Appunti di Paradigmi

4 ottobre 2022

Rosso Carlo

Contents

1 Introduzione

Def. 1.1 (Paradigma) *Dal greco paradeigma: modello, esempio. Modello di riferimento, termine di paragone. In informatica, un paradigma l'insieme delle tecniche e dei metodi considerati adeguati ad affrontare una classe determinata, anche se ampia, di problemi.*

Esempi:

- concorrenza: più linee di esecuzione contemporanee, asincrone, che condividono l'uso di un insieme di risorse in modo (possibilmente) coordinato;
- parallelismo: più linee di esecuzione contemporanee, sincrone, che eseguono in modo coordinato lo stesso calcolo su di un partizione dei dati di ingresso;
- in rete: collaborazione con altri sistemi attraverso la comunicazion esincrona su di una interfaccia di rete;
- distribuzione: un sistema è costituito da nodi indipendenti che, attraverso una rete non affidabile, devono coordinare l'esecuzione dello stesso lavoro, condividendo il consenso sullo stato complessivo del sistema;
- reattività: un sistema distribuito costruito sulle basi della comunicazione asincrona tramite messaggi da cui ottiene caratteristiche di flessibilità, resilienza, scalabilità e reattività;

Utilizziamo Java per implementare i paradigmi di programmazione. Java è un linguaggio OO, a memoria gestita (GC), con controllo statico dei tipi, basato su classi ad ereditarietà singola, con una sintassi simile a C e C++. Java è compilato in un linguaggio intermedio (bytecode), il cui modello di esecuzione è descritto nella specifica della JVM. Il bytecode può essere eseguito in molti modi: interpretato, compilato durante l'esecuzione (aka JIT), compilato prima dell'esecuzione (GraalVM).

1.1 Java

Ogni file deve chiamarsi come l'oggetto pubblico che contiene: NomeClasse.java.
Il percorso delle directory deve corrispondere al package in cui l'oggetto si trova.
Nei sorgenti, o nel CLASSPATH, devono esserci tutti i tipi nominati dai sorgenti.
L'ordine di compilazione non è importante.
Il comando "java" avvia la JVM ed esegue il bytecode contenuto nel CLASSPATH.
Il formato .jar è il più comune formato di distribuzione del codice nella piattaforma Java. Si tratta di un archivio compresso zip contenente alcuni file specifici che descrivono il suo contenuto e come usarlo.
L'unità principale di organizzazione del codice è la Classe. Ogni oggetto fa necessariamente riferimento alla definizione di un aClasse, che determina la struttura del suo stato ed il codice che opera su tale stato. Il codice eseguibile deve necessariamente essere incluso in una Classe: come metodo dotato di un nome e richiamabile da altri oggetti, o come blocco anonimo eseguito alla creazione di ciascuna istanza. Le classi formano l'insieme dei Tipi che il compilatore riconosce in un programma Java. Per convensione i package sono denominati con nomi di dominio, in ordine inverso.
Una classe non può usare un'altra classe qualsiasi: deve averne visibilità e dichiarare l'intenzione di usarla. Una classe può usare una qualsiasi altra classe all'interno dello stesso package senza indicazioni particolari. Un file sorgente può contenere al più una classe pubblica, e deve chiamarsi come la classe contenuta. "import", per importare una classe. Una classe può contenere variabili, metodi, altre classi, blocchi di codice anonimi.

Interfacce: una interfaccia dichiara le caratterisctiche di un tipo senza fornire una sua implementazione. Le classi possono dichiarare di implementare una interfaccia fornendo l'implementazione richiesta. Una interfaccia può essere estesa solo da un'altra interfaccia. Una interfaccia può avere visibilità pubblica o di package.

