❖ Erlang 入門

Erlang はエリクソンが開発したプログラミング言語。特徴は:

- ・並列処理をサポート
- ・分散処理をサポート
- ・Fault Tolerance(耐障害性)が高い

関数型言語という特徴は強調してないみたいだが、まぁ関数型言語と呼んでいいと思う。天下のエリクソンが交換機の開発に使っているくらいなので、見どころはあるに違いない。公式ホームページにある Erlang 講座の資料を中心に、少し勉強してみたのでまとめる。

本書の構成は:

- ❖ データ構造
- ❖ バインドとパターンマッチ
- ❖ プログラム(モジュール、関数、文)と、その実行
- ❖ 関数定義の小技
- ❖ リストの操作
- ❖ 制御構文
- ❖ 並列処理
- ❖ エラーハンドリング
- ❖ 例外処理
- ❖ その他

❖ データ構造

主なデータタイプは 5 種類[1]。

- ・Integer
- ・Float
- ・Atom
- ・Tuple
- ・List

[1] 他にも、プロセス ID、Port、Binary などがある。

具体例を挙げる。

データタイプの例

```
Integer:
8                                   10 進数。
-25
16#DEADBEAF                         #で区切って、16 進数。
2#0101                              2 進数。
$a                                  $を付けると、ascii コード。この例では、16#61 と同じ。

Float:
3.14
-5.33
1.2E-5.

Atom:
atom1                               Atom は Ruby のシンボルみたいなものか。
lower                               小文字で始める。
can_be_long
'Upper'                             大文字で始める場合はシングルクオートで囲む。
'including spaces'                  スペースを含む場合も囲む。

Tuple:
{}                                  Tuple は配列(固定長)。
{1, atom}                           どんなデータ構造でも要素になれる。
{3, {a, b, c}, 4}

List:
[]                                  List はリスト(可変長)。
[1, atom]                           どんなデータ構造でも要素になれる。
[3, {a, b, c}, 4]
"abc"                               ダブルクオートで囲むと ascii コードが要素になる。
                                    この例では、[16#61, 16#62, 16#63]と同じ。
""
```

予約語はリテラルに使えない。

予約語

```
after and andalso band begin bnot bor bsl bsr bxor case catch cond
div end fun inf let not of or orelse query receive rem try when xor
```

リテラルを格納するのが変数。変数の特徴は:

- 変数名は大文字で始める
- 変数にリテラルを代入(格納)することを、バインドと呼ぶ
- 変数にリテラルをバインドできるのは 1 回だけ
- 1 度バインドしたら、変数の値は決して変わらない
- 変数は、バインド済みであれば、Tuple や List の要素に使うことができる

❖ バインドとパターンマッチ[2]

等号(=)は、バインドとパターンマッチを兼ねた演算子。バインドされてない変数に使えばバインドの意味になるし、バインドされた変数に使えばパターンマッチと解釈される。

バインドとパターンマッチの例

```
A = 10                                  変数 A に 10 をバインド。
{B, C, D} = {5, foo, bar}               配列でも OK。変数 B, C, D をバインド。
{A, B} = {10, 5}                        A,B と 10,5 をパターンマッチ。マッチする。
{A, A, D} = {10, 10, foo}               マッチしない。
{E, E, E} = {3, 3, 3}                   E をバインド。
{F, F, G} = {1, 2, 3}                   式自体が失敗。F も G もバインドされない。
{E, F, G} = {3, 2, 1}                   E はマッチ。F と G をバインド。
{H, I} = {1, 2, 3}                      式自体が失敗。要素数が多くても少なくてもダメ。
[H, I] = [1, 2]                         リストでも OK。
[J, K] = [1, 2, 3]                      式自体が失敗。リストでも、要素数が違うとダメ。
_ = 10                                  _は「読み捨てる」の意味。
{J, K, _} = {1, 2, 3}                   J と K にバインド。
L = {1, 2, 3}                           配列自体を L にバインド。
```

