# Erlang基础知识集锦

声明:此文档只作为对erlang的认知之用，如果需要学习并使用erlang请系统学习介绍erlang的书。

## 简介

* Erlang是一个并行编程语言和运行时系统，最初由爱立信（Ericsson）于1987年为电信行业所开发。1998年爱立信在改良的MPL（Mozilla Public License）许可下将Erlang发布于开源社区。
* Erlang是：

1. 一种“小众”语言。
2. 一种函数式语言 （变量只能赋值一次）。
3. 一种动态类型语言（变量类型在运行时决定，代码需要编译后才能执行，与Python，Ruby等不一样）。
4. 一种强类型语言。
5. 一种面向并发（Concurrency Oriented）的语言。

## 特性

* 并发性 - Erlang支持超大量级的并发线程，并且不需要操作系统具有并发机制。
* 分布式 - 一个分布式Erlang系统是多个Erlang节点组成的网络（通常每个处理器被作为一个节点）。
* 健壮性 - Erlang具有多种基本的错误检测能力，它们能够用于构建容错系统。
* 软实时性- Erlang支持可编程的“软”实时系统，使用了递增式垃圾收集技术。
* 热代码升级-Erlang允许程序代码在运行系统中被修改。旧代码能被逐步淘汰而后被新代码替换。在此过渡期间，新旧代码是共存的。
* 递增式代码装载-用户能够控制代码如何被装载的细节。
* 外部接口-Erlang进程与外部世界之间的通讯使用和在Erlang进程之间相同的消息传送机制。

## 数据类型

### 变量

* + 1. 在erlang中的变量只能被赋值一次，变量第一个字母要是大写的。
    2. 符号”=”是值类型匹配操作(带有赋值的意思)。（pattern m！atching）
    3. 在erlang中变量是不可变的，也没有共享内存的概念，也没有锁。

### 浮点数

* + 1. 整数除法示例: 4 div 2。
    2. 求余示例: 5 rem 2。
    3. “/”永远返回浮点数。

### Atoms(原子)

* + 1. Atoms表示不同的非数字常量值。
    2. 示例: Xss = 'Asss'。

### 元组

* + 1. 将若干个以逗号分隔的值用一对大括号括起来就形成了一个元组。
    2. 元组示例: Person = {person, {name, joe}, {height, 1.83}, {footsize, 42}}.
    3. 从元组中提取数据：

Point = {point, 10, 45}.

{point, X, Y} = Point.( pattern match)

此后X=10, Y=45.

### 列表

* + 1. 将若干个以逗号分隔的值用一对方括号括起来，就形成了一个列表。
    2. 示例: ThingsToBuy = [{apple, 10}, {pear, 6}, {milk, 3}].
    3. 列表中的第一个元素叫做列表头，剩下的部分叫做列表尾。一般来说列表头可以是任何东西，列表尾经常是一个列表。
    4. 访问列表头元素是一个非常高效的操作。
    5. 从列表中解析元素:

ThingsToBuy1 = [{oranges, 4}, {newspaper, 1}|ThingsToBuy].

[Buy1|ThingsToBuy2] = ThingsToBuy1.

此后:ThingsToBuy1 = [{oranges,4},{newspaper,1},{apple,10},{pear,6},{milk,3}].

Buy1 = {oranges,4}.

ThingsToBuy2 = [{newspaper,1},{apple,10},{pear,6},{milk,3}].

### 字符串

在erlang中没有字符串，字符串仅仅是整数列表。用双引号将一串字符括起来就是一个字符串。

示例:

Name = "Hello".

5>I = $s.

115

8> [I-32, $u, $r, $p, $r, $i, $s, $e].

"Surprise"

## Erlang基本语法

### 模块

* + 1. Erlang源文件是以.erl做为扩展名的，头文件是以.hrl做为扩展名的。
    2. 要运行一个模块，首先要编译它，编译以后的文件是以.beam做为扩展名的。文件编译命令：c(Modulename).
    3. 改变erlang的工作目录

在操作系统用户文件夹下找到目前登录用户的文件夹在其中建立一个.erlang文件夹，在这个文件中写上:c:cd(＂you work dir＂). 重启erlang shell就改变了工作路径。

* + 1. 在erlang中符号的使用:

逗号(,)用来分隔函数调用，数据构造器(列表元组中的逗号)以及模式中的参数。

句号(.)(后面跟一个空白符号)用来在shell中分隔完整的函数和表达式。

分号(;)用来分隔子句，在以下几种情况下都会用到子句:分段的函数定义,case子句, if语句, try…catch语句， 以及receive表达式。

* + 1. 调用模块中的方法:Modules:ExportFunc(…)

### 匿名函数(fun关键字)

#### 使用示例

1> Hypot = fun(X, Y) -> math:sqrt(X\*X + Y\*Y) end.

#Fun<erl\_eval.12.113037538>

2> Hypot(3, 4).

