はじめに

マルチコアのコンピュータが市場に出回っています。もうしばらくすれば、何十、何百のコアをつんだコンピュータが当たり前になるかもしれません。

そのようなマルチコア時代にプログラムに要求されるのは並列処理です。

並列処理と言えば Java 等で使用されているスレッドでしょう。しかし、スレッドを使用したプログラミングは非常に難しいものです。スレッド間の共有データはいつもデッドロックを起こすか、複数のスレッドに同時に書き換えられて壊れているかのどちらかです。複数のスレッドはいつもプログラマの予想を越えた動きし、混沌の神々の恩寵を与えてくれます。デバッグは GC 以前のメモリバグ以上に困難なものです。

設計段階から安全で効率的な並列処理を念頭に作られたプログラミング語が必要なのです。Erlang はそのような言語で、安全で効率的な並列処理を行うためにメッセージパッシング、シングルアサインメント、プロセス間のエラーハンドリング等の様々な工夫がこらされています。

Erlang はマルチコア時代に求められているプログラミング言語なのです。

目次

表目次

図目次

第 1 章

Erlang とは

Erlang とは汎用プログラミング言語であり、その実行環境です。
並列処理、分散処理、耐障害性を備えた言語です。

1.1 歴史

Erlang はスウェーデンのエリクソン社によりテレコミュニケーションシステムのために設計された言語です。テレコミュニケーションの性質上、並列処理、耐障害性等を組み込んだ言語として設計されています。

現在ではオープンソースとして公開されとおり、商用の実装（http://www.erlang.se）とオープンソースの実装（http://www.erlang.org）があります。

Erlang の歴史はテレコミュニケーションシステムに最適な言語の探索から始まりました。

1982 年から 1985 年にかけて、エリクソンの研究所でテレコミュニケーションシステムのために 20 以上の言語が試されました。そこでの結論は Lisp、Prolog、Parlog 等のシンボリックな言語が必要だというものでした。

1985 年から 1986 年にかけて Lisp、Prolog、Parlog が試されましたが、非同期な並列処理と耐障害性が必要であるという結論から、並列処理と耐障害性を組み込んだ独自の言語を開発することにしました。

1987 年にエリクソンの研究所で初期実装の Erlang による実験が行われました。この実装は Prolog で書かれたインタプリタでした。

1988 年に初期実装の Erlang が実業務のプロトタイプ開発に使用され、Erlang が初めて研究所の外で使用されました。

1991 年に一般向けの最初の Erlang の実装がリリースされました。この時点ではまだオープンソースではなく、1998 年にオープンソースとして公開されました。

Erlang は通信トラヒック工学の開祖である数学者のアグナー・アーラン（Agner Krarup Erlang）にちなんで命名されました。Erlang はアーランと読みます。

1.2 特徴

Erlang と特徴を簡単に挙げていきます。

並列処理 OS のプロセスあるいはスレッドとは異なる Erlang 自身が管理するプロセスにより並列処理を行います。Erlang のプロセスは非常に軽量であり、2,000 万個のプロセスを使用したベンチマークが行われたこともあります[*1]。

[*1] http://groups.google.com/group/comp.lang.functional/msg/33b7a62afb727a4f?dmode=source

また Yaws という Erlang で書かれた Web サーバは、Apache が同時接続数約 4,000 で応答しなくなったところ、同時接続数 80,000 以上まで応答できたというベンチマークもあります[*2]。

分散処理 Erlang のプロセス間の通信はメッセージパッシング方式によって行われますが、物理的に別の計算機で動いている Erlang のプロセスとの通信も同じメッセージパッシング方式で行うことができます。既に分散処理の仕組みが組込まれているのです。

耐障害性 シーケンシャルなプログラムの耐障害性では throw、catch の例外処理でも可能ですが、並列処理を想定した場合は十分ではありません。Erlang には 1 つのプロセスで障害が発生した場合、他のプロセスへ障害発生を通知し、後始末をする仕組みが組込まれています。

ソフトリアルタイム プロセス間通信のタイムアウト処理等により、ミリ秒単位の精度のリアルタイム処理を可能とします。

無停止稼動 ホットスワップという稼働中にプログラムを更新する仕組みにより、システムを停止することなくシステムの更新が可能です。

宣言的文法 変数への代入は 1 回のみ可能であり、2 回目の代入はエラーになります。この特徴のために Erlang は関数型言語に分類されることもあります。

パターンマッチング ML や Haskell と同様なパターンマッチングを備えています。

シンボリックプログラミング言語 データタイプのアトムにより、Lisp 等と同様にシンボリックなプログラミング言語になっています。

動的型付け 高い生産性を獲得するために動的型付けが選択されました。

ガーベッジ・コレクション ガーベッジ・コレクションが実装されているため、メモリ管理に気を使う必要はありません。

シンプル Erlang は他の言語と比較すると文法や機能がシンプルになっています。システムダウンを排除するための様々な工夫の結果、言語自体をシンプルにするとうい選択が採られたためです。

1.3 はじめの一歩

Eshell を起動して、少し Erlang を動かしてみます。

Eshell とは Erlang の対話環境で、関数の定義等はできませんが、Erlang のプログラムを対話的に動かすことができます。

Emacs を使用していれば、M-x run-erlang で Eshell が起動します。

TODO Windows ではどうやって起動するんだっけか？

Linux では erl コマンドを実行します。

```
~% erl
Erlang (BEAM) emulator version 5.5.2 [source] [async-threads:0] [hipe]
 [kernel-poll:false]

Eshell V5.5.2  (abort with ^G)
1>
```

プロンプトの 1> に続けて、“.”（ピリオド）で終る式を入力すると、実行結果が表示されます。

例えば、“1 + 2.” と入力し Enter キーを押下すると、その結果 “3” とが表示されます。

[*2] http://www.sics.se/ joe/apachevsyaws.html

```
1> 1 + 2.
3
```

次のように http://www.erlang.org/ にアクセスして、そのページの内容の取得することもできます。

```
2> http:request("http://www.erlang.org/").

=INFO REPORT==== 11-Mar-2007::18:37:16 ===
The inets application was not started. Has now been started as a temporary
    application.
{ok,{{"HTTP/1.1",200,"OK"},
     [{"connection","Keep-Alive"},
      {"date","Sun, 11 Mar 2007 09:37:05 GMT"},
      {"accept-ranges","bytes"},
      {"etag","\"3c3d8bc0ac1cb806843fd142775768d9bd2aff69\""},
      {"server","Apache/1.3.29 (Unix)"},
      {"content-length","9881"},
      {"content-type","text/html"},
      {"last-modified","Fri, 09 Mar 2007 12:27:10 GMT"},
      {"keep-alive","timeout=150, max=100"}],
     "<!-- This file was generated from index.ehtml
    （以下略）
```

help(). でシェルの内部コマンドの一覧が表示されます。

init:restart(). で、Eshell の再起動ができます。また、init:stop(). または q(). で終了します。

```
Erlang (BEAM) emulator version 5.5.3 [source] [async-threads:0] [hipe]

Eshell V5.5.3  (abort with ^G)
1> X = 1.
1
2> init:restart().
ok
3> Erlang (BEAM) emulator version 5.5.3 [source] [async-threads:0] [hipe]

Eshell V5.5.3  (abort with ^G)
1> X = 2.
2
2> init:stop().
ok
3>
Process inferior-erlang<2> finished
```

