2023-11-06 - Threads classici, Modello ad attori, Creazione di attori, Invio e ricezione di messaggi, Registrazione attori

Threads classici

Un thread è un **flusso di esecuzione** di un processo. Diversi thread **condividono codice** (`.text`) e **memoria statica** (`.data`), ma hanno **stack** ed **heap indipendenti**. Comunicano attraverso un area di **memoria condivisa**.

I thread sono il modo *"tradizionale"* di offrire **concorrenza**, ma questo fa sorgere il **problema** delle **race conditions**: due thread *accedono* ad una risorsa e, dopo aver effettuato un *controllo*, la *modificano*. Ma in caso l'esecuzione di un thread venga *interrotta* dallo scheduler in alcuni punti, possono verificarsi dei *problemi*, ed ottenere dello *stato inconsistente* o *persi* alcuni aggiornamenti della risorsa (**update loss**).

Esempio: prelievo da ATM

thread$_1$: `withdraw(10)`	thread$_2$: `withdraw(10)`	balance
`if (balance - amount >= 0)`		15€
	`if (balance - amount >= 0)`	15€
	`balance -= amount`	5€
`balance -= amount`		-5€

Lo scheduler ha *bloccato l'esecuzione* del primo thread subito *dopo il controllo*, dando tempo di CPU al secondo, che ha effettuato un *aggiornamento*. Quando il primo thread *riparte*, il controllo era *già* stato *effettuato*, quindi viene *aggiornata* la risorsa (anche se il controllo sarebbe *ora falso*).

Per risolvere questo problema è possibile utilizzare i **lock** offerti dal **sistema operativo**, che **bloccano l'accesso** ad una risorsa quando un thread lo richiede, in modo che nessun altro thread possa **modificarla** mentre è bloccata.

Questo meccanismo, oltre ad essere **dispendioso** *(bisogna delegare l'operazione di lock al sistema operativo)*, può portare a dei problemi, come ad esempio il **deadlock**, ovvero un **ciclo di lock** che si aspettano a vicenda, **bloccando all'infinito** l'esecuzione.

Erlang *(e **Scala** attraverso la libreria Akka)* usano un approccio diverso, il **modello ad attori**.

Modello ad attori

Nel modello ad attori, ogni attore è un **light-weight process** della macchina virtuale di Erlang *(che gestisce anche la schedulazione di questi thread più leggeri rispetto a quelli del sistema operativo)* ed è completamente **indipendente** dagli altri *(non condividono nulla, in modo da poter distribuire in maniera molto più efficace gli attori)*.
La **comunicazione** avviene solamente tramite **scambio di messaggi asincrono** *(asincrono: non è richiesto un ascolto attivo e una risposta immediata)*. Ogni attore è caratterizzato da un **nome univoco** e si può mandare un **messaggio** ad un attore dato il *suo nome*.

In Erlang, **ogni oggetto** è un **attore** ed ha una coda di messaggi (**mailbox**) e un comportamento (**behaviour**, *il suo codice*). La computazione è **data-driven**, un attore esegue *qualcosa* all'arrivo di un **messaggio**.
Ogni attore:

- può **creare** uno o più **attori**
- può **mandare** uno o più **messaggi** ad altri attori
- può **assumere un comportamento diverso** per gestire i messaggi in coda

Creazione di attori

È possibile **creare un attore** attraverso la funzione `spawn(...)`, che riceve come **parametro** la **funzione** da far eseguire al nuovo attore e **restituisce** il **nome** del nuovo attore.

L'attore *"padre"* **conosce** il nome del *"figlio"* (restituito dalla spawn) e quindi può **mandargli messaggi**, mentre il *"figlio"* non conosce il nome del *"padre"*. Per permettere anche la **comunicazione nell'altro senso**, deve essere passato il nome del *"padre"* come **parametro della funzione** che il *"figlio"* esegue (passata alla `spawn`).

Un attore può ottenere il **proprio nome** (**pid**) con `self()` ed il **proprio nodo** *(macchina virtuale che sta eseguendo il codice)* con `self()`.

Sono definite diverse funzioni `spawn` (con **stesso comportamento**) con diverso numero di argomenti:

```
% spawn/1: spawn(Fun)
spawn(fun() -> io:format("hello") end).
% spawn/2: spawn(Node, Fun)
spawn(node(), fun() -> io:write("hello") end).
% spawn/3: spawn(Module, Function, Args)
spawn(io, write, ["hello"]).
% spawn/4: spawn(Node, Module, Function, Args)
spawn(node(), io, write, ["hello"]).
```

Quando viene stampato, il nome di un attore è nel formato `<X.Y.Z>`, dove `X`, `Y` e `Z` sono dei numeri *(dove `X` rappresenta il nodo e `Y` l'attore all'interno del nodo)*. Ma questo nome **non** è "costruibile", è solo **ottenibile** attraverso la funzione `self()`, dato che *"sottobanco"* è una struttura dati molto più complessa *(che memorizza altre informazioni dell'attore)*, che viene semplificata per essere stampata.

