Programmazione concorrente

Thread

```
/**
* Allocates a new Thread object so that it has target as
* its run object, has the specified name as its name, and
* belongs to the thread group referred to by group, and
* has the specified stack size.
*
*/
public Thread(ThreadGroup group, Runnable target, String name, long stackSize);

/**
Il principale metodo è start(), che avvia un nuovo
percorso di esecuzione (similmente ad una fork) che
lavora all'interno della stessa JVM, condividendo lo
stesso heap e quindi lo stesso stato complessivo.

* Causes this thread to begin execution; the Java Virtual
* Machine calls the run method of this thread.
*/
void start();

/**
* Causes the currently executing thread to sleep
* (temporarily cease execution) for the specified number of
* milliseconds, subject to the precision and accuracy of
* system timers and schedulers
*
* @param millis the length of time to sleep in milliseconds
*
*/
public static void sleep(long millis)
throws InterruptedException

/**
* The Runnable interface should be implemented by any
* class whose instances are intended to be executed
* by a thread.
*/
@FunctionalInterface
public interface Runnable
{
    /**
    * The general contract of the method run is that
    * it may take any action whatsoever.
    */
    void run();
}

/**
* Returns this thread's name.
*/
```

```
public String getName();

/**
* Tests if this thread is alive.
*/
public boolean isAlive();

/**
* Returns a reference to the currently executing thread
* object.
*
*/
public static Thread currentThread();

/**
* Interrupts this thread.
*
*/
public void interrupt();

/**
* Set the handler invoked when this thread abruptly
* terminates due to an uncaught exception.
*/
public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh);
```

Executor

```
/**
* Executes the given command at some time in the future.
* The command may execute in a new thread, in a pooled
* thread, or in the calling thread, at the discretion
* of the Executor implementation.
*
* @param command the runnable task
*
*/
void execute(Runnable command);
```

Tipo	Funzionamento
CachedThreadPool	Riusa thread già creati, ne crea nuovi se necessario
FixedThreadPool	Riusa un insieme di thread di dimensione fissa
ScheduledThreadPool	Esegue i compiti con una temporizzazione
SingleThreadExecutor	Usa un solo thread per tutti

Callables

```
/**
* A task that returns a result and may throw an exception.
*/
@FunctionalInterface
```

```
public interface Callable < V >
{
    /**
    * Computes a result, or throws an exception if unable
    * to do so.
    *
    * @return computed result
    * @throws Exception - if unable to compute a result
    */
    V call() throws Exception;
}

/**
* An Executor that provides methods to manage termination
* and methods that can produce a Future for tracking
* progress of one or more asynchronous tasks.
*
*/
public interface ExecutorService extends Executor;

/**
* Submits a value-returning task for execution and
* returns a Future representing the pending results
* of the task.
*
* @param T - the type of the task's result
* @param task - the task to submit
* @return a Future representing pending completion
* of the task
*
*/
<T> Future<T> submit(Callable<T> task);

/**
* A Future represents the result of an asynchronous
* computation. Methods are provided to check if the
* computation is complete, to wait for its completion,
* and to retrieve the result of the computation.
*
*/
public interface Future< V >;

/**
* Waits if necessary for the computation to complete,
* and then retrieves its result.
*
*/
T get();

/**
* Returns true if this task completed.
*
*/
boolean isDone();
```

ExecutorService

```
/**
* 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 < ? extends Callable< T > > tasks);

/**
* Executes the given tasks, returning a list of Futures
* holding their status and results when all complete.
* Future.isDone() is true for each element of the
* returned list.
*/
< T > List< Future< T > > invokeAll(Collection< ? extends Callable< T > > tasks);

/**
* Initiates an orderly shutdown in which previously
* submitted tasks are executed, but no new tasks will
* be accepted.
*
*/
void shutdown();

/**
* Blocks until all tasks have completed execution after a
* shutdown request, or the timeout occurs, or the current
* thread is interrupted, whichever happens first.
*
*/
boolean awaitTermination(long timeout, TimeUnit unit);

/**
* Returns true if all tasks have completed following
* shut down.
*
*/
boolean isTerminated();

/**
* Attempts to stop all actively executing tasks, halts
* the processing of waiting tasks, and returns a list
* of the tasks that were awaiting execution.
*
*/
List< Runnable > shutdownNow()
```

Dati thread-safe

Una struttura dati non thread-safe non consente a più thread di operare contemporaneamente.