1.4 並列処理

Erlang のプロセスはネイティブスレッドではなくユーザスレッドに分類できるものですが、-smp オプションを付けて起動するとマルチプロセッサ、あるいはマルチコアに対応します。 /.emacs に次の設定を記述しておきましょう。

```
(setq inferior-erlang-machine-options '("-smp" "auto"))
```

次のソースファイルをコンパイルして、fib_mp:main(40). を実行すると、2 つの CPU の使用率が 100% になるを観察できると思います。

```
-module(fib_mp).
-export([main/1, fib/1]).

main(N) ->
    start(N),                                   % 1つ目のプロセスを起動
    start(N).                                   % 2つ目のプロセスを起動

start(N) ->
    spawn(?MODULE, fib, [N]).                   % プロセスを起動する

fib(1) ->
    1;
fib(2) ->
    1;
fib(N) ->
    fib(N - 1) + fib(N - 2).
```

第 2 章

Erlang の基礎

Erlang の文法を簡単に説明します。

2.1 使用可能文字

ソース中での使用可能文字は ISO-8859-1（Latin-1）となっています。残念ながら日本語の変数名は使用できません。アトムについてはシングルクォートで囲めば日本語も使用できますが、表示時にはエスケープされてしまいます。

2.2 予約語

Erlang には次の 28 個の予約語があります。予約語は関数名等に使用することはできません。

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

2.3 ソースファイル

Erlang は 1 ファイル 1 モジュールです。拡張子には erl を使います。

fib.erl

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

fib(1) ->
    1;
fib(2) ->
    1;
fib(N) ->
    fib(N-1) + fib(N-2).
```

-module でモジュール名を指定します。モジュール名は通常ファイル名と同じにします。

-export でそのモジュールの外から使用できる関数とその関数の引数の個数をリストで指定します。テスト用のモジュール等で、いちいち export に関数名を記述するのが煩雑に思えるときは、-export([fib/1]). のかわり

に-compile(export_all). と書けば、全ての関数をエクスポートすることができます。

拡張子 hrl のファイルにレコード定義や定数定義を書いておいて-include_lib("tables.hrl"). でインクルードできます。

2.4 日本語のあつかい

Erlang は文字列のエンコーディングについては、一切関知しません。そもそも Erlang では文字列は整数のリストです。リストの中に好きなようにコードを並べてください、というスタンスのようです。

そのためソースファイルの文字コードを SJIS、EUC、UTF-8 等にしておけば、それらの文字コードで文字列リテラルを書くことができます。

入出力では、文字列は整数のリストなので、バイナリとしてあつかうことになります。

CEAN で iconv パッケージをインストールすれば、文字コードの変換が可能になります。

2.5 コメント

“%” で始まる行はコメントになります。TEX と同じですね。

2.6 変数

変数は英字の大文字で始まり、英字、数字と “@” が使用できます。ちなみに、英字の小文字で始めた場合、それは変数ではなくアトムになってしまいます。

```
Num
N
Num01
Num@01
```

2.7 データタイプ

Erlang は動的型付けの言語です。ここではどのような型があるのか見ていきたいと思います。

表 2.1 はデータタイプの一覧です。型は 2 つのタイプに分類できます。単純型と複合型です。単純型にはアトム、整数、浮動小数点数、リファレンス、バイナリ、プロセス ID およびポートがあります。複合型にはタプルとリストがあります。

キャラクタと文字列がないのを不思議に思うかもしれません。キャラクタは $a のように書きますが、これは整数の表記方の一種です。また文字列はリストの表記方の一種です。

```
1> $a.
97
2> [$a, $b, $c].
"abc"
```

また、レコードというものもありますが、それもタプルの表記方の一種です。

表 2.1 データタイプ

分類	型	説明
単純型	アトム	hello
	整数	1, 1234567890
	浮動小数点数	3.14, 1.2e+6
	リファレンス	グローバルユニークなもの
	バイナリ	«127, 0, 0, 1», «"wold"»
	プロセス ID（PID）	self() で自分の PID が取得できる
	ポート	外部とのインターフェース
複合型	タプル	this, is ,tuple
	リスト	[this, is, list]

2.8 アトム

アトムは自分自身がその値である変数のようなものです[*1]。

変数名のようの名前を持ちます。名前は英字の小文字から始まるり、英数字、“_”（アンダーバー）、“@”（アットマーク）からなります。また、シングルクォートで囲めば任意の文字を使用できます。

```
1> this_is_@_atom.
this_is_@_atom
2> true.
true
3> 'Hello␣World!\n'.
'Hello␣World!\n'
```

true と false は特別なアトムで真偽値を表すブーリアンです。この true と false といアトムにのみ論理演算子（not, and, or, xor）は作用します。

```
1> false and true.
false
2> false or true.
true
3> not false and true.
true
```

2.9 整数

通常の 10 進数表記の他に、“#”を使用した 2 進数から 36 進数までの表記もあります。

```
0
123
```

[*1] Common Lisp のキーワードシンボルに似ています。

```
+21
-21
2#0101      % 2進数
-8#277      % 負の8進数
16#FAE0     % 16進数
36#1z       % 36進数
```

2.10 浮動小数点数

Erlang には分数がないため、1 を 2 で割ると浮動小数点数になります。

```
1> 1 / 2.
0.500000
2> 1.0.
1.00000
3> -2.1.
-2.10000
4> 1.2e+6.
1.20000e+6
5> 0.987E-20.
9.87000e-21
```

2.11 ポート

ポートとは Erlang と外界との入出力を司るものです。例えば、C で書かれた外部関数を呼んだりするときに使います。また、ファイルの入出力もポートをとおして行っています。

2.12 Pid（プロセス ID）

Erlang の各プロセスの ID です。self で自分の Pid を取得できます。spawn は新しいプロセスを開始し、その Pid を返します。

```
1> self().
<0.64.0>
2> Pid = spawn(file, list_dir, ["/"]).
<0.67.0>
3> Pid.
<0.67.0>
```

2.13 リファレンス

リファレンスは make_ref で作成するユニークなものです。

送信時にリファレンスを一緒に送り、受信時に送ったリファレンスでパターンマッチングを行うことにより、送信と受信の対応付けを確実の行うことができます。

```
1> Ref = make_ref().
#Ref<0.0.0.241>
2> self() ! {Ref, hello}.
{#Ref<0.0.0.241>,hello}
3> receive {Ref, Var} -> Var end.
hello
```

2.14 リスト

リストは [要素 1, 要素 2, ..., 要素 n] のように書きます。空のリストは [] です。リストの要素には任意のものを指定できます。1 つのリストの中に異なる型の要素を混在させることができます。

また、[最初の要素|残りの要素] という書き方もでます。この書き方はパターンマッチングでよく使われます。

length 関数でリストの要素の個数を取得できます。