5.0

* + 1. 匿名函数可以作为函数的参数，并且作为函数的返回值。
    2. 定义自己的抽象控制流程：

for(Max, Max, F) -> [F(Max)];

for(I, Max, F) -> [F(I)|for(I+1, Max, F)].

#### 简单列表处理

* + 1. map函数示例:

map(\_, []) -> [];

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

1. L = [1,2,3,4,5].

[1,2,3,4,5]

map(fun(X) -> 2\*X end, L).

输出:[2,4,6,8].

#### 列表理解

* + 1. 示例:

1> L = [1,2,3,4].

[1,2,3,4]

2> [2\*X || X <- L].

[2,4,6,8]

1> Buy = [{oranges, 4}, {newspaper, 1}, {apples, 10}, {pears, 6}, {milk, 3}].

[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]

2> [{Name, 2\*Number} || {Name, Number} <- Buy].

[{oranges,8},{newspaper,2},{apples,20},{pears,12},{milk,6}]

#### 算数表达式(以及优先级)

参见下图:



#### Guard

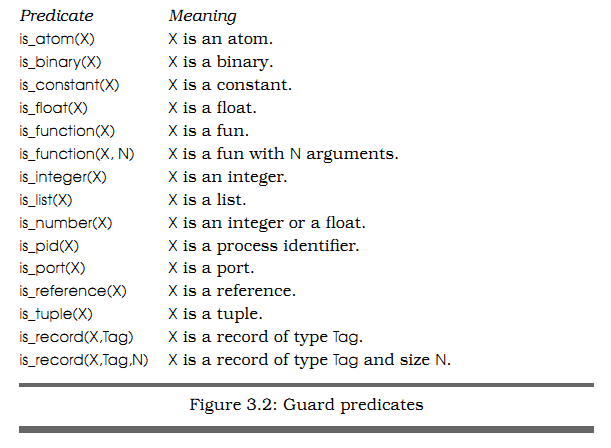
* + 1. Guard是一种强化模式匹配功能的结构，使用Guard可以在一个模式上做一些简单的变量测试和比较。
    2. 示例:

max(X, Y) -> when X > Y -> X;

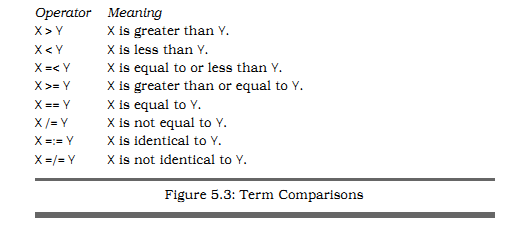
max(X, Y) -> Y.

* + 1. Guard组合关系
* Guard1;Guard2...;Guardn 条件式中每个条件式是or关系。
* Guard1,Guard2...,Guardn 条件式中每个条件式是and关系。
  + 1. 合法的Guard表达式：
* 原子true。
* 其它常量(字典或者列表中的条目或者绑定变量), 这些在断言表达式中都会被求值为false。
* 表下表一中的这些内建函数。
* 比较表达式,参见表二。
* 算数表达式。参见表三。
* 布尔表达式。
* 短路布尔表达式

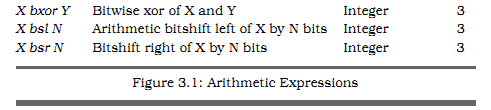
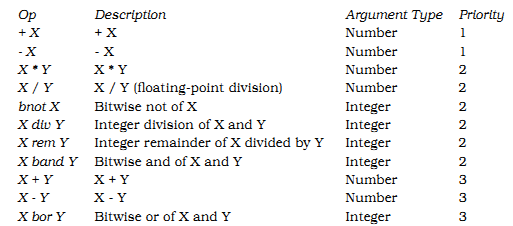
表一：



表二：



表三:



* + 1. Guard例子

f(X, Y) when is\_integer(X), X > Y, Y < 6 ->...

* + 1. true Guard的使用:

true Guard可以在if表达式的末尾处理所有为被匹配的Guard。

if

Guard -> Expression1;

Guard2 -> Expression2;

…

True -> Expressions

end

#### 记录(Records)

* + 1. 使用场景

当元组的元素数量比较大的时候我们可能不知道某个【元素】的具体意义，这个时候就可以使用记录。

* + 1. 申明示例:

-record(todo, {status=reminder,who=joe,text}).

* + 1. 在shell中读取recode的方法:

rr(SomeErlangHeadFile).

* + 1. 使用示例:

1>#todo{}.

#todo{status = reminder, who=joe, text = undefined}

3>X1 = #todo{status = urgent, text = "Fix errata in book"}.

#todo{status = urgent, who = joe, text = "Fix errata in book"}

4>X2 = X1#todo{status = Done}

#todo{status = done, who = joe, text = "Fix errata in book"}

5>#todo{who = W, text, Txt} = X2.

