2023-10-30 - Erlang, Moduli, Numeri, Atomi, Tuple, Liste, Stringhe, Variabili, Pattern matching, Funzioni, Comprehensions

Erlang

Erlang (**Er**icsson **Lang**uage) è un linguaggio di programmazione sviluppato dalla *Ericsson* (azienda di telecomunicazioni svedese), creato unendo **caratteristiche** di diversi *linguaggi* (e *paradigmi*) con l'intento di realizzare il miglior linguaggio da usare in campo di **telecomunicazioni**.

Erlang gira su una macchina virtuale (in passato **JAM**, ora **BEAM**), per ottimizzare la gestione di **thread** e **parallelismo** (light-weight process). *I classici linguaggi di programmazione ad alto livello delegano queste operazioni al sistema operativo (operazione molto dispendiosa).*

La *"standard library"* di Erlang è la **Open Telecom Platform** (OTP).

Caratteristiche fondamentali

- orientato alla **concorrenza** *(un **processo** è alla base di ogni computazione)*, adotta il modello ad **attori**:
  - **scambio di messaggi asincrono** (**unico** meccanismo di **comunicazione** tra attori)
  - **non** è presente **memoria condivisa**
- **funzionale**, quindi **stateless** *(caratteristica molto utile in telecomunicazioni)*
- **dinamicamente tipizzato** *(i tipi vengono controllati a runtime)*
- distribuito
- tollerante agli errori
- supporto ad hot-swapping *(aggiornamento del codice senza interrompere l'esecuzione)*

Concorrenza vs Distribuzione vs Parallelismo

Concorrenza: più processi sulla **stessa macchina** concorrono per una risorsa (il tempo di CPU). Viene eseguito un processo alla volta, una piccola parte alla volta *(scheduling)*.

Parallelismo: più processi vengono eseguiti sulla **stessa macchina**, ma su **CPU diverse**, ognuna ha tutto il tempo di CPU dedicato *(meccanismo utilizzato dalle GPU)*.

Distribuzione: più processi lavorano per ottenere un risultato comune ma su **macchine diverse**, ognuna ha le proprie risorse.

Moduli

Un modulo viene dichiarato tramite `-module(nome).`, il nome deve **coincidere** con il nome del file.

Vengono **esposte** dal modulo le funzioni elencate in `-export([nome/2,nome/1]).`, ogni funzione è identificata dal suo **nome** e dal **numero di parametri** (due funzioni con **nome uguale** ma numero di **parametri diverso** sono **diverse**).

Le **funzioni** sono definire per **casi** (**a tratti**), ogni caso è *indipendente* dagli altri. I casi elencati sono separati da `;`, la definizione è terminata con `.`.

```
-module(fact).
-export([fact/1]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).
```

Per importare il modulo nella BEAM shell (lanciata con il comando `erl`) è necessario **compilarlo** attraverso il comando `c(nome)`, che restituisce `{ok, nome}` in caso di successo, oppure l'errore.
Le funzioni esportate dal modulo sono poi raggiungibili attraverso `nome:funzione(parametri).`.

```
$ erl

1> c(fact).
{ok,fact}
```

```
2> fact:fact(7).
5040
```

La shell funziona come un **REPL** (read, eval, print, loop), risponde stampando il valore restituito dall'**ultima** espressione valutata *(prima del punto)*.

```
$ erl

1> A = "hello", B = "world".
"world"
2> A.
"hello"
3> B.
"world"
4> A, B.
"world"
```

Strutture dati

Numeri

La dimensione dei numeri è **dinamica** ed **illimitata**. È possibile definire un numero in una determinata **base** utilizzando la sintassi `base#valore`. È possibile estrarre il valore di un **carattere ASCII** `$carattere.`.

```
10. % 10
16#FF. % 255
-12.34e-2. % 0.1234
$A. % 65
$a. % 97
```

Atomi

Gli atomi sono delle **etichette**, delle **costanti** con un **nome** *(il valore della costante è il nome stesso)*. Servono per **rappresentare qualcosa**, attraverso quella stringa. Un atomo ha un **significato**, ad esempio anche `true` e `false` sono atomi.
Un atomo valido è una stringa composta da **qualsiasi carattere** *(anche newline)*, in caso **non inizi** con una **lettera minuscola** deve essere scritto tra **apici**.

```
cazzola@di.unimi.it. % 'cazzola@di.unimi.it'
'Walter Cazzola'. % 'Walter Cazzola'
'Walter
Cazzola'. % 'Walter\nCazzola'
```

Ciò che non comincia con lettera minuscola (o apici) non è un atomo, ma una variabile.