```
1> [1, "hello", {abc, def}, [a, b]].
[1,"hello",{abc,def},[a,b]]
2> [].
[]
3> [a|[]].
[a]
4> [a|[b|[]]].
[a,b]
5> length([1, "hello", {abc, def}, [a, b]]).
4
```

2.15 タプル

タプルは 要素 1, 要素 2, ..., 要素 n のように書きます。空のタプルは です。タプルの要素には任意のものを指定できます。1 つのリストの中に異なる型の要素を混在させることができます。

size 関数でタプルの要素の個数を取得できます。element 関数で位置を指定してタプルの要素を取り出すことができます。最初の要素の位置は 0 ではなく 1 です。

```
1> Tuple = {1, "hello", {abc, def}, [a, b]}.
{1,"hello",{abc,def},[a,b]}
2> size(Tuple).
4
3> element(1, Tuple).
1
4> element(4, Tuple).
[a,b]
```

```
5>
```

```
start() ->
    Point1 = #point{x = 1, y = 3},
    Point2 = #point{x = 3, y = 2},
    Point3 = add(Point1, Point2),
    io:format("~p~n", [Point3]).

add(#point{x = X1, y = Y1}, #point{x = X2, y = Y2}) ->
    #point{x = X1 + X2, y = Y1 + Y2}.
```

レコードは実際はタプルなので#point{x = 1, y = 3} は{point, 1, 3} と同じです。

2.18 関数

同じ名前で異る引数を持つ関数の定義ができます。また、動的型付けなので関数の返り値も場合により異る型を返すことができます。

引数はパターンマッチングを行うパターンになります。セミコロンで区切って、異なる引数のパターンマッチングを定義できます。ただし、引数の数が異なると別の関数になってしまいます。引数で区切られたそれぞれを function clasuse（関数節）と呼びます。

また、ガード（2.23 参照）により適用する function clasuse を選択することができます。

Erlang のドキュメントで関数を表記する際は、次のようにモジュール名と関数名と引数の数を使います。引数の数が異なると違う関数になるので、引数の数も必要なのです。lists モジュールの引数が 1 つの reverse 関数を意味します。

lists:reverse/1

関数名にはアトムを使用します。なのでシングルクォートで囲めば日本語も使えないことはありません。

```
-module(fun_test1).
-export(['かんすう'/2, test1/0]).

%% ガードシーケンスを使う
'かんすう'(X, Y) when X == 1, Y == 2 ->
    lists:flatten(io_lib:format("X:~p,␣Y:~p", [X, Y]));
%% リストとタプルでのパターンマッチング
'かんすう'([H|T],
```

```
    "H:a,␣T:[b,c],␣Y1:d,␣Y2:e" = 'かんすう'([a, b, c], {d, e}),
    anything_else = 'かんすう'(hello, world).
```

2.19 fun

fun により名前ない関数を作成できます。fun と end 中に通常の関数定義から関数名を除いたものを書きます。引数のよるパターンマッチングや when も通常の関数と同じように動作します。

また fun モジュール名:関数名/引数の個数 で関数を指定することもできます。

```
78> F = fun(X) -> X + 1 end.
#Fun<erl_eval.6.72228031>
79> F(1).
2
80> fun(X) -> X + 1 end(2).
3
81> fun erlang:list_to_tuple/1([1, 2, 3]).
{1,2,3}
82> lists:map(fun erlang:size/1, [{}, {1}, {1,2}]).
[0,1,2]
83> F2 = fun(1) -> one; (2) -> two; (_) -> other end.
#Fun<erl_eval.6.72228031>
84> [F2(1), F2(2), F2(3)].
[one,two,other]
```

2.20 BIF

BIF とは built in function の略で組み込み関数です。ほとんどの組み込み関数は erlang モジュールで定義されています。Eshell で m(erlang). とすると、組み込み関数の一覧が表示されます。

```
1> m(erlang).
Module erlang compiled: No compile time info available
Compiler options:  []
Object file: preloaded
Exports:
'!'/2                           list_to_existing_atom/1
'$erase'/1                      list_to_float/1
'$erase'/0                      list_to_integer/2
'$get'/1                        list_to_integer/1
'$get'/0                        list_to_pid/1
（以下略）
```

2.21 並列処理

Erlang の並列処理の非常に軽量な Erlang 独自のプロセスによって行われます。プロセス間に共有するデータはなく、メッセージの非同期送受信（メッセージパッシング）による、プロセス間で情報の受け渡しを行います。

プロセスを生成するには spawn または spawn_link を使います。

メッセージを送信するには次のように！を使います。

プロセス ID！メッセージ.

送信されたメッセージを受信するには receive を使います。receive の構文は次のようになります。

```
receive
    MessagePatten1 ->
        Action1;
    MessagePatten2 ->
        Action2
after
    TimeOutExpr ->
        TimeOutAction
end.
```

MessagePatten は受信メッセージに対するパターンマッチングです。マッチした場合 Action が実行されます。

after はタイムアウト処理のためにあります。受信すべきメッセージがなく TimeOutExpr（ミリ秒）経過すると TimeOutAction が実行されます。after は省略可能で、省略した場合は永遠にメッセージを待ちつづけます。

次のメッセージ送受信の例は、メッセージを送信するプロセスを作り、2 回メッセージを受信しています。送信したメッセージは 1 つだけなので、2 回目の receive はタイムアウトしています。

```
1> Pid = self().
<0.53.0>
2> spawn(fun() -> Pid ! hello end).
<0.63.0>
3> receive A -> A after 1 -> timeout end.
hello
4> receive A -> A after 1 -> timeout end.
timeout
```

2.22 パターンマッチング

パターンマッチングは、関数呼出し、case、receive、try および = で利用されます。

= によるパターンマッチング

次の “=” の使い方を見てください

```
1> X = 1.
1
2> Y = 1.
1
3> X = 1.
1
4> X = Y.
1
5> X = 2.

=ERROR REPORT==== 18-Mar-2007::15:40:21 ===
Error in process <0.29.0> with exit value: {{badmatch,2},[{erl_eval,expr,3}]}

** exited: {{badmatch,2},[{erl_eval,expr,3}]} **
```

（1>）は X に 1 を代入しています。（2>）は Y に 1 を代入しています。（3>）は X に再び 1 を代入しています。（4>）は X に Y を代入しています。（5>）は X に 2 を代入して、エラーとなりました。

Erlang は単一代入（シングルアサインメント）なので代入は 1 回しかできないはずです。（3>）で X に 2 回目の代入をした時点でエラーになるのでは、と思うかもしれません。しかし、実行してみると（5>）でようやくエラーとなっています。

これは、“=” が行う主な仕事が代入ではなく、むしろパターンマッチングであるためです。“=” はパターンマッチングを行なう際、変数にまだ値が設定されていな場合のみ、代入を行うのです。

（1>）で X と 1 のパターンマッチングを行います。ここで X はまだ値が設定されていまん。変数にまだ値が設定されていないので、X に 1 を代入します。次の（2>）も同様です。（3>）では X と 1 のパターンマッチングですが、すでに X には（1>）で 1 が設定されています。1 と 1 のパターンマッチングなので、パターンマッチングは成功します。（4>）では X と Y のパターンマッチングです。X も Y もともに 1 が設定されているため、パターンマッチングは成功します。最後に（5>）で X と 2 のパターンマッチングを行いますが、X は 2 になっているため、1 と 2 のパターンマッチングになり失敗します。エラーメッセージの “badmatch,2” は、1 が設定されている X は 2 とはマッチしません、という意味なのです。

複雑なパターンマッチング

全く意味のない例ですが、[1, "Hello", [first, "World"], $!] から Hello と World と！を抜き出して表示してみます。