#todo{status = done, who = joe, text = "Fix errata in book"}

6>W.

joe

7>X2#todo.text.

"Fix errata in book"

* + 1. 在函数中对记录进行模式匹配：

clear\_status(#todo{status=S, who=W}) ->

R#todo{status=finished}

* + 1. 记录只是元组的伪装

11>X2.

#todo{status = done, who = joe, text = "Fix errata in book"}

12>rf(todo).

Ok

13>X2.

{todo, done, joe, "Fix errata in book"}

#### case和if表达式

* + 1. case语句的语法

case Expression of

Pattern1 [when Guard1] -> Expr\_seq1;、

Pattern2 [when Guard2] -> Expr\_seq1;

...

End

* + 1. case语句示例:

filter(P, [H|T]) ->

case P(H) of

true -> [H|filter(P, T)];

false -> filter(P, T)

end;

filter(P, []) ->

[].

* + 1. if语句的语法

if

Guard1 ->

Expr\_seq1;

Guard ->

Expr\_seq1;

...

End

### erlang中的异常

* 1. erlang异常的产生

异常经常在系统内部错误或在代码中显式调用throw(Exception), exit(Exception), erlang:error(Exception).

#### 自己生成异常的方法

* exit(Why)

当想要终止当前进程的时候，就需要用到这个函数，如果这个异常未被捕获，那么系统会向所有与当前进程相连接的进程广播{“EXIT”, Pid, Why}消息。

* 这个函数用于抛出一个调用者可能会捕获的异常，在这种情况下，有必要为函数添加注释说明它会抛出这个异常。这个函数的调用者有两种选择，要么忽略这些异常，要么将这个调用包含在try…catch表达式中并对错误进行处理。
* erlang:error(Why)

这个函数用于抛出那些”崩溃错误”。这些异常应该是那些调用者不会真正意识到去处理的致命错误，可以将它等同于内部产生的系统错误。

#### try…catch

* + 1. 语法形式:

try FuncOrExpressionSequence of

Pattern1 [when Guard1] -> Expressions1;

Pattern2 [when Guard2] -> Expressions2;

...

catch

ExceptionType: ExPattern1 [when ExGuard1] ->ExExpressions1;

ExceptionType: ExPattern2 [when ExGuard2] ->ExExpressions2;

...

after %%after部分可以省略

AfterExpressions %%这个地方的返回值将会被忽略

end

在以上语法中ExceptionType是throw, exit, error中的一种假如ExceptionType被忽略则它的默认类型为throw.

* + 1. 捕获所有的错误

try Expr

catch

\_:\_ -> ...Code to handle all exceptions.

End

* + 1. 使用示例:



**(ej@tcim.com)1>** c(try\_test).

{ok,try\_test}

**(ej@tcim.com)2>** try\_test:demo1().

[{1,normal,a},

{2,caught,thrown,a},

{3,caught,exited,a},

{4,normal,{'EXIT',a}},

{5,caught,error,a}]

**(ej@tcim.com)3>** try\_test:demo2().

[{1,a},

{2,a},

{3,{'EXIT',a}},

{4,{'EXIT',a}},

{5,

{'EXIT',{a,[{try\_test,generate\_exception,1},

{try\_test,'-demo2/0-lc$^0/1-0-',1},

{try\_test,'-demo2/0-lc$^0/1-0-',1},

{erl\_eval,do\_apply,5},

{shell,exprs,7},

{shell,eval\_exprs,7},

{shell,eval\_loop,3}]}}}]**(ej@tcim.com)**

### Erlang高级语法

* 1. 二进制数据操作：

示例一:

1> Bin1 = <<1,2,3>>.

<<1,2,3>>

2> Bin2 = <<4,5>>.

<<4,5>>

3> Bin3 = <<6>>.

<<6>>

4> list\_to\_binary([Bin1, 1, [2,3,Bin2],4|Bin3]).

<<1,2,3,1,2,3,4,5,4,6>>

示例二:

1> Red = 2.

2

2> Green = 61.

61

1. Blue = 20.

20

1. Mem = <<Red:5, Green:6, Blue:5>>.

<<23,180>>

5> <<R:5, G:6, B:5>> = Mem.

<<23,180>>

6> R.

2

#### apply关键字

* + 1. 格式

apply(Mod, Func, [Agr1, arg2,...,ArgN])

此方法调用模块Mod中的参数为[Agr1, arg2,...,ArgN]的Func方法。

* + 1. 建议

如果可能的话应避免用此种方法调用其它模块中的方法，当一个函数的参数个数提前知道，更好的调用方式是：M:F(Arg1, Arg2,....ArgN).

* + 1. 使用示例:

1> apply(erlang, atom\_to\_list, [hello]).

"hello"

2> erlang:atom\_to\_list(hello).

