Laboratorio di Reti

Federico Matteoni

1 Thread

Processo Istanza di un programma in esecuzione

Thread **Flusso di esecuzione** all'interno di un processo ⇒ Ogni processo ha almeno un thread.
I **thread condividono le risorse di un processo**.
Possono essere eseguiti sia su single-core (es. interleaving, time-sharing...) che su multicore (più flussi di esecuzione in parallelo)

Multitasking Si può riferire a

Processi, controllato esclusivamente dal S.O.

Thread, controllato in parte dal programmatore

Contesto di un processo Insieme delle informazioni necessarie per ristabilire esattamente lo stato in cui si trova il S.O. nel momento in cui si interrompe l'esecuzione di un processo per passare ad un altro: registri del processore, memoria del processo...

Perché? Per gestire più funzionalità contemporaneamente, come gestire input, visualizzare a schermo, monitorare la rete ed eseguire calcoli.
Esempi noti: browser web, videogame multiplayer. Si creano **più componenti interagenti** in modo da:

Usare meglio le risorse

Migliorare le performance per applicazioni che richiedono grossi calcoli: si dividono i task per eseguirli in parallelo.

Anche problemi: difficile debugging e manutenzione, sincronizzazione, deadlocks...

In Java Il main thread, invocato dalla JVM all'esecuzione del programma, può attivare altri thread. La JVM attiva automaticamente altri thread come il garbage collector.
Un thread è un oggetto. Per creare un thread si definisce un task che implementi l'interfaccia `Runnable` e si crea un thread passandogli l'istanza del task creato. Altrimenti si può estendere la classe `java.lang.Thread`.

Runnable Appartiene a `java.language`, contiene solo la firma del metodo `void run()`. Un oggetto che la implementa è un frammento di codice che può essere eseguito in un thread.

Stati

Created/New: subito dopo l'istruzione `new`, variabili allocate ed inizializzate. Thread in attesa di passare in esecuzione

Runnable/Running: thread in esecuzione o in attesa per ottenere la CPU (Java non separa i due stati).

Not Runnable (**Blocked/Waiting**): thread non può essere messo in esecuzione, può accadere quando attende un'operazione I/O o ha invocato metodi come `sleep()` oppure `wait()`.

Dead: termine naturale o dopo l'invocazione di `stop()` da parte di altri thread (deprecato).

1.1 Threadpool

Perché? In caso di task leggeri molto frequenti risulta impraticabile attivare ulteriori thread. Diventa quindi utile definire un limite massimo di thread che possono essere attivati contemporaneamente, così da sfruttare meglio i processori, evitare troppi thread in competizione e diminuire i costi di attivazione/terminazione dei thread.

Threadpool Struttura dati la cui dimensione massima può essere prefissata, contenente riferimenti ad un insieme di thread. I thread possono essere riutilizzati: la sottomissione di un task al threadpool è **disaccoppiata** dall'esecuzione del thread. L'esecuzione può essere ritardata se non vi sono risorse disponibili.
La **politica di gestione dei thread** stabilisce quando i thread vengono attivati (al momento della creazione del pool, on demand, all'arrivo di un nuovo task...) e quando è opportuno terminare l'esecuzione di un thread.
Il threadpool, quindi, al momento della sottomissione di un task può:

Usare un thread attivato in precedenza e al momento inattivo

Creare un nuovo thread

Memorizzare il task in una coda, in attesa

Respingere la richiesta

Callable Classe per definire un task che può restituire un risultato e sollevare eccezioni

Future Rappresenta il risultato di una computazione asincrona. Definisce metodi per controllare se la computazione è terminata, attendere la terminazione oppure cancellarla. Viene implementata nella classe `FutureTask`.

2 Monitor

Lock Implicite Ogni oggetto ha associate una lock implicita ed una coda. La lock si acquisisce mediante metodi o blocchi di codice `synchronized`. Quando questo viene invocato:

Se nessun metodo `synchronized` della classe è in esecuzione, l'oggetto viene bloccato (la lock viene acquisita ed il metodo viene eseguito.