2 Concorrenza

Def. 2.1 (programma concorrente) *Teoria e tecniche per la gestione di più processi sulla stessa macchina che operano contemporaneamente condividendo le risorse disponibili.*

Questo richiede sempre maggiore gestione della concorrenza e dell'accesso contemporaneo alle stesse risorse. Inoltre, alcune di queste tecniche si sono rivelate problematiche dal punto di vista della sicurezza. Per gestire più linee di esecuzione all'interno dello stesso processo è stato ideato il concetto di thread. I thread condividono le risorse di uno stesso processo, rendendo più economico il costo di passaggio da un ramo di esecuzione all'altro. La programmazione distribuita implica la comunicazione fra entità che non possono avere stato condiviso. La programmazione funzionale tratta preferibilmente dati immutabili, con qualche concessione alla mutabilità per lo stretto necessario. Lo stato può essere distinto o condiviso. La programmazione concorrente si pone nel qudrante più difficile, dove lo stato è mutabile e condiviso, e quindi l'accesso e l'intervento su di esso va coordinato e gestito.

Problemi della concorrenza:

- non determinismo;
- starvation;
- race condition;
- deadlock;

Tipologie di concorrenza:

- collaborativa: co-routine;
- pre-emptive: processi, threads;
- real-time: processi, threads;
- event driven (o async): future, events, treams;

collaborativa I programmi devono esplicitamente cedere il controllo ad intervalli regolari. Casi d'uso: embedded, very high perfomance.

pre-emptive Il sistema operativo è in grado di interrompere l'esecuzione di un programma e sottrargli il controllo delle risorse per affidarle al programma seguente.

real-time Il sistema operativo garantisce prestazioni precise e prefissate nella suddivisione delle risorse fra i programmi.

event driven/async I programmi dichiarano esplicitamente le operazioni che vanno eseguite e lasciano all'ambiente di esecuzione la decisione di quando eseguirle e come assegnare le risorse. Si tratta di un metodo poco comune a livello di sistema operativo, ma molto popolare nell'organizzazione delle applicazioni.

Def. 2.2 (Future) *Un oggetto Future rappresenta un calcolo che prima o poi ritornerà un valore. Si può verificare se il calcolo è stato completato, e quindi ottenere il valore risultante, o controllare se sia ancora in corso.*

Def. 2.3 (sezione critica) *Si dice sezione critica la parte di codice in cui vengono acceduti i dati condivisi. Permettere a più thread di trovarsi contemporaneamente nella sezione critica porta ad errori (race condition).*

syncronized è una parola chiave che applicata ad un blocco di istruzioni impedisce che sia percorso contemporaneamente da più thread. Può decorare due tipi di raggruppamenti di istruzioni: un blocco di istruzioni semplice ($\{\dots\}$) o un metodo. In entrambi i casi è utilizzato un monitor lock o intrinsic lock.
Si può anche controllare manualmente una sezione critica, con un lock o con un semaforo. Un lock è un semaforo che può essere acquisito da un solo thread alla volta. Un semaforo è un contatore che può essere incrementato o decrementato da più thread contemporaneamente (più o meno).

2.1 Thread-safety

Se possibile è meglio evitare di condividere dati fra i thread; se necessario si utilizzano alcuni strumenti specifici.

Problemi:

- deadlock
- race condition: il risultato non è deterministico

Atomicità

Per incrementare un contatore condiviso, si utilizza una delle classi del package "java.util.concurrent.atomic":

- Integer: AtomicInteger;
- Long: AtomicLong;
- Object: AtomicReference;
- IntegerArray: AtomicIntegerArray;
- ... (altri array);

Queste classi garantiscono che la modifica del valore che contengono sia "atomica" (quindi thread-safe). Inoltre, si assicurano di evitare deadlock.
L'AtomicReference è formata da un puntatore e da un boolean che funge da semaforo.

Volatile

La parola chiave volatile viene utilizzata per dichiarare una variabile che deve sempre essere letta "dalla memoria principale". Questa parola impone la regola "Happens-Before Order": la scrittura su un campo volatile avviene prima di ogni altra lettura su quel campo. La garanzia fornita da volatile è utile alla correttezza del programma solo se nessun thread scrive nella variabile volatile un valore dipendente dal valore che ha appena letto dalla stessa variabile.

Concurrent Data Structures

Nel package "java.util.concurrent" sono presenti le collezioni ottimizzate per la concorrenza. Queste collezioni danno garanzia di atomicità ed ordinamento delle operazioni.
Alcune implementazioni offrono metodi come reduce, search o foreach, che suddividono automaticamente il lavoro fra i thread.