```
1> X = [1, {"Hello", [first, "World"]}, $!].
[1,{"Hello",[first,"World"]},33]
2> [_, {Arg1, [_, Arg2]}, Arg3] = [1, {"Hello", [first, "World"]}, $!].
[1,{"Hello",[first,"World"]},33]
3> io:format("~s␣~s~c~n", [Arg1, Arg2, Arg3]).
Hello World!
ok
```

データ隠蔽できない。データ構造を直に利用する。レコードを使う。

2.23 ガード

ガードとは関数定義、receive、case、if、try 等で使われる条件式です。式の結果が true なら条件に該当、true 以外なら該当しないとみなされます。

コンマ（,）またはセミコロン（;）で区切って複数の式を書くことができます。そのため、ガードシーケンスと呼ぶこともあります。もちろん 1 つの式だけでも有効です。

コンマで区切った場合は、各式が全て true になったときのみそのガード全体が true とみなされます。セミコロンで区切った場合は、いずれかの式が true となればそのガード全体が true とみなされます。

ガードを評価中に副作用が発生することを排除するために、ガードには書ける式は限られています。表 2.2 がガードで書ける式です。

表 2.2 ガードで有効な式

アトムの “true”
アトムの “true” 以外の定数　全て false とみなされます。
表 2.3 の BIF
比較式
算術式
論理式
短絡論理式

表 2.3　ガードで有効な BIF

is_atom/1
is_binary/1
is_constant/1
is_float/1
is_function/1
is_function/2
is_integer/1
is_list/1
is_number/1
is_pid/1
is_port/1
is_reference/1
is_tuple/1
is_record/2
is_record/3
abs(Number)
element(N, Tuple)
float(Term)
hd(List)
length(List)
node()
node(Pid|Ref|Port)
round(Number)
self()
size(Tuple|Binary)
tl(List)
trunc(Number)

```
-module(guard).
-export([start/0]).

-define(P(Exp), io:format("~s␣->␣~p~n", [??Exp, Exp])).

start() ->
    ?P(and_guard(1)),
    ?P(and_guard(5)),
    ?P(or_guard(1)),
    ?P(or_guard([a, b])),
    ?P(or_guard({a, b})).
```

```
and_guard(X) when is_number(X), X < 5 ->
    true;
and_guard(_) ->
    false.

or_guard(X) when is_number(X); is_list(X) ->
    true;
or_guard(_) ->
    false.

%% 実行結果
%% 6> guard:start().
%% and_guard ( 1 ) -> true
%% and_guard ( 5 ) -> false
%% or_guard ( 1 ) -> true
%% or_guard ( [ a , b ] ) -> true
%% or_guard ( { a , b } ) -> false
%% ok
```

2.24 比較演算子

値の大小関係を判定する演算子です。

表 2.4 比較演算子

>	大きい	左辺が右辺より大きければ true を返します。
<	小さい	左辺が右辺より小さければ true を返します。
>=	以上	左辺が右辺と同じか大きければ true を返します。
=<	以下	左辺が右辺と同じか小さければ true を返します。
==	等しい	左辺と右辺が等しければ true を返します。
/=	等しくない	左辺と右辺が等しくなければ true を返します。
=:=	厳密な ==	== と同じですが 1 =:= 1.0 は false です。
=/=	厳密な /=	/= と同じですが 1 =/= 1.0 は true です。

型 (データタイプ) が異なると比較できないと思われるかもしれませんが、異なる型の間では次のような大小関係が成り立ちます。

number < atom < reference < fun < port < pid < tuple < list < binary

2.25 算術演算子

数値計算を行う演算子と、ビット演算を行う演算子です。表 2.5 で引数の型が数値となっているものは整数と浮動小数点数の両方に使用できる演算子です。

表 2.5　算術演算子

演算子	引数の型	説明
+	数値	単項の +
-	数値	単項の -
+	数値	足し算
-	数値	引き算
*	数値	かけ算
/	数値	割り算（結果は浮動小数点数になります）
div	整数	割り算（結果は小数点以下切り捨てで整数になります）
rem	整数	剰余
bont	整数	ビットごとの単項否定演算子
band	整数	ビットごとの論理積
bor	整数	ビットごとの論理和
bxor	整数	ビットごとの排他論理和
bsl	整数	左論理シフト
bsr	整数	右論理シット

2.26　論理演算子

論理演算子です。

表 2.6　論理演算子

演算子	説明
not	単項の論理否定
and	論理積
or	論理和
xor	排他論理和

2.27　演算子の優先順位

演算子の優先順位です。次の表で上のものほど優先順位が高くなります。同じ優先順位のものは結合規則により優先順位が決まります。結合規則が左結合の場合は、左側が優先順位が高くなります。反対に右結合の場合は、右側が優先順位が高くなります。

2.28　制御構文

case, if のみ.

表 2.7 演算子の優先順位

No	演算子	結合規則	説明
1	:		モジュール名と関数名を繋ぐ。lists:reverse
2	#		レコードを表す。FileInfo#file_info.size
3	単項の +, -, bnot, not		単項の算術、ビット、論理演算子
4	/, *, div, rem, band, and	左結合	二項の算術、ビット、論理演算子
5	+, -, bor, bxor, bsl, bsr, or, xor	左結合	二項の算術、ビット、論理演算子
6	++, –	右結合	リストの結合と削除
7	==, /=, =<, <, >=, >, =:=, =/=		比較演算子
8	andalso		短絡の and 論理演算子
9	orelse		短絡の or 論理演算子
10	=, !	右結合	パターンマッチング、メッセージ送信
11	catch		例外補足

2.29 エスケープシーケンス

キャラクタ、文字列およびクォートされたアトムでのエスケープシーケンスには “\” で始まる、1 文字または 3 桁の 8 進数を使用します。

- \b \008 バックスペース (BS)
- \d \177 デリート (DEL)
- \e \033 バックスペース (ESC|
- \f \014 フォームフィード (FF)
- \n \012 ラインフィード (LF)
- \r \015 キャリッジリターン (CR)
- \s \040 スペース (SPC)
- \t \011 水平タブ (HT)
- \v \013 垂直タブ (VT)
- \\ \008 バックスラッシュ
- \’ \047 シングルクォート (’)
- \" \042 ダブルクォート (")
- \^a 〜 \^z, \^A 〜 \^Z コントロールエスケープ (Ctrl+A 等)
- \000 〜 \777 8 進数エスケープ
  10,
  1 等も有効

```
"\"hello\"\n"
"her\bllo"
$\042
```

2.30 リスト内包表記

リスト内包表記（List Comprehensions）とはリストを作成する表記です。構文は次のとおりです。

[式 || パターン <- リスト式, フィルタ]

パターン <- リスト式 は 1 つ以上必要です。フィルタはなくてもよく、あるいは複数指定することもできます。指定した全てのフィルタが true になる組み合わせのリストが作成されます。