Se l'oggetto è già bloccato, il thread viene sospeso nella coda associata finché la lock non viene rilasciata.

Notare che la lock è **associata all'istanza della classe**, non alla classe. Metodi su istanze (oggetti) diverse della stessa classe possono essere eseguiti concorrentemente.
I costruttori non possono essere dichiarati `synchronized` (errore di compilazione), perché solo il thread che crea l'oggetto deve poterci accedere mentre l'oggetto viene creato.
Non ha senso specificare `synchronized` nelle interfacce poiché è riferito all'implementazione.
Inoltre il `synchronized` non viene ereditato.

Monitor Meccanismo linguistico ad alto livello per la sincronizzazione. classe di oggetti utilizzabili concorrentemente in modo safe. La risorsa è un oggetto passivo, le sue operazioni vengono invocate da entità attive (thread). La sincronizzazione sullo stato della risorsa è garantita esplicitamente: mutua esclusione sulla struttura garantita dalla lock implicita (un solo thread per volta è all'interno del monitor), meccanismi per sospensione/risveglio sullo stato dell'oggetto condiviso simili a variabili di condizione (`wait/notify`)
Il monitor è quindi un **oggetto** con un insieme di metodi `synchronized` che incapsula lo stato di una risorsa condivisa. Ha due code gestite implicitamente: entry set (thread in attesa di acquisire la lock) e wait set (thread che hanno eseguito una wait e attendono una notify)

Wait Sospende il thread in attesa che si verifichi una condizione (opzionalmente per un tempo massimo). Rilascia la lock (a differenza di `sleep` e `yield`).

Notify Sveglia ad un thread in attesa il verificarsi di una certa condizione (`notifyAll` sveglia tutti i thread in attesa)

Deadlock Due o più thread bloccati per sempre in attesa uno dell'altro

Starvation Thread ha difficoltà ad accedere ad una risorsa condivisa e quindi difficoltà a procedere. In generale task "greedy" che invocano spesso metodi lunghi obbligando gli altri ad aspettare

Livelock Programma che genera una sequenza ciclica di operazioni inutili ai fini dell'effettivo avanzamento della computazione.

3 Concurrent collections

Collezioni di oggetti Insieme di classi che consentono di lavorare con gruppi di oggetti. L'essere o meno thread-safe varia da classe a classe. In generale, tre tipi di collezioni: **senza supporto** per multithreading, **synchronized** collections e **concurrent** collections (introdotte in `java.util.concurrent`).

Vector Contenitore elastico (dimensione variabile) e non generico. Thread-safe conservative locking.

ArrayList Elastico come `Vector`. Prima di JDK5 poteva contenere solo elementi di tipo `Object`, adesso è parametrico (generic) rispetto al tipo di oggetti contenuti.
Elementi possono essere acceduti in modo diretto tramite l'indice. **Non thread-safe** di default: nessuna sincronizzazione per maggiore efficienza.

Unsynchronized Collections Come `ArrayList`, un loro uso incontrollato in un programma multithread può portare a risultati scorretti.

3.1 Synchronized Collections

`Collections` contiene metodi statici per l'elaborazione delle collezioni, **factory methods** per creare versioni sincronizzate di `list/set/map`.

Input: una collezione

Output: la stessa collezione con le operazioni sincronizzate.

La collection risultante è protetta da **lock sull'intera collezione** → degradazione di performance

Nota bene Nessun thread deve accedere all'oggetto originale, requisito ottenibile con istruzioni del tipo
`List<String> synchList = Collections.synchronizedList(new ArrayList<String>());`

Svantaggi Garantiscono la thread-safety a scapito della scalabilità del programma.

