0, by default

   public void run()
   {
      for (int i = 3; i < 100000; i += 2)
      {
           if (negative)
               pi -= (1.0 / i);
           else
               pi += (1.0 / i);
           negative = !negative;
      }
      pi += 1.0;
      pi *= 4.0;
      System.out.println("Finished calculating PI");
   }
}
```

El hilo inicial de CalcPI2 duerme en intervalos de 10 milisegundos, hasta que mt.isAlive() devuelve false. Cuando esto sucede, el hilo de salida sale de su bucle while e imprime el contenido de pi. Si ejecuta este programa, verá salida similar (pero probablemente no idéntica) a lo siguiente:

```
Finished calculating PI
pi = 3.1415726535897894
```

Ahora, ¿no parece más exacto?

Está vivo?

Un hilo podría llamar al método isAlive() sobre sí mismo. Sin embargo, eso no tiene sentido porque isAlive() siempre devolverá true.

Unir fuerzas

Debido a que la técnica del método loop / isAlive() / sleep() resulta útil, Sun lo empaquetó en un trío de métodos: join(), join(long millis) y join(long millis, int nanos). El subproceso actual llama a join(), a través de la referencia que tiene del otro hilo cuando quiere esperar a que este otro hilo termine. En cambio, el hilo actual llama a join(long millis) o join(long millis, int nanos) cuando quiere o bien esperar a que el otro hilo termine o esperar hasta que una combinación de millis milésimas de segundo y nanos nanosegundos haya pasado. (Al igual que con los métodos sleep(), el código de manejo de subprocesos JVM redondeará los valores de los argumentos de los métodos join (long millis) y join (long millis, int nanos). El código fuente CalcPI3 del Listado 5 muestra una llamada para join():

Listing 5. CalcPI3.java (implementar este método)

```
// CalcPI3.java
public class CalcPI3
{
   public static void main (String[] args)
   {
      MyThread mt = new MyThread();
      mt.start();
      try
      {
          mt.join();
      }
      catch (InterruptedException e)
      {
      }
      System.out.println("pi = " + mt.pi);
   }
}

class MyThread extends Thread
{
   boolean negative = true;
   double pi; // Initializes to 0.0, by default
   public void run()
   {
      for (int i = 3; i < 100000; i += 2)
```

```
      {
           if (negative)
               pi -= (1.0 / i);
           else
               pi += (1.0 / i);
           negative = !negative;
      }
      pi += 1.0;
      pi *= 4.0;
      System.out.println("Finished calculating PI");
   }
}
```

El subproceso main de CalcPI3 espera a que finalice el subproceso asociado con el objeto MyThread, referido por mt. El hilo main imprime entonces el valor de pi, que es idéntico al valor que CalcPI2 genera.

No intente unir el hilo actual a sí mismo porque el hilo actual esperará para siempre.

Censos

En algunas situaciones, es posible que desee saber qué subprocesos se ejecutan activamente en su programa. Thread proporciona un par de métodos para ayudarle con esa tarea: activeCount () y enumerate(Thread [] thdarray). Pero esos métodos sólo funcionan en el contexto del grupo de subprocesos del subproceso actual. En otras palabras, estos métodos identifican sólo subprocesos activos que pertenecen al mismo grupo de subprocesos que el subproceso actual.

El método static ActiveCount() devuelve un recuento de los subprocesos que se ejecutan activamente en el grupo de subprocesos del subproceso actual. Un programa utiliza el valor de retorno entero de este método para dimensionar una matriz de referencias de hilos. Para recuperar esas referencias, el programa debe llamar al método static enumerate(Thread[] thdarray). El valor de retorno de entero de este método identifica el número total de referencias de Thread que enumeran (Thread[] thdarray) almacena en la matriz. Para ver cómo funcionan estos métodos, consulte el Listado 6:

Listado 6. Census.java

```
// Census.java
class Census
{
   public static void main(String[] args)
   {
      Thread[] threads = new Thread[Thread.activeCount()];
      int n = Thread.enumerate(threads);
      for (int i = 0; i < n; i++)
```