"hello"

#### (模块)属性

* + 1. 常见模块属性

-module(modname).

-import(...).

-export(...).

-compile(Options).

-vsn(Version).

最后一个选项为增加编译选项，Options可以是单个选项或者选项列表。

-compile(export\_all)属性经常被使用在调试程序中。

-vsn(Version).中的Version可以是任意字符条目。

* + 1. 自定义属性

格式:

-SomeTag(Value).

(例如:-purpose("example of attributes")

* + 1. 常看模块属性

Module:module\_info().

Module:module\_info(exports).

#### 块表达式(Block Expressions)

* + 1. 语法形式

begin

Expr1,

...

ExprN

end

块表达式用来把一串表达式组织成一个类似子句的实体，块表达式的值为最后一个表达式的返回值。

* 1. 布尔类型

Erlang中没有独立的布尔类型，原子true和false取而代之被赋予了特殊的布尔语义，它们通常作为布尔符号使用。

#### 布尔表达式

* + 1. 4种布尔表达式：

not B1: 逻辑非

B1 and B2: 逻辑与

B1 or B2: 逻辑或

B1 xor B2: 逻辑异或

* + 1. 使用举例

1>not true.

false

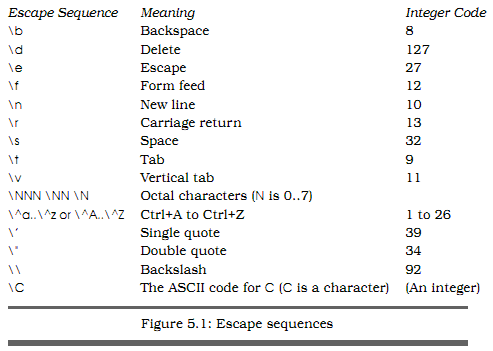
2>true and false.

false

3>true or false.

true

* 1. 转义符
     1. 转义列表



* + 1. 使用举例

1. io:format("~w~n ", ["\b\d\e\f\n\r\s\t\v "]).

[8,127,27,12,10,13,32,9,11,32]

* + 1. 表达式和表达式序列

在erlang中，任何可以被求出值的东西都被称作表达式。这意味着catch, if,try…catch等都是表达式，而类似记录和模块属性等，这些不能被求值的都不是表达式。

表达式序列是一系列由逗号分开的表达式。这些表达式都应该紧接放置于->之后，表达式序列E1,E2,…,En的值被定义为序列中最后一个表达式的值。

#### 包含文件

-include(Filename)

Filename应该包好一个绝对或相对路径以便预处理器能够定位到相应的文件。

-include\_lib(Filename)

这是包含了erlang库中的头文件。

#### 列表操作符++和—

1> [1,2,3]++[4,5].

[1,2,3,4,5]

2> [a,b,c,1,d,e,1,x,y,1]--[1].

[a,b,c,d,e,1,x,y,1]

4> [a,b,c,1,d,e,1,x,y,1]--[1,1].

[a,b,c,d,e,x,y,1]

#### 宏

* + 1. 语法

-define(Constant, Replacement).

-define(Func(Var1, Var2,...,Var), Replacement).

使用

-define(macro1(X, Y), {a, X, Y}).

…

foo(A)

?macro1(A+10, b)

* + 1. 预定义宏

?FILE 展开后为目前文件名

?MODULE 展开后为目前模块名

?LINE 展开后为目前行号

* + 1. 宏的流程控制

-undef(Macro) .取消改宏定义，在这个语句之后不能调用这个宏。

-ifdef(Macro).只有Macro被定义后，才对该行以下的代码进行运算。

-ifndef(Macro).只有在不定义Macro的情况下，才对该行以下的代码进行运算。

-else. 只能在-ifdef或-ifndef之后出现，如果条件为false, 那么改语句后的代码才被执行。

-endif.标记-ifdef或-ifndef语句的结束。

* + 1. 使用举例

-undefine(Macro).

-ifdef(Macro).

-ifndef(Macro).

-else.

-endif.

-module(m1). %%module name is m1

-export([start/0]).

-ifdef(debug).

-define(TRACE(X), io:format("TRACE ~p:~p ~p~n",[?MODULE, ?LINE, X])).

-else.

-define(TRACE(X), void).

-endif.

start() -> loop(5).

loop(0) ->

void;

loop(N) ->

?TRACE(N),

loop(N-1).

编译以上脚本的方法:c(m1,{d, debug}).

运行方法：m1:start().

#### 在模式中使用匹配操作符

在函数参数中将传进来的参数在参数列表中尽量赋给一个临时变量, 这样在接下来的操作中会比较方便，如:

func1([{tag1, A, B}=Z|T])

#### 数值类型

* + 1. 整数
* 传统语法 直接书写10， 50， -7。
* K进制整数 语法K#Digits来表示

1> 7#456114.