```
>1 [N * N || N <- lists:seq(1,10), N rem 2 == 0]. % 1〜10で偶数を二乗した数のリスト。
[4,16,36,64,100]
>2 [[X, Y] || X <- "AB", Y <- "abc"]. % 全ての組み合わせリストを作成
["Aa","Ab","Ac","Ba","Bb","Bc"]
```

2.31 例外・エラー処理

例外・エラー処理として try catch があります。構文は次のとおりです。

```
try Expr of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
catch
    [Class1:]ExceptionPattern1 [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    ...;
    [ClassN:]ExceptionPatternN [when ExceptionGuardSeqN] ->
        ExceptionBodyN
after
    AfterBody
end
```

try の後の Expr がエラー処理の対象となります。try Expr of は case Expr of と同様のパターンマッチングが可能ですが、Body1（BodyNN）で発生した例外はキャッチされません。

catch の Class1（ClassN）は表 2.8 に挙げてある 3 種類があります。Class を省略した場合は、ランタイムエラーと erlang:fault で発生したエラーは補足することができません。全て補足したい場合は _:ExceptionPattern のように書きます。

AfterBody は例外・エラーが発生するしないにかかわらず必ず実行される式です。例えば、ファイルのクローズ処理等を行うのに適しています。

表 2.8　例外・エラークラス

Class	説明
error	1 + a を実行したような場合に発生するランタイムエラー。または、erlang:error/1,2、erlang:fault/1,2 によって発生します。
exit	erlang:exit/1,2 が呼ばれた場合に発生します。
throw	erlang:throw/1 が呼ばれた場合に発生します。

第 3 章

シーケンシャルプログラミング

シーケンシャルプログラミング。

3.1 ファイル入出力

Erlang は外界と通信する時はポートを使用します。ポートを開き、それが 1 つのプロセスであるかのように、！と receive でメッセージの非同期送受信を行います。そのため、Erlang の言語仕様にはファイル入出力といったものはありません。

ファイル入出力も同様にポートを使用して行います。ただし、ポートを使用している部分は、ファイルサーバプロセスが管理しています。通常のプログラムでははは、そのファイルサーバプロセスの API により、ポートは隠蔽されています。

Erlang のファイル入出力はバイナリが基本です。file:read_file というファイルの中身を全て取得する関数がありますが、文字列ではなくバイナリとして取得します。文字列として取得したい場合は binary_to_list で変換する必要があります。

```
-module(file01).
-export([get_file_contents/1]).

get_file_contents(File) ->
    {ok, Binary} = file:read_file(File),
    binary_to_list(Binary).
```

1 行ずつ読み込む関数 io:get_line もあります。それを使って、ファイルの中身を行番号付きで表示してみます。

```
-module(file02).
-export([print_with_line/1]).

print_with_line(File) ->
    {ok, IoDevice} = file:open(File, read),
    print_with_line(IoDevice, 1),
    file:close(IoDevice).

print_with_line(IoDevice, LineNumber) ->
    case io:get_line(IoDevice, "") of
```

```
        eof ->
            ok;
        Line ->
            io:format("~b␣~s", [LineNumber, Line]),
            print_with_line(IoDevice, LineNumber + 1)
    end.
```

今度は表示するのではなく、ファイルの書き出してみます。書き込みファイルをオープンするには file:open の第 2 引数に write を指定します。io:format の第 1 引数に、書き込み先のデバイスを指定します。

```
-module(file03).
-export([print_with_line/2]).

print_with_line(ReadFile, WriteFile) ->
    {ok, In} = file:open(ReadFile, read),
    {ok, Out} = file:open(WriteFile, write),
    print_with_line(In, Out, 1),
    file:close(Out),
    file:close(In).

print_with_line(In, Out, LineNumber) ->
    case io:get_line(In, "") of
        eof ->
            ok;
        Line ->
            io:format(Out, "~b␣~s", [LineNumber, Line]),
            print_with_line(In, Out, LineNumber + 1)
    end.
```

次はファイルのコピー、名前変更、削除のサンプルです。

```
-module(file04).
-export([start/0]).

start() ->
    %% a.txt に Hello と書く
    file:write_file("a.txt", <<"Hello">>),
    io:format("a.txt(~p):␣~p~n",
              [filelib:is_file("a.txt"), file:read_file("a.txt")]),
    %% a.txt を b.txt にコピー
    file:copy("a.txt", "b.txt"),
    io:format("b.txt(~p):␣~p~n",
              [filelib:is_file("b.txt"), file:read_file("b.txt")]),
    %% a.txt を c.txt に名前を変える
```

```
    file:rename("a.txt", "c.txt"),
    io:format("c.txt(~p):␣~p~n",
              [filelib:is_file("c.txt"), file:read_file("c.txt")]),
    %% b.txt を消す
    file:delete("b.txt"),
    %% c.txt を消す
    file:delete("c.txt"),
    io:format("a.txt(~p),␣b.txt(~p),␣c.txt(~p)~n",
              [filelib:is_file("a.txt"),
               filelib:is_file("b.txt"),
               filelib:is_file("c.txt")]).
```

第 4 章

コンカレントプログラミング

この章ではコンカレントプログラミング、並列処理についてみていきます。

4.1 並列処理の仕組み

Erlang の並列処理では、OS のプロセスやスレッドとは異なる、Erlang 独自の「プロセス」を使用します。

「ユーザスレッド」と呼び「プロセス」とは呼ばないのは、Erlang のプロセス間では何も共有されないため、スレッドという呼び名はふさわしくない、との理由のようです。

メッセージパッシングという、任意のメッセージの非同期送受信によって、プロセス間の通信は行われます。

各プロセスは自分用のメールボックスを持っています。あるプロセスが他のプロセスに何かを伝えたい場合は、そのプロセスに！で伝えたいメッセージを送信します。メッセージは送信されると、受信するプロセスのメールボックスに入れられます。メールボックスのメッセージを見るには、receive を使用します。receive ではパターンマッチングを利用して、パターンにマッチしたメッセージのみ受け取ることができます。

フラッシュ

タイムアウト

メールボックス

4.2 プロセスの簡単なサンプル

プロセスを使用した簡単なサンプルです。Server がいます。そこに Client が相方を探しにやってきます。Server は相方がそろったら、それぞれの Client に相方の名前を教えてやります。Client はお互いに挨拶をします。pair:example() を実行すると Alice と Tom がお互いに挨拶をします。

```
-module(pair).
-export([start/0, loop/1, client/2, client_proc/2, example/0]).

start() ->
    spawn(?MODULE, loop, [{}]).

loop({}) ->
    receive
        A ->
            loop({A})
```

```
    end;
loop({A}) ->
    receive
        B ->
            A ! {A, B},
            B ! {B, A},
            loop({})
    end.

client(Server, MyName) ->
    spawn(?MODULE, client_proc, [Server, MyName]).

client_proc(Server, MyName) ->
    Server ! self(),
    receive
        {_Self, Other} ->
            Other ! MyName,
            receive
                YourName ->
                    io:format("Hello␣~s,␣My␣name␣is␣~s~n", [YourName, MyName]
                        )
            end
    end.

