Corso di Programmazione Conconcorrente

Lightweight Executable Framework

Valter Crescenzi
http://crescenzi.inf.uniroma3.it

Sommario

- Lightweight Executable Framework
  - Introduzione: Task / Worker Thread / Executor Service
  - **java.util.concurrent.*Executor*** ed ***ExecutorService***
  - Task: ***Callable***, ***Runnable***, ***Future***, **FutureTask**
  - Esempio: Una Cache Asincrona
  - ***ScheduledExecutorService***

Limiti dei Meccanismi Nativi per Creare Thread in Java

- Con i meccanismi nativi di java dobbiamo contestualmente
  - creare thread e...
    - eseguendo **`new Thread()`**...
    - .. e quindi invocando il metodo **`Thread.start()`**
  - ...e sottomettere *task* da eseguire
    - sarà quello specificato dal metodo **`run()`** di una classe che implementa l'interfaccia ***`Runnable`***
    - Indipendentente da come sia stata specificata:
      - estendendo la classe **`java.lang.Thread`**
      - oppure, fornendo un **`Runnable`** al suo costruttore
- Vogliamo disaccoppiare i due aspetti, per poter variare uno senza dover cambiare anche l’altro

Disaccoppiamento della Creazione di Thread dalla Sottomissione di Task

- Disaccoppiando...
  - la *sottomissione* di **`Runnable`** (in generale *task*)
  - la *creazione* di thread che li prendano in carico

  ...è possibile variare la politica di *creazione* ed utilizzo dei thread indipendentemente dall'ordine di *sottomissione* dei task da eseguire

- Ad es. si possono applicare politiche tese a:
  - contenere il numero di thread creati
    - pooling dei thread; caching e riuso dei thread
  - attuare una politica di scheduling personalizzata
  - programmare temporalmente l'esecuzione

Thread vs *Worker* Thread

- Idea di base: lo stesso thread può far avanzare molteplici task anche non correlati
  - si parla specificatamente di *Worker Thread*
  - si risparmia il costo di istanziare il thread
    - talvolta può risultare preponderante rispetto ai benefici della parallelizzazione
    - non solo in termini di tempo ma anche di memoria
      - su molte piattaforme lo stack di un thread implica un consumo di memoria non trascurabile
      - ✔ N.B. la diffusione delle applicazioni mobile rendono sempre più frequenti scenari con centinaia di migliaia di connessioni poco utilizzate
- C'è bisogno di un servizio che si occupi di
  - creare e gestire i worker thread
  - smistare i task verso i worker che li devono eseguire

Lightweight Executable Framework

- Al “centro” del package **`java.util.concurrent`**
- L’interfaccia alla base di questo framework è **`java.util.concurrent.`*`Executor`***

Method Summary	
void	**execute**(Runnable command) Executes the given command at some time in the future.

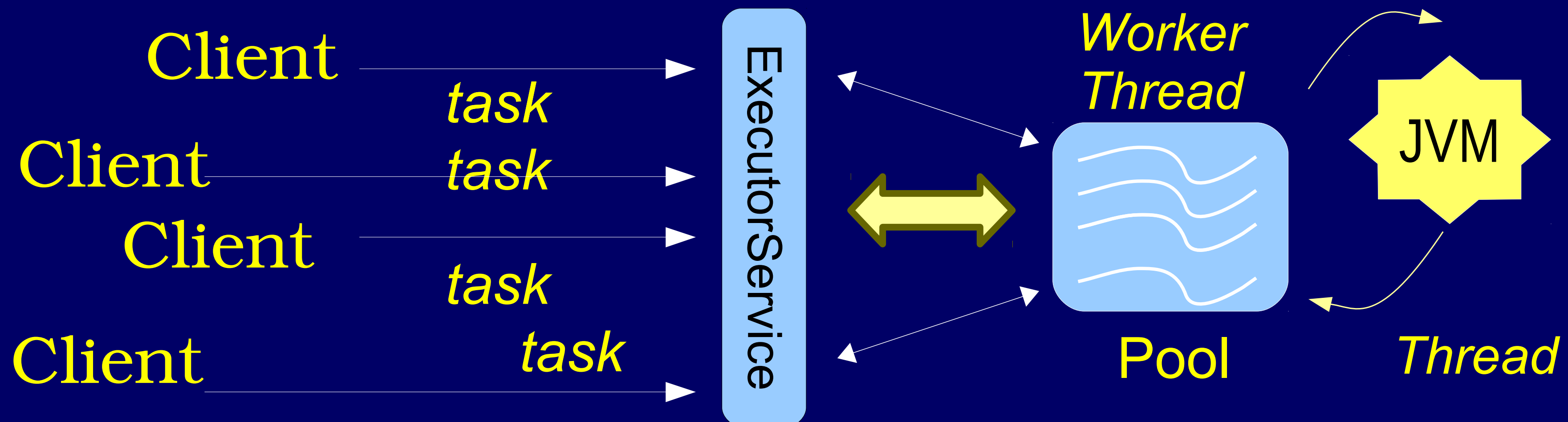
- Possibili implementazioni di maggiore utilizzo fornite dai metodi statici factory della classe di utilità **`java.util.concurrent.Executors`**

