

Laboratorio di Reti

Federico Matteoni

1 Thread

Processo Istanza di un programma in esecuzione

Thread Flusso di esecuzione all'interno di un processo \Rightarrow 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.lang**, 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 Implicito 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 blocchi `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**.

Direct Buffers: buffer allocati fuori dalla JVM, nella memoria gestita dal S.O.

Selector Oggetto in grado di monitorare un insieme di canali. Intercetta 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.

5 Serializzazione

Oggetti Caratterizzati da stato (vive con l'istanza) e comportamento (specificato dalla classe).

Serialization Scrittura e lettura di oggetti. Si basa sulla possibilità di scrivere lo stato di un oggetto in una forma sequenziale, sufficiente per ricostruire l'oggetto quando viene riletto.

Il flattening/salvataggio dell'oggetto riguarda lo stato e può avvenire in diversi modi:

Accedendo ai singoli campi, salvando i valori per esempio su un file di testo in un formato opportuno

Trasformando automaticamente l'oggetto in uno stream di byte mediante il **meccanismo della object serialization** di Java

La serializzazione consente di persistere degli oggetti al di fuori del ciclo di vita della JVM. Si crea una rappresentazione dell'oggetto indipendente dalla JVM che ha generato l'oggetto stesso. Un'altra JVM, se ha accesso alla classe ed all'oggetto serializzato, può ricostruirlo.

Persistenza Fornisce un meccanismo di persistenza dei programmi, consentendo l'archiviazione di un oggetto su un file. Ad esempio, per memorizzare lo stato di una sessione.

Interoperabilità Meccanismo di interoperabilità mediante oggetti condivisi tra diverse applicazioni.

Approcci

Java Object Serialization Utilizzabile solo se chi serializza e chi deserializza è scritto in Java.

Testo Come XML e JSON (JavaScript Object Notation), formati dalla sintassi semplice e facilmente riproducibile. Interpretabile da qualsiasi linguaggio.

6 JSON

Formato leggero per interscambio di dati, indipendente dalla piattaforma poiché è semplice testo. Non dipende dal linguaggio di programmazione ed è self-describing.

Strutture Coppie chiave:valore e liste ordinate di valori. Una risorsa JSON ha una struttura ad albero.

7 RMI

Remote Method Invocation Un'applicazione RMI è composta da due programmi: server e client. L'obiettivo è di **permettere al programmatore di sviluppare applicazioni distribuite** usando la **stessa sintassi** e semantica **dei programmi non distribuiti**.

Viene raggiunto solo in parte, ha una buona trasparenza ma non totale.

Server Crea un oggetto remoto e **pubblica** un riferimento all'oggetto. Dopodiché, **attende** che i client invochino metodi sull'oggetto remoto.

Client **Ottiene** un riferimento all'oggetto remoto e **invoca** i suoi metodi.

Applicazione distribuita Un'applicazione JAVA distribuita è composta da computazioni eseguite su JVM differenti, possibilmente in esecuzione su host differenti comunicanti tra loro. Un'applicazione multithread non è distribuita, perché è eseguita sulla solita JVM.

Socket Richiedono la progettazione di protocolli di comunicazione e la verifica delle loro funzionalità. Sono meccanismi flessibili ma di basso livello. La **serializzazione** consente di ridurre la complessità dei protocolli inviando dati strutturati.

RPC Un'alternativa è una tecnologia di più alto livello originariamente chiamata **Remote Procedure Call**, la cui interfaccia di comunicazione è rappresentata dall'invocazione di una procedura remota invece che dall'utilizzo diretto di un socket.

Sfrutta il paradigma **domanda/risposta**:

1. Client invoca procedura del server remoto
2. Server esegue la procedura con i parametri passati dal client e restituisce al client il risultato

La connessione remota è trasparente rispetto a client e server, prevede meccanismi di affidabilità a livello sottostante.

Funzionamento Il programmatore non si preoccupa di sviluppare i protocolli per lo scambio di dati, la verifica, la codifica e decodifica. Le **operazioni** sono **interamente gestite dal supporto** tramite **stub**/proxy presenti sul client.

Marshalling Preparazione e impacchettamento dell'informazione per la trasmissione. Ad esempio, traduce formati non portabili in formati portabili o canonici. Viene spesso **implementato con stub generati automaticamente**: il programmatore si limita a sviluppare l'interfaccia per la RPC e lo stub generator prende la definizione dell'interfaccia e crea gli stub per il server e il client.

Gli stub si occupano di fare il marshalling/unmarshalling dei parametri, oltre che della comunicazione e dell'invocazione delle procedure. Questo è possibile perché queste azioni sono facilmente definibili dati i parametri e l'identità della procedura.

Limiti della RPC La programmazione è essenzialmente procedurale, inoltre la locazione del server non è trasparente (il client deve conoscere IP e porta del server). Quindi vengono proposte diverse tecnologie per superarli, tra cui l'RMI.

RMI Permette di usare veri e propri **oggetti remoti** come se fossero oggetti come tutti gli altri, senza preoccuparsi della realizzazione di protocolli, connessione ecc. . .

Il loro utilizzo è largamente trasparente: una volta che viene localizzato si possono usare i metodi dell'oggetto come se fosse un oggetto locale. La codifica, decodifica, verifica e trasmissione dei dati sono effettuati dal supporto RMI in maniera trasparente.

Problemi da gestire Il client deve in qualche modo trovare il riferimento all'oggetto remoto, e l'oggetto remoto deve esplicitamente rendere i suoi servizi invocabili e rendersi reperibile.