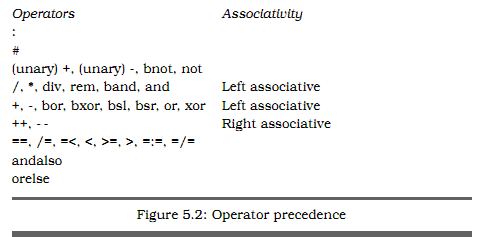
81351

* $语法 语法$C表示ASCII字符C的整数值。
  + 1. 浮点数

1. 3.14 -2.3E+6 23.56e-27

浮点数的范围:-10^323~10^308

#### 操作符优先级



#### 进程字典

Erlang的每个进程都有自己的私有数据存储，叫做进程字典。进程字典是由一系列组成的关联数组,进程字典的操作方法如下：

* @spec put(Key, Value) -> OldValue

增加一个键值到进程字典，如果进程字典中原来就有这个键，那么这个值将会被返回，如果进程字典中原来没有这个键，那么将会返回atom类型的undefined。

* @spec get(Key) ->Value
* @spec get() -> [{Key, Value}...]
* @spec get\_keys(Value) ->[Key].
* @spec erase(Key) ->Value.
* @spec erase() ->[{Key, Value}]. --擦除进程字典中的所有键值

#### 引用

引用是全局唯一的erlang值，使用BIF erlang:make\_ref()来创建引用，引用适用于创建那些唯一标签的场合，例如:在一个BUG跟踪系统中，可以给每一个新的bug报告加入一个引用，以便改记录赋予一个唯一标识。

#### 短路布尔表达式

Expr1 orelse Expr2

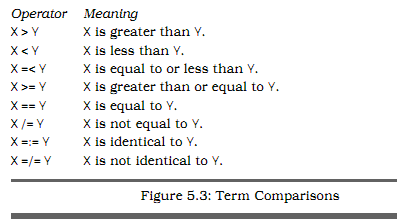
首先求值的是表达式Expr1。如果Expr1为true,那么Expr2就不会被求值，如果Expr1运算结果为false,才会对Expr2进行求值。

#### 比较表达式

* + 1. 为了进行比较，erlang按照下面的方式为所有类型定义了大小比较顺序

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

* + 1. 比较表达式



除了=:=和=/=, 所有的条目比较运算符在比较数字的时候有如下规则：

* 在比较的过程中如果有一个数字为整数另一个为浮点数则在比较前整数会转变为浮点数。
* 如果都是整数或者浮点数则没有数据转换。

示例:

1>15 =:= 15.0 数据类型不转的情况

False

2>15 == 15.0 数据类型转的情况

True

* 1. Underscore Variables(强调变量，下划线变量)

正常情况下，假如一个变量在一个子句(clause)中仅被使用了一次，编译器将会生成一个警告, 因为这经常被认为是一种错误的标志，如果在一个变量前加一个下划线将不会生成这样一个警告。

### 编译和运行erlang程序

* 1. 和运行环境相关的API

以下函数设置erlang程序运行时模块的搜索路径:

@spec code:get\_path()

@spec code:add\_patha(Dir) => true|{error, bad\_directory} %%在模块搜索目录头上增加搜索目录

@spec code:add\_pathz(Dir) => true|{error, bad\_directory} %%在模块搜索目录尾上增加搜索目录

@spec code:all\_loaded()

获取erlang程序运行的起始目录:

init:get\_argument(home).

### 并行编程

#### Erlang中进程特点

* 创建和摧毁进程的速度很快。(2-4ms(微妙),win7 32 bit 操作系统，双核CPU，3.37G内存)
* 进程的行为在所有的操作系统中表现的一样。
* 我们可以有大量的进程。
* 进程之间没有内存共享，它们是完全独立的。
* 进程之间交互的唯一方式是消息传递。

#### Erlang进程的理解:



**(ej@tcim.com)1>** c(processes).

{ok,processes}

**(ej@tcim.com)2>** processes:max(20000).

Maximum allowed processes:32768

Process spawn time=3.1 (3.15) microseconds

ok

修改默认最大进程数的方法:

erl +P 100000 将系统的最大线程数设置为10万。

#### 并发原语

* Pid = spawn(Fun)

建立一个新进程 此进程和调用进程并行工作。

* Pid|Message (Pid1 | Pid2 |...|M)

"|"代表发送操作，消息发送是异步的。(在实际代码中|是!)

#### 设置接收超时值

* + 1. 语法：

receive

Pattern1 [when Guard1] ->

Expressions1;

Pattern2 [when Guard2] ->

Expressions2;

...

after Time -> %%Time为milliseconds毫秒

Expressions

end

after部分只有在消息队列中的所有消息被遍历过之后才会被执行.

after section is checked only after pattern matching has been performed on all entries in mailbox.