Creazione di *Executor*

Method Summary	
static ExecutorService	**newCachedThreadPool**() Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when available.
static ExecutorService	**newFixedThreadPool**(int nThreads) Creates a thread pool that reuses a fixed set of threads operating off a shared unbounded queue.
static ScheduledExecutorService	**newScheduledThreadPool** (int corePoolSize) Creates a thread pool that can schedule cmds to run after a given delay, or to execute periodically.
static ExecutorService	**newSingleThreadExecutor**() Creates an Executor that uses a single worker thread operating off an unbounded queue.
static ScheduledExecutorService	**newSingleThreadScheduledExecutor**() Creates a single-threaded executor that can schedule cmds to run after a given delay, or to execute periodically.

Terminologia

- *Task:* specifica di ciò che un f.d.e. deve fare
- *Thread*: f.d.e. come servizio offerto dalla piattaforma sottostante (la JVM)
- *Worker Thread*: thread che prende in carico, uno dopo l'altro, diversi task – anche non correlati tra loro – sottomessi ad un executor service
- *Executor Service:* servizio che accetta sottomissioni di task e ne smista l'esecuzione ad alcuni worker thread



Esempio di Utilizzo: Thread-Pooling Lato Server

```
class NetworkService {
    private final ServerSocket serverSocket;
    private final ExecutorService pool;

    public NetworkService(int port, int poolSize)
            throws IOException {
      serverSocket = new ServerSocket(port);
      pool = Executors.newFixedThreadPool(poolSize);
    }
    public void serve() {
      try { for (;;)
        pool.execute(new Handler(serverSocket.accept()));
      } catch (IOException ex) { pool.shutdown(); }
    }
  }

  class Handler implements Runnable {
    private final Socket socket;
    Handler(Socket socket) { this.socket = socket; }
    public void run() {…} // read and service request
 }
```

Vantaggi dei Lightweight Executable Framework

- Supponiamo di dover realizzare un **`NetworkService`** a cui possono concorrentemente accedere dalla rete molteplici client che per semplicità ipotizziamo indipendenti tra loro
  - Scelta più semplice: creare un thread dedicato alla gestione di ogni richiesta, un thread nuovo per ogni richiesta che termina al fine del suo processamento
  - Vincolo aggiuntivo: occupazione delle risorse del server limitato superiormente indipendentemente dal numero e dalla frequenza delle richieste
- Nella pratica *ogni* server che voglia contrastare attacchi del tipo DoS *deve* rispettare questo vincolo
- Il pooling dei thread è una *necessità* in molti server
- Altri vantaggi: *decomposizione parallela* e speed-up >>

Alcuni Svantaggi dei Lightweight Executable Framework

- Perdita di identità corrispondenza thread / task
  - i thread possono essere riciclati e comunque si perde l'identificatilità dei task con gli oggetti **`java.lang.Thread`**
- Le dipendenze tra thread divengono pericolose perché la politica di esecuzione potrebbe l'invalidare l'*Assunzione di Progresso Finito*.
- Possibili soluzioni:
  - prevedere un adeguato numero di worker thread
  - usarli solo nel caso che è noto che non possono esserci dipendenze, ad es. sessioni *http*
  - creare politiche di gestione delle code di task che tengano conto delle dipendenze