- **nel migliore dei casi** lancia una java.util.ConcurrentModificationException
- **nel caso intermedio** lo stato diventa inconsistente
- **nel peggiore dei casi** ottengo un'altra eccezione

Atomic variables

La funzionalità richiede la disponibilità del supporto dell'hardware attraverso istruzioni CAS (Compare-and-swap).

In mancanza di queste l'implementazione ripiega su metodi più convenzionali (meno efficienti, che bloccano il thread).

Tipo	Array
Integer	AtomicInteger
Long	AtomicLong
Object	AtomicReference
Integer[]	AtomicIntegerArray
Long[]	AtomicLongArray
Object[]	AtomicReferenceArray

Concurrent data structures

ConcurrentMap

```
/**
* A Map providing thread safety and atomicity
* guarantees.
*/
public interface ConcurrentMap< K,V > extends Map< K,V >;

/**
* If the specified key is not already associated with a
* value, associate it with the given value.
* The action is performed atomically.
*/
V putIfAbsent(K key, V Value);

/**
* Replaces the entry for a key only if currently mapped
* to some value.
* The action is performed atomically.
*/
V replace(K key, V Value);

/**
* Replaces the entry for a key only if currently mapped
* to a given value.
* The action is performed atomically.
*/
V replace(K key, V oldValue, V newValue);

/**
* Returns the result of accumulating the given
* transformation of all (key, value) pairs using the
* given reducer to combine values, or null if none.
*
* @param the elements needed to switch to parallel
```

```
* @param the transformation for an element
* @param a commutative associative combining function
*/
public < U > U reduce(long parallelismThreshold, BiFunction< ? super K,? super
V,? extends U> transformer, BiFunction< ? super U,? super U,? extends U>
reducer);

/**
* Returns a non-null result from applying the given
* search function on each (key, value), or null if none.
* Upon success, further element processing is suppressed.
*
* @param the elements needed to switch to parallel
* @param a search function, that returns non-null on
*success
*/
public < U > U search(long parallelismThreshold, BiFunction< ? super K,? super
V,? extends U> searchFunction);

/**
* Performs the given action for each (key, value).
*
* @param the elements needed to switch to parallel
* @param the action (can have side-effects)
*/
public void forEach(long parallelismThreshold, BiConsumer< ? super K,? super V>
action);
```

BlockingQueue

Metodo	Risultato negativo
add(e)	eccezione
offer(e)	false
put(e)	attesa
offer(e, time, unit)	attesa limitata
remove()	eccezione
poll()	null
take()	attesa
poll(time, unit)	attesa limitata
element()	eccezione
peek()	null

```
/**
* Removes all available elements from this
* queue and adds them to the given collection.
* This operation may be more efficient than
* repeatedly polling this queue.
*
* @param c the collection to transfer elements into
* @return the number of elements transferred
*/
int drainTo(Collection< ? super E > c);
```

Varianti:

- **TransferQueue**: interfaccia per una coda in cui i produttori aspettano i consumatori
- **BlockingDeque**: interfaccia che permette di prendere un elemento dalla coda o dalla testa
- **ArrayBlockingQueue**: implementazione basata su array, con possibilità di fairness
- **LinkedBlockingDeque**, **LinkedBlockingQueue**, **LinkedTransferQueue**: implementazioni basate su liste collegate
- **PriorityBlockingQueue**: coda ordinata per priorità
- **DelayQueue**: un elemento non può essere preso prima di un ritardo impostato
- **SynchronousQueue**: ogni produttore deve attendere un consumatore (capacità nulla)

Thread local

```
/**
* These variables differ from their normal counterparts
* in that each thread that accesses one (via its get
* or set method) has its own, independently initialized
* copy of the variable.
*/
public class ThreadLocal< T >;

/**
* Returns the value in the current thread's copy of this
* thread-local variable. If the variable has no value
* for the current thread, it is first initialized to the
* value returned by an invocation of the initialValue()
* method.
*/
public T get();

/**
* Removes the current thread's value for this thread-local
* variable.
*/
public void remove();

/**
* Sets the current thread's copy of this thread-local
* variable to the specified value.
*/
public void set(T value);

/**
* Returns the current thread's "initial value" for
* this thread-local variable.
```

```
*
*/
protected T initialValue();
```