example() ->
    Server = pair:start(),
    client(Server, "Alice"),
    client(Server, "Tom").
```

TODO find をシーケンシャルとコンカレントで性能比較してみる。

4.3 プログラムの粒度

Eshell では i(). でプロセスの一覧を見ることができます。Eshell 起動直後に i() を実行すると、既に 23 個のプロセスが動いていることが分かります。

プログラムの構成要素としてオブジェクトよりもプロセスの方が相応しいように思います。プログラムが複雑になるにつれて、オブジェクトでは粒度が小さ過ぎて、より大きな粒度の要素が欲くなります。そこでコンポーネントなどが出ききますが、それらは粒とうより複合物のように感じられます。

構成要素というよりも、むしろ粒度といった語感です。もちろん、数値やリストといったオブジェクトもありますし、関数というまとまった要素もあります。それよりもより上位の要素としてプロセスはとてもしっくりくるように感じられます。

パッケージや名前空間では無意味過ぎます。単なる寄せ集めです。

第 5 章

分散プログラミング

5.1 分散ノード

分散プログラミングでは、複数の Erlang ランタイムがお互いにコミュニケーションをとりながらシステムを構成します。それぞれの Erlang ランタイムは「分散ノード」または単に「ノード」と呼ばれます。

分散ノードはお互いに TCP/IP でのメッセージパッシングによりコミュニケーションをとります。

5.2 マジッククッキー

分散ノード同士がコミュニケーション可能かどうかはマジッククッキーが一致するかどうかで判断されます。

マジッククッキーはホームディレクトリの .erlang.cookie というファイルに格納されています。このファイルがない場合は、ランダムなクッキーでファイルが作成されます。

他のコンピュータのノードと通信するには、 /.erlang.cookie に書かれている内容を同じにしておかなければなりません。

セキュリティ上の理由により /.erlang.cookie はファイルのパーミッションモード 0400（オーナによるリードのみ可能）になっているので、編集時は注意してください。

あらかじめ各コンピュータの /.erlang.cookie の中身を hello に変更しておきましょう。

erlang:get_cookie() でローカルノード（自分自身）のクッキーを取得できます。

ノード毎に違うクッキーを使用したい場合は、set_cookie(Node, Cookie) でノードと使用するクッキーを指定することができます。Node に node() を指定すれば、ローカルノードのクッキーを変更できます。

5.3 分散ノードの開始

Erlang を普通に起動した場合は、非分散ノードとなってしまいます。

分散ノードを開始するには net_kernel:start を使用します。または、-name か -sname オプションでノード名を指定して erl を実行します。

ノード名には通常 “任意の名称@ホスト名” を使います。-name を使用、または net_kernel:start で longnames を指定した場合、「ホスト名」は FQDN（完全修飾ドメイン名）になります。

```
ancient@vmubu:~\% erl -name ubu@172.22.10.22
Erlang (BEAM) emulator version 5.5.4 [source] [async-threads:0] [hipe] [
    kernel-poll:false]
```

```
Eshell V5.5.4  (abort with ^G)
(ubu@172.22.10.22)1>
```

```
1> net_kernel:start(['master@172.22.10.15', longnames]).
{ok,<0.32.0>}
(master@172.22.10.15)2>
```

5.4　ping pong pang

LAN の接続を確認する ping コマンドがあります。それと同様な net_adm:ping 関数でノード間の接続を確認できます。

sheep と bano ノードを起ち上げて ping をうってみましょう。

sheep

```
~% erl -sname sheep
Erlang (BEAM) emulator version 5.5.3 [source] [async-threads:0] [hipe]

Eshell V5.5.3  (abort with ^G)
(sheep@Macintosh)1>
```

bano

```
~% erl -sname bano
Erlang (BEAM) emulator version 5.5.3 [source] [async-threads:0] [hipe]

Eshell V5.5.3  (abort with ^G)
(bano@Macintosh)1> net_adm:ping(sheep@Macintosh).
pong
```

pong が返ってきました。つながっているときは pong が返ってくるのです。では、sheep を終了させてもう一度 ping をうってみましょう。

```
(bano@Macintosh)2> net_adm:ping(sheep@Macintosh).
pang
```

今度は pong ではなく pang が返ってきました。接続できなときは pang なのですね。

nodes() でつながっているノードのリストを取得できます。あらたに tonbu ノードを起ち上げて、nodes() を実行してみます。

```
~% erl -sname tonbu
Erlang (BEAM) emulator version 5.5.3 [source] [async-threads:0] [hipe]

Eshell V5.5.3  (abort with ^G)
(tonbu@Macintosh)1> nodes().
[]
```

```
(tonbu@Macintosh)2> net_adm:ping(sheep@Macintosh).
pong
(tonbu@Macintosh)3> nodes().
[sheep@Macintosh,bano@Macintosh]
```

最初の nodes() ではまだ、どのノードとも繋っていないので、空リストが返ってきます。sheep に ping をした後、再度 nodes() を実行すると、今度は [sheep@Macintosh,bano@Macintosh] sheep だけではなく、bano も返ってきました。一つのノードにつながると、そのノードが構成している分散システム全体のノードと繋がるのです。

5.5 epmd

erl を -name オプション付きで起動すると、epmd デーモン起動されます。epmd は Erlang Port Mapper Daemon で、Erlang の分散システムにおいて、ネームサーバのような機能をはたします。

Linux, Mac OS X なら ps コマンドで epmd が起動されていることを確認することができます。

```
~% ps ax | grep epmd
 5316  ??  S      0:00.27 /opt/local/lib/erlang/erts-5.5.3/bin/epmd -daemon
```

5.6 リモートノードへのモジュールロード

あるノードでモジュールをロードしても、他のノードでロードされなければ、他のノードではそのモジュールを使用することはできません。rpc:call でモジュールをロードしているノードから結果だけもらうことはできますが、実行はモジュールをロードしてるノードなので負荷分散にはなりません。

やはり、リモートノードもモジュールをロードしたいところです。そのために使用する関数が code:get_object_code, code:load_binary, rpc:call です。

code:get_object_code でモジュールのバイナリコードを取得して rpc:call でリモートの code:load_binary をコールします。code:load_binary によってモジュールがロードされます。

```
(cho@Macintosh)37> {Module, Binary, Filename} = code:get_object_code(todo).
{todo
   ,<<70,79,82,49,0,0,16,144,66,69,65,77,65,116,111,109,0,0,3,102,0,0,0,82,4,
        116,111,...>>,
     "/Users/ancient/letter/tex/erlang/sample/mnesia/todo.beam"}
(cho@Macintosh)38> rpc:call('babi@localhost', code, load_binary, [Module,
   Filename, Binary]).
{module,todo}
```

次のような関数で、つながっている全てのノードにモジュールをロードさせることができます。

```
load_nodes(Module) ->
    {_Module, Binary, Filename} = code:get_object_code(Module),
    load_nodes(nodes(), [Module, Filename, Binary]).

load_nodes([Node|Nodes], MFB) ->
```

```
    rpc:call(Node, code, load_binary, MFB),
    load_nodes(Nodes, MFB);
load_nodes([], _) ->
    ok.
```

第 6 章

ネットワークプログラミング

6.1 FTP

Inets の ftp モジュールを使用すれば FTP クライアントの機能を簡単に利用できます。