#### receive的工作方式

* 当进入一个receive语句时，启动一个计时器(只有在表达式有after部分才计时)。
* 从mailbox（邮箱）中取出第一个消息，然后尝试对Pattern1，Pattern2等进行模式匹配。如果匹配成功，消息就从邮箱中删除，对应的模式之后的表达式就会被求值。
* 如果邮箱中的第一个消息不能匹配receive语句的任何一个模式，那么就会将第一个消息从邮箱中删除并送入一个“保存队列中”，然后继续尝试邮箱中的第二个消息。这个过程会不断的重复直到找到匹配的消息或者邮箱中所有的消息全部被检查了。
* 如果邮箱中所有的消息都不能匹配，那么就挂起进程，直到下一次又有新的消息进入邮箱时在对该进程进行重新调度执行。注意，当一个新消息到达时，只对新消息进行匹配而不会对保存队列中的消息进行再次匹配
* 一个消息如果被匹配，那么存入保存队列中的所有消息就会按照它们到达进程的时间先后顺序重新放回到邮箱中，这时，如果开启了一个计时器，那么计时器将会被清空。
* 如果在我们等待一个消息时触发了计时器，那么先对超时部分中的语句进行求值然后把存入保存队列中的所有消息按照它们达到进程的时间先后顺序重新放回到邮箱中。

#### 超时使用示例:



1. c(stimer).

ok

1. Pid = stimer:start(5000, fun() -> io:format("time event~n") end).

<0.146.0>

time event

1. Pid1 = stimer:start(25000, fun() -> io:format("timer event~n") end).

<0.143.0>

1. stimer:cancel(Pid1).

cancel

#### 注册进程

erlang进程能够发布一个进程的ID以至于任何在系统中的进程可以和他交互。有以下4个方法可以操作注册进程:

* spec register(AnAtom, Pid)

注册一个名字叫做AnAtom，ID是Pid的进程，假如AnAtom已经被注册过了，那么这个注册操作将会失败。

* spec unregister(AnAtom)

移除一个与AnAtom有关的注册。

假如一个进程死掉，它将会自动的取消注册。

* spec whereis(AnAtom) -> Pid | Undefined
* registered() -> [AnAtom::atom()]

返回在系统中注册的所有的进程的列表。

* 1. 从另一个模块中启动进程的方法

spawn(Mod, FuncName, args)

### 并行编程中的错误处理

#### 错误处理的细节

* + 1. 错误处理中的3种基础概念
* 连接(Links): 一个连接定义了一个错误传播路径，假如两个进程连接到一起然后其中的一个死掉，接下来一个退出信号将会被发送到另一个进程，连接到一个进程的其它进程被叫做这个进程的进程集合。
* 退出信号(Exit Signals)：一个进程死掉的消息会发送给这个进程的进程集合。进程可以调用exit(Reason)或者程序内部发生错误将会发送进程退出的消息。除此之外Pid1还可以调用exit(Pid2, X)发送{"EXIT", Pid, X}消息给Pid2, 但是Pid1不会死掉。
* 系统进程(system processes): 可以将其它进程的退出消息转化为普通的消息的进程叫做系统进程。一般进程可以调用内建函数process\_flag(trap\_exit, true)来将自己转化为系统进程。

#### 错误处理原语(Error Handing Primitive)

* @spec spawn\_link(Fun) ->Pid

spawn\_link是原子操作，其不等于spawn之后调用link，因为调用spawn之后的进程可能死掉。

* @spec process\_flag(trap\_exit, true)

将目前的进程转化为系统进程。

也可以用process\_flag(trap\_exit, false) 将系统进程转化为一般进程。

* @spec link(Pid) ->true

假如连接一个不存在的进程，则noproc异常抛出。

* @spec unlink(Pid) ->true

解连接。

* @spec exit(Why) -> none()

这条原语引起目前进程因为Why原因结束。

* @spec exit(Pid, Why) -> true

此条原语发送原因为Why的退出信号。

* @spec erlang:monitor(process, item) -> MonitorRef

此条原语安装了一个监督者，item是PID或者一个进程的注册名。

### 分布式编程

#### 名字服务服务器

参见中文<<erlang程序设计>> p145。

#### 分布式原语

erlang分布式的中心概念是节点，一个节点。相互交流的节点之间cookie要相同。erlang:set\_cookie() 可设置节点的cookie值。

* @spec spawn(Node, Fun) ->Pid

约等于spawn(Fun)