リストを使ったバインドの応用例

```
[A, B | C] = [1, 2, 3, 4, 5]            |は、「残り全部」の意味。リスト内でのみ使用可能。
                                        A, B, C が、それぞれ、1, 2, [3, 4, 5]とバインド。
[D, E | _] = [1, 2, 3, 4, 5]            _は、「読み捨てる」の意味。
[F|G] = [abc]                           H が abc, T が[]にバインド。
[H|I] = []                              式が失敗。要素数が少ない。
{H, _, [I | _], {I}} = {abc, {x, y, z}, [3, 2, 1], {3}}
                                        H, I が、それぞれ、abc, 3 にバインド。
```

❖ プログラム(モジュール、関数、文)と、その実行

ソースファイルにモジュールを定義してみよう。モジュールには関数を、関数には文を書く。

プログラムの例(mymod.erl)

```
-module(mymod).                         % モジュール名を宣言。モジュール名は Atom。
                                        % ピリオド(.)が文の終わり。
                                        % パーセント(%)以降はコメント。
-export([apifunc1/0, apifunc2/2]).      % 外部に見せる関数名(と引数の数)を列挙。関数名も Atom。

apifunc1() ->                           % 関数定義開始。
  hello,                                % 式と式の間はカンマ(,)。
  world,
  goodbye.                              % 最後の式が関数の return 値。

apifunc2(A, B) ->                       % 引数は変数なので大文字で開始ね。
  impl2(A, B).                          % 内部関数を呼び出す。

impl2(A, B) ->
  A + B.
```

[2] 普通のプログラマなら、なぜ突然パターンマッチの説明が始まるのか? と違和感を感じるだろう。まぁ、そこが Erlang の特徴なんでしょう。

Erlang シェルを起動して、プログラムをコンパイルし実行する。

mymod.erl の実行

```
maru@genpon ~/erlang $ erl                          シェルを起動。
Erlang (BEAM) emulator version 5.6.5 [source] [async-threads:0]

Eshell V5.6.5  (abort with ^G)
1> c(mymod).                                        シェル関数 c()を呼んで mymod.erl をコンパイル。
{ok,mymod}
2> mymod:apifunc1().                                関数呼び出し。
goodbye
3> mymod:apifunc2(3,4).
7
4> mymod:impl2(3,4).                                公開してない関数は呼べない。
** exception error: undefined function mymod:impl2/2
5> apifunc1().                                      モジュール名を付けないとダメ。
** exception error: undefined shell command apifunc1/0
6> mymod:module_info().                             モジュール mymod に関する情報を表示。
[{exports,[{apifunc1,0},
           {apifunc2,2}]),
 {imports,[]},
 ...]
7> maru@genpon ~/erlang $
```

上記の例で使った c()や q()はシェルコマンドと呼ばれる関数である。例を挙げる。

シェル関数の例

```
h()                                                 ヒストリ(直前の 20 個のコマンド)を表示。
b()                                                 全変数のバインド内容を表示。
f()                                                 全変数のバインドを解除。
f(Var)                                              変数 Var のバインドを解除。
e(12)                                               ヒストリの 12 番のコマンドを再実行。
e(-1)                                               直前のコマンドを再実行。
help()                                              シェルのヘルプを表示。
q()                                                 シェルを終了。
```

シェル関数とは別に、BIF(Build In Functions)と呼ばれる関数群を erlang モジュールが提供している[3]。BIF を呼ぶときは、モジュール名を省略してよい。BIF の例を挙げる。

BIF の例

```
date()
time()
length([1, 2, 3, 4, 5])                            List 用。
size({1, 2, 3})                                    Tuple 用。
atom_to_list(atom)
list_to_tuple([1, 2, 3])
tuple_to_list({})
```

[3] 標準関数ライブラリみたいなもの。

Gentoo で、(USE フラグの doc 付きで)emerge した場合、HTML 形式のマニュアルがインストールされているはず。

Erlang のマニュアル

```
/usr/share/doc/erlang-12.2.5-r1/html/doc/index.html
```

また、コマンドラインからモジュールのヘルプを見ることも可能。

モジュールのヘルプを見る

```
$ erl -man erlang
```

❖ 関数定義の小技

引数の値に応じて関数の挙動を変えたい場合、Erlang ではパターンマッチを使う。

関数引数のパターンマッチング(boolean.erl)

```
-module(boolean).
-export([bool/1]).

bool(true) ->                                   % 引数が true(という Atom)にマッチするなら、本節を評価して終了。
                                                % マッチしなければ次節へ。
  1;                                            % 節と節の間はセミコロン(;)。
bool(false) ->
  hello,
  world,
  0;
bool(Other) ->                                  % 上記のどれにもマッチしなければ、本節を評価。
                                                % Other は変数なので何にでもマッチする(というか、ここでバインド)。
  {invalid_object, Other}.
```

boolean.erl の実行

```
13> c(boolean).
{ok,boolean}
14> boolean:bool(true).
1
15> boolean:bool(false).
0
16> boolean:bool(1).
{invalid_object,1}
17>
```