ExecutorService

java.util.concurrent
Interface ExecutorService
All Superinterfaces:
Executor
All Known Subinterfaces:
ScheduledExecutorService

- Aggiunge metodi per la gestione della ciclo di vita del servizio di esecuzione

Method Summary	
boolean	**awaitTermination**(long timeout, TimeUnit unit) Blocks until all tasks have completed execution …
boolean	**isShutdown**() Returns true if this executor has been shut down.
boolean	**isTerminated**() Returns true if all tasks have completed…
void	**shutdown**() Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
List<Runnable>	**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.

Cancellazione nel Lightweight Executable Framework

- Bisogna distinguere tra cancellazione
  - di un *task*
    - potrebbe o meno già essere stato preso in carico da un worker thread che può o meno essere ricettivo alle “istigazioni al suicidio”
  - di un *worker thread*
    - ad esempio perché la dinamica delle sottomissioni è tale da non richiedere tutti i worker thread esistenti (cfr. con **`Executors.newCachedThreadPool()`**)
  - di un ***`ExecutorService`***
    - gestendo il transitorio di chiusura del servizio, ad esempio decidendo cosa fare per i task
      - già sottomessi ma non ancora eseguiti
      - già sottomessi e già presi in carico da un worker thread
      - sottomessi solo dopo la richiesta di cancellazione del servizio

ExecutorService

java.util.concurrent
Interface ExecutorService
All Superinterfaces:
Executor
All Known Subinterfaces:
ScheduledExecutorService

- Altri metodi per la sottomissione di nuovi task

Method Summary	
<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

java.util.concurrent.*Callable*

- Introdotti per gli ***ExecutorService***
- Simile a **java.lang.*Runnable*** ma consente di restituire un valore

java.util.concurrent

Interface Callable<V>

Type Parameters:

V - the result type of method call

Method Summary	
V	**call**() Computes a result, or throws an exception if unable to do so.

Un Esempio di Utilizzo di *Callable*

```
interface ArchiveSearcher { String search(String t); }
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);
         }
      });
    doOtherThings(); // do other things …
    try {
      displayText(future.get()); // use future
    } catch(ExecutionException ex){ cleanup(); return; }
  }
}
```

`java.util.concurrent.`***`Future`*** (1)

- I task saranno eseguiti dagli executor service, *at some time in the future*
- In seguito alla sottomissione gli ***`Executor`*** non restituiscono i risultati della computazione, ma ***`Future`***
  - Oggetti che incapsulano un risultato potenzialmente ancora da calcolare
  - Gestiscono l'*hand-off* tra il worker thread che esegue il task ed il thread che fruisce del risultato
  - Se si cerca di leggere da un ***`Future`*** un risultato non ancora disponibile, ci si deve sincronizzare sull'effettiva disponibilità
    - ad esempio in attesa passiva

Future (2)

java.util.concurrent
Interface Future<V>

Type Parameters:

V - The result type returned by this Future's get method

All Known Subinterfaces:

ScheduledFuture<V>

All Known Implementing Classes:

FutureTask

Method Summary

boolean	**cancel**(boolean mayInterruptIfRunning) Attempts to cancel execution of this task.
V	**get**() Waits if necessary for the computation to complete, and then retrieves its result.
V	**get**(long timeout, TimeUnit unit) Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
boolean	**isCancelled**() Returns true if this task was cancelled before it completed normally.
boolean	**isDone**() Returns true if this task completed.