在一个节点上产生一个进程

* @spec spawn(Node, Mod, Func, ArgList) -> Pid

约等于spawn(Mod, Func, Arglist)。

这个方法是原子操作执行过程中不能被打断。

* @spec spawn\_link(Node, Fun) ->Pid

类似:spawn\_link(Fun)。

* @spec spawn\_link(Node, Mod, Func, ArgList) -> Pid

类似于:spawn(Node, Mod, Func, ArgList)。

新创建的进程链接到目前的进程。

* @spec disconnect\_node(Node) -> bool |ignored

强制中断一个节点。

* @spec monitor\_node(Node, Flag) -> true

假如Flag=true，监视被打开，Flag = false, 监视被关闭。

当flag = true时调用此内建函数的进程将会收到{nodeup, Node}，

当某个节点不在此进程的监视集合中的时候{nodedown, Node}将会被收到。

* @spec node() -> Node

此方法返回本地节点的名字，

* @spec node(Arg) -> Node

此函数返回Arg所在的节点，Arg可以是PID, 一个名字引用，或者一个port。

* @spec nodes() -> [Node]

返回一个连接上的所有节点的列表。

* @spec is\_alive() ->bool()

### 接口技术

接口技术就是让erlang与其他语言结合，erlang与其它语言结合的方式也是发送消息机制。

### gen\_server

* 1. 简介

gen\_server实现了一个client\_server模式，在这个模式中的client可以有任意个。这种模式通常被使用在不同的客户端想要共享共同的资源，此中情况下服务端负责管理这些共享的资源。

* 1. 代码



* 1. 使用方法

**(ej@tcim.com)1>** c(my\_bank).

{ok,my\_bank}

**(ej@tcim.com)2>** my\_bank:start().

{ok,<0.79.0>}

**(ej@tcim.com)3>** my\_bank:new\_account("joe").

{welcome,"joe"}

**(ej@tcim.com)4>** my\_bank:deposit("joe", 20).

{thanks,"joe",your\_balance\_is,20}

**(ej@tcim.com)5>** my\_bank:deposit("joe", 50).

{thanks,"joe",your\_balance\_is,70}

**(ej@tcim.com)6>** my\_bank:withdraw("joe", 15).

{thanks,"joe",your\_balance\_is,55}

**(ej@tcim.com)7>** my\_bank:withdraw("joe", 30).

{thanks,"joe",your\_balance\_is,25}

**(ej@tcim.com)8>** my\_bank:stop().

stopped

* 1. my\_bank.erl文件代码分析
     1. –behaviour(gen\_server).(11行)此句的作用是当我们忘记需要实现的回调函数的时候编译器会产生警告或者错误消息。
     2. start().(19行)此句为启动一个本地(回调)服务，将第一个列表参数中的local换为global参数，将启动一个能在由erlang节点组成的集群中全局访问的(回调)服务。此函数经过gen\_server中的事件分发最终会调用my\_bank.erl模块中的init([])(28行)函数。
     3. stop().(20行)此句为停止本地回调服务。此函数经过gen\_server中的事件分发最终会调用my\_bank.erl模块中的handle\_call(stop, \_From, Tab)(57行)函数。
     4. new\_account(…), deposit(…), withdraw(…)为在开户，存款，取款功能。此函数最终会调用my\_bank.erl模块中的handle\_call(…)函数。
     5. handle\_cast(…)(60行)是gen\_server:cast(…)的回调。cast函数是一种没有返回值的调用。(返回的时候的atom：noreply).
     6. Handle\_info(…)用来处理发送给服务器的原生消息。原生消息就是:如果服务器和其他的进程建立了连接，并且正在捕捉退出事件，那么它有可能会突然收到意外的{‘EXIT’, Pid, What}这样的消息。又或者，系统中的其它进程获得了服务器程序的PID，那么它可能会给服务器发送消息。
     7. terminate(…).服务器终止的时候调用的回调函数。
     8. code\_change(…).在热代码升级的时候被回调。

### gen\_event

* 1. 简介

一个事件处理器就是一个命名对象(一个命名进程)，这个命名对象可以接收相应的事件并且处理，在下面的代码中event\_handler.erl就相当于gen\_event.erl.

* 1. 代码



* 1. 使用方法

(ej@tcim.com)1> **event\_handler:make(errors).**

true

(ej@tcim.com)2> **event\_handler:event(errors, hi).**

{event,hi}

fun ret is:void

([ej@tcim.com)3>**motor\_controller:add\_event\_handler()**](mailto:ej@tcim.com)3%3emotor_controller:add_event_handler())**.**

{add,#Fun<motor\_controller.0.130547767>}

(ej@tcim.com)4> **event\_handler:event(errors, hi).**

motor\_controller ignored event: hi

{event,hi}

fun ret is:void

* 1. event\_handler.erl和motor\_control.erl代码分析
* event\_handler.erl分析
* make函数(14行)注册一个名称为Name的进程，进程中执行的函数为my\_handler(…).
* add\_handler(…)(17行)改变my\_handler函数中的参数(函数指针)。
* event(…)(20行)向Name进程发送消息。

* motor\_controller.erl分析
* add\_event\_handler(…)调用event\_handler.erl中的add\_handler函数。
* controller(…)为add\_event\_handler传递给add\_handler的”函数指针”。

### gen\_fsm

* 1. 简介

有限状态机可以被描述为以下的关系:

State(S) x Event(E) -> Action(A), State(S’)

假如我们在状态S，此时发生了一个E事件，接下来应该执行动作A，然后改变状态机的状态至S’.