より細かいパターンマッチを行いたい場合は、「ガード」と呼ばれる条件式を指定することができる。例えば前述の boolean.erl で、bool(Other)を細分化してみよう。

boolean.erl(改)

```
bool(Other) when is_atom(Other) ->          % when の後ろがガード。Other が Atom ならマッチ。
  {invalid_atom, Other};
bool(Other) ->
  {param_is_not_atom, Other}.
```

boolean.erl(改)の実行

```
21> boolean:bool(abc).
{invalid_atom,abc}
22> boolean:bool({}).
{param_is_not_atom,{}}
```

ガードには、true/false を返す BIF(の一部)や比較式などを書くことができる。また、カンマ(,)で区切って複数の条件を書くと論理積として扱われる。セミコロン(;)なら論理和となる。

ガードシーケンス

```
bool(Other) when Other == 1; Other == 2 -> ...;
bool(Other) when is_tuple(Other), size(Other) > 3 -> ...;
...
```

apply()は BIF の 1 つで、任意のモジュールの任意の関数を任意の引数で呼ぶときに使える。これは、ある関数を呼ぶとき、実行時まで引数の数が確定しないようなケースで便利。

apply()の例

```
foo(a_list) ->
  apply(a_module, a_func, a_list).          % モジュール a_module には a_func/2 と a_func/3 が定義
                                            % されているが、a_list の length が 2 か 3 か不明。
```

❖ リストの操作

関数型言語と言えばリスト操作ですな。

listlib.erl

```
-module(listlib).
-export([ave/1, double/1, member/2, ave2/1]).

ave(X) -> sum(X) / len(X).

sum([H|T]) -> H + sum(T);
sum([]) -> 0.

len([_|T]) -> 1 + len(T);
len([]) -> 0.
```

```
double([H|T]) -> [2*H|double(T)];
double([]) -> [].

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

ave2(X) -> ave2(X, 0, 0).                    % ave()よりも高速。

ave2([H|T], Length, Sum) -> ave2(T, Length + 1, Sum + H);
ave2([], Length, Sum) -> Sum / Length.
```

listlib.erl の実行

```
22> c(listlib).
{ok,listlib}
23> listlib:ave([18,34,20]).
24.0
24> listlib:double([18,34,20]).
"$D("                                        ascii 文字列で表示されてしまう。↓ Tuple に変換する。
25> list_to_tuple(listlib:double([18,34,20])).
{36,68,40}
26> listlib:member(20, [18,34,20]).
true
27> listlib:member(2, [18,34,20]).
false
28> listlib:ave2([18,34,20]).
24.0
29>
```

❖ 制御構文

関数型言語に制御構文は無いはず? Erlang にはある。

制御構文

```
case listlib:member(A, X) of
  true -> ...;
  false -> ...
end.

if
  integer(X) -> ...;
  tuple(X) -> ...;
end.
```

❖ 並列処理

Erlang では並列処理の単位をプロセスと呼ぶ。Windows や Linux のプロセスを連想してしまうが、それらに比べると非常に軽いらしい。プロセスを作るには、spawn()にエントリポイントと引数リストを渡す。spawn()はプロセス ID を返す。プロセス ID は Integer ではなく、特殊な型である。自プロセスの ID を得るには self()を使う。

プロセスの生成

```
5> Pid2 = spawn(a_module, a_func, a_arg_list).
<0.40.0>
6> self().
<0.38.0>
7> processes().                          全プロセスのプロセス ID を返す。
[<0.0.0>,<0.2.0>,....]
8> process_inf(self()).                  プロセス情報を返す。
[{current_function,{...}},
 {initial_call,{...}},
...]
9>
```

