code URJC

Programación Concurrente

Tema 5

Programación Concurrente en Java

Micael Gallego
micael.gallego@urjc.es
@micael_gallego

Universidad Rey Juan Carlos

code URJC

Programación Concurrente

Tema 5.6

Ejecución concurrente y asíncrona de tareas

Micael Gallego
micael.gallego@urjc.es
@micael_gallego

Universidad Rey Juan Carlos

Ejecución de tareas en hilos

- **Ejecución de tareas**
- Ejecución secuencial de tareas
- Ejecución de tareas en un nuevo hilo
- Framework de ejecución de tareas
- Ejecución de tareas con grupos de hilos
- Ejecución asíncrona de tareas
- Conclusiones

Ejecución de tareas

code URJC

- La mayoría de las aplicaciones **concurrentes** se estructuran en base a la ejecución de **tareas**
- Las **tareas** son unidades de **trabajo abstractas e independientes** entre sí
- La división en tareas tiene las siguientes **ventajas**:
  - **Simplifica** la organización del programa
  - Facilita la recuperación de **errores**
  - Promueve la **concurrencia** porque permite paralelizar la ejecución de tareas

Ejecución de tareas

code URJC

- El **ámbito** de cada tarea tiene que seleccionarse de forma adecuada
- Las tareas deberían ser lo más **independientes** entre sí. Una tarea no debería depender del **estado**, el **resultado** o lo que **realice** otra tarea
- La independencia favorece la ejecución en **paralelo** (aprovechamiento de **recursos**)
- Para favorecer la **escalabilidad** y el **balanceo** de carga, las tareas no deberían ser muy **grandes**

Ejecución de tareas

- En las **aplicaciones web** se considera que atender a una **petición** de un usuario representa una **tarea**
- Las tareas se ejecutan **concurrentemente**:
  - Si una tarea **tarda mucho** en procesarse, **no afecta** a los demás usuarios
  - Si existen **varios procesadores**, se podrán **repartir** el **trabajo entre ellos** cuando varios usuarios hagan peticiones concurrentes

Ejecución de tareas en hilos

- Ejecución de tareas
- **Ejecución secuencial de tareas**
- Ejecución de tareas en un nuevo hilo
- Framework de ejecución de tareas
- Ejecución de tareas con grupos de hilos
- Ejecución asíncrona de tareas
- Conclusiones

Ejecución secuencial de tareas

code URJC

- Existen **varias políticas** para ejecutar las tareas de una aplicación
- La forma más sencilla de ejecutar tareas es hacerlo de **forma secuencial**
- Una aplicación que ejecuta tareas de forma secuencial sólo necesita un **único hilo de ejecución (*single thread*)**

Ejecución secuencial de tareas

code URJC

- **Ejecución secuencial**

```
class ServidorWebUnicoHilo {
  public static void main(String[] args) throws IOException {

    ServerSocket socket = new ServerSocket(80);

    while (true) {
      Socket conexion = socket.accept();
      procesarPeticion(conexion);
    }
  }
}
```

Se bloquea a la espera de una nueva petición http

Ejecución secuencial de tareas

- El **problema** con esta solución es que el servidor **no puede aceptar** una nueva **conexión** mientras está **procesando** la petición actual
- Además, si el servidor tiene **varios procesadores** o atender la petición requiere acceso de **entrada/salida** se estarán **desaprovechando** los recursos del sistema

Ejecución de tareas en hilos

- Ejecución de tareas
- Ejecución secuencial de tareas
- **Ejecución de tareas en un nuevo hilo**
- Framework de ejecución de tareas
- Ejecución de tareas con grupos de hilos
- Ejecución asíncrona de tareas
- Conclusiones

Ejecución de tareas en un nuevo hilo

code URJC

- Para solucionar el problema, se puede ejecutar cada tarea en su **propio hilo de ejecución**
- De esa forma se pueden **aceptar nuevas** conexiones aunque se esté **procesando** la petición actual
- Se **aprovechan** mejor los recursos cuando se producen **peticiones concurrentes**

Ejecución de tareas en un nuevo hilo

- **Nuevo hilo por cada tarea**

```
class ServidorWebHiloPorTarea {
  public static void main(String[] args) throws IOException {

    ServerSocket socket = new ServerSocket(80);

    while (true) {
      Socket conexion = socket.accept();
      Runnable tarea = ()-> procesarPeticion(conexion);
      new Thread(tarea).start();
    }
  }
}
```