BlockingQueue è una coda che rende possibile scegliere la semantica dell'operazione di accodamento e prelievo. Permette quindi maggiore flessibilità nella gestione dell'accesso ai dati su più thread.

ThreadLocal

Una variabile "ThreadLocal¡T¿" esiste in una copia differente ed indipendente per ciascun Thread che attravera la sua dichiarazione. Ovvero, ogni volta che tale variabile è richiesta in un thread diverso, suddetto thread la copia e crea una nuova istanza. Assomigliano alle variabili globali.

2.2 Parallel Streams

Lo Stream rappresenta l'iterazione su di un insieme di cardinalità non nota, potenzialmente infinita. Si tratta di un modello in cui l'esecuzione sequenziale e quella parallela non richiedono modifiche di codice. Questo è possibile perchè sono permesse operazioni con proprietà più restrittive.
Uno Stream richiede la definizione dell'algoritmo di calcolo che avverrà sopra i suoi elementi, indipendentemente dal loro recupero, che è fissato.

Ci sono alcune flag, per indicare le proprietà dell'operazione:

- concurrent;

- distinct: garantisce che non ci siano duplicati secondo equals();
- immutable;
- nonnull;
- ordered: l'ordine di elaborazione è lo stesso dell'ordine di origine;
- sized: garantisce che il numero di elementi sia noto;
- sorted: mantiene gli elementi ordinati secondo il loro ordinamento naturale o dato da un Comparator;
- subsized;

SplitIterator

Per esprimere una classe parallelizzabile servono dei metodi che consentano alla sorgente di esplicitare la suddivisione del lavoro.
Con "tryAdvance()" lo stream fornisce un elemento al thread che ci lavora.

Collector

Nell'operazione di riduzione, l'accumulazione del risultato avviene creando nuovi valori (altrimenti si usa forEach). Questo non è sempre conveniente. Lo Stream, di default, decide autonomamente quanto parallelismo usare attraverso il ForkJoinPool.

Approfondimento

La descrizione degli stream fino ad ora proposta è incompleta: manca un protocollo esplicito per gestire la terminazione dello stream; e gli errori sono gestiti come eccezioni. Per questo, sono sviluppati i Reactive Streams. Le basi del modello sono i seguenti concetti:

- Observable: emette i dati, è l'equivalente di uno stream;
- Scheduler: esegue i task;
- Subscriber: riceve i dati, è l'equivalente di un Consumer;
- Subject: può consumare un Observable per produrre un altro Observable, permette di introdurre modifiche sostanziali nel flusso di dati;

L'idea del modello è un'implementazione dell'Observer Pattern. In questo modo si ottiene una semantica più ricca, maggiore regolarità nella composizione e indipendenza dal modello di esecuzione.

3 Programmazione Distribuita

La programmazione distribuita sono la teoria e le tecniche per la gestione di più processi su macchine diverse che operano in modo coordinato allo svolgimento di un unico compito. Un insieme di macchine che esegue un algoritmo distribuito è detto sistema distribuito. Caratteristiche:

- affidabilità: anche se alcuni nodi sono fermi o in errore, il sistema continua a funzionare;
- scalabilità: il sistema può essere esteso con l'aggiunta di nuovi nodi, se quelli già presenti non fossero sufficienti;
- diffusione: il sistema può essere distribuito su un'area geografica ampia;
- concorrenza: i vari nodi di esecuzione operano contemporaneamente;
- totale asincronia: l'ordine temporale delle operazioni non è strettamente condiviso; se fosse necessario sarebbe imposto con opportuni mezzi (meglio evitare);
- fallimenti imperscrutabili: un guasto è indistinguibile da un ritardo. L'unica risorsa in comune tra i nodi è il collegamento di rete, per cui le principali astrazioni disponibili riguardano l'invio di un messaggio e l'attesa della ricezione di un messaggio (la sincronizzazione rallenta parecchio);

3.1 Serializzazione

Un passo fondamentale è la serializzazione, ovvero il metodo con cui un oggetto viene predisposto per la trasmissione in un messaggio. La serializzazione prende un oggetto e lo traduce in un messaggio, utilizzando il metodo "marshall()". La deserializzazione è il processo inverso ed usa il metodo "unmarshall()".
Tendenzialmente sono utilizzate connessioni HTTP, per cui si usa XML o JSON, in modo che i messaggi siano leggibili anche da un uomo.