プロセス間通信のしくみとしては、メッセージ送受信のみが提供されている。メッセージを送信するには、あて先のプロセス ID と送りたいデータを指定する。

プロセス間のメッセージ送受信

```
Pid2!{Pid1, foo}.                        プロセス Pid2 へ Tuple を送信。
Pid2!stop.                               プロセス Pid2 へ Atom を送信。

receive                                  メッセージ受信待ち。
  {From, Msg} ->                         受信したデータがこれにマッチするなら、
    ...,                                 これと
    ...;                                 これを実行して終了。
                                         マッチしないなら、次節へ。
  stop ->                                stop を受信したなら。
    ...                                  これを実行。
end.
```

receive 内のどの条件にもマッチしないメッセージは、mailbox に残る。また特に指定しなければ、receive はマッチするメッセージを受信するまで待ち続ける。タイムアウト時間を指定することもできる。

タイムアウト付きのメッセージ待ち

```
receive                                  メッセージ受信待ち。
  {From, Msg} -> ...;
  stop -> ...
after
  1000 ->                                1000msec たったら、
    ...,                                 これと
    ...                                  これを実行して終了。
end.
```

メッセージをエコーバックするプロセスを書いてみる。

echo.erl

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

go() ->
  Pid2 = spawn(echo, loop, []),
  Pid2!{self(), hello},
  receive
    {Pid2, Msg} ->
      io:format("P1 ~w~n", [Msg])          C の printf()みたいなもの。~w は引数をそのまま表示。~n は改行。
  end,
  Pid2!stop.                               最後の式が go()の return 値。

loop() ->
  receive
    {From, Msg} ->
      From!{self(), Msg},
      loop();                              これでもスタックオーバーフローしないみたい。
                                           再帰呼び出し後の From と Msg は、呼び出し前とは別物。
    stop -> true
  end.
```

echo.erl の実行

```
22> echo:go().
P1 hello
stop
23>
```

register()を使うと、プロセス ID に別名を付けることができる。

プロセスに別名を付ける

```
register(echo_man, Pid2).
echo_man!{self(), goodbye}.
```

プロセス間通信のリアルな例をコードの断片で示す。

テレフォニーの例

```
ringing_a(A, B) ->                         % A から B への呼を抽象化するプロセス上で実行される。
  receive
    {A, on_hook} ->                        % A が電話を切った。
      A!{stop_tone, ring},
      B!terminate,
      idle(A);
    {B, answered} ->                       % B が電話に出た。
      A!{stop_tone, ring},
      switch!{connect, A, B},
      conversation_a(A, B)
  end.
```

クライアント・サーバの例

サーバ:

```
server(State) ->
  receive
    {From, {request, X}} ->
      {R, NextState} = stm(X, State),
      From!{Server, {reply, R}},
      server(NextState)
  end.
```

クライアント:

```
request(Req) ->
  Server!{self(), {request, Req}},
  receive
    {Server, {reply, Rep}} ->
      handle(Rep)
  end.
```

receive のタイムアウト機能の応用例を挙げる。

受信タイマーの応用例

Tmsec 寝る:

```
sleep(T) ->
  receive
  after
    T -> true
  end.
```

サスペンドする:

```
suspend() ->
  receive
  after
    infinity -> true                          % 永遠に待つ。
  end.
```

Tmsec 後にメッセージ What を送る:

```
alarm(T, What) ->
  spawn(timer, set, [self(), T, What]).

set(Pid, T, Msg) ->
  receive
  after
    T -> Pid!Msg
  end.
```

mailbox を空にする:

```
flush() ->
  receive
    Any -> flush()
  after
    0 -> true
  end.
```

❖ エラーハンドリング

Fault Tolerance が高いとうたっている Erlang の、エラーハンドリングの仕組みを見てみよう。基本原理は以下の通り。

- あるプロセスがクラッシュすると、そのプロセスがリンクしている全プロセスへ EXIT シグナルを送る
- EXIT シグナルを受け取ったプロセスは、死んで、そのプロセスがリンクしている全プロセスへ EXIT シグナルを送る
  →ただし、(a)そのプロセスの trap_exit フラグが true、または(b)その EXIT シグナルが後述の exit(normal)により送られたものの場合は除く

このように、EXIT シグナルはリンクしている全プロセスへ伝播(propagate)していく。いくつかのレイヤに分けてプロセス群を構成し、レイヤ間を、EXIT シグナルを捕捉(Trap)するプロセスでリンクさせておけば、あるレイヤでエラーが起きても他のレイヤが影響を受けないようなシステムを実現することができる。

EXIT シグナルハンドリングのための BIF を紹介する。

EXIT シグナルのハンドリング

`link(Pid)`	*現プロセスとプロセス Pid を相互にリンクする。*
`unlink(Pid)`	*現プロセスとプロセス Pid 間のリンクを切る。*
`process_flag(trap_exit, true)`	*現プロセスの trap_exit フラグを true に変える。* *これにより、EXIT シグナルをメッセージとして受信できるようになる。*
`exit(Reason)`	*プロセスを終了し、リンクしている全プロセスへ EXIT シグナルを送る。* *Reason に normal を指定すると、受け取ったプロセスは EXIT シグナルを無視する。*

`receive`
`  {'EXIT', Pid, Reason} -> ...`
`end.`
	trap_exit フラグが true なら、この形式により、EXIT シグナルをメッセージとして受け取ることができる。

EXIT シグナルの応用例を挙げる。クライアントの要求に応じてリソースを確保/解放するサーバを考えてみよう。あるクライアントがリソースを確保したままクラッシュしてしまったら、サーバが、そのリソースを解放するように設計したい。

リソース管理サーバの例

```
main(Free, Allocated) ->                    % サーバプロセスのエントリポイント。Free は未使用リソースのリスト。
                                            Allocated は使用中リソースのリスト。
  receive
    {Pid, alloc} ->                         % リソース確保。ここで Pid をリンクする。
      myalloc(Free, Allocated, Pid);
    {Pid, {release, Res}} ->                % リソース解放。ここで Pid とのリンクを切る。
      unlink(Pid),
      NewAllocated = myfree({Res, Pid}, Allocated),
      main([Res | Free], NewAllocated);
    {'EXIT', Pid, Reason} ->                % Pid が死に、EXIT シグナルが送られてきた。
      Res = mylookup(Pid, Allocated),       % Pid に割り当てたリソースを探し解放する。
      NewAllocated = myfree({Res, Pid}, Allocated),
      main([Res | Free], NewAllocated)
  end.

myalloc([], Allocated, Pid) ->              % リソース確保関数。
  Pid!no_more_res,                          % 未使用リストが空っぽなら失敗。
  main([], Allocated);
myalloc([Res | Free], Allocated, Pid) ->
  Pid!{allocated, Res},                     % 確保できたら、Pid をリンクする。
  link(Pid),
  main(Free, [{Res, Pid}|Allocated]).

myfree(H, [H|T]) ->                         % リソース解放。リンクを切るのは呼び出し側の仕事。
  T;
myfree(X, [H|T]) ->
  [H|myfree(X, T)].

mylookup(Pid, [{Res, Pid}|_]) ->            % Pid が確保したリソースを探す。
  Res;
mylookup(Pid, [_|Allocated]) ->
  mylookup(Pid, Allocated).
```

❖ 例外処理

例外処理の原理は以下の通り。

- throw(ex)により例外 ex を投げる
- catch 付きで呼んだ関数内で投げられた例外は捕捉(catch)することができる
- catch 無しで呼んだ関数内で例外が投げられると、そのプロセスは死に、EXIT シグナル(nocatch)がブロードキャストされる
- catch を付けておけば、EXIT シグナルも捕捉できる

例を示そう。

例外処理の例

```
% mymod.erlの中身。
-module(mymod).
-export([foo/1]).

foo(1) -> hello;
foo(2) -> throw({myerror, a_error});       % 例外{myerror, a_error}を投げる。
foo(3) -> tuple_to_list(s);                % EXITシグナル(badarg)が発生する。
foo(4) -> exit({myexit, another_error}).   % EXITシグナル({myexit, another_error})が発生する。
           % 1～4のどれにもマッチしなければEXITシグナル(function_clause)が発生する。