Se crea un hilo que ejecuta la tarea para atender la petición

Ejecución de tareas en un nuevo hilo

- Hay que tener en cuenta que ahora se pueden estar procesando **varias peticiones de forma concurrente**
- Si se **comparte** algún objeto al procesar la petición, ese objeto debe ser ***thread-safe*** o estar apropiadamente **sincronizado**

Ejecución de tareas en un nuevo hilo

- **Problemas con la creación ilimitada de hilos**
  - Si existen **pocas** peticiones concurrentes, la creación de un hilo por petición puede ser **aceptable**
  - Pero para un sistema en **producción** con **mucha** carga presenta varios **problemas**:
    - La **creación** de un hilo es **costosa**
    - Cada **hilo activo** consume **memoria** (para la pila de ejecución) y **tiempo de CPU** en cambios de contexto
    - Está limitado el **número máximo** de hilos (depende del sistema operativo, JVM, configuraciones, etc…)

Ejecución de tareas en un nuevo hilo

- **Problemas con la creación ilimitada de hilos**
  - La aplicación puede **funcionar** hasta cierto **límite** y alcanzado ese límite puede **fallar repentinamente** por problemas de **memoria**
  - Es mejor que se **degrade el rendimiento** de forma **gradual** y en un punto determinado que **no se pueda atender** a nuevos clientes

Ejecución de tareas en hilos

- Ejecución de tareas
- Ejecución secuencial de tareas
- Ejecución de tareas en un nuevo hilo
- **Framework de ejecución de tareas**
- Ejecución de tareas con grupos de hilos
- Ejecución asíncrona de tareas
- Conclusiones

Framework de ejecución de tareas

code URJC

- Las **tareas** son **unidades lógicas** de trabajo, y los **hilos** permiten **ejecutar las tareas** de forma asíncrona
- Ni la **ejecución secuencial** de tareas ni la **creación de un hilo por tarea** son soluciones aceptables
- La **solución** es **limitar el número máximo** de hilos posibles y **reutilizar** los hilos en vez de crear uno nuevo por cada tarea

Framework de ejecución de tareas

- La biblioteca estándar de Java dispone de un **interfaz** para **ejecutar tareas** llamado **Executor**

```
package java.util.concurrent;

public interface Executor {
  void execute(Runnable task);
}
```

- Existen varias clases que implementan el interfaz con **diferentes políticas de ejecución**

Framework de ejecución de tareas

code URJC

- Si se quisiera, se podrían implementar **ejecutores** que se comportaran como los **ejemplos anteriores**

```
public class HiloPorTareaExecutor implements Executor {
  public void execute(Runnable tarea) {
    new Thread(tarea).start();
  };
}
```

```
public class TareasSecuenciaExecutor implements Executor {
  public void execute(Runnable tarea) {
    tarea.run();
  };
}
```

Framework de ejecución de tareas

code URJC

```
class ServidorWebHiloPorTarea {

  private static Executor executor =
    new HiloPorTareaExecutor();

  public static void main(String[] args) throws IOException {

    ServerSocket socket = new ServerSocket(80);

    while (true) {
      Socket conexion = socket.accept();
      Runnable tarea = ()-> procesarPeticion(conexion);
      executor.execute(tarea);
    }
  }
}
```

Usamos el ejecutor para que la política de ejecución concreta la decida el ejecutor

Framework de ejecución de tareas

code URJC

- Las **tareas** son **unidades lógicas** de trabajo, y los **hilos** permiten **ejecutar las tareas** de forma asíncrona
- Ni la **ejecución secuencial** de tareas ni la **creación de un hilo por tarea** son soluciones aceptables
- La **solución** es **limitar el número máximo** de hilos posibles y **reutilizar** los hilos en vez de crear uno nuevo por cada tarea

Framework de ejecución de tareas

code URJC

- La ventaja de usar un **ejecutor** es que se puede elegir de forma sencilla la **política de ejecución**
  - Cuántas tareas concurrentemente (número máximo de hilos)
  - Reutilización de hilos para varias tareas (evita crear un hilo por tarea)
  - En qué orden se deben ejecutar (FIFO, prioridad)
  - Cuántas tareas pendientes de ejecución
  - …

Framework de ejecución de tareas

code URJC

- **No existe una política** de ejecución **adecuada** para todos los casos
- Depende de:
  - Los **recursos** disponibles (número de cores, memoria)
  - Si las tareas hacen mucho uso de **entrada/salida** bloqueante
  - Si es preferible **reducir el rendimiento** para un cliente o bien **rechazar** a nuevos clientes