3.2 Concurrent Collections

Idea Accettare un compromesso sulla semantica delle operazioni, così da rendere possibile il mantenimento di un livello di concorrenza tale da garantire una buona scalabilità.
Invece di sincronizzare l'intera struttura, si sincronizza solo la parte interessata. Esempio: una hash table ha diversi buckets, quindi sincronizzo solo il bucket a cui accedo.

Concurrent Collections Implementate in `java.util.concurrent`, superano l'approccio "*sincronizza l'intera struttura dati*" garantendo quindi un supporto più sofisticato per la sincronizzazione permettendo l'overlapping di operazioni sugli elementi della struttura.

Vantaggio Maggior livello di concorrenza e quindi migliori performance

Prezzo Modifica della semantica di alcune operazioni.

Lock Striping Invece di una sola lock per tutta la struttura, mantengo lock per sezioni. Ad esempio, una hash table suddivisa in sezioni rende possibili write simultanee se modificano sezioni diverse.
Posso usare 16 lock per controllare l'accesso alla struttura, un numero arbitrario di lettori ed un numero fisso massimo di scrittori (16) che lavorano simultaneamente. Così lettori e scrittori possono convivere.

Vantaggi Maggiore parallelismo e scalabilità

Svantaggi Non si può eseguire la lock sull'intera struttura. Si approssima la semantica di alcune operazioni, ad esempio `size()` e `isEmpty()`, che restituiscono un valore approssimato.
Diventa impossibile sincronizzare funzioni composte da operazioni elementari a livello utente (nelle `synchronized` si usavano blochi `synchronized` che lockavano l'intera collezione).
Per risolvere questo problema si mettono a disposizione operazioni eseguite atomicamente, come `putIfAbsent(K key, V val)`, `removeIfEqual(K key, V val)`, `replaceIfEqual(K key, V oldVal, V newVal)`

4 Java NIO

Obiettivi Incrementare la performance dell'I/O, fornire un insieme eterogeneo di funzionalità per l'I/O e aumentare l'espressività delle applicazioni.

Non semplice Per migliorare le performance è necessario definire le primitive a livelli di astrazione più bassi. Inoltre i risultati dipendono dalla piattaforma di esecuzione. Si rende necessario uno sforzo di progettazione maggiore.

Costrutti base

Canali e buffer Invece dell'I/O standard Java che opera su stream di byte o caratteri. I dati sono trasferiti da canali a buffer o viceversa, ed i buffer vengono gestiti esplicitamente dal programmatore.
Un **channel** è simile ad uno stream. I dati possono essere letti dal channel in un buffer e viceversa scritti dal buffer in un channel.

Buffer Contenitori di dati di dimensione fissa. Contengono dati appena letti o da scrivere.
Sono oggetti della classe `java.nio.Buffer` **non thread-safe**.

Selector Oggetto in grado di monitorare un insieme di canali. Itercetta eventi provenienti dai canali monitorati: dati arrivati, connessione aperta...
Sono simili agli stream, ma **bidirezionali**, con **scattered read** (distribuisce i dati letti da un canale in uno o più buffer), **gathering write** (scrive su un canale i dati recuperati da più buffer) e supporta **trasferimenti diretti tra canali**.

Channel Rappresentano connessioni con entità capaci di eseguire operazioni I/O. `Channel` è un'interfaccia radice di una gerarchia di interfacce:

`FileChannel`: legge/scrive dati su file

`DatagramChannel`: legge/scrive dati su rete via `UDP`

`SocketChannel`: legge/scrive dati su rete via `TCP`

`ServerSocketChannel`: attende richieste di connessioni `TCP` e crea un `SocketChannel` per ogni connessione creata

Sono **bidirezionali**. Tutti i dati sono gestiti tramite oggetti di tipo `Buffer`, quindi senza leggere/scrivere direttamente sul canale. Inoltre possono essere **bloccanti o meno**, non bloccati utili soprattutto per le comunicazioni in cui i dati arrivano in modo incrementale come i collegamenti di rete.