Synchronized

Tutti i blocchi sincronizzati di un oggetto condividono lo stesso monitor lock o intrinsic lock. Può decorare due tipi di raggruppamenti di istruzioni:

- un blocco di istruzioni semplice { ... }
- un metodo
- un oggetto

Wait

Ci costringe a gestire lo stato del blocco. Un Thread può farlo:

- eseguendo un metodo synchronized dell'oggetto
- eseguendo un blocco synchronized all'interno dell'oggetto
- se l'oggetto è una Class, eseguendone un metodo synchronized static

```
/**
* Causes the current thread to wait until another
* thread invokes the notify() method or the notifyAll()
* method for this object.
*/
void wait() throws InterruptedException;

/**
* Wakes up a single thread that is waiting on this
* object's monitor.
*/
void notify();

/**
* Wakes up all threads that are waiting on this
* object's monitor.
*/
void notifyAll();
```

Lock

```
/**
* Lock implementations provide more extensive locking
* operations than can be obtained using synchronized
* methods and statements.
*
*/
public interface Lock;

/**
* Acquires the lock.
*
*/
```

```
void lock();

/**
* Questa chiamata ha l'effetto di sbloccare un thread (tipicamente, uno a caso)
fra quelli
* che attendevano di acquisire il lock.
* Releases the lock.
*
*/
void unlock();

/**
* Acquires the lock only if it is free at the time of invocation.
*
*/
boolean tryLock();
```

ReentrantLock

```
/**
* Creates an instance of ReentrantLock with the given
* fairness policy.
*
*/
public ReentrantLock(boolean fair);
```

Conditions

Una Condition permette di separare l'accodamento in attesa dal possesso del lock che controlla l'attesa.
Lo scopo è da poter gestire, su di un solo lock, di più condizioni di attesa distinte.

```
/**
* una Condition ci viene fornita dal lock su cui deve sussistere. Ciascuna
Condition di uno
* stesso lock consente di gestire un insieme distinto di Thread in attesa.
* Returns a new Condition instance that is bound to this
* Lock instance.
*
*/
public Condition newCondition();

/**
* Causes the current thread to wait until it is signalled
* or interrupted.
*
*/
public void await();

/**
* Wakes up one waiting thread.
*
*/
public void signal();
```

```
/**
* Wakes up all waiting threads.
*
*/
public void signalAll();
```

Semafori

Il valore iniziale del semaforo non è un limite: può essere superato, e può essere anche negativo inizialmente. Mantenere la coerenza semantica sta all'utilizzatore.

La maggior parte dei metodi di Semaphore:

- può lanciare InterruptedException se il thread viene interrotto durante l'attesa
- lancia IllegalArgumentException se il parametro è negativo

```
/**
* Creates a Semaphore with the given number of permits and
* the given fairness setting.
*
*/
public Semaphore(int permits, boolean fair);

/**
* Acquires a permit from this semaphore, blocking until one
* is available, or the thread is interrupted.
*
*/
public void acquire();

/**
* Acquires the given number of permits from this semaphore,
* blocking until all are available, or the thread is
* interrupted.
*
* @param the number of permits to acquire
*/
public void acquire(int permits);

/**
* Releases a permit, returning it to the semaphore.
*
*/
public void release();

/**
* Releases the given number of permits, returning them to
* the semaphore.
*
* @param the number of permits to release
*/
public void release(int permits);

/**
* Shrinks the number of available permits by the
* indicated reduction.
```

```
*
*/
protected void reducePermits(int reduction);

/**
* Acquires a permit from this semaphore, only if one is
* available at the time of invocation.
*
*/
public boolean tryAcquire();

/**
* Acquires the given number of permits from this semaphore,
* if all become available within the given waiting time and
* the current thread has not been interrupted.
* tryAcquire ritorna immediatamente, con risultato falso se non ha ottenuto un
permesso.
* E' in grado di violare la fairness del semaforo.
*/
public boolean tryAcquire(int permits, long timeout, TimeUnit unit);
```