```
            System.out.println(threads[i].toString());
    }
}
```

Cuando se ejecuta, este programa produce una salida similar a la siguiente:

```
Thread[main, 5, main]
```

La salida muestra que se está ejecutando un hilo, el hilo de arranque. El main de la izquierda identifica el nombre de ese hilo. El 5 indica la prioridad del hilo, y el main de la derecha identifica el grupo de hilos del hilo. Es posible que le decepcione que no pueda ver ningún hilo del sistema, como el hilo del recolector de elementos no utilizados, en la salida. Esa limitación resulta del método enumerate de Thread (Thread[] thdarray), que interroga sólo el grupo de subprocesos del subproceso actual para los subprocesos activos. Sin embargo, la clase ThreadGroup contiene varios métodos enumerate() que permiten capturar referencias a todos los subprocesos activos, independientemente del grupo de subprocesos.

ActiveCount() y NullPointerException
No dependa del valor de retorno de activeCount() al iterar sobre una matriz. Si lo hace, su programa corre el riesgo de lanzar objetos NullPointerException. ¿Por qué? Entre las llamadas a activeCount() y enumerate (Thread[] thdarray), uno o más subprocesos pueden terminar. Como resultado, enumerate (Thread[] thdarray) copiará menos referencias de hilo en su matriz. Por lo tanto, considere el valor devuelto de activeCount() como un valor máximo para propósitos de dimensionamiento de matriz. Además, piense en le valor de retorno de enumerate(Thread[] thdarray) como la representación del número de subprocesos activos en el momento de la llamada de un programa a ese método.

Antibugging

Si el programa no funciona correctamente y sospecha que el problema se encuentra en un subproceso, puede obtener más información sobre ese subproceso llamando a los métodos dumpStack() y toString() del Thread. El método dumpStack() estático, que proporciona una envoltura alrededor de `new Excepción ("Traza de pila" ).PrintStackTrace();`, imprime un rastreo de la pila para el subproceso actual. ToString () devuelve un objeto String que describe el nombre del subproceso, la prioridad y el grupo de subprocesos de acuerdo con el siguiente formato: Thread [thread-name, priority, thread-group].

El sistema de castas

No todos los subprocesos se crean iguales. Se dividen en dos categorías: usuario y daemon. Un hilo de usuario realiza un trabajo importante para el usuario del programa, trabajo que debe finalizar antes de que finalice la aplicación. En cambio, un hilo daemon realiza tareas domésticas (como recolección de basura) y otras tareas de fondo que

probablemente no contribuyen al trabajo principal de la aplicación, pero son necesarias para que la aplicación continúe su trabajo principal. A diferencia de los subprocesos de usuario, los subprocesos de demonio no necesitan finalizar antes de que finalice la aplicación. Cuando el subproceso de inicio de una aplicación (que es un subproceso de usuario) finaliza, la JVM comprueba si hay otros subprocesos de usuario en ejecución. Si algunos lo son, la JVM impide la finalización de la aplicación. De lo contrario, la JVM finaliza la aplicación, independientemente de si se están ejecutando subprocesos de demonio.

Cuándo utilizar currentThread()

En varios lugares, este artículo se refiere al concepto de un hilo actual. Si necesita acceso a un objeto Thread que describe el subproceso actual, llame al método estático currentThread() de Thread. Ejemplo: Thread current = Thread.currentThread();.

Cuando un subproceso llama al método start() de un objeto Thread, el subproceso recién iniciado es un subproceso de usuario. Ese es el valor predeterminado. Para establecer un subproceso como un subproceso daemon, el programa debe llamar al método setDaemon(boolean isDaemon) de Thread con un valor de argumento verdadero booleano antes de la llamada a start(). Más adelante, puede comprobar si un subproceso es daemon llamando al método isDaemon() de Thread. Ese método devuelve un valor verdadero booleano si el subproceso es daemon.

Para dejarte jugar con el usuario y los hilos de daemon, escribe UserDaemonThreadDemo:

Listing 7. UserDaemonThreadDemo.java (implementar este ejemplo)