```
-module(ftp_upload).
-export([upload/6]).

upload(Host, User, Password, RemoteDir, LocalDir, FileList) ->
    %% FTP ホストに接続します。
    {ok, Ftp} = ftp:open(Host),
    %% ログインします。
    ftp:user(Ftp, User, Password),
    %% アップロード先ディレクトリに移動します。
    ftp:cd(Ftp, RemoteDir),
    %% アップロード元ディレクトリに移動します。
    ftp:lcd(Ftp, LocalDir),
    %% 各ファイルをアップロードします。
    lists:foreach(fun(File) ->
                          io:format("uploading␣~s~n", [File]),
                          ftp:send(Ftp, File)
                  end,
                  FileList),
    %% FTP ホストから切断します。
    ftp:close(Ftp).
```

第 7 章

OTP

OTP

第 8 章

Mnesia

8.1 Mnesia の概要

Mnesis は分散データベース管理システム（DBMS）です。テレコミュニケーションシステムで要求される要件、ノンストップかつリアルタイムを満すべく開発された DBMS です。

複数の Erlang ノードでデータのコピーが可能で、データがどのノードにあるかを意識することなくデータアクセスが可能です。

ハッシュテーブルのようにキーと値の保持するのが基本です。

実行中にスキーマの再定義が可能です。

SQL のかわりとなる QLC があります。

8.2 Mnesia を使ってみる

Mnesia は Erlang のディストリビューションに含まれています。

Mensia を使用するには、Mensia がデータベースとして使用するディレクトリを指定して Erlang を起動します。具体的には erl -mnesia dir '"ディレクトリ名"' として起動します。

ディレクトリ名はダブルクォートで囲った文字列として指定する必要があるため、シングルクォートで囲った上で、ダブルクォートで囲った文字列を指定することになります。

```
~% erl -mnesia dir '"/tmp/mnesia"'
Erlang (BEAM) emulator version 5.5.2 [source] [async-threads:0] [hipe] [
    kernel-poll:false]

Eshell V5.5.2  (abort with ^G)
1> mnesia:create_schema([node()]).
ok
2> mnesia:start().
ok
3> mnesia:info().
---> Processes holding locks <---
---> Processes waiting for locks <---
---> Participant transactions <---
---> Coordinator transactions <---
```

```
---> Uncertain transactions <---
---> Active tables <---
schema         : with 1        records occupying 390      words of mem
===> System info in version "4.3.3", debug level = none <===
opt_disc. Directory "/tmp/mnesia" is used.
use fallback at restart = false
running db nodes   = [nonode@nohost]
stopped db nodes   = []
master node tables = []
remote             = []
ram_copies         = []
disc_copies        = [schema]
disc_only_copies   = []
[{nonode@nohost,disc_copies}] = [schema]
2 transactions committed, 0 aborted, 0 restarted, 0 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
ok
```

mnesia:create_schema([node()]). でスキーマを初期化します。mnesia:start(). で Mnesia を開始します。mnesia:info(). で Mnesia の情報を表示します。

Emacs から run-erlnag で起動する場合は、次のように inferior-erlang-machine-options に erl コマンドの引数を指定するように /.emacs に書いておきます。

```
(setq inferior-erlang-machine-options
      '("-smp" "auto" ;; マルチコアの利用
"-mnesia" "dir" "\"/tmp/mnesia\""))
```

テーブルの作成は mnesia:create_table 関数を使用します。レコードまたはリストでテーブルのレイアウト（属性）を指定います。

```
10> mnesia:create_table(bano, [{attributes, [ba, bi, no]}]).
{atomic,ok}
11> mnesia:info().
---> Processes holding locks <---
---> Processes waiting for locks <---
---> Participant transactions <---
---> Coordinator transactions <---
---> Uncertain transactions <---
---> Active tables <---
bano           : with 0        records occupying 288      words of mem
funky          : with 0        records occupying 288      words of mem
schema         : with 3        records occupying 623      words of mem
```

```
===> System info in version "4.3.3", debug level = none <===
opt_disc. Directory "/tmp/mnesia" is used.
use fallback at restart = false
running db nodes   = [nonode@nohost]
stopped db nodes   = []
master node tables = []
remote             = []
ram_copies         = [bano,funky]
disc_copies        = [schema]
disc_only_copies   = []
[{nonode@nohost,disc_copies}] = [schema]
[{nonode@nohost,ram_copies}] = [funky,bano]
4 transactions committed, 1 aborted, 0 restarted, 2 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
ok
12>
```

8.3 テーブルの作成

全て既定値でテーブルを作成するには、次の関数を実行します。

```
mnesia:create_table(table1, []).
```

table1 というテーブル名でテーブルが作成されます。ローカルノードのラムコピーで、属性は key と val です。

レコードを登録するには、transaction の中で write を使います。

```
mnesia:transaction(fun() -> mnesia:write({table1, key1, kal1}) end).
```

8.4 テーブル記憶タイプ

Mnesia ではテーブルの記憶タイプに次の 3 があります。

ram_copies　テーブルのデータはメモリに記憶されます。リード、ライトともに高速です。Mnesia を停止するとデータは消去されます。ただしテーブルのコピーを持っているノードが 1 つでも生きていれば、データは残っています。

disc_copies　テーブルのデータはディスクとメモリの両方に記憶されます。リード時は ram_copies と同様メモリからデータを取り出すので高速ですが、ライト時はディスクに書き込むため低速です。Mensia を停止してもデータはディスクに残り、次回 Mnesia を起動したときに参照可能です。

disc_only_copies　テーブルのデータはディスクのみに記憶されます。そのためリード時も低速ですが、メモリの使用量が少くなります。Mensia を停止してもデータはディスクに残り、次回 Mnesia を起動したときに参照可能です。

8.5 テーブルのコピー

Mnesia は分散データベース管理システムです。テーブルのデータをどの Erlang ノードに保持するかを指定することができます。複数のノードを指定した場合は、それらのノードで同期化されます。データのアクセスはテーブルのデータを保持していないノードからも可能であり、またどとノードに保持されているかを意識する必要もありません。

8.6 データの書き込み

データの書き込みは mnesia:dirty_write または mnesia:write を使用します。

Mensia の関数全般について dirty から始まるものは、トランザクション外で実行される関数で、それらと対になる dirty のついていない関数は実行時にトランザクションを必要とします。

引数にはレコードを指定します。同じキーのレコードがある場合は、上書きされます。

```
(cho@Macintosh)84> mnesia:create_table(table1, []).
{atomic,ok}
(cho@Macintosh)85> mnesia:dirty_write({table1, key1, val1}).
ok
(cho@Macintosh)86> mnesia:transaction(fun() -> mnesia:write({table1, key2,
    val2}) end).
{atomic,ok}
(cho@Macintosh)87> mnesia:transaction(fun() -> mnesia:write({table1, key2,
    val22}) end).
{atomic,ok}
```

8.7 データの読み取り

データの読み取りは mnesia:dirty_read または mnesia:read を使用します。引数にはテーブル名とキーの値のタプルを指定します。