Tuple

Liste di **dimensione fissata**. Possono contenere un qualsiasi (ma fisso) numero di elementi, di **qualsiasi tipo** ed è possibile **innestare** tuple dentro altre tuple.
È possibile effettuare pattern matching sulla struttura delle tuple.

```
{123, "walter", cazzola}.
{}.
{abc, {'Walter', 'Cazzola'}, 3.14}.
{{1,2},3} == {1,{2,3}}. % false
{{1,2},3} == {{1,2},3}. % true
```

Liste

Liste di dimensione variabile *(ma non mutabili)*, possono essere disomogenee. Possono essere definite attraverso il costruttore `[HEAD|TAIL]` oppure con gli elementi separati da `,`.
Sono presenti le operazioni di concatenazione `++` e sottrazione `--`.

```
[]. % []
[1 | []]. % [1]
[1 | [2]]. % [1,2]
[1,2]. % [1,2]
[{1,2}, ok, []]. % [{1,2}, ok, []]
length([{1,2}, ok, []]). % 3
[{1,2}, ok, []] == [{1,2}, ok, []]. % true
[1,2,3] ++ [3,4,5]. % [1,2,3,3,4,5]
[1,2,3] -- [3,4,5]. % [1,2]
```

Stringhe

Le stringhe non sono altro che delle liste di caratteri, infatti possono essere definite anche come una lista di valori di caratteri ASCII (`$carattere`).
Sono presenti le operazioni di concatenazione e sottrazione esattamente come sulle liste.

```
A=[$W, $a, $l, $t, $e, $r]. % "Walter"
B=[$C, $a, $z, $z, $o, $l, $a]. % "Cazzola"
A ++ " " ++ B. % "Walter Cazzola"
A -- B. % "Wter"
```

"Variabili" e Pattern matching

Le "variabili" *(non sono vere variabili, ma più etichette a valori)* iniziano con una **lettera maiuscola** *(altrimenti sarebbero atomi)*, sono assegnabili **una sola volta**, attraverso **pattern matching**. In caso non sia possibile effettuare un matching viene **lanciato un errore**.
La variabile speciale `_` è una **wildcard** e una variabile anonima *(un "cestino")*.

```
A = 1. % 1
A = 2. % ** exception error: no match of right hand side value 2
[B|L] = [a,b,c]. % [a,b,c]
B. % a
L. % [b,c]
{X, X} = {B, B}. % {a, a}
{Y, Y} = {X, b} % exception error: no match of right hand side value {a,b}
1 = A. % 1
1 = Z. % * 1: variable 'Z' is unbound
{A1, _, [B1|_], {B1}} = {abc, 23, [22|x], {22}}. % {abc,23,[22|x],{22}}
A1. % abc
B1. % 22
```

Funzioni e Funzioni anonime

Le **funzioni** sono definire per **casi** (**a tratti**), ogni caso è *indipendente* dagli altri. I casi elencati sono separati da `;`, la definizione è terminata con `.`.
Ogni caso può anche avere una **sequenza di guardie**. Tutte le guardie devono essere **prive di side-effects**, quindi sono permesse solo alcune espressioni:

- atomi `true`, `false` ed altri atomi costanti
- chiamate ad alcune funzioni built-in *(**NON** tutte)*
- espressioni aritmetiche e booleane (anche cortocircuitate con `andalso` e `orelse`)

```
name(pattern1, pattern2, ... patternn) [when guard] -> body;
name(pattern1, pattern2, ... patternn) [when guard] -> body;
...
name(pattern1, pattern2, ... patternn) [when guard1] -> body.
```

