PROGRAMACION CONCURRENTE Y DISTRIBUIDA

IV.1 Executor framework: Threads pool

J.M.Drake
L.Barros

Notas:

Tareas y Threads

- Existen dos modelo básicos de programas concurrentes:
  - Un programa resulta de la actividad de objetos activos que interaccionan entre si directamente (peer to peer) o a través de recursos y servicios pasivos.
  - Un programa resulta de la ejecución concurrente de tareas. Cada tarea es una unidad de trabajo abstracta y discreta que idealmente puede realizarse con independencia de las otras tareas.
- En las aplicaciones con estrategia cliente/servidor, las tareas resultan:
  - Para implementar los clientes
  - En los servidores para atender los requerimientos de los clientes.
- Organizar las actividades de los servicios como tareas concurrentes:
  - Incrementa la capacidad de prestar servicios (*throughput*).
  - Mejora los tiempos de respuesta (*responsiveness*) con carga de trabajo normal.
  - Presenta una degradación gradual de prestaciones cuando la carga se incrementa.

Notas:

Modelo cliente/servidor

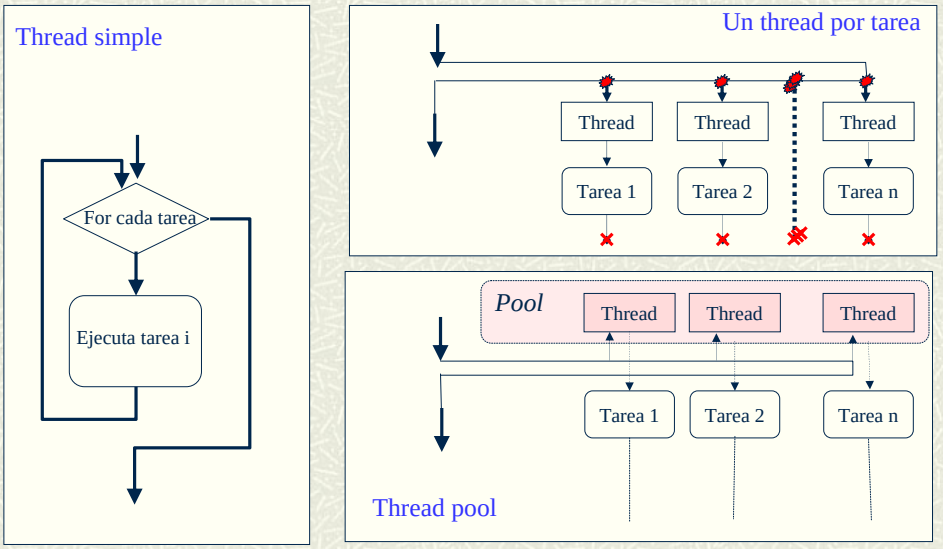
Notas:

Estrategias de ejecución de las tareas

- Hay diferentes estrategias para ejecutar las tareas que constituyen un programa:
  - Ejecución secuencial de las tareas.
    - Ofrecen bajo throughput y tiempo de respuesta.
  - Creación explícita de un thread por cada tarea.
    - La ejecución concurrente de las tareas mejora el uso de los recursos.
    - Cuando las tareas son muy breves, la creación/destrucción de thread supone una sobre carga relevante.
    - Con baja carga tiene buena respuesta. Con alta carga tiene peligro de fallo por agotamiento de los recursos.
  - Creación de un grupo de thread (pool) para posteriormente aplicarlos
    - Se elimina la sobrecarga de creación y destrucción de thread.
    - Se limita la cantidad de recursos que se acaparan.
    - Se consigue la estabilidad con alta carga.

Notas:

Tres políticas de ejecución de tareas



Notas:

Executor framework

- Executor es la base de un framework flexible y potente de ejecución asíncrona de tareas que independiza la definición de las tareas como unidades lógicas de trabajo y los threads como los mecanismos con los que las tareas se ejecutan concurrentemente y asíncronamente.
- Tiene como objeto diferenciar la interfaz java.util.concurrent.Executor:

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

- Flexibiliza el mantenimiento de programas concurrentes a lo largo de su ciclo de vida, posibilitando el cambio de las políticas de ejecución sin cambiar el código de la propia ejecución.