* 1. 代码



* 1. 使用方法、

**1> c(code\_lock).**

{ok,code\_lock}

**2> code\_lock:start([1,2,3]).**

{ok,<0.36.0>}

**3> code\_lock:button(1).**

Now the code you input is: [1]

ok

**4> code\_lock:button(2).**

Now the code you input is: [1,2]

ok

**5> code\_lock:button(3).**

Now the code you input is: [1,2,3]

ok

**6> code\_lock:button(1).**

Now the code you input is: [1]

ok

**7> code\_lock:button(2).**

Now the code you input is: [1,2]

ok

**8> code\_lock:button(2).**

Now the code you input is: [1,2,2]

ok

Wrong Start

* 1. code\_lock.erl文件解释

参见代码注释。

### 监督树

#### 简介

监督树是监督一个或者多个其它进程的进程，监督进程一般负责启动挂掉的进程。在启动子进程的时候可以设置重启被监督进程的策略..

#### 代码



#### 使用方法

由于指令比较多这里只列出指令:

* erl -boot start\_sasl -config elog2.config(启动erlang)
* sellaprime\_supervisor:start\_in\_shell\_for\_testing().
* area\_server:area({square,10}).
* area\_server:area({rectange,10, 20}).
* area\_server:area({square, 25}).
* prime\_server:new\_prime(20).

说明:

start\_sasl启动标志建立一个适合运行产品级的的系统，(System Architecture Support Library, SASL)将会关注于错误日志，过载保护等等。

注意修改elog2.config中的日志文件路径。

#### sellaprime\_supervisor.erl文件解释

* + 1. start\_in\_shell\_for\_testing(…)开启一个监督进程，并且将调用进程与此监督进程分离。
    2. init([]).(27行)启动两个子监督进程。关于开启子进程中进程列表中每一项的意义参见：erlang程序设计中文版P298, 英文版P353。

### 热代码切换

* 1. 代码



* 1. 使用方法

**(ej@tcim.com)1>** server3:start(name\_server, name\_server1).

**true**

**(ej@tcim.com)2>** name\_server1:add(joe, "at home").

ok

**(ej@tcim.com)3>** name\_server1:add(helen, "at work").

ok

**(ej@tcim.com)4>** server3:swap\_code(name\_server, new\_name\_server).

ack

**(ej@tcim.com)5>** new\_name\_server:all\_names().

[joe,helen]

* 1. 文件解释

略。

### 完整的OTP应用程序

* 1. 代码

在上一节监督树的那几个文件的基础上在加上以下文件：



* 1. 使用方法

切换测试文件所在路径执行命令:erl -boot start\_sasl -config elog2.config,然后执行一下erlang指令:

1> application:loaded\_applications().

[{kernel,"ERTS CXC 138 10","2.13.5"},

{sasl,"SASL CXC 138 11","2.1.9"},

{stdlib,"ERTS CXC 138 10","1.16.5"}]

2> application:load(sellaprime).

ok

3> application:loaded\_applications().

[{kernel,"ERTS CXC 138 10","2.13.5"},

{sasl,"SASL CXC 138 11","2.1.9"},

{stdlib,"ERTS CXC 138 10","1.16.5"},

{sellaprime,"The Prime Number Shop","1.0"}]

4> application:start(sellaprime).

\*\*\* my\_alarm\_handler init:{xyz,{alarm\_handler,[]}}

area\_server starting

prime\_server start\_link

prime\_server starting

ok

5> application:stop(sellaprime).

prime\_server stopping

area\_server stopping

=INFO REPORT==== 1-Feb-2012::10:21:50 ===

application: sellaprime

exited: stopped

type: temporary

ok

6> application:loaded\_applications().

[{kernel,"ERTS CXC 138 10","2.13.5"},

{sasl,"SASL CXC 138 11","2.1.9"},

{stdlib,"ERTS CXC 138 10","1.16.5"},

{sellaprime,"The Prime Number Shop","1.0"}]

7> application:unload(sellaprime).

ok

8> application:loaded\_applications().

[{kernel,"ERTS CXC 138 10","2.13.5"},

{sasl,"SASL CXC 138 11","2.1.9"},

{stdlib,"ERTS CXC 138 10","1.16.5"}]

说明: 第一条指令检查erlang运行系统中正在运行的应用程序。

第二条指令加载了应用程序，加载动作会载入所有的代码，但不会启动应用程序。

第四条指令启动了应用程序。

第五条指令将这个应用程序停止。

第七条指令卸载sellaprime应用程序。

* 1. Sellaprime.app文件解释

此文件包含了应用程序的说明，已经此应用程序要加载的模块等应用程序加载相关的信息。