Le **funzioni anonime** vengono definite attraverso la sintassi `fun(parametri) -> body end`.

```
DOUBLE = fun(X) -> X*2 end.
% DOUBLE(4). % 8
% DOUBLE(1). % 2
```

```
IS_EVEN = fun(X) -> X rem 2 == 0 end.
% IS_EVEN(4). % true
% IS_EVEN(1). % false
SUM = fun(X, Y) -> X + Y end.
% SUM(5, 10). % 15
% SUM(0, 0). % 0
```

È anche possibile dare un **nome interno** a funzioni *"anonime"*, in modo da poter essere **ricorsive**.

```
INF_ITER = fun Recur(Cur) -> io:format("~B~n", [Cur]), Recur(Cur+1) end.
```

Le funzioni supportano le **chiusure** (closures).

Map, Filter, Reduce

I pattern per la manipolazione di liste `map`, `filter` e `reduce` sono facilmente definibili, ma sono anche già presenti nel modulo di libreria `lists`.

```
-module(map_filter_reduce).
-export([map/2, filter/2, reduce/2]).

map(_, []) -> [];
map(F, [H|TL]) -> [F(H) | map(F, TL)].

% map_filter_reduce:map(fun(X) -> X*2 end, [2,3,4,5,6]).
% [4,6,8,10,12]

filter(_, []) -> [];
filter(P, [H|TL]) -> filter(P(H), P, H, TL).

filter(true, P, H, L) -> [H|filter(P, L)];
filter(false, P, _, L) -> filter(P, L).

% map_filter_reduce:filter(fun(X) -> X rem 2 == 0 end, [2,3,4,5,6]).
% [2,4,6]

reduce(F, [H|TL]) -> reduce(F, H, TL).

reduce(_, Q, []) -> Q;
reduce(F, Q, [H|TL]) -> reduce(F, F(Q,H), TL).

% map_filter_reduce:reduce(fun(X, ACC) -> ACC + X end, [2,3,4,5,6]).
% 20
```

Comprensione di una lista (List Comprehension)

Una **comprehension** è la definizione di una lista attraverso le **caratteristiche che soddisfa**, *in opposizione alla classica definizione per elencazione.*
La sintassi in Erlang è `[X || qualifier1, ... qualifiern]`, dove `X` è un'**espressione** utilizzabile nei `qualifier` e i `qualifier` sono le **proprietà** che la lista deve rispettare. Ogni **qualifier** è nella forma `pattern <- listexpr`, dove `listexpr` viene valutato ad una lista.

```
[X || X <- lists:seq(2, 10)]. % [2,3,4,5,6,7,8,9,10]
[X || X <- lists:seq(2, 10), (X rem 2) == 0]. % [2,4,6,8,10]
```

Esempio: numeri primi Viene usata una comprehension per lista tutti i numeri **da 2 ad N**, poi vengono *"filtrati"* solo quelli che rispettano `lenght(...) == 0`, che lista tutti i **divisori** del numero attuale fino alla **sua radice**.

```
primes(0) -> [];
primes(1) -> [];
primes(N) -> [X || X <-
```

```
  lists:seq(2,N),
  length([Y || Y <-
    lists:seq(2, trunc(math:sqrt(X))),
    X rem Y == 0
  ]) == 0
].

% primes(20).
% [2,3,5,7,11,13,17,19]
```

Esempio: quick sort Viene usata una comprehension che lista tutti gli elementi **minori del pivot** ed una per tutti quelli **maggiori del pivot**. Vengono poi **concatenate** dopo aver chiamato ricorsivamente `qsort` sulle parti.

```
qsort(_, []) -> [];
qsort(CMP, [PIVOT|TAIL]) ->
  qsort(CMP, [X || X <-
    TAIL,
    CMP(PIVOT, X) < 0
  ]) ++
  [PIVOT] ++
  qsort(CMP, [X || X <-
    TAIL,
    CMP(PIVOT, X) >= 0
  ]).

% qsort(fun(X, Y) -> Y-X end, [10,5,4,1,38,3,9,0]).
% [0,1,3,4,5,9,10,38]
```