```
(cho@Macintosh)91> mnesia:dirty_read({table1, key1}).
[{table1,key1,val1}]
(cho@Macintosh)92> mnesia:transaction(fun() -> mnesia:read({table1, key2})
    end).
{atomic,[{table1,key2,val22}]}
```

キーでの検索ではなく、もっと複雑な検索を行うときは mnesia:dirty_select または mnesia:select を使います。

8.8 QLC

QLC は Mnesia で利用できる問合せインターフェースモジュールです。Mnesia 専用ではなく ETS（Erlang ビルトインタームストレージ）や Dets（ディスクベースタームストーレジ）にも使用できます。

QLC の問い合せではリスト内包表記を使用します。それを QLCs（Query List Comprehensions）と呼びます。

8.9 耐障害性

あるノードが停止しているとき。停止していたノードが復帰したとき。

第 9 章

Yaws

Yaws は Erlang で書かれた Web サーバです。

9.1 Yaws のインストール

Mac OS X では MacPorts でインストールできます。

```
sudo port install yaws
```

ターミナルを開いて、/opt/local/etc にある yaws.conf.template を yaws.conf とい名前にコピーします。

```
~% cd /opt/local/etc
/opt/local/etc% cp yaws.conf.template yaws.conf
```

これが Yaws の設定ファイルになります。この設定ファイルには 3 つのバーチャルホストが定義されています。ポートは 80 番と 443 番を使う設定になっています。既に Apache 等でそれらのポートを使用している場合は、未使用なポート番号に書換えてください。

```
<server Macintosh.local>
        port = 4080         # 80番から書換えた
        listen = 0.0.0.0
        docroot = /opt/local/var/yaws/www
</server>

<server localhost>
        port = 4080         # 80番から書換えた
        listen = 0.0.0.0
        docroot = /opt/local/tmp
        dir_listings = true
        dav = true
        <auth>
                realm = foobar
                dir = /
                user = foo:bar
                user = baz:bar
```

```
        </auth>
</server>



# And then an ssl server

<server Macintosh.local>
        port = 4443         # 443番から書換えた
        docroot = /opt/local/tmp
        listen = 0.0.0.0
        dir_listings = true
        <ssl>
                keyfile = /opt/local/etc/yaws-key.pem
                certfile = /opt/local/etc/yaws-cert.pem
        </ssl>
</server>
```

また、/opt/local/tmp をドキュメントルートとして使用しているので、/opt/local/tmp がない場合はターミナルから作成してください。

```
/opt/local/etc% mkdir /opt/local/tmp
```

ターミナルから yaws -i を実行すると Yaws が開始します。

```
/opt/local/etc% yaws -i
Erlang (BEAM) emulator version 5.5.3 [source] [async-threads:0] [hipe]

Eshell V5.5.3  (abort with ^G)
1>
=INFO REPORT==== 21-Mar-2007::15:06:04 ===
Yaws: Using config file ./yaws.conf
yaws:Add path "/opt/local/lib/yaws/ebin"
yaws:Add path "/opt/local/lib/yaws/examples/ebin"
yaws:Running with id=default
Running with debug checks turned on (slower server)
Logging to directory "/opt/local/var/log/yaws"

=INFO REPORT==== 21-Mar-2007::15:06:04 ===
Yaws: Listening to 0.0.0.0:4443 for servers
 - https://Macintosh.local:4443 under /opt/local/tmp

=INFO REPORT==== 21-Mar-2007::15:06:04 ===
```

```
Yaws: Listening to 0.0.0.0:4080 for servers
 - http://Macintosh.local:4080 under /opt/local/var/yaws/www
 - http://localhost:4080 under /opt/local/tmp
```

ブラウザで http://Macintosh.local:4080 を開くと Yaws のページが表示されます。右側のリンクから各種ドキュメントやサンプルページにアクセスできます。

また、http://localhost:4080 はベーシック認証のサンプル、https://Macintosh.local:4443 は SSL のサンプルになっています。

9.2 埋め込みモードでの起動

yaws:start_embedded で Yaws を起動できます。引数にはドキュメントルートとなるディレクトリを指定します。

```
1> yaws:start_embedded("/home/ancient/public_html").

=INFO REPORT==== 11-Apr-2007::21:36:48 ===
Yaws: Listening to 127.0.0.1:8000 for servers
 - http://localhost:8000 under /home/ancient/public_html
ok
```

外部からアクセスできるようにするには、listen と servername の指定も必要です。

```
SC = [{port, 8880},                          % ポート番号
      {servername, "www.example.com"},       % サーバ名
      {listen, {0, 0, 0, 0}}],               % リッスンアドレス
GC = [{logdir, "/tmp"}],                     % ログ出力ディレクトリ
yaws:start_embedded("/var/www", SC, GC).     % 組み込みモードでスタート
```

9.3 動的なページの作成

動的なページを作成するにはドキュメントルート以下に拡張子を yaws にしたファイル作成します。

<erl>タグの間に out(Arg) 関数を定義します。out(Arg) は 1 番目の要素が html、2 番目の要素が文字列のタプル {html, String} を返します。表示時に<erl>タグから</erl>タグが返したタプルの 2 番目要素であるの文字列で置き換えられます。

次は現在日時を表示するサンプルです。なお、関数 f は io_lib:format/2 のエイリアスです。

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html;␣charset=UTF-8">
  <title>最初のページ</title>
</head>
<body>
<h1>最初のページ</h1>
<erl>
```

```
out(Arg) ->
  {{Y, M, D}, {H, Mi, S}} = erlang:localtime(),
  {html, f("今は~b年~b月~b日~b時~b分~b秒です。", [Y, M, D, H, Mi, S])}.
</erl>
</body>
</html>
```

out(Arg) は {html, String} を返すかわりに {ehtml, EHTML} を返すこともできます。
EHTML の部分は文字列、バイナリ、{TAG, Attrs, Body}、あるいは EHTML のリストです。
TAG は HTML のタグをアトムで記述します。
Attrs は HTML のタグの属性で [border, 1, bgcolor, grey] のように、属性名とその値のタプルのリストです。
Body は HTML のタグに囲まれている中身です。ここにまたタグを書くこともできます。

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html;␣charset=UTF-8">
  <title>EHTMLの例1</title>
</head>
<body>
<h1>EHTMLの例1</h1>
<erl>
out(Arg) ->
  {{Y, M, D}, {H, Mi, S}} = erlang:localtime(),
  {ehtml, [{pre, [], "テーブルです。"},
           {table, [{border, 1}, {bgcolor, grey}],
             [{tr, [],
                lists:map(fun(X) -> {th, [], X} end,
                          ["年","月","日","時","分","秒"])},
              {tr, [],
                lists:map(fun(X) -> {td, [], integer_to_list(X)} end,
                          [Y, M, D, H, Mi, S])}]}]}.
</erl>
</body>
</html>
```

9.4 セッション

第 10 章

CEAN

CEAN とは Comprehensive Erlang Archive Network の略で TEX の CTAN あるいは Perl の CPAN にあたるものです。

http://cean.process-one.net/ を参照してください。

Erlang のインストール

Erlang のインストールを説明します。

あとがき

あとがきです。