Funzionamento RMI permette di costruire riferimento remoti.

Due **proxy**:

Proxy locale: **stub** nel client, su cui vengono fatte le invocazioni destinate all'oggetto remoto

Entità remota: **skeleton** nel server, che riceve le invocazioni fatte sullo stub e le realizza effettuando le corrispondenti chiamate sul server.

Pattern proxy: questi componenti nascondono al livello applicativo la natura distribuita dell'applicazione. Non è possibile riferire direttamente l'oggetto remoto, si usa una **variabile d'interfaccia** per contenere il riferimento al proxy, che permette di controllare e preparare il passaggio da client a server.

Remote Reference Layer RRL, responsabile della gestione dei riferimenti agli oggetti remoti, dei parametri e delle astrazioni stream-oriented connection.

Registry Servizio di naming che agisce da "pagine gialle":

Registra i nomi e i riferimenti agli oggetti i cui metodi possono essere invocati da remoto

Gli oggetti devono essere registrati (**bind**) con un nome pubblico

I client possono richiedere (**lookup**) gli oggetti chiedendone un riferimento a partire dal nome pubblico

Operazioni Le principali operazioni necessarie rispetto all'uso di un oggetto locale sono:

Server deve **esportare** gli oggetti remoti e registrarli tramite la **bind**

Client deve **individuare** un riferimento all'oggetto remoto, cercandoli tramite la **lookup** nel registry

Il client invoca il servizio mediante le chiamate di metodi che sono gli stessi delle invocazioni dei metodi locali.

Livello logico identico all'invocazione di metodi locali

Livello di supporto: gestita dall'RMI che provvede a trasformare i parametri della chiamata remota in dati da spedire in rete. Il network support provvede all'invio vero e proprio.

Considerazioni pratiche

Separazione tra definizione del comportamento (**interfaccia**) e sua implementazione (**classe**)

Componenti remoti riferiti tramite variabili d'interfaccia.

Le interfacce devono estendere `java.rmi.Remote` ed ogni metodo deve propagare `java.rmi.RemoteException`.

Le classi devono implementare l'interfaccia definita ed estendere `java.rmi.UnicastRemoteObject`.

Interfaccia L'interfaccia è remota solo se estende `java.rmi.Remote`. `Remote` non definisce metodi ma identifica gli oggetti che possono essere usati in remoto.

8 REST

Web Service Programma che utilizza la connessione web e richiede ad un server esterno di fornire dati o eseguire un algoritmo. Tipologie possibili: RESTful, RPC, ibrido.

REST Il **RE**presentational **S**tate **T**ransfer è un **stile architetturale** per lo sviluppo di web services. Viene usato per costruire sistemi debolmente accoppiati: il web è una *istanza* di questo stile architetturale.

Principi base ROA (Resource Oriented Architecture) è un insieme di linee guida per implementare un web service RESTful. I principi base REST sono:

Resource Identification

Una **risorsa** è qualsiasi cosa sia sufficientemente importante da poter essere considerata come un'entità. Bisogna **dare un nome a tutto ciò di cui si vuole parlare**, utilizzando gli **URI**.

Uniform Interface

Stesso piccolo insieme di operazioni applicato a **tutto**. Un piccolo insieme di **verbi universali** (non inventati in base all'applicazione) applicato ad un largo insieme di **sostantivi**. Se servono nuovi verbi a molte applicazioni si estende l'interfaccia.

POST, crea una sottorisorsa

GET, restituisce una rappresentazione della risorsa

PUT, inizializza o aggiorna lo stato di una risorsa

DELETE, elimina una risorsa

Altre: HEAD per leggere i metadati, OPTIONS per le operazioni disponibili, PATCH per modificare una parte.

Self-Descriptive Messages

Ogni messaggio contiene le informazioni necessarie per la propria gestione. Le risorse sono entità **astratte** (non utilizzabili di per sé). L'accesso alla risorsa avviene mediante la **rappresentazione**, cioè allo stato attuale. Il formato è negoziabile, ma deve supportare i link.

Hypermedia as the Engine of Application State (HATEOAS)

Sono trasferite rappresentazioni delle risorse contenenti link, con cui il client può procedere al passo successivo dell'interazione. Le risorse e lo stato sono **navigati** tramite i link → **le applicazioni RESTful navigano invece di chiamare**.

Stateless Interactions

Ogni richiesta al server deve contenere le informazioni necessarie per capire completamente la richiesta, **indipendentemente dalle richieste precedenti**. La richiesta avviene in completo isolamento.

Ciò non significa applicazioni stateless: l'idea è di evitare transazioni di lunga durata nelle applicazioni. Lo **stato della risorsa è gestito sul server**, è lo stesso per ogni client quindi se uno lo cambia gli altri devono vedere il cambio. Lo **stato del client è gestito sul client**, specifico per ognuno e potrebbe avere effetto sulle richieste mandate al server ma non sulle risorse stesse.

Ad esempio è il client a comunicare che mi trovo in una data directory o che ho visualizzato la figura n. 3.

9 URI

URI Essenziali per implementare la resource identification. Gli URI sono identificatori **universali** per le cose, generalmente leggibili dagli umani. Struttura:

`<scheme>://<authority><path>?<query>#<fragment>`

Esempio: `http://www.google.it/search?q=rest&start=10#1`

I fragment non sono inviati al server.

Linee guida Preferire i nomi ai verbi, mantenere URI brevi e non cambiarli (usare i redirect).

Gli URI REST sono identificatori che devono essere scoperti seguendo i link e non costruiti dal client.