3.2 Comunicazioni

Le astrazioni di vase che abbiamo a disposizione per la comunicazione tra più JVM corrispondono direttamente alle caratteristiche del protocollo TCP/IP: socket e datagram.

3.3 Socket

Un Socket è un'astrazione per la comunicazione bidirezionale punto-punto fra due sistemi. Il Socket è composto da un client e un server. Un Socket fornisce un InputStream e un OutputStream per ricevere e trasmettere dati nel collegamento. Regole:

- sono thread-safe, ma un solo thread può scrivere o leggere per volta;
- i buffer sono limitati, quindi si rischia di perdere dati;
- lettura e scrittura possono bloccare il thread (deadlock);
- le connessioni possono avere caratteristiche particolari (urgent data, ...);
- una volta terminato, il socket va chiuso esplicitamente;
- le stream sono continue, per cui è necessario un carattere di separazione;

3.4 Datagram

Non c'è garanzia di ricezione o ordinamento in arrivo. La dimensione massima è di 64Kb. Per inviare o ricevere abbiamo una sola classe, senza distinzione di operatività. Con i Datagram la dimensione del messaggio è nota, inoltre è possibile inviare messaggi a più indirizzi contemporaneamente. Contro (rispetto ai Socket): non c'è garanzia nè segnale di consegna; i messaggi sono inviati in una sola direzione; messaggi lunghi subiscono una forte penalità di affidabilità.

3.5 URL

Permette connessioni del tipo HTTP a basso livello.

3.6 HTTP client

Si tratta di un'evoluzione della classe URL. Utilizza il build pattern. Una volta costruita una richiesta, la si esegue per ottenere la risposta.

3.7 Channel

Un Channel è un'astrazione per la comunicazione bidirezionale punto-punto fra due sistemi. Per comunicazioni tra più di due sistemi, si consiglia di utilizzare i channel, che permettono una gestione più sintetica e leggibile.

3.8 Punti Deboli

- the network is reliable;
- latency is zero;
- bandwidth is infinite;
- the network is secure;
- topology doesn't change;
- there is one administrator;
- transport cost is zero;
- the network is homogeneous;

Oltre a queste tecnologie a basso livello, sono disponibili i framework, talvolta opensource. Per cui si utilizza del codice già scritto, per risparmiare tempo. I framework sono librerie in cui alcuni casi d'uso sono già implementati. Per scegliere il framework più adatto bisogna leggere la documentazione per capire quali proprietà sono garantite e quali sono messe in secondo piano. Usare un framework significa accettare le sue proprietà e i suoi limiti. Inoltre, può succedere che un framework sia aggiornato, nel qual caso può risultare necessario un refactoring.

3.9 CRDT (Conflict-free Replicated Data Type)

Un CRDT è un tipo di dato che può essere replicato in modo asincrono su più nodi, fornendo la garanzia che esiste un modo di riconciliare tutte le possibili modifiche risolvendo ogni possibile conflitto. Ecco alcuni esempi di queste strutture dati:

- grow-only counter;
- positive-negative counter;
- grow-only set;
- 2-phase set: set di elementi aggiunti e set di elementi rimossi;
- last-writer-wins set;

3.10 Back-pressure

Con back-pressure si intende la resistenza che il componente successivo può opporre ai dati provenienti dal componente precedente della catena di elaborazione. Questa strategia permette ad ogni nodo della catena di dichiarare quanti dati è in grado di elaborare.

3.11 Reactive Streams

- Publisher: fornisce un numero arbitrario di elementi di tipo T;
- Subscriber: consuma gli elementi di tipo T e controlla il flusso degli elementi in arrivo;
- Subscription: rappresenta la relazione tra un Publisher e un Subscriber;
- Processor: estende Publisher e Subscriber, permettendo di trasformare ed elaborare gli elementi;