```
// UserDaemonThreadDemo.java
public class UserDaemonThreadDemo
{
   public static void main(String[] args)
   {
      if (args.length == 0)
         new MyThread().start();
      else
      {
         MyThread mt = new MyThread();
         mt.setDaemon(true);
         mt.start();
      }
      try
      {
         Thread.sleep(100);
      }
      catch (InterruptedException e)
      {
      }
   }
}
```

```
class MyThread extends Thread
{
   public void run()
   {
      System.out.println("Daemon is " + isDaemon());
      while (true);
   }
}
```

Después de compilar el código, ejecute UserDaemonThreadDemo a través del comando java de Java2 SDK. Si ejecuta el programa sin argumentos de línea de comandos, como en `java UserDaemonThreadDemo`, por ejemplo, new MyThread() .start(); se ejecuta. Ese fragmento de código inicia un subproceso de usuario que imprime Daemon es falso antes de entrar en un bucle infinito. (Debe presionar Ctrl-C o una combinación de teclas equivalente para finalizar ese bucle infinito.) Debido a que el nuevo subproceso es un subproceso de usuario, la aplicación continúa ejecutándose después de finalizar el subproceso de inicio. Sin embargo, si especifica al menos un argumento de línea de comandos, como en java UserDaemonThreadDemo x, por ejemplo, mt.setDaemon (true); Ejecuta, y el nuevo hilo será un daemon. Como resultado, una vez que el hilo de inicio se despierta de su sueño de 100 milisegundos y termina, el nuevo hilo de demonio también terminará.

Una excepción de setDaemon()

Tenga en cuenta que el método `setDaemon(boolean isDaemon)` lanza una excepción `IllegalThreadStateException` si se realiza una llamada a ese método después de que el subproceso inicia la ejecución.

Runnables

Después de estudiar los ejemplos de la sección anterior, podría pensar que introducir multithreading en una clase siempre requiere que extienda Thread y que su subclase reemplace el método run() de Thread. Sin embargo, eso no siempre es una opción. El cumplimiento de la herencia de implementación de Java prohíbe a una clase extender dos o más superclases. Como resultado, si una clase extiende una clase que no es Thread, esa clase tampoco puede extender Thread. Dada esa restricción, ¿cómo es posible introducir multithreading en una clase que ya extiende alguna otra clase? Afortunadamente, los diseñadores de Java se dieron cuenta de que surgirían situaciones en las que no sería posible subclasificar Thread. Esa realización llevó a la interfaz java.lang.Runnable y constructores de Thread con parámetros Runnable, como Thread (Runnable target).

La interfaz Runnable declara una sola firma de método: void run() ;. Esa firma es idéntica a la firma del método run() de Thread y sirve como entrada de ejecución de un subproceso. Debido a que Runnable es una interfaz, cualquier clase puede implementar esa interfaz adjuntando una cláusula implements al encabezado de la clase y proporcionando un

método run() apropiado. En el momento de la ejecución, el código de programa puede crear un objeto de esa clase y pasar la referencia del runnable a un constructor de Thread apropiado. El constructor almacena esa referencia dentro del objeto Thread y asegura que un nuevo subproceso llame al método run() de runnable después de una llamada al método start() del objeto Thread, que el Listado 8 demuestra:

Listing 8. RunnableDemo.java

```
// RunnableDemo.java

public class RunnableDemo extends java.applet.Applet implements Runnable
{
   private Thread t;

   public void run()
   {
      while (t == Thread.currentThread())
      {
           int width = rnd(30);
           if (width < 2)
               width += 2;

           int height = rnd(10);
           if (height < 2)
               height += 2;

           draw (width, height);
      }
   }

   public void start()
   {
      if (t == null)
      {
         t = new Thread(this);
         t.start();
      }
   }

   public void stop()
   {
      if (t != null)
         t = null;
   }

   private void draw (int width, int height)
   {
      for (int c = 0; c < width; c++)
           System.out.print('*');
```