Ejecución de tareas en hilos

- Ejecución de tareas
- Ejecución secuencial de tareas
- Ejecución de tareas en un nuevo hilo
- Framework de ejecución de tareas
- **Ejecución de tareas con grupos de hilos**
- Ejecución asíncrona de tareas
- Conclusiones

Ejecución de tareas con grupos de hilos

code URJC

- **Grupos de hilos (Thread Pools)**
  - **Executors** implementados con un conjunto de hilos que ejecutan las tareas
  - Las tareas están en una **cola** y los hilos las **ejecutan** según van llegando.
  - Cuando un hilo termina de ejecutar una tarea, selecciona la **siguiente tarea** de la cola. Si no hay, se bloquea a la **espera de una nueva**
  - Siguen un esquema **productor / consumidor**

Ejecución de tareas con grupos de hilos

code URJC

- **Grupos de hilos (Thread Pools)**
  - Reutilizar un hilo **reduce el tiempo de creación** y los recursos de CPU asociados a costa de algo de memoria
  - Se inicia la ejecución de las tareas más rápido (***responsiveness***)
  - Es una forma de **limitar el número máximo** de hilos ejecutándose a la vez y reduce el riesgo de sobrecarga

Ejecución de tareas con grupos de hilos

- **Grupos de hilos (Thread Pools)**

```
Executor executor = Executors.newFixedThreadPool(10);

Runnable tarea = ...;

executor.execute(tarea);
```

Conjunto fijo de hilos. Si no hay hilo disponible, la tarea se encola y se espera hasta que haya un hilo disponible

Ejecución de tareas con grupos de hilos

code URJC

- **Grupos de hilos (Thread Pools)**
  (métodos estáticos de la clase **Executors**)
  - **newFixedThreadPool:** Inicialmente crea un conjunto de hilos fijo para ejecutar las tareas. Si no hay hilo disponible, la tarea se encola.
  - **newCachedThreadPool:** Cada tarea se ejecuta en un hilo. Si no existe ninguno libre, se crea uno. Un hilo permanece libre durante cierto tiempo, si no se usa, se elimina.
  - **newSingleThreadExecutor:** Un único hilo de ejecución para las tareas (las tareas se ejecutarán de forma secuencial)
  - **newScheduledThreadPool:** Ejecuta tareas a intervalos regulares de tiempo (p.e.: Ejecutar esta tarea cada 2 segundos).

Ejecución de tareas con grupos de hilos

code URJC

- **Ciclo de vida de los executors**
  - Los executors deben finalizarse de forma adecuada (para que la aplicación pueda finalizar su ejecución)
  - Como las tareas se ejecutan de forma asíncrona, en un momento dado las tareas pueden estar:
    - En ejecución
    - A la espera
    - Ejecutadas

Ejecución de tareas con grupos de hilos

- **Ciclo de vida de los executors**
  - Se puede finalizar un executor con más o menos "delicadeza".
  - Hay diferentes opciones entre los extremos:
    - **Shutdown:** Ejecutar las tareas enviadas pero no aceptar nuevas tareas.
    - **Shutdown now:** Finalizar (interrumpir) todas las tareas en ejecución y descartar las tareas previamente enviadas.

Ejecución de tareas con grupos de hilos

code URJC

- **Ciclo de vida de los executors**
  - Se controla mediante el interfaz ExecutorService

```
public interface ExecutorService extends Executor {

  void shutdown();
  List<Runnable> shutdownNow();

  boolean isShutdown();

  boolean isTerminated();
  boolean awaitTermination(long timeout, TimeUnit
unit)
     throws InterruptedException;

  // Otros métodos...
}
```

Terminación ordenada

Terminación forzada

Ejecución de tareas con grupos de hilos

code URJC

- **Ciclo de vida de los executors**
  - Se controla mediante el interfaz ExecutorService

```
public interface ExecutorService extends Executor {

  void shutdown();
  List<Runnable> shutdownNow();

  boolean isShutdown();

  boolean isTerminated();
  boolean awaitTermination(long timeout, TimeUnit
unit)
     throws InterruptedException;

  // Otros métodos...
}
```

En proceso de terminación?