% mymod.erlを使用。
catch mymod:foo(1).                        % 普通に終了。
catch mymod:foo(2).                        % 例外が捕捉され、{myerror, a_error}が評価される。
catch mymod:foo(3).                        % EXITシグナルが捕捉され、{'EXIT', badarg}が評価される。
catch mymod:foo(4).                        % EXITシグナルが捕捉され、{'EXIT', {myexit,
                                           another_error}}が評価される。
catch mymod:foo(5).                        % EXITシグナルが捕捉され、{'EXIT', function_clause}が評価
                                           される。

mymod:foo(1).                              % 普通に終了。
mymod:foo(2).                              % 例外が捕捉されず、プロセスが死に、EXITシグナル(nocatch)が全
                                           リンクプロセスに送られる。
mymod:foo(3).                              % EXITシグナル(badarg)が全リンクプロセスに送られる。
mymod:foo(4).                              % EXITシグナル({myexit, another_error})が全リンクプロセス
                                           に送られる。
mymod:foo(5).                              % EXITシグナル(function_clause)が全リンクプロセスに送られる。

% 例外の内容に応じて処理を変えるなら、例えばcaseを使う。
f(X) ->
  case catch foo(X) of
    {'EXIT', {myexit, Reason}} -> ...;     % 想定内のプロセス終了。
    {'EXIT', Reason} -> ...;               % 想定外のエラー。
    {myerror, Reason} -> ...;              % 想定内のエラー。
    Success -> ...                         % 正常終了。
  end.
```

❖ その他

ある関数が呼ばれ、その関数が見つからない場合、error_handler モジュールが提供する関数 undefined_call/3 が呼ばれる。

undefine_call()の呼び出し

```
a_mod:a_func(p1, p2, p3).                  % この関数が見つからない場合、↓が呼ばれる。

error_handler:undefined_call(a_mod, a_func, [p1, p2, p3]).
         % この中では、a_modがロードされてなければ、ロードしてapply(a_mod, a_func, [p1, p2, p3])する。
         % a_modがロード済みの場合は、exit({undefined_function, {a_mod, a_func, [p1, p2, p3]}})する。

process_flag(error_handler, myhandlermod). % これにより、error_handler:undefined_call()をフックできる。
```

モジュールのロードについて。

モジュールのロード

```
-module(mymod).
-export([server/0]).

server() ->
  receive
    reload ->
      mymod:server()                    % モジュール指定付きなら、最新コードをロードする。
    Msg ->
      ...,
      server()                          % モジュール指定なしなら、現行コードを使い続ける。
  end.
```

Port は特殊な型で、Erlang から起動したネイティブプロセスを抽象化したものである。

- Port のオープンは、ネイティブプロセスの起動に相当する
- Port は、普通の Erlang プロセスと同様にリンクできるし、クラッシュしたら EXIT シグナルを送信する
- Port と、Port をオープンした Erlang プロセスは、バイトストリームを介してデータを送受信できる(ネイティブプロセスからは stdin/stdout として見える)

Port オープンの例

```
Port = open_port({spawn, Process}, {packet, 2}).
          % ネイティブプロセス Process を起動し、バイトストリームでつなぐ。
          % バイトストリームを流れるパケットは、長さ(ビッグエンディアンの 2 バイト)と、それに続くデータで構成される。
```

Binary は特殊な型で、リニアな(構造を持たない)メモリブロックである。Port へ(Port から)Binary を送受信することも可能。

Binary を操作する BIF の例

```
binary_to_term(Bin)
term_to_binary(Term)
binary_to_list(Bin)
split_binary(Bin, Pos)
concat_binary([Bin1, Bin2])
is_binary(Obj)
```

Reference は特殊な型で、「一意な ID」という用途のために用意されている。

Reference の例

```
Ref1 = make_ref().
Ref2 = make_ref().                      % Ref1 と Ref2 は、(ほぼ)絶対に一致しない。
```

Process Dictionary は、プロセス間でデータ共有するためのしくみ[4]。Process Dictionary は List であり、その要素はキーと値からなる Tuple である(つまりハッシュリスト)。

Process Dictionary を操作する BIF の例

```
get()                    % 全体(Tuple の List)を得る。
get(Key)                 % Key に対応する値を得る。なければ undefined が返る。
put(Key, Value)          % Key の値として Value を保存。旧値か undefined を返す。
erase()                  % 全体を消す。
erase(Key)               % Key とその値のペアを消す。値か undefined を返す。
get_keys(Value)          % 値として Value を持つキーのリストを返す。
```

[4] 要するに巨大なグローバル変数である。なるべく避けるべきであろう。