Notas:

Ejemplo de aplicación del framework Executor (1)

```
public class Tarea implements Runnable{
    // Atributos
    String nombre;
    // Constructor
    public Tarea(String elNombre){
            nombre=elNombre;
    }
    // Tarea realizar
    public void run(){
       buscaEImprime(nombre);
      }
   }
}
```


Notas:

Ejemplo de aplicación del framework Executor (2)

```
import java.util.*;
import java.util.concurrent.*;
public class Gestor {
   public static void main(String[] args) {
        Vector<String> listaNombres=new Vector<String>();
        listaNombres.add("-");
        //Executor exec=new ThreadPerTaskExecutor();
        //Executor exec=new OneThreadExecutor();
         final Executor exec= Executors.newFixedThreadPool(10);

        try{Sopa.cargaSopa(args[0]);}catch(IOException e){};

        while (listaNombres.size()!=0){
            try{listaNombres=Sopa.leeNombres();} catch (IOException e){}
            for (int i=0; i<listaNombres; i++){
                  exec.execute (new Tarea(listaNombres.elementAt(i))); }
        }
    }
}
```


Notas:

Implementación de ejecutores

```
import java.util.concurrent.*;
public class ThreadPerTaskExecutor implements Executor{
    public void execute(Runnable r){
        new Thread(r).start();
    }
}
```

```
import java.util.concurrent.*;
public class OneThreadExecutor implements Executor{
    public void execute(Runnable r){
        r.run();
    }
}
```


Notas:

Políticas de ejecución

- La utilidad de usar el framework Executor es poder modificar la política de ejecución de un conjunto de tareas con sólo el cambio mínimo de declarar un tipo de ejecutor diferente.
- Una política de ejecución incluye:
  - Definir en que thread se ejecutan las tareas.
  - Definir el orden en el que se ejecutan las tareas (FIFO,LIFO, Por Prioridad, etc.
  - El número de tareas que se permiten ejecutar concurrentemente.
  - El número de tareas que pueden encolarse en espera de ejecución.
  - Selección de la tarea que debe ser rechazada si hay sobrecarga.
  - Definir las acciones que deben ejecutarse antes y después de ejecutarse una tarea.
- La política de ejecución de tareas en una herramienta de gestión de recursos, y la óptima en cada caso depende de los recursos disponibles y de la calidad de servicio (*throughput*)que se requiere para la ejecución.

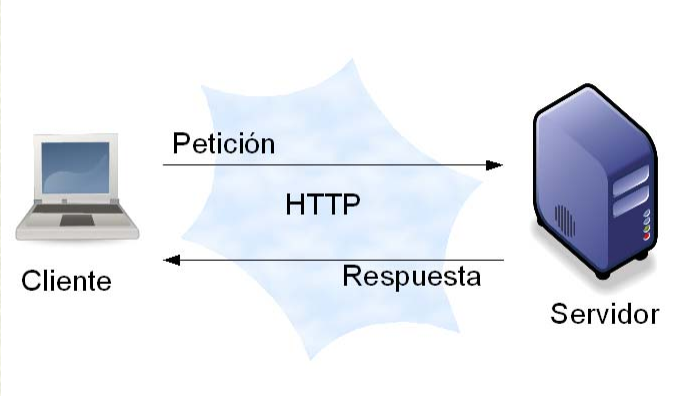
Notas:

Thread pools

- Un thread pool es una estructura que gestiona un conjunto **homogéneo** de threads dispuestos a ejecutar la tarea que se les encargue a través de una cola de trabajos
- El ciclo de vida de un thread de un thread pool es muy simple:
  - Espera en la cola de trabajo un nuevo trabajo.
  - Los ejecuta
  - Retorna a la cola a esperar el siguiente
- Ejecutar tareas a través de un pool de threads tiene ventajas:
  - Reutilizar Thread ya creados implica ahorrase la actividades de creación y destrucción de nuevas tareas
  - Disminuye el tiempo de respuesta, ya que las tareas están creadas cuando llega el requerimiento de ejecución del trabajo.
  - Regulando la dimensión del pool se puede equilibrar la carga con los recursos de CPU y memoria que se requieren.

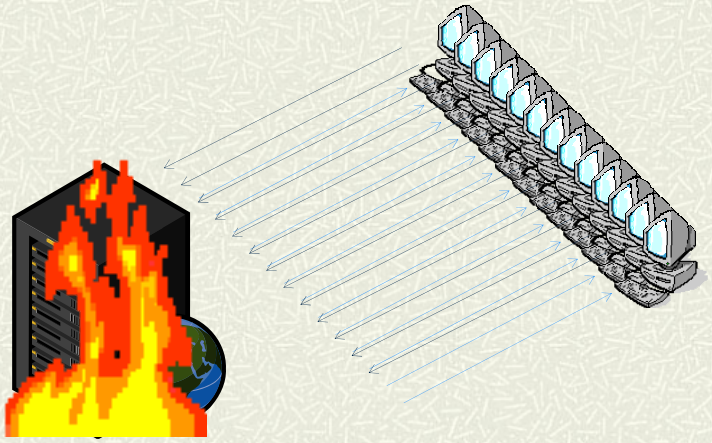
Notas:

