Erlang

H_2O

平成 21 年 5 月 7 日

1 出自

- エリクソンの電子交換機プログラムの研究部門
- 最初は Prolog 応用から始まった
- ATM 交換機、DBMS など実用システムに使われている。(関数型プログラムは魅力的だが、履歴依存システムを関数で記述するのは困難と思っていた。それを覆したのだから素晴らしい。)
- 関数型言語
  関数型言語には、Cf. ML, Haskel,,,, と色々あるけれど、実システム開発では Erlang が唯一。
- 動的タイピング言語
  Cf. 静的タイピング言語
- 並列処理
  メッセージを交換しあうプロセスの集まり。　非同期通信。
- CSP とも通ずるモデル
- マルチコアに容易に対応
- 分散処理

2 関数型言語

- 副作用のない関数が基本
- 宣言型
- 単一代入
- Tail recursion Cf. `void foo(int i) { ..... foo(i+1);}` を実行したらスタックはどうなるか?
- 状態の表現
- 関数は副作用を持たない
- 状態依存の処理: Tail recursion で表現。

3　データタイプ

3.1　プリミティブデータタイプ

atom
Integer
Float
Pid　　プロセスの識別子
Ref
fun　　関数

3.2　複合データタイプ

タプル tuples 要素数は不変。要素のタイプは実行時に決まる。

```
{a, 12, 34}
```

リスト list 様相数は可変。要素のタイプは実行時に決まる。

```
[a, 23, hi, lo]
```

レコード（構造体）フィールド名を付けたタプル

```
-record(Name, {FieldName1, ,,,, FieldNameN}.
#Name.FieldName
```

3.3　変数

○変数は Capital letter で始まる

3.4　パターンマッチ

なかなか使いやすい。

```
パターン = 式
Ex.  {A, B} = {12, apple}
     {C, [Head | Tail]} = {{222, man},[a, b, c]}
```

4　シーケンシャル処理

○　Factorial

```
-module(math).
-export([fac/1]).
fac(N) when N > 0 -> N * fac(N-1);
fac(0)            -> 1.

> math:fac(25).
```

○ binary tree lookup

```
lookup(Key, {Key, Val, _, _})  ->  {ok, Val};
lookup(Key, {Key1, Val, S, B}) when Key < Key1  ->  lookup(Key, S);
lookup(Key, {Key1, Val, S, B})  ->  lookup(Key, B);
lookup(Key, nil)  ->  not_found.
```

○リストのアペンド

```
append([H | T], L)  ->  [H | append(T, L)];
append([], L)       ->  L.
```

○ List member

```
member(H, [H | _])  ->  true;
member(H, [_ | T])  ->  member(H, T);
member(_, [])       ->  false.
```

○ Case 式

```
case Expr  of
    Pattern1 [when ガード 1]  -> Seq1;
    Pattern2 [when ガード 2]  -> Seq2;
    ...
end;
```

○ If 式

```
if
   Guard1 -> Sequence1;
   Guard2 -> Sequence2;
   ...
end
```

5 関数オブジェクト

```
fun(Arg1,.., ArgN)  ->  ..... end.
```

○ 関数も First class 値なので、パラメータや返値として記述できる。

○ 定義例

```
K = 2,
F = fun(X) -> X * K end.                %% 変数 F の値は関数
```

○ 定義例

```
adder(C)  ->  fun(X) -> X + C end.        %% addr(C) は関数を返す
> Add10 = adder(10).
#Fun
> Add10(8).
18
```

○ Lisp のマップ機能

```
map(F, [H | T])   ->  [F(H) | map(T)];
map(F, [])        ->  [].

> map(Add10, [1,2,3,4,5]).
[11,12,13,14,15]
```

○リスト内包 (comprehension)

```
[Term || P1, P2,,, Pn]
     where Pi is Pattern <- Exporesion or
                 prediate.
```

○ quick ソート

```
sort([X | Xs])  ->
    sort([Y || Y <- Xs, Y < X])  ++
    [X] ++
    sort([Y || Y <- Xs, Y >= X]);
    sort([])       ->  [].

++: append operator
```