La creazione di attori è **molto veloce**, decine di migliaia di processi in microsecondi *(20000 in 2.5 microsecondi)*. Il **numero di attori** *"spawnabili"* è **limitato** *(ad un numero che dipende dall'architettura e dalla BEAM)*, ma è *modificabile* al lancio della shell con `erl +P <massimo_numero_attori>`.
Questa velocità è dovuta alla **gestione degli attori** a livello della **macchina virtuale** di Erlang (**BEAM**) e non del sistema operativo, che risulterebbe decisamente troppo *"overkill"* per gestire dei processi così leggeri (attori). La BEAM utilizza uno scheduling sugli attori di tipo **preemptive**.

Invio di messaggi

Un attore può mandare un **messaggio** *(una qualsiasi espressione valida)* ad un processo di cui **conosce il Pid**, attraverso l'operatore `!`.

L'operazione di invio **non è bloccante** *(mai)* e non verrà **mai notificato** un errore (nemmeno se il destinatario è irraggiungibile). Il destinatario **non** conosce il **mittente** *(a meno che non venga passato nel messaggio)*.
L'invio di messaggi **non è ordinato**, il destinatario può gestirli in ordine sparso *(ordinati in base al tempo di ricezione, NON invio)*.

```
Pid = spawn(...). % creazione altro attore
% invio messaggio (il nuovo attore non conosce il mittente)
Pid ! {ciaoanonimo, "Ciao"}.
% invio messaggio (il nuovo attore conosce il mittente)
Pid ! {ciaopalese, "Ciao", self()}.
```

Ricezione di messaggi

La **ricezione** di messaggi avviene attraverso **pattern matching** `receive ... end`.

Il matching avviene in ordine sulla coda dei messaggi in arrivo *(mailbox)*, dal più vecchio al più nuovo *(in ordine di **arrivo**, **NON invio**)*. Quando un messaggio viene matchato viene rimosso dalla coda.

La ricezione è **bloccante**, se non è presente *nessun* messaggio che rispetta un pattern l'attore rimane **in attesa** di un messaggio che rispetti un pattern. È possibile *interrompere l'attesa* con la clausola `after` alla fine dei pattern.

```
% viene matchato qualsiasi messaggio
receive
    Any -> do_something(Any)
end.
```

```
% viene ricevuto solo il messaggio che rispetta questo pattern
receive
    {Pid, something} -> do_something(Pid)
end.

% matcha due pattern, se non arriva un messagio consono in 1000ms
% allora viene eseguito body()
receive
    {pattern1, Num} -> function1(Num);
    {pattern2, Num} -> function2(Num);
    after 1000 -> body()
end.
```

Esempio: convertitore di temperature

```
-module(converter).
-export([t_converter/0]).

t_converter() ->
  receive
    {toF, C} -> io:format("~p °C is ~p °F~n", [C, 32+C*9/5]), t_converter();
    {toC, F} -> io:format("~p °F is ~p °C~n", [F, (F-32)*5/9]), t_converter();
    {stop} -> io:format("Stopping~n");
    Other -> io:format("Unknown: ~p~n", [Other]), t_converter()
  end.

% Pid = spawn(converter, t_converter, []).
% Pid ! {toC, 32}. % 32 °F is 0.0 °C
% Pid ! {toF, 100}. % 100 °C is 212.0 °F
% Pid ! ehehehe. % Unknown: ehehehe
% Pid ! {stop}. % Stopping!
```

Dopo che avviene il matching *(tranne su stop)* viene **rilanciata ricorsivamente** la funzione `t_converter`, in modo da continuare a fare il matching su altri messaggi *(altrimenti il "server" dopo aver matchato il primo messaggio si fermerebbe)*.
Questa ricorsione è **ottimizzata**, lo **stack non cresce**.

Dare nomi ad attori (registrazione)

È possibile **assegnare un nome** ad un attore *(registrare)*, in modo che chiunque **conosca questo nome** *(atomo)* sia in grado di **comunicare** *(mandare messaggi)* all'attore. Gli attori registrati sono **comuni** al **nodo**. Per effettuare queste operazioni sono presenti le funzioni di libreria:

- `register(nome, Pid)`: registra l'attore dandogli il nome (non è possibile *ri-registrare* un nome)
- `unregister(nome)`: rimuove il nome dall'attore
- `whereis(nome)`: restituisce il `Pid` dell'attore o `undefined`
- `registered()`: restituisce la lista (atomi) degli attori registrati

```
register(nome, Pid).
unregister(nome).
whereis(nome). % Pid | undefined
registered(). % [nome1, nome2, ...]
```

Esempio: orologio

```
-module(clock).
-export([start/2, stop/0]).

start(Time, Fun) ->
  ClockPid = spawn(fun() -> tick(Time, Fun) end),
  register(clock, ClockPid).
```

```
stop() ->
  clock ! stop.

tick(Time, Fun) ->
  receive
    stop -> void
    after Time -> Fun(), tick(Time, Fun)
  end.

% clock:start(5000, fun() -> io:format("TICK ~p~n",[erlang:now()]) end).
% TICK {1699,886068,300544}
% TICK {1699,886073,301701}
% TICK {1699,886078,302892}
% clock:stop().
```

Viene **eseguita** la funzione passata a start (`Fun`) **ogni** `Time` millisecondi. Per **interrompere** l'infinita esecuzione è necessario mandare il messaggio `stop`.
Ma dato che non abbiamo il `Pid` dell'attore, è necessario **registrarlo** assegnandogli il nome `clock`, in modo da poter **mandare** messaggi.