Ejecución de tareas con grupos de hilos

- **Ciclo de vida de los executors**
  - Se controla mediante el interfaz ExecutorService

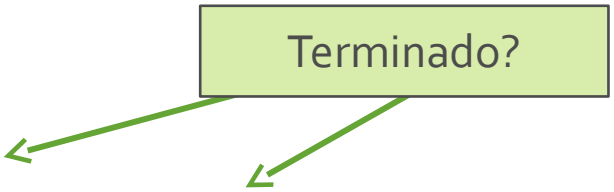
```
public interface ExecutorService extends Executor {

  void shutdown();
  List<Runnable> shutdownNow();

  boolean isShutdown();

  boolean isTerminated();
  boolean awaitTermination(long timeout, TimeUnit
unit)
     throws InterruptedException;

  // Otros métodos...
}
```



Ejecución de tareas en hilos

- Ejecución de tareas
- Ejecución secuencial de tareas
- Ejecución de tareas en un nuevo hilo
- Framework de ejecución de tareas
- Ejecución de tareas con grupos de hilos
- **Ejecución asíncrona de tareas**
- Conclusiones

Ejecución asíncrona de tareas

code URJC

- Hasta ahora hemos visto que un código **envía una tarea** para su ejecución y se **olvida de ella**
- Hay veces que se quiere tener un **mayor control** de la tarea que se ha **enviado** al **executor**:
  - Esperar a que termine (o hasta un timeout)
  - Obtener un valor obtenido por la tarea
  - Gestionar una excepción producida en la tarea

Ejecución asíncrona de tareas

- Si se quiere tener un mayor **control** sobre las tareas deben implementan **Callable<V>** (en vez de **Runnable**)

```
public interface Callable<V> {
    V call() throws Exception;
}
```

- Los executors tienen el método

  submit(Callable<V> task)

Ejecución asíncrona de tareas

code URJC

- Además, el método **submit(Callable<V> c)** devuelve un objeto de la clase **Future<V>** para controlar el estado de la tarea

```
Callable<String> tarea = new DescargaTweets();
Future<String> f = executor.submit(tarea);
try {
  String tweets = f.get();
  // Procesar los tweets...

} catch (ExecutionException e) {
  Throwable cause = e.getCause();
  // Tratar la excepción producida en la tarea
}
```

Ejecución asíncrona de tareas

```
public interface Future<V> {

  V get() throws InterruptedException,
    ExecutionException, CancellationException;

  V get(long timeout, TimeUnit unit) throws
    InterruptedException, ExecutionException,
    CancellationException, TimeoutException;

  boolean cancel(boolean mayInterruptIfRunning);

  boolean isCancelled();
  boolean isDone();
}
```

Acceder al valor obtenido por la tarea

Ejecución asíncrona de tareas

```
public interface Future<V> {

  V get() throws InterruptedException,
    ExecutionException, CancellationException;

  V get(long timeout, TimeUnit unit) throws
    InterruptedException, ExecutionException,
    CancellationException, TimeoutException;

  boolean cancel(boolean mayInterruptIfRunning);

  boolean isCancelled();
  boolean isDone();
}
```

Cancelar la tarea (indicando si debe interrumpir si se está ejecutando)

Ejecución asíncrona de tareas

```
public interface Future<V> {

  V get() throws InterruptedException,
    ExecutionException, CancellationException;

  V get(long timeout, TimeUnit unit) throws
    InterruptedException, ExecutionException,
    CancellationException, TimeoutException;

  boolean cancel(boolean mayInterruptIfRunning);

  boolean isCancelled();
  boolean isDone();
}
```

Consulta del estado de la tarea

Ejecución asíncrona de tareas

code URJC

- Hay veces que **controlar a una única tarea** no es suficiente. Es necesario gestionar un **conjunto de tareas a la vez**
- Por ejemplo, si queremos **descargar** contenido de twitter, facebook, tuenti, instagram **a la vez** y mostrar el contenido según va llegando tenemos que **gestionar todas las descargas** de forma conjunta

Ejecución asíncrona de tareas

code URJC

- Para gestionar varias tareas a la vez se usa un **ExecutorCompletionService**.
  - Se pueden **enviar tareas** (como cualquier **executor**)
  - Dispone del método **take()** para obtener el **Future<V>** de las tareas según van completando
  - Tiene internamente una **cola (queue)** con los resultados de las tareas creando un esquema **productor-consumidor**

Ejecución asíncrona de tareas

code URJC

Creación de un **ExecutorCompletionService** usando un executor creado previamente.

```
ExecutorService executor =
  Executors.newFixedThreadPool(10);

CompletionService<String> completionService =
  new ExecutorCompletionService<>(executor);

completionService.submit(new ConsultaTwitterTask());
completionService.submit(new ConsultaFacebookTask());
completionService.submit(new ConsultaInstagramTask());

for (int i = 0, i = 3; i++) {
  try {
    Future<String> f = completionService.take();
    String info = f.get();
    procesarInfo(info);

  } catch (ExecutionException e) {
    throw e.getCause();
  } catch (Exception e){...}
}
```