6 コンカレント処理

(1) 基本

○プロセスの生成

```
spawn(echo, loop, [])
```

○サーバの例

```
-module(echo).
-export([start/0, loop/0]).

start()  ->  spawn(echo, loop, []).

loop() ->
   receive
     {From, Message} ->
              From ! Message,
              loop()
   end.

...
Pid = echo:start(),
Pid ! {self(), hello}
....
```

○メッセージ受信: パターンマッチングが行われる

```
 receive
     Message1 ->
     ......;
     Message2 ->
      ......
end
```

(2) 銀行システムの例

```
-module(bank_server).
-export([start/0, server/1]).

start()  -> register(bank_server, spawn(bank_server, server, [[]])).

server(Data)  ->
        receive
           {From, {ask, Who}}  ->
               reply(From, lookup(Who, Data)),
               server(Data);                  %% Tail recursion。状態を引き継ぐ。
           {From, {deposit, Who, Amount}} ->
               reply(From, ok),
               server(deposit(Who, Amount, Data));
           {From, {Withdraw, Who, Amount}} ->
               case  lookup(Who, Data) of
                    undefined ->
                           reply(From, no),
                           server(Data);     %% Tail recursion。状態を引き継ぐ。
                     Balance  when Balance > Amount  ->
                           reply(From, ok),
                           server(deposit(Who, -Amount, Data));  %% Tail recursion。

                     _   ->
                           reply(From, no),
                           server(Data)      %% Tail recursion。状態を引き継ぐ。
               end
           end.

reply(To, X)  -> To ! {bank_server, X}.

lookup(Who, [{Who, Value} | _])   ->  Value;
lookup(Who, [_ | T])              ->  lookup(Who, T);
lookup(_, _)                      ->  undefined.

deposit(Who, X, [{Who, Balance} | T]) -> [{Who, Balance + X} | T];
deposit(Who, X [H | T]) ->   [H | deposit(Who, X, T)];
```

```
deposit(Who, X, [])  ->  [{Who, X}].


-------------------------------------
-module(bank_client).
-export([ask/1, deposit/2, withdraw]).

ask(Who)  ->  rpc({ask, Who}).
deposit(Who, Amount)  ->  rpc({deposit, Who, Amount}).
withdraw(Who, Amount) -> rpc({withdraw, Who, Amount}).

rpc(Msg)  ->
    bank_server ! {self(), Msg},
    receive
         {bank_server, Reply} -> Reply
    end.
```

7 分散処理

○リモートノード上にプロセスを生成し、通信できる。

```
Pid = spawn(Node, Module, Func, ArgList).

bank_server ! {self(), Msg},

{bank_server, 'host@domainname'} ! { self(), Msg}
```

8 mapreduce

Google の map-reduce アルゴリズムも簡潔に記述できる。
マルチコア向き。

【例】

```
-module(mapredudemdl).
-export([mapreduce/4]).

mapreduce(F1, F2, Acc0, l) ->
    S = self(),
    Pid = spawn(fun() -> reducew(S, F1, F2, Acc0, L) end),
    receive
        (Pid, Result) -> Result
    end.

reduce(Parent, F1, F2, Acc0, L) ->
    process_flag(trap_exit, true),
```

```
    ReducePid = self(),
    foreach(fun(X) -> spawn_link(fun() -> do_job(ReducePid, F1, X) end) end, L),
    N = length(L),
    Dict0 = dict:new(),
    Dict1 = collect_replies(N, Dict0),
    Acc = dict:fold(F2, Acc0, Dict1),
    Parent ! { self(), Acc }.

collect_replies(0, Dist) -> Dict;
collect_replies(N, Dict) ->
    receive
        {Key, Val} -> case dict:is_key(Key, Dict) of
                          true -> Dict1 = dict:append(Key, Val, Dict),
                                  collect_replies(N, Dict1);
                          false-> Dict1 = dist:store(Key, [Val], Dict),
                                  collect_replies(N, Dict1)
                      end;
       {'EXIT', _, Why} -> collect_replies(N-1, Dict)
    end.

do_job(ReducePid, F, X) -> F(ReducePid, X).
```

9 こんなことをやりたかった

○○○○○○○○○○○○