Servidor Web I



Notas:

Servidor Web II: creación de un thread por cada petición



Notas:

Servidor Web III: Solución

- Si la solicitud simplemente crea un nuevo hilo para cada nueva petición HTTP y el sistema recibe más peticiones de las que puede manejar de inmediato, la aplicación de pronto dejará de responder a todas las solicitudes.
- Con un límite en el número de los hilos que se pueden crear, la aplicación no servirá a las solicitudes HTTP tan pronto como entren, pero el servicio será lo más rápido que el sistema puede soportar.
- Una ventaja importante del pool de threads fijo es que las aplicaciones que lo utilizan se degradan gradualmente.

Notas:

Clase ThreadPoolExecutor

- Las instancias Java de thread pool se declaran con la clase java.util.concurrent. ThreadPoolExecutor que implementa ExecutorService (extends Executor):

```
class ThreadPoolExecutor implements ExecutorService(
    int corePoolSize,   // Número base de thread que se crean en el pool
    int maximumPoolSize, //  Número máximo de threads que se pueden crear en el pool
    long keepAliveTime, TimeUnit unit, // Tiempo en el que los thread que hay en
                        // exceso respecto de corePoolSize se destruyen
    BlockingQueue<Runnable> workQueue,  //Tipo de cola de tareas del pool
    ThreadFactory threadFactory,  //Define la política con las que se crean los thread
                        // dinámicamente (Prioridad, Grupo, naturaleza)
    RejectedExecutionHandler handler) // Define la política con la que se descartan las
                        // tareas cuya ejecución es rechazada, Bien por el proceso de shutdown
                        // o por que la cola de tareas es acotada.
)
```


Notas:

Ejecutores definidos en las librerías Java

- En *java.util.concurrent.executors* hay cuatro factories que generan intancias concretas que implementan Executor y ofrecen políticas de ejecución diferentes:
  - newFixedRhreadPool(int nThreads) => Genera un pool de threads con un número de elementos constantes. Si alguno de ellos muere es sustituido por otro.
  - newCachedThreadPool() => Genera un pool un número base de threads. Este número se incrementa si la carga de trabajo del thread crece.
  - newSingleThreadExecutor() => crea un único thread que ejecuta secuencialmente las tareas que se le encominedan. Si el thread muere es sustituido por otro.
  - newScheduledThreadPool(int nThread) =>Crea un pool de threads con un número de elemento prejijado. Los threads ofrecen la interfaz *ScheduledExecutorService* con la que se puede planificar la ejecución periodica de los threads.

Notas:

Ejemplo

- Ejemplo:
  - Sistema inactivo
  - llegan 10 tareas simultáneamente
  - pasan 10 segundos
  - llegan 5 tareas simultáneamente
  - pasan 5 minutos
  - Llegan 20 tareas simultáneamente.
  - Cada tarea toma entre 5 y 15 segundos en procesarse.

Notas:

Ejemplo:newFixedThreadPool(int numThreads)

- Ejemplo:
  - Sistema inactivo
  - llegan 10 tareas simultáneamente
  - pasan 10 segundos
  - llegan 5 tareas simultáneamente
  - pasan 5 minutos
  - Llegan 20 tareas simultáneamente.
  - Cada tarea toma entre 5 y 15 segundos en procesarse.
- las primeras 10 tareas que llegan se ponen en la cola y los threads desocupados toman cada uno una tarea y la procesan.
- cuando llegan las siguientes 5 tareas, se ponen en la cola y los primeros threads que se desocupen irán tomando las tareas de la cola para procesarlas.
- cuando lleguen 20 tareas 5 minutos después, se ponen en la cola y los threads van a tomar las primeras 10 (dado que están todos desocupados).
- cada thread cuando termine de procesar su tarea, tomará otra tarea de la cola.

Notas:

Ejemplo:newCachedThreadPool()

- Ejemplo:
  - Sistema inactivo
  - llegan 10 tareas simultáneamente
  - pasan 10 segundos
  - llegan 5 tareas simultáneamente
  - pasan 5 minutos
  - Llegan 20 tareas simultáneamente.
  - Cada tarea toma entre 5 y 15 segundos en procesarse.