Ejecución asíncrona de tareas

```
ExecutorService executor =
  Executors.newFixedThreadPool(10);

CompletionService<String> completionService =
  new ExecutorCompletionService<>(executor);

completionService.submit(new ConsultaTwitterTask());
completionService.submit(new ConsultaFacebookTask());
completionService.submit(new ConsultaInstagramTask());

for (int i = 0, i = 3; i++) {
  try {
    Future<String> f = completionService.take();
    String info = f.get();
    procesarInfo(info);

  } catch (ExecutionException e) {
    throw e.getCause();
  } catch (Exception e){...}
}
```

Creación de las tareas y envío al executor

Ejecución asíncrona de tareas

```
ExecutorService executor =
  Executors.newFixedThreadPool(10);

CompletionService<String> completionService =
  new ExecutorCompletionService<>(executor);

completionService.submit(new ConsultaTwitterTask());
completionService.submit(new ConsultaFacebookTask());
completionService.submit(new ConsultaInstagramTask());

for (int i = 0, i = 3; i++) {
  try {
    Future<String> f = completionService.take();
    String info = f.get();
    procesarInfo(info);

  } catch (ExecutionException e) {
    throw e.getCause();
  } catch (Exception e){...}
}
```

Obtención del Future<V> que permite acceder al valor de la tarea que acaba de terminar

Ejecución asíncrona de tareas

```
ExecutorService executor =
  Executors.newFixedThreadPool(10);

CompletionService<String> completionService =
  new ExecutorCompletionService<>(executor);

completionService.submit(new ConsultaTwitterTask());
completionService.submit(new ConsultaFacebookTask());
completionService.submit(new ConsultaInstagramTask());

for (int i = 0, i = 3; i++) {
  try {
    Future<String> f = completionService.take();
    String info = f.get();
    procesarInfo(info);

  } catch (ExecutionException e) {
    throw e.getCause();
  } catch (Exception e){...}
}
```

Uso del valor

Ejecución de tareas en hilos

- Ejecución de tareas
- Ejecución secuencial de tareas
- Ejecución de tareas en un nuevo hilo
- Framework de ejecución de tareas
- Ejecución de tareas con grupos de hilos
- Ejecución asíncrona de tareas
- **Conclusiones**

Conclusiones

- En general **no es buena idea gestionar los hilos** directamente, es mejor utilizar un **executor**.
- En vez de hilos que ejecutan infinitamente, hay que diseñar el **programa en base a tareas:**
  - Acotadas en el tiempo
  - Independientes entre sí

Conclusiones

- Habitualmente el **pool de hilos (ThreadPool)** es una buen executor porque:
  - **Reduce el tiempo de ejecución** (se evita la creación de nuevos hilos)
  - Se controla los **recursos usados** (máximo número de hilos)
- Aunque es conveniente **evaluar todas las alternativas** por su hubiese un executor más adecuado a las necesidades

Conclusiones

- Las tareas se pueden representar como:
  - **Runnable:** Básicos. Sin valor devuelto ni excepciones.
  - **Callable<V>:** Completo. Valor devuelto y excepciones.
- Para enviar una tarea a un executor se usa el método **execute(Runnable r)** o **submit(Callable<V> c)**

Conclusiones

- Para gestionar las tareas una vez enviadas al executor se usa el **Future<V>**:
  - **Control de una única tarea:** al enviar la tarea al executor devuelve el Future asociado a la tarea.
  - **Control de varias tareas:**
    - Se usa un **ExecutorCompletionService** para enviar las tareas a un executor
    - Se usa el método **submit(…)** para enviar
    - El método **take()** devuelve el Future asociado a la tarea que acaba de finalizar (y así con todas las que van terminando)

Ejercicio 1

- Se ejecutarán N tareas de forma concurrente
- Cada tarea tendrá el siguiente comportamiento
  - Se esperará un tiempo aleatorio entre 0 y 500ms
  - Aleatoriamente se decidirá si termina correctamente o con fallo:
    - Correctamente: Devolverá el String "Tarea X correcta"
    - Con fallo: Elevará una excepción RuntimeException con mensaje "Tarea X con error"
- El hilo principal creará las tareas y las ejecutará
- Cuando alguna de las tareas finalice correctamente, el hilo principal mostrará su valor por pantalla
- En cuanto alguna de las tareas finalice con error, el hilo principal elevará la excepción generada en la tarea (lo que hará que el programa termine)

Ejercicio 2

- Refactoriza el ejercicio de búsqueda de ficheros duplicados para que use un ExecutorService en vez de crear los hilos manualmente.