`Future` Read-Only

- Concettualmente i future sono molto semplici:
  - offrono le stesse funzionalità del buffer unitario sottostante un classico produttori/consumatori
- I Future pre-Java 8 sono *in sola lettura*!
  - al tempo, appariva una scelta conservativa per mantenere l'API piccola e di immediata comprensione
  - ed è andata esattamente così...
- Tuttavia i tempi sono decisamente cambiati con Java 8
  - introduzione del lambda calcolo in Java>>

- Da Java 8 in poi sono stati affiancati da una versione che permette anche la scrittura del risultato
  - Prima era possibile solo dall'interno dell'***`ExecutorService`*** che conosceva il tipo concreto della implementazione

Future VS *CompletableFuture*

- **java.util.concurrent.CompletableFuture**
  - Aggiungono molto altro, come vedremo
    - computazione asincrone (o “monadiche”) >>
    - altri linguaggi (Scala, JavaScript) li chiamano **Promise**

Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

CompletionStage<T>, Future<T>

```
public class CompletableFuture<T>
extends Object
implements Future<T>, CompletionStage<T>
```

`java.util.concurrent.FutureTask` (1)

java.util.concurrent

Class FutureTask<V>

java.lang.Object

java.util.concurrent.FutureTask<V>

Type Parameters:

V - The result type returned by this FutureTask's get method

All Implemented Interfaces:

Runnable, Future<V>

- Una implementazione concreta di ***`Future`***
- Svolge anche il ruolo di *task* in quanto implementa l'interfaccia **`Runnable`**
- In questo modo è possibile sottomettere il *task* dalla cui esecuzione si vuole ottenere un risultato

`java.util.concurrent.FutureTask` (2)

- Comoda ed utile: ma confonde diversi aspetti
  - Sottomissione di un *task*
  - Raccolta del risultato della sua esecuzione
- Consente:
  - di crearsi esplicitamente oggetti ***`Future`*** (altrimenti la creazione sarebbe interna all'***`ExecutorService`*** utilizzato)
  - di usare i ***`Future`*** con **`Executor.execute()`**

Constructor Summary
FutureTask(**Callable**<V> callable) Creates a FutureTask that will upon running, execute the given Callable.
FutureTask(**Runnable** runnable, V result) Creates a FutureTask that will upon running, execute the given Runnable, and arrange that get will return the given result on successful completion.

Esempio di Utilizzo di **FutureTask**

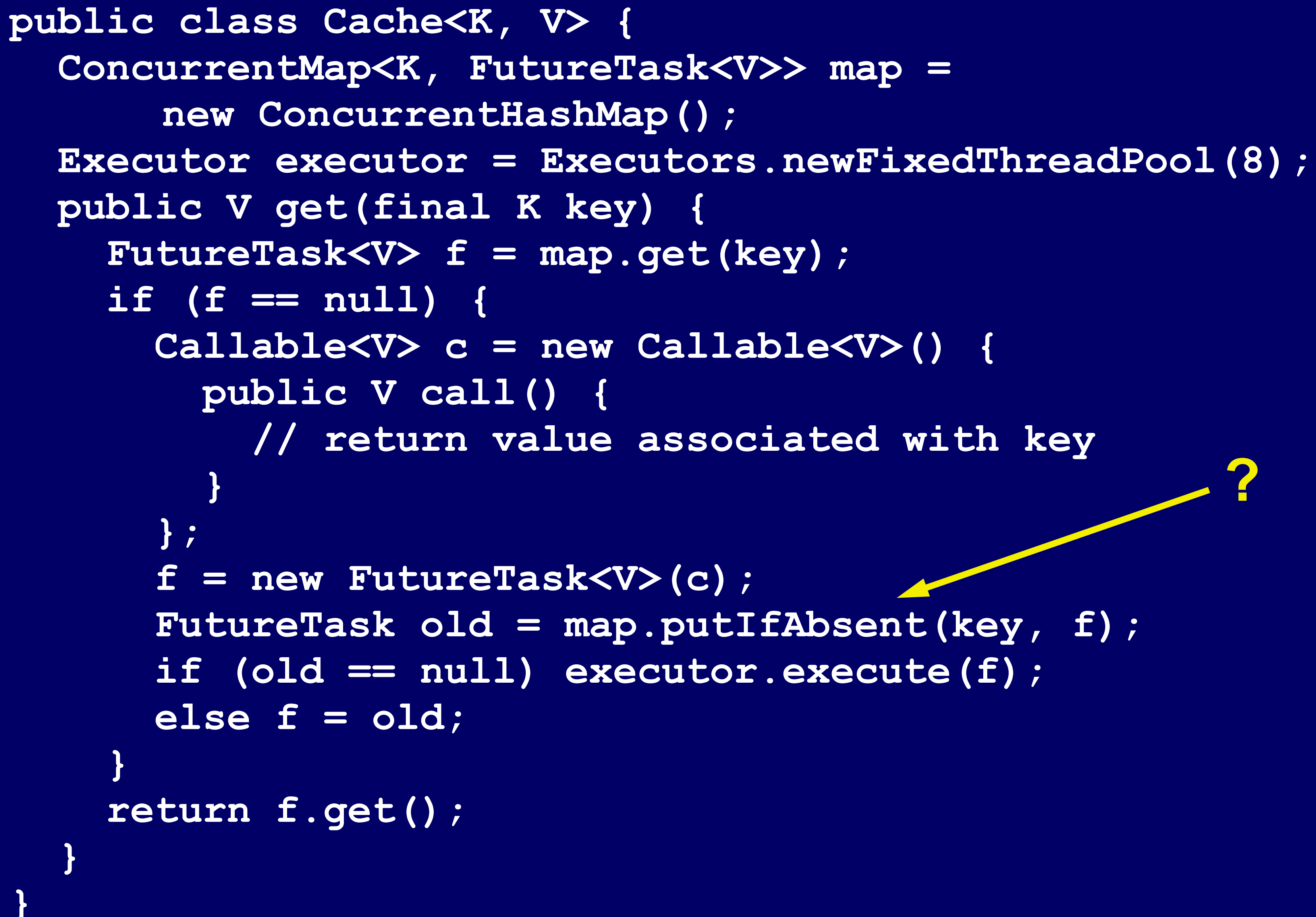
- Esempio **ArchiveSearcher**; con utilizzo di **execute()** al posto di **submit()**:

```
FutureTask<String> future =
    new FutureTask<String>(
       new Callable<String>() {
           public String call() {
             return searcher.search(target);
           }
       }
    );
executor.execute(future);
```