- cada una de las 10 tareas que llegan primero, se ejecutan cada una en un thread nuevo.
- Cuando llegan las siguientes 5 tareas, se buscan primero los threads que ya hayan terminado de procesar su tarea; si hay threads libres, se ponen las tareas a procesarse dentro de dichos threads, y las tareas faltantes se procesan en threads nuevos:
  - Por ejemplo, si ya hay 3 threads libres, se procesan ahi 3 de las tareas nuevas, y se crean 2 threads nuevos para procesar las 2 tareas faltantes
- En este momento se tienen 12 threads funcionando, que al final habrán procesado 15 tareas.
- Los threads que llevan mucho tiempo inactivos son terminados automáticamente por el pool, de manera que el número de threads para procesar tareas va bajando cuando el sistema está inactivo e incluso se puede quedar vacío, como al principio.
- Cuando lleguen nuevas tareas, se crearán los threads necesarios para procesarlas.
  - Es decir, cuando lleguen 20 tareas 5 minutos después, seguramente van a correr en 20 threads nuevos.

Notas:

Ejemplo: newSingleThreadExecutor()

- Ejemplo:
  - Sistema inactivo
  - llegan 10 tareas simultáneamente
  - pasan 10 segundos
  - llegan 5 tareas simultáneamente
  - pasan 5 minutos
  - Llegan 20 tareas simultáneamente.
  - Cada tarea toma entre 5 y 15 segundos en procesarse.

- crea un pool de un solo thread, con una cola en donde se ponen las tareas a procesar.
- el thread toma una tarea de la cola, la procesa y toma la siguiente, en un ciclo.
- llegan 10 tareas, y se van a procesar de manera secuencial, las siguientes 5 se van a encolar y se procesarán cuando se terminen las primeras 10, y las ultimas 20 se procesarán una tras otra.

Notas:

Ejemplo:newScheduledExecutor()

- Ejemplo:
  - Sistema inactivo
  - llegan 10 tareas simultáneamente
  - pasan 10 segundos
  - llegan 5 tareas simultáneamente
  - pasan 5 minutos
  - Llegan 20 tareas simultáneamente.
  - Cada tarea toma entre 5 y 15 segundos en procesarse.
- crea un pool que va a ejecutar tareas programadas cada cierto tiempo, ya sea una sola vez o de manera repetitiva.
- Es parecido a un timer, pero con la diferencia de que puede tener varios threads que irán realizando las tareas programadas conforme se desocupen.

Notas:

Ciclo de vida de los Executors

- La interfaz ExecutorService que extiende Executor ofrece diferentes servicios para controlar el ciclo de vida de los threads del pool:

- Un thread pool sigue un ciclo de vida con tres estados:
  - running: Estado en que se encuentra cuando se crea un thread pool. Admite trabajos y cuando tiene thread disponibles los ejecuta.
  - shutting down: Estado de finalizacíón gradual. No se acepta nuevas tareas, pero aquellas que habían sido ordenadas (aunque no se hayan iniciado su ejecución) se ejecutan.
  - terminated: Finalización abrupta. Trata de finalizar las tareas en ejecución, y ni admite nuevas tareas, ni comienza la ejecución de tareas ordenadas.

Notas:

Interfaz ExecutorService

```
public interface ExecutorService extends Executor{
    void shutDown();   // Initiates an orderly shutdown in which previously submitted
                       // tasks are executed, but no new tasks will be accepted.
    List<Runable> shutdownNow(); // Attempts to stop all actively executing tasks,
                       // halts the processing of waiting tasks, and returns a list of the tasks
                       // that were awaiting execution.
    boolean isShutDown(); // Returns true if this executor has been shut down.
    boolean is Terminated(); // Returns true if all tasks have completed following shut down.
    boolean awaitTermination(long timeout, TimeUnit unit)
                                        throw interruptedException;
      // Blocks until all tasks have completed execution after a shutdown request, or the
      // timeout occurs, or the current thread is interrupted, whichever happens first.
  // .. Hay otros metodos relativos a encargos de tareas
  }
```


Notas:

Ejemplo de finalización de las tareas del pool

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;
public class Gestor {
    public static void main(String[] args) {
        Vector<String> listaNombres=new Vector<String>();
        listaNombres.add("_");

        final ExecutorService exec= Executors.newFixedThreadPool(2);

        try{Sopa.cargaSopa(args[0]);}catch(IOException e){};

        while (listaNombres.size()!=0){
              try{listaNombres=Sopa.leeNombres();}catch (IOException e){}
             for (int i=0; i<listaNombres.size(); i++){
                 exec.execute(new Tarea(listaNombres.elementAt(i)));}
        }
        exec.shutdownNow();
    }
}
```


Notas:

Ejemplo II: Ejemplo de finalización de las tareas del pool

```
public class MyRunnable implements Runnable {
private final long countUntil;