```
        System.out.print('\n');

        for (int r = 0; r < height - 2; r++)
        {
             System.out.print('*');

             for (int c = 0; c < width - 2; c++)
                  System.out.print(' ');

             System.out.print('*');

             System.out.print('\n');
        }

        for (int c = 0; c < width; c++)
             System.out.print('*');

        System.out.print('\n');
    }

    private int rnd(int limit)
    {
        // Return a random number x in the range 0 <= x < limit.

        return (int)(Math.random() * limit);
    }
}
```

RunnableDemo describe un applet para la salida repetida de contornos de rectángulos basados en asterisco en la salida estándar. Para llevar a cabo esta tarea, Runnable debe extender la clase java.applet.Applet (java.applet identifica el paquete en el que se encuentra Applet e implementa la interfaz Runnable.

Un applet proporciona un método void start() público, que es llamado (normalmente por un navegador Web) cuando un applet debe comenzar a ejecutarse, y proporciona un método void stop() público, que se llama cuando un applet debe dejar de ejecutarse.

El método start() es el lugar perfecto para crear e iniciar un subproceso y RunnableDemo logra esta tarea ejecutando t = new Thread (this); T.start () ;. Paso esto al constructor de Thread porque el applet es un runnable debido a RunnableDemo implementando Runnable.

El método stop() del applet es el lugar perfecto para detener un subproceso, asignando null a la variable Thread. No puedo usar el método public void stop() de Thread para esta tarea porque este método ha sido descontinuado - no es seguro usarlo.

El método run () contiene un bucle infinito que se ejecuta por tanto tiempo como Thread.currentThread () devuelve la misma referencia de hilo que se encuentra en la

variable de hilo. La referencia en esta variable se anula cuando se llama al método stop() del applet.

Debido a que la nueva salida de RunnableDemo resultaría demasiado larga para incluirse en este artículo, sugiero que compile y ejecute ese programa usted mismo.

Tendrá que utilizar la herramienta appletviewer y un archivo HTML para ejecutar el applet. El Listado 9 presenta un archivo HTML adecuado: el ancho y la altura se establecen en 0 porque no se genera ninguna salida gráfica.

Listado 9. RunnableDemo.html

```
<Applet code = "RunnableDemo" width = "0" height = "0"> </ applet>
```

Especifique appletviewer RunnableDemo.html para ejecutar este applet.

¿Thread contra Runnable?
Cuando enfrentas una situación en la que una clase puede extender Thread o implementar Runnable, ¿qué enfoque escoges? Si la clase ya extiende otra clase, debe implementar Runnable. Sin embargo, si esa clase no extiende ninguna otra clase, piense en el nombre de la clase. Ese nombre sugiere que los objetos de la clase son activos o pasivos. Por ejemplo, el nombre Ticker sugiere que sus objetos son activos. Por lo tanto, la clase Ticker extendería Thread, y los objetos Ticker serían objetos Thread especializados.

Revisión

Los usuarios esperan que los programas logren un rendimiento sólido. Una forma de lograr esa tarea es utilizar subprocesos. Un hilo es una ruta de ejecución independiente a través del código de programa. Los hilos benefician los programas basados en GUI porque permiten que esos programas permanezcan receptivos a los usuarios mientras realizan otras tareas. Además, los programas con hilos terminan típicamente más rápidamente que sus homólogos sin hilos. Esto es especialmente cierto en los hilos que se ejecutan en una máquina multiprocesador, donde cada hilo tiene su propio procesador. Los objetos de las subclases de Thread y Thread describen hilos y se asocian con esas entidades. Para aquellas clases que no pueden extender Thread, debe crear un runnable para aprovechar el multithreading.

Bibliografía

1. Java 101: Descripción de los subprocesos Java, Parte 1: Introducción de subprocesos y ejecutables
http://www.javaworld.com/article/2074217/java-concurrency/java-101--understanding-java-threads--part-1--introducing-threads-and-runnables.html