Un Piccolo ma Completo Esempio

- Si vuole calcolare un valore di tipo V univocamente associato ad una chiave di tipo K
  - il calcolo dei valori è computazionalmente costoso
  - ottimizzazioni volute:
    - pool di thread per calcolare i valori
    - i valori già calcolati sono conservati in una cache
    - non si usa mai più di un thread per calcolare il valore associato alla medesima chiave
- Si utilizzano
  - `Future`
  - `ConcurrentHashMap`
  - `Executors`
  - `Executor`

Esempio: Una Cache Asincrona




```
public class Cache<K, V> {
  ConcurrentMap<K, FutureTask<V>> map =
      new ConcurrentHashMap();
  Executor executor = Executors.newFixedThreadPool(8);
  public V get(final K key) {
    FutureTask<V> f = map.get(key);
    if (f == null) {
      Callable<V> c = new Callable<V>() {
        public V call() {
          // return value associated with key
        }
      };
      f = new FutureTask<V>(c);
      FutureTask old = map.putIfAbsent(key, f);
      if (old == null) executor.execute(f);
      else f = old;
    }
    return f.get();
  }
}
```

java.util.concurrent.*ScheduledExecutorService*

java.util.concurrent
Interface ScheduledExecutorService
All Superinterfaces:
Executor, ExecutorService

- Aggiunge metodi per esecuzioni pianificate nel tempo

Method Summary	
<V> ScheduledFuture<V>	**schedule(Callable**<V> callable, long delay, TimeUnit unit) Creates and executes a ScheduledFuture that becomes enabled after the given delay.
ScheduledFuture<?>	**schedule(Runnable** command, long delay, TimeUnit unit) Creates and executes a one-shot action that becomes enabled after the given delay.
ScheduledFuture<?>	**scheduleAtFixedRate(Runnable** command, long initialDelay, long period, TimeUnit unit) Creates and executes a periodic action ….
ScheduledFuture<?>	**scheduleWithFixedDelay(Runnable** command, long initialDelay, long delay, TimeUnit unit) Creates and executes a periodic action …