MyRunnable(long countUntil) {
this.countUntil = countUntil;
}

@Override
public void run() {
long sum = 0;
for (long i = 1; i < countUntil; i++) {
sum += i;
}
System.out.println(sum);
}
}
```

```
public class Main {
private static final int NTHREADS = 10;

public static void main(String[] args) {
ExecutorService executor =
      Executors.newFixedThreadPool(NTHREADS);
for (int i = 0; i < 20; i++) {
Runnable worker = new MyRunnable(10L + i);
executor.execute(worker);
}
// This will make the executor accept no new threads
// and finish all existing threads in the queue
executor.shutdown();
// Wait until all threads are finish
while (!executor.isTerminated()) {

}
System.out.println("Finished all threads");
}
}
```


Notas:

Tarea con resultados demorados

- El framework Executor utiliza la representacion de las tareas mediante instancias que ofrecen la interfaz Runnable. Esta interfaz tiene ciertas limitaciones:
  - No puede retornar valores.
  - No puede lanzar excepciones.
- Muchas tareas representan segmentos de computación aplazada que deben ser verificados posteriormente cuando terminen. Para estos casos la abstracción Callable es más adecuada.
- La tarea que las lanza obtiene información en la invocación de forma que, más adelante puede obtener los resultados y el estado de la tarea.
- *Future* es una clase cuyas instancias representan el estado de una tarea a lo largo de su ciclo de vida, y a través de ella una tarea puede retornar los resultados, conocer su estado o ser gestionada.

Notas:

Interfaces Callable y Future

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

public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled()
    boolean isDone();
    V get() throws InterruptedException, ExecutionException, CancellationException;
    V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException,
                        CancellationException, TimeoutException;
}
```


Notas:

Interface ExecutorService (2)

```
public interface ExecutorService extends Executor{
    <T> List <Future<T>> invokeAll(Collection<Callable<T> tasks);    // Executes the given
        tasks, returning a list of Futures holding their status and results when all complete.
    <T> List <Future<T>> invokeAll(Collection<Callable<T> tasks, long timeout, TimeUnit
        unit);    // Executes the given tasks, returning a list of Futures holding their status and
        results when all complete or the timeout expires, whichever happens first.
    <T> T invokeAny(Collection <Callable<T> tasks) ; // Executes the given tasks, returning
        the result of one that has completed successfully (i.e., without throwing an exception), if
        any do.
    <T> T invokeAny(Collection <Callable<T> tasks, long timeout, TimeUnit unit) ;
        // Executes the given tasks, returning the result of one that has completed successfully
        (i.e., without throwing an exception), if any do before the given timeout elapses.
    <T> Future<T> submit(Callable<T> task) ; // Submits a value-returning task for execution
        and returns a Future representing the pending results of the task.
    Future<?> submit(Runnable task) ; // Submits a Runnable task for execution and returns a
        Future representing that task.
    <T> Future<T> submit(Runnable task, T result) ; // Submits a Runnable task for execution
        and returns a Future representing that task that will upon completion return the given
        result.
}
```


Notas:

Ejemplo de tarea aplazada

```
interface ArchiveSearcher {
    String search(String target);
}

 class App {
    ExecutorService executor = ...
    ArchiveSearcher searcher = ...
    void showSearch(final String target) throws InterruptedException {
         Future<String> future = executor.submit(
               new Callable<String>() {
                    public String call() { return searcher.search(target); }
               }
          );
        displayOtherThings(); // do other things while searching
        try { displayText(future.get()); // use future
        } catch (ExecutionException ex) { cleanup(); return; }
   }
}
```


Notas:

Ejemplo de devolución de parámetros

```
import java.util.*;
import java.util.concurrent.*;

public class CallableExample {

  public static class WordLengthCallable
      implements Callable {
    private String word;
    public WordLengthCallable(String word) {
      this.word = word;
    }
    public Integer call() {
      return Integer.valueOf(word.length());
    }
  }
```

```
  public static void main(String args[]) throws Exception {
    ExecutorService pool = Executors.newFixedThreadPool(3);
    Set<Future<Integer>> set = new HashSet<Future<Integer>>();
    for (String word: args) {
      Callable<Integer> callable = new WordLengthCallable(word);
      Future<Integer> future = pool.submit(callable);
      set.add(future);
    }
    int sum = 0;
    for (Future<Integer> future : set) {
      sum += future.get();
    }
    System.out.printf("The sum of lengths is %s%n", sum);
    System.exit(sum);
  }
}
```

Ejecución:
Params: hola hola hola hola
The sum of lengths is 16

Notas: