**Erlang学习博文：**

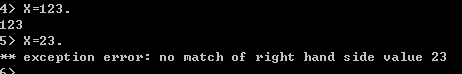
[**http://www.cnblogs.com/me-sa/category/304370.html**](http://www.cnblogs.com/me-sa/category/304370.html)

**erlang OPT的基础模块proc\_lib**

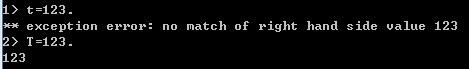
opt的behaviour都是使用proc\_lib实现创建新进程，我们也可以直接使用这个模块来创建符合OPT原则的进程

**1 Erl表达式都必须以一个句号后面跟一个空白字符结尾（空白字符指空格、Tab或者回车符）**

**2 erl中的=操作符不能重新绑定，它不是一个赋值操作符，而是一个模式匹配操作符**



**3 erl中的变量是以大写字母开头，小写字母开口的是原子**



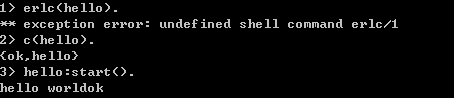
**4 erl中的hello world代码采用erlc编译器编译**

-module(hello).

-export([start/0]).

start() ->

io:format("hello world").



**5 并发程序**

erl的基本并发单元是进程（process）。一个进程是一个轻量级的虚拟机，只能通过发送和接受消息来与其他进程通信。进程与模块类似于对象和类

**文件服务器进程：**

-module(afile\_server).

-export([start/1, loop/1]).

start(Dir) ->

spawn(afile\_server, loop, [Dir]).

loop(Dir) ->

receive

{Client, list\_dir} ->

Client ! {self(), file:list\_dir(Dir)};

{Client, {get\_file, File}} ->

Full = filename:join(Dir, File),

Client ! {self(), file:read\_file(Full)}

end,

loop(Dir).

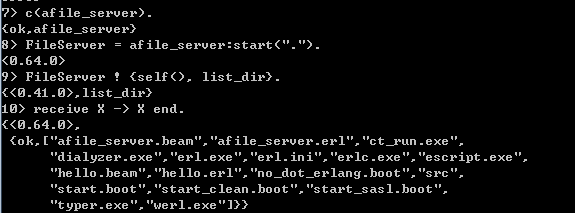
这就是erl编写无限循环的方法。变量Dir包含了文件服务器当前的工作目录。我们在这个循环内等待指令，接收到指令时我们会等待，然后再次调用自身来获取下一个指令，不必担心最后的自身调用，这不会耗尽栈空间。Erlang采用了一种所谓“尾部调用”的优化，意思是此函数的运行空间的固定的。

另外，loop函数永远不会返回，在顺序编程语言里，必须要极其小心避免无限循环，因为只有一条控制线，如果这条线卡在循环里就有麻烦了。Erlang则没有这个问题，服务器知识一个在无限循环里处理请求的程序，与我们想要执行的其他任务并行运行

中间的这段receive…end的代码表示：

如果接受到{Client，list\_dir}消息，就应该回复一个文件列表；如果接受到{Client，{get\_file,File}}，则回复这个文件。作为模式匹配过程的一部分，Client变量在收到消息时会被绑定

Spawn是erlang的一个基本函数，它会创建一个新进程，当进程创建完毕之后，它便开始执行参数所指定的代码



（1）FileServer=afile\_server:start(“.”).

这样就创建出了一个新的并行进程来执行函数afile\_server:loop(Dir)，并返回一个进程标识符，可以用它来与此进程进行通信

（2）FileServer ! {self(), list\_dir}

这里给服务提进行发送了一条{self(), list\_dir}消息，Pid ! Message的返回值被规定为Message，因此结果为{self(), list\_dir}的值

（3）receive X -> X end

这里接受文件服务器发送的回复

**文件客户端进程**

-module(afile\_client).

-export([ls/1, get\_file/2]).

ls(Server) ->

Server ! {self(), list\_dir},

receive

{Server, FileList} ->

FileList

end.

get\_file(Server, File) ->

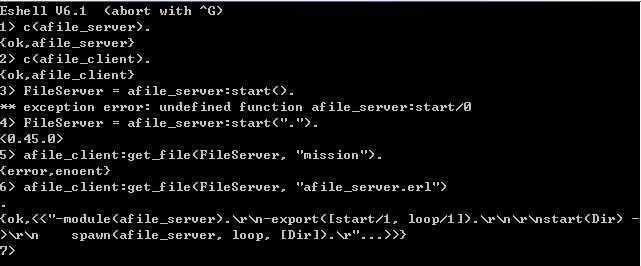
Server ! {self(), {get\_file, File}},

receive

{Server, Content} ->

Content

end.

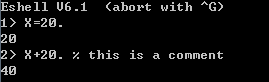


**6 erl终止命令**

q()以一种受控的方式停止系统。所有打开的文件都被刷入缓存并关闭，数据库（如果被打开的话）会被停止，所有的应用程序都以有序的方式关闭。q()是init:stop()命令在shell里的别名

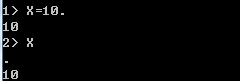
halt()为立即停止系统

**7 erlang基础**

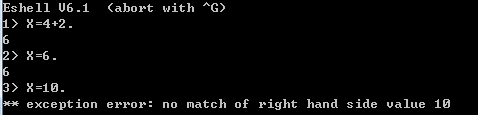


%代表一段注释的起点。从百分号到行尾的所有文字都被当做注释，shell和erlang编译器会忽略它们

Erl中的所有变量值都必须以大写字母开头，如果想知道一个变量的值，只需要输入变量名即可。Erl的变量是一次性变量，即只能被赋值一次，已被指派的一个值的变量称为绑定变量，否则称为未绑定变量。如果X出现在许多不同的函数里，那么所有这些X的值都是不相干的

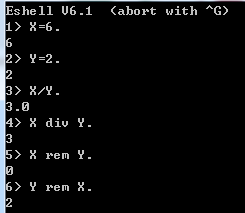


注意这里是模式匹配的函数，而不是赋值操作符，在第一次的时候会相当于直接赋值，在后面就会相当于做匹配



**浮点数的运算**

使用/使用得到的结果数浮点数，使用div才会得到整数结果



**原子**

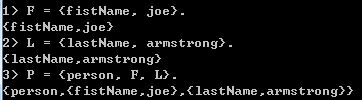
Erl中，原子被用于表示常量值，原子以小写字母开头，原子是全局性的，而且不需要宏定义或者包含文件就能实现，例如red、december等。

同时，原子还可以放在单引号’’内，可以用这种方式创建以大写字母开头（否则会被解释为变量）或包含字母数字以为字符的原子，例如’Monday’，’Tuesday’等，在erl中，单引号和双引号是不能互换的。如’a’和a的意思完全一致，单引号的用法如前面，双引号用于给字符串字面量定界

一个原子的值就是它本身，erl是一种函数式编程语言，每个表达式都必须有一个值

**元组**

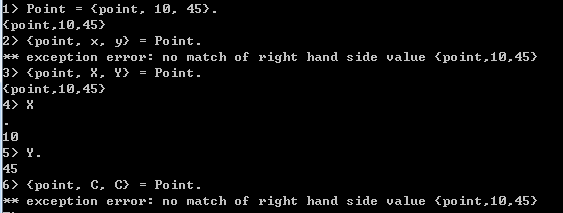
如果想把一些数量固定的项目归组为单一的实体，就会使用元祖，创建元组的方法是用大括号把想要表示的值括起来，并用逗号分隔它们，元组是匿名的，元组会在声明它们的时候自动创建，不再使用时候则被销毁。Erlang使用一个垃圾器来回收所有未使用的内存，这样就不必担心内存分配的问题了。如果在构建新元组时用到变量，那么新的元组会共享该变量所引用数据结构的值。



如果试图用为定义的变量创建数据结构，则会得到一个错误结果

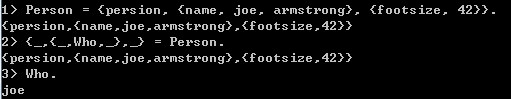


提取元组的值，需要使用到模式匹配操作符=



在上面的粒子中，首先不能写成小写字母，原子不能改变，然后是C，C表示相同的变量，不能被赋不同的值

提取复杂元组，同样的方法。

，

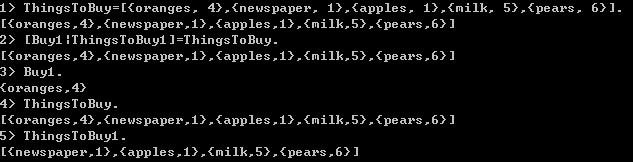
注意，在erlang中，\_表示占位符，用于表示不感兴趣的那些变量，称为匿名变量，与正规变量不同，同一模式中多个\_不必绑定相同的值

**列表**

列表被用来存放任意数量的事物。创建列表的方法是用中括号把列表元素括起来，并用逗号分隔它们

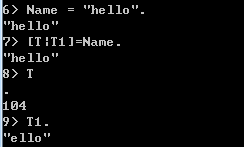
列表的第一个元素称为列表头，假设把头去掉，剩下的就称为列表尾，注意，列表头可以是任何事物，但列表尾通常仍然是一个列表。访问列表头是一种非常高效的操作，因此基本上所有的列表处理函数都从提取列表头开始，然后对它做一些操作，接着处理列表尾

提取列表元素，采用模式匹配

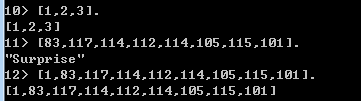


**字符串**

严格来说，erlang中没有字符串。要在erlang中表示字符串，可以选择一个由整数组成的列表或者一个二进制型。当字符串表示为一个整数列表时，列表里的每个元素都代表了一个Unicode代码点，可以用字符串面量来创建这样一个列表，其实就是用双引号



注意，当shell打印某个列表的时候，如果列表内的所有整数都代表可打印字符，它就会将它打印为字符串字面量，否则，打印成列表记法

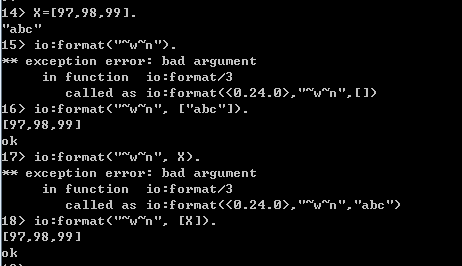


在上面的例子中，1,2,3不能转换为字符码，因此打印成列表，在第二组中所有的整数都有对应的字符，因此打印成字符串

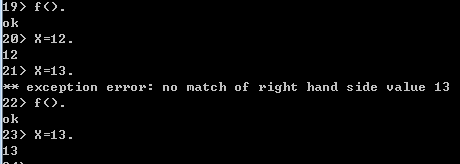
如果需要知道某个字符的整数值，可以使用$符号



如果shell将某个整数列表打印成字符串，但我们希望打印成整数，就必须采用格式化语句进行转化：

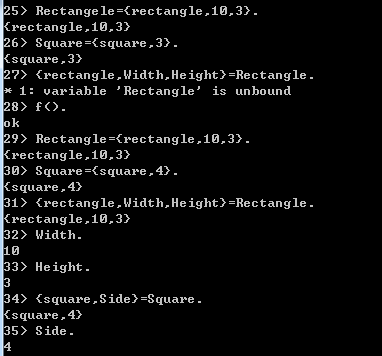


一个很有用的指令f()，让shell忘记现有的任何绑定，即所有变量都变为未绑定状态



**模块**

模块保存在扩展名为.erl的文件里，而且必须先编译才能运行模块里的代码。编译后的模块以.beam作为扩展名

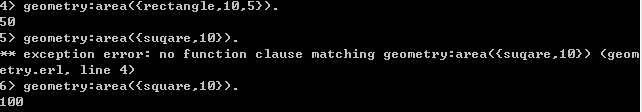


-module(geometry).

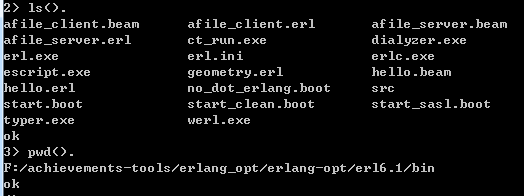
-export([area/1]).

area({rectangle,Width,Height}) -> Width \* Height;

area({square,Side}) -> Side \* Side.



**目录和代码路径**



注意，它们是函数，需要用括号

求一个列表清单的总价格：

-module(shop).

-export([cost/1, total/1]).

cost(orange) -> 4;

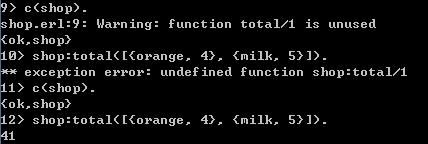
cost(milk) -> 5;

cost(apple) ->10;

cost(pears) -> 6.

total([{What, N}|T]) -> cost(What) \* N + total(T);

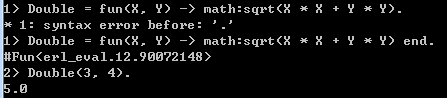
total([]) -> 0.



千万要注意total([]) -> 0. 它是递归调用的出口

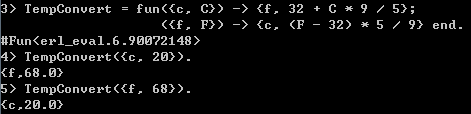
**Fun函数：基本的抽象单元**

Erlang是一种函数式编程语言。此外，函数式编程语言还表示函数可以被用作其他函数的参数，也可以返回函数。操作其他函数的函数被称为高阶函数，而的erlang中用于代表函数的数据类型被称为fun



上面是定义一个fun函数指派给一个变量，注意fun函数最后需要用end结束

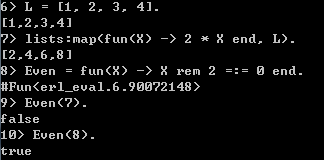
Fun函数可以有多个不同的子句：



**以fun作为参数的函数**

标准库里的lists模块导出了一些以fun作为参数的函数。它们之中最有用的是lists:map(F,L)。这个函数返回的是一个列表，它通过给列表L里的各个元素应用fun F而成。

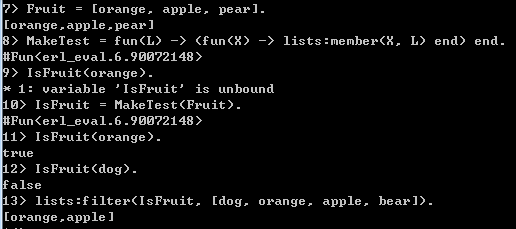
另一个有用的函数是lists:filter(P, L)，它返回一个新的列表，内涵L中所有符合条件的元素（条件是对元素E而言P(E)为true）。Map和filter等函数能再一次调用里对整个列表执行某种操作，我们把它们称为一次一列表式操作





**返回fun的函数**

函数不仅可以使用fun作为参数（例如map和filter），还可以返回fun



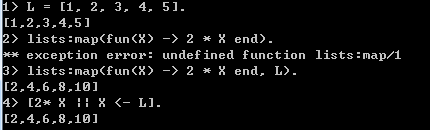
关于理解上面的返回fun的函数

关键的理解就是对于fun函数来说，**括号里的东西就是返回值**

在上面的代码中 MakeTest(Fruit)执行后返回fun(X) -> list:member(X, L) end，其中的L被替换为Fruit

**列表推导**

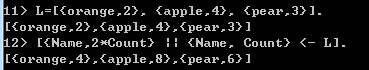
列表推导无须使用fun、map或filter就能创建列表的表达式

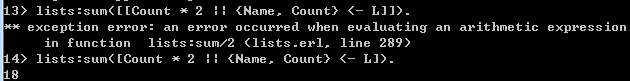


[F(X) || X <- L]意义是X从列表中取出，进行F(X)的操作



举一个例子：





注意，列表推导里的生成器部分起着过滤器的作用



排序：

-module(lib\_misc).

-export([qsort/1]).

qsort([]) -> [];

qsort([Pivot | T]) ->

qsort([X || X <- T, X < Pivot])

++ [Pivot] ++

qsort([X || X <- T, X >= Pivot]





注意++的效率很低下

**毕达哥拉斯三元数组**

pythag(N) ->

[{A, B, C} ||

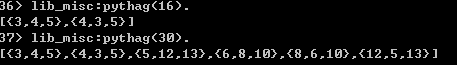
A <- lists:seq(1, N),

B <- lists:seq(1, N),

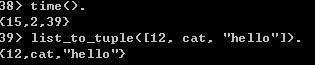
C <- lists:seq(1, N),

A + B + C =< N,

A \* A + B \* B =:= C \* C].



**内置函数**



Time获取当前时间，list\_to\_tuple将一个列表转换为元组

**关卡**

关卡是一种结构，可以用它来增加模式匹配的威力。通过使用关卡，可以对某个模式里的变量执行简单的测试与比较

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

max(X, Y) -> Y.



关卡序列：

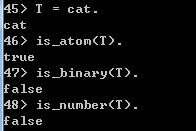
（1）X =:= dog; X =:= cat

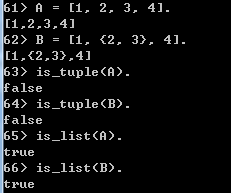
（2）Is\_integer(X), X > Y; abs(Y) < 23

第一个关卡的意思是X是一个dog或者是一个cat

第二个关卡的意思是X是一个整数，并且X大于Y，或者Y的绝对值小于23

关卡内置函数的测试



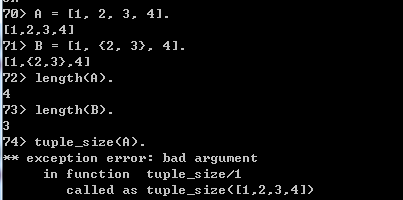


关于length与size的区别

Length-求列表的长度

Size-求元组或二进制型的大小

tuple\_size-元组的长度



**Case 表达式**

Case的语法如下：

Case Expression of

Pattern1 [when Guard1] -> Expr\_seq1;

Pattern2 [when Guard2] -> Expr\_seq2;

…

end

**If表达式**

If

Guard1 ->

Expr\_seq1;

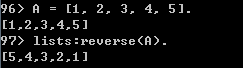
Guard2 ->

Expr\_seq2;

…

End

注意，if表达式必须至少有一个关卡的执行结果为true，否则就会发生异常错误，所以通常在if的最后写上一个true关卡表达式



**归集器**

经常会想让一个函数返回两个列表，比如将一个列表的奇数与偶数分离

-module(lib\_misc).

-export([odds\_and\_evens\_acc/1, odds\_and\_evens\_get/3]).

odds\_and\_evens\_acc(L) ->

odds\_and\_evens\_get(L, [], []).

odds\_and\_evens\_get([H|T], Odds, Evens) ->

case (H rem 2) of

1 -> odds\_and\_evens\_get(T, [H|Odds], Evens);

0 -> odds\_and\_evens\_get(T, Odds, [H|Evens])

end;

odds\_and\_evens\_get([], Odds, Evens) ->

{Odds, Evens}.



目前已经讨论了两种数据容器，一个是元组，一个是列表。元组用于保存固定数量的元素，而列表用于保存可变数目的元素

**记录与映射组**

记录其实就是元组的另一种形式，通过使用记录，可以给元组里的各个元素关联一个名字

映射组是键-值对。键可以是任意erlang数据结构

记录使用一组固定且预定义的名称，而映射组则可以动态添加新的名称

记录是元组的另一种形式，因此他们的存数与性能特性和元组一样，映射组比元组占用更多的存储空间，查找起来也更慢，但是比元组要灵活的多

**记录的语法：**

-record(Name, {

%% 以下两个键带有默认值

key1 = Default1,

key2 = Default2,

%% 以下一个键相当于Key3 = undefined

Key3,

})

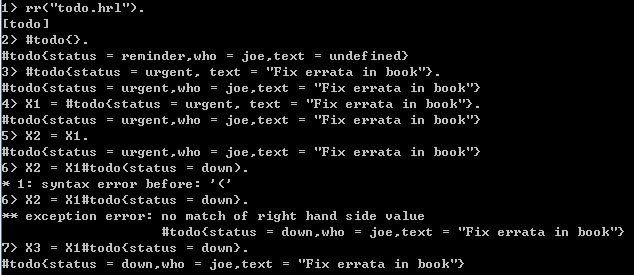
Name是记录名。Key1.。key3这些是记录所含各个字段的名称，它们必须是原子。

在定义记录的时，可以保存在erlang源代码文件里，也可以由扩展名为.hrl的文件保存，然后包含在erlang源代码里，注意，文件包含是能确保多个erlang模块共享相同记录定义的方式

在todo.hrl文件中定义

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

在shell中编译为

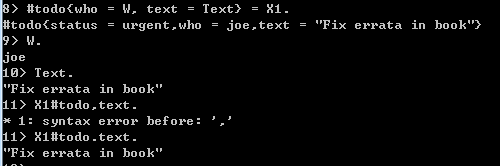


上面的第二行与第三行创建了新的记录。语法#todo{ksy1=val1…}用于创建一个类型为todo的新记录。所有的键都是原子，而且必须与记录定义里所用的一致。如果省略了一个键，系统就会用记录定义里的值作为该键的默认值。

在第四行复制了一个现有的记录。语法X1#todo{status=done}的意思是创建一个X1的副本，并修改字段status的值为done。记住，这么做生成的是一个原始记录的副本，原始记录没有变化

**提取记录的字段**

模式匹配



上面X1#todo.text用于取X1中的某一个字段

**映射组语法**

#{key1 Op val1, key2 Op val2, … , keyN Op valN}

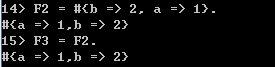
其语法与记录类似，但是之后没有记录名，而Op是=>或:=这两个符号的其中之一



创建带有非原子键的映射组



映射组在系统内部是作为有序集合存储的，打印时总是先给各键排序后的顺序，与映射组的创建方式无关



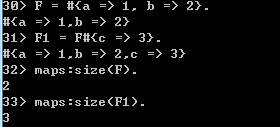
更新一个映射组

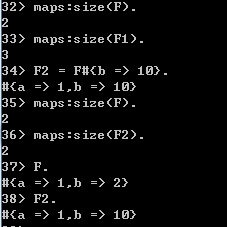
NewMap = OldMap #{K1 Op V1, … , KN Op VN}

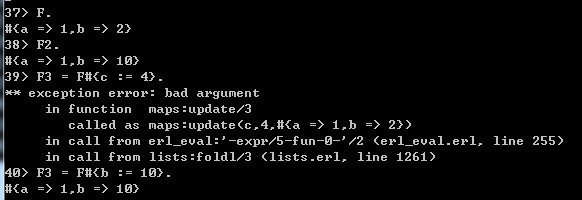
其中Op是=>或者=:

（1）使用K=>V有两种作用，一种是将现有K的值更新为新值V，另一种是给映射组添加一个全新的K-V对，这个操作总是成功的

（2）使用K:=V的作用是将现有键K的值更新为V。如果被更新的映射组不包含键K，这个操作就会失败s





注意上面的=>操作和:=操作的区别

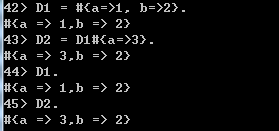
**使用映射组的最佳方式是在首次定义某个键时总是使用Key=>Val，而在修改某个键值时候都是用key := Val**

为什么要这么做？

首先，如果拼错了新键的名称，希望能报错

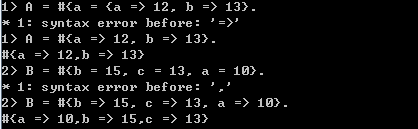
其次，与效率有关。如果映射组更新操作里只使用:=操作符，那么我们就知道新旧映射组都带有一组相同的键，因此可以共享相同的键描述符。假如我们有一个包含数百万映射组的列表，并且它们的各个键都相同，那么节省的空间是很可观的

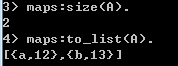
查看下面的一段代码：



在erlang中，给变量赋值后它们都不会改变相互的值，给人的表现就是深层复制了一份。实际上，深层复制并没有发生，erlang系统只是复制了内部结构里的某些必要部分，以形成了复制物的假象。因此，创建一个看似某个对象深层复制物的操作是极其轻量的。

映射组的内置函数







**异常处理**

Erlang有两种方法来捕捉异常。第一种是把抛出异常错误的调用函数封装在一个try…catch中，另一种方法是把调用封装在一个catch表达式中

1 用try…catch捕捉异常

try FuncOrexpressionSeq of

Pattern1 [when Guard1] -> Expressions1;

Pattern2 [when Guard2] -> Expressions2;

Catch

ExceptionType1: Expattern1 [when ExGuard1] -> Expressions1;

ExceptionType2: Expattern2 [when Exgrard2] -> Expressions2;

After

AfterExpressions

End

注意上面的语句，try…catch就像是case的加强版。它基本上就是case表达式加上最后catch和after区块

注意，Exception是一个原子（throw、exit和error其中之一），它告诉我们异常发生错误是如何生成的。如果省略了ExceptionType，就会默认为throw

关键字after之后的代码是用来在FuncOrExpressionSeq结束后执行清理的，这段代码一定会被执行，哪怕有异常错误抛出也是如此。After区块的代码会在try或catch区块里的Expressions代码完成后立即执行。AfterExpressions的返回值会被丢弃

-module(try\_test).

-export([generate\_exception/1, demo1/0, demo2/0, catcher/1]).

generate\_exception(1) -> a;

generate\_exception(2) -> throw(a);

generate\_exception(3) -> exit(a);

generate\_exception(4) -> {'EXIT', a};

generate\_exception(5) -> error(a).

demo1() ->

[catcher(I) || I <- [1, 2, 3, 4, 5]].

demo2() ->

[{I, (catch generate\_exception(I))} || I <- [1, 2, 3, 4, 5]].

catcher(N) ->

try generate\_exception(N) of

Val -> {N, normal, Val}

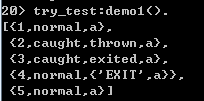
catch

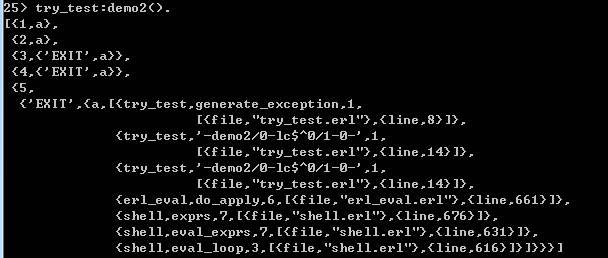
throw: X -> {N, caught, thrown, X};

exit: X -> {N, caught, exited, X};

error: X -> {N, caught, error, X}

end.





可以看出，就会发现这两种方法提供了不同量级的调试信息，第一种方式概括了信息，第二种则提供了详细的栈跟踪信息

捕捉一切可能的异常错误：

Try Expr

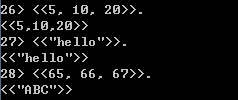
Catch

\_:\_ -> …处理所有的错误代码

end

**二进制型和位语法**

二进制型的编写和答应形式是双小于与双大于号之间的一列整数或字符串

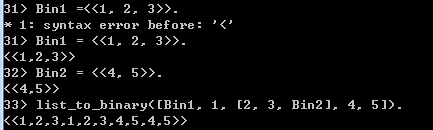


在二进制型使用整数时，它们必须属于0-255这个范围，与字符串类似，如果某个二进制型的内容是可打印的字符串，shell就会将这个二进制型打印成字符串，否则就打印成一列整数

**操作二进制型的内置函数**

（1）List\_to\_binary(L) -> B

List\_to\_binary返回一个二进制型，



注意，第一行两边的空白是必须的。如果没有空白，erlang的分词器就会把第二个符号看做是原子=<（小于等于操作）。有时候必须的二进制型数据的周围加上空白或括号来避免语法错误

（2）split\_binary(Bin, Pos) -> {Bin1, Bin2}

这个函数在Pos处把二进制型Bin一分为二

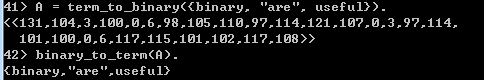


（3）term\_to\_binary -> Bin

（4）binary\_to\_term –> Term

（5）byte\_size -> Size

Erlang的任何数据结构与二进制型的转换





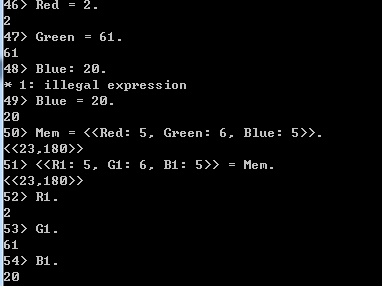
**位语法**

位语法是一种表示法，用于从二进制数据里提取或加入单独的位或位串。

假设要把三个变量（X,Y和Z）打包进一个16位的内存区域。X应当在结果中占据3位，Y应当占据7位，而Z应当占据6位，只需这么写：

M = <<X: 3, Y: 7, Z: 6>>

例如，打包和解包16位颜色



位语法表达式被用来构建二进制型或位串，形式如下：

<<>>

<<E1, E2, … ,En>>

每个Ei元素都标识出二进制型或位串里的一个片段。

如果表达式的总位数是8的整数倍，就会创建一个二进制型，否则创建一个位串

**模块属性（预定义的模块属性与用户定义的模块属性）**

下列模块有着预先定义的含义，必须放在任何函数定义之前

-module -import -export -compile -vsn

**注意**-compile(export\_all)它会导出模块中所有的函数，无需再显式使用-export了

模块的预定义属性：   
-module(Module).    声明模块名称，必须与文件名相同   
-export(Functions).   指定向外界导出的函数列表   
-import(Module,Functions).   引入函数，引入的函数可以被当作本地定义的函数使用   
-compile(Options).     设置编译选项，比如export\_all   
-vsn(Vsn).         模块版本，设置了此项，可以通过beam\_lib:version/1 获取此项信息   
可以通过-include和-include\_lib来包含文件，两者的区别是include-lib不能通过绝对路径查找文件，而是在你当前Erlang的lib目录进行查找。

用户定义属性语法如下：

-SomeTag(value)

例子：

-module(attrs).

-vsn(1234).

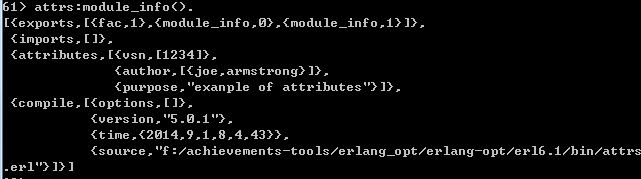
-author({joe, armstrong}).

-purpose("exanple of attributes").

-export([fac/1]).

fac(1) -> 1;

fac(N) -> N \* fac(N - 1).



注意：上面的version是编译器的版本号，不要与模块属性里定义的vsn混淆

**动态代码载入**

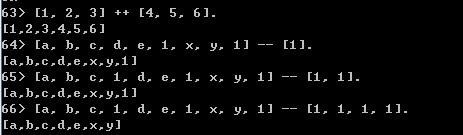
动态代码载入是erlang核心最惊人的特性之一。它的思想很简单：每当调用someModule:someFunction()时。调用的总是最新版模式里的最新版函数，哪怕当代码在模块里运行时重新编译了该模块也是如此。

**列表操作符**

++和—是用于列表添加和移除的中缀操作符

A ++ B（也就是附加）

A –- B 移除的意思是B中所有元素都会从A里面去除，注意：如果符号X在B里只出现了K次，那么A只会移除前K个X



**宏**

Erlang中宏的定义方式如下：

-define(Constant, Replacement).

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

当erlang的预处理器epp碰到一个?MacroName形式的表达式时，就会展开这个宏。宏定义里出现的变量会匹配对应宏调用位置的完成形式,预定义宏如下所示：

?FILE展开成当前文件名

?MODULE展开成当前模块名

?LINE展开成当前行号

-module(define).

-compile(export\_all).

testMax1(X, Y) ->

?LINE.

测试：

-module(m1).

-export([loop/1]).

-ifdef(debug\_flag).

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

-else.

-define(DEBUG(X), void).

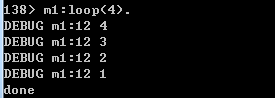
-endif.

loop(0) -> done;

loop(N) ->

?DEBUG(N),

loop(N - 1).



匹配操作符可以用在模式里的任何位置，因此如果有两个需要重建的数据结构，比如下面的：

Func([{tag, {one, A}, B}|T]) ->

…

… f(… {tag, {one, A}, B}, …)

… g(… {one, A}, …)

就可以引入两个心变量Z1和Z2，写成下面的形式

Func([{tag, {one, A} = Z1, B} = Z2|T]) ->

…

… f(… Z2, …)

… g(… Z1, …)

**短路布尔表达式**

Expr1 orelse Expr2

它会首先执行Expr1，如果Expr1的执行结果是true，Exor2就不再执行。如果Expr1的执行结果是false，则会执行Expr2

Expr1 andalso expr2

它会首先执行Expr1，如果Expr1的执行结果是true，则会执行Exor2。如果Expr1的执行结果是false，则不再执行Expr2

注意：在对应的布尔表达式里（A or B和A and B），两边的参数总会被执行，即使表达式的真值只需要第一个表达式的值就能确定也是如此

**类型**

类型定义可以使用以下的非正式语法：

T1 :: A | B | C

它的意思是T1被定义为A、B或C其中之一

定义新的类型可以使用以下语法：

-type NewTypeName(TVar1, Tvar2, … TvarN) :: Type.

**并发编程**

编写并发编程，只需要三个新的基本函数：spawn、send和receive

Spawn - 创建一个并行进程

Send - 向某个进程发送消息

Receive – 接受消息

在erlangli，进程隶属于编程语言，而非操作系统，这就意味着erlang的进程在任何操作系统上都会具有相同的逻辑行为

在erlang中：

1）创建和销毁进程是非常快速的

2）在进程间发送消息是非常快速的

3）进程在所有操作系统上都具有相同的行为方式

4）可以拥有大量进程

5）进程不共享任何内存，是完全独立的

6）进程唯一的交互方式就是消息传递

**基本并发函数：**

（1）Pid = spawn(Mod, Func, Args).

创建一个新的并发进程来执行apply(Mode, Func, Args)。这个新进程和调用进程并列运行

（2）Pid = spawn(Fun)

创建一个新的并发进程来执行Fun。这种形式的spawn总是使用被执行fun的当前值，而且这个fun无需从模块里导出

（3）Pid ! Message

向标识符为Pid的进程发送消息Message。消息发送是异步的，发送方并不等待，而是会继续之前的工作

（4）receive … end

接受发送给某个进程的消息，它的语法如下：

Receive

Pattern1 [when Guard1] ->

Expressions1;

Pattern2 [when Guard2] ->

Expressions2;

End

消息到达进程后，系统会尝试与Pattern1、Pattern2进行匹配，如果没有匹配的模式，消息就会被保存起来供以后使用，进程会开始等待下一条消息

-module(area\_server).

-compile(export\_all).

loop() ->

receive

{rectangle, Width, Height} ->

io:format("Area of rectangle is ~p~n", [Width \* Height]),

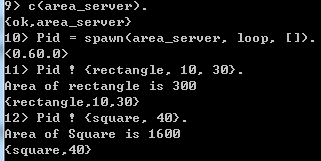
loop();

{square, Side} ->

io:format("Area of Square is ~p~n", [Side \* Side]),

loop()

end.



如果需要对方给自己回复，最好自己在添加一个self标识位进去，让服务器知道应该回复给谁。最佳实践是确认发送给进程的每一个消息都已收到。如果发送给进程的消息不匹配原始接受语句里的任何一个模式，这条消息就会遗留在进程邮箱里，永远无法接受。为了解决这个问题，我们在接受语句的最后加了一个子句，让它能匹配所有发送给此进程的消息。

-module(area\_server).

-compile(export\_all).

rpc(Pid, Request) ->

Pid ! {self(), Request},

receive

Response ->

io:format("match error ~p~n", [Response])

end.

loop() ->

receive

{From, {rectangle, Width, Height}} ->

io:format("Area of rectangle is ~p~n", [Width \* Height]),

From ! Width \* Height,

loop();

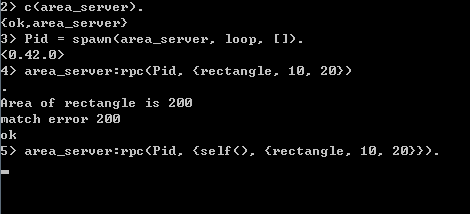
{From, {square, Side}} ->

io:format("Area of Square is ~p~n", [Side \* Side]),

From ! Side \* Side,

loop()

end.



上面的一段代码会有一个问题，在rpc/2函数中，我们向服务器发送请求然后等待响应，但是并不是等待来自服务器的响应，而是等待任何响应。如果其他某个进程在客户端等待来自服务器的消息时向它发送了一个消息，客户端就会将此消息错误解读为来自服务器的响应。

**计算创建进程的时间消耗**

-module(process).

-compile(export\_all).

max(N) ->

Max = erlang:system\_info(process\_limit),

io:format("Maximum allowed processes: ~p~n", [Max]),

statistics(runtime),

statistics(wall\_clock),

io:format("runtime = ~p~n", [runtime]),

io:format("wall\_clock = ~p~n", [wall\_clock]),

L = for(1, N, fun() -> spawn(fun() -> wait() end) end),

{\_, Time1} = statistics(runtime),

{\_, Time2} = statistics(wall\_clock),

lists:foreach(fun(Pid) -> Pid ! die end, L),

U1 = Time1 \* 1000 / N,

U2 = Time2 \* 1000 / N,

io:format("Process spawn time = ~p (~p) microseconds~n", [U1, U2]).

wait() ->

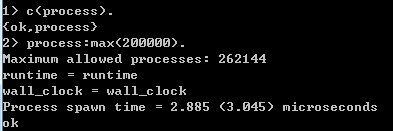
receive

die -> void

end.

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

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



如果调用process:max(300000)是就会出错。

Erlang:system\_info(process\_limit)来找出所允许的最大进程数量，其中有些是系统保留的进程，索尼我们实际上使用没有那么多，当超出限制值时，系统会拒绝启动更多的进程并生成一个错误报告。系统内设的值是262144个进程

如果要超越这一限制，必须用+P标识启动Erlang仿真器如下：

Erl +P 3000000

**带超时的接受**

可以给接受语句添加一个超市器，设定进程等待消息的最长时间，语法如下：

Receive

Pattern1 [when Guard1] ->

Expressions1;

Pattern2 [when Guard2] ->

Expressions2

After Time ->

Expressions

end

如果在进入接受表达式的Time毫秒后还没有收到匹配的消息，进程就会停止等待消息，转而执行Expressions

注意，只有当邮箱里的所有条目都进行过模式匹配后，才会检查after部分

创建一个定时器

-module(stimer).

-export([start/2]).

start(Time, Fun) -> spawn(fun() -> timer(Time, Fun) end).

timer(Time, Fun) ->

receive

cancel ->

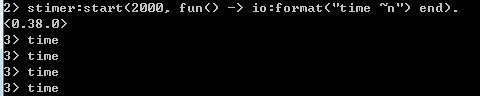
void

after Time ->

Fun(),

timer(Time, Fun)

end.



**注意：如果接受语句里的超时值是原子infinity（无穷大），就永远不会触发超时**

**注册进程**

如果想给一个进程发送消息，就需要知道它的PID，但是当进程创建时，只有父进程才知道它的PID。系统里没有其他进程知道它的存在。Erlang有一种公布进程标识的方法，它让系统里的任何进程都能与该进程通信，这样的进程被称为**注册进程**。管理注册进程的内置函数有四个：

**register(AnAtom, Pid)**

用AnAtom（一个原子）作为名称来注册进程Pid。如果AnAtom已被用于注册某个进程，这此注册就失败

**unregister(AnAtom)**

移除与AnAtom关联的所有注册信息

**whereis(AnAtom) -> Pid | undefined**

监察AnAtom是否已被注册，如果是就返回进程标识符Pid，如果没有找到与AnAtom关联的进程就返回原子undefined

**registerd() -> [AnAtom:: atom()]**

返回一个包含系统里所有注册进程的列表

**关于尾递归的说明**

可以对尾递归的函数进行特别编译，把语句序列里的最后一次函数调用替换成跳至被调用函数的开头。这就意味着尾递归的函数无需消耗栈空间也能抑制循环下去。

**注意：**如果不采用尾递归，而是在递归之后再执行某种操作，则它会将后面的调用压栈，抑制循环下去，会导致系统的空间被消耗完

一个并发程序的模板：

-module(ctempplate).

-compile(export\_all).

start() ->

spawn(?MODULE, loop, []).

rpc(Pid, Request) ->

Pid ! {self(), Request},

receive

{Pid, Response} -> Response

end.

loop(X) ->

receive

Any ->

Io:format(“Receive: ~p~n”, [Any]),

Loop()

end.

**并发程序的错误**

两个原则：“让其他进程修复错误”和“任其崩溃”

在erlang中，会将应用程序构建成两个部分：一部分负责解决问题，另一部分负责在错误发生时纠正它们。纠正错误的部分往往是通用的，因此同一段错误纠正代码可以用在许多不同的应用程序里。

**错误处理的术语含义**

**进程**

进程有两种，普通进程和系统进程。Spawn创建的是普通进程。普通进程可以通过执行内置函数process\_flag(trap\_exit, true)编程系统进程

**连接**

进程可以相互连接。如果A和B两个进程有连接，而A处于某种原因终止了，就会向B发出一个错误信息，反之亦然

**连接组**

进程P的连接组是指与P相连的一组进程

**监视**

监视和连接很相似，但监视是单向的。如果A监视B，而B处于某种原因终止了，就会向A发送一个“宕机”消息，但反过来就不行

**消息和错误信号**

进程写作的方式是交换消息或错误信号。消息是通过基本函数send函数发送的，错误信号则是进程崩溃或进程终止时自动发送的。错误信号会发送给终止进程的连接组

**错误信号的接受**

当系统进程受到错误信号时，该信号会被转换成{‘EXIT’, Pid, Why}形式的消息。Pid值终止进程的标识，Why是终止原因。如果进程是无错误终止，Wht就是源自normal，否则Wht会是错误的描述

当普通进程受到错误信号时，如果推出原因不是normal，该进程就会终止。当它终止时，同样会向它的连接组广播一个退出信号

**显式错误信号**

任何执行exit(Why)的进程都会终止（如果不是在catch或try的范围内执行的话），并向它的连接组广播一个带有原因Why的退出信号

**不可捕捉的退出进程**

系统进程受到摧毁信号(kill signal)时会终止。摧毁信号是通过调用exit(Pid, kill)生成的。这种信号会绕过常规的错误信号处理机制，不会被转换成消息。摧毁信号值应该用在其他错误处理机制无法终止的顽固进程上。

**文件编程**

用于文件操作的函数被归为四个模块：

File

它包含打开、关闭、读取和写入文件的方法，还有列出目录等等

Filename

这个模块里的方法能够以跨平台的方式操作文件名，这样就可以在许多不同的操作系统上运行相同的代码了

Filelib

这个模块是file的扩展，其中大多数都是使用file里的函数编写的

Io

这个模块有一些操作已打开文件的方法。它包含的方法能够解析文件里的数据，或者把格式化写入数据

现在创建data.dat与temp.dat文件

Data.dat

{person, "joe", "armstrong",

[{occupation, programmer},

{favoriteLanguage, erlang}]}.

{cat, {name, "zorro"},

{owner, "joe"}}.

Temp.dat

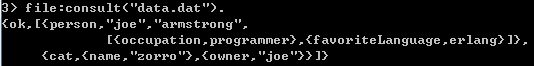
"this is my first file"

"do you konw?"

**读取文件**

（1）读取文件里的所有数据类型

可以调用file:consule来读取所有的数据类型

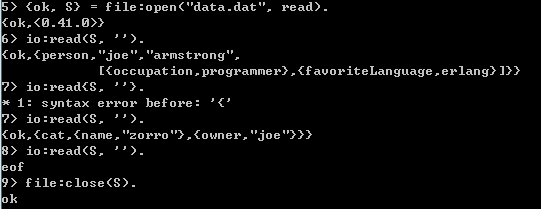


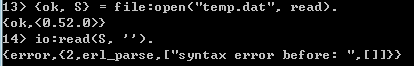


file:consult(File)假定File包含一个由Erlang数据类型组成的序列。如果它能读取文件里的所有数据类型，就会返回{ok, [Term]}，否则就会返回{error, Reason}

（2）分次读取文件里的数据类型

如果想从文件中一次读取一个数据类型，就要首先用file:open打开文件，然后用io:read逐个读取数据类型，直到文件末尾，最后再用file:close关闭文件





注意上面的不同：

1）File:open(File, read) -> {ok, IoDevice | {error, Why}}

尝试打开File进行读取。如果它能打开文件就会返回{ok, IoDevice}，否则返回{error, Reason}。IoDevice是一个用来访问文件的I/O对象

2）io:read(IoDevice, Prompt) -> {ok, Term} | {error, Why} | eof

从IoDevice读取一个**erlang数据类型**Term。如果IoDevice代表一个被打开的文件，Prompt就会被忽略。只有用io:read读取标准输入时，才会用Prompt提供一个提示符

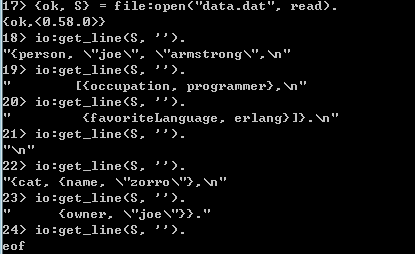
3）file:close(IoDevice) -> ok | {error, Why}

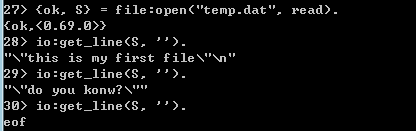
关闭IoDevice

查找已载入的源代码：



（3）分次读取文件里的行





io:get\_line会一直读取字符，直到遇到换行符或者文件尾

（4）读取整个文件到二进制型中

可以用file:read\_file(File)把整个文件读入一个二进制型中，这是一次原子操作

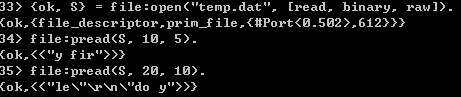
如果成功，则返回{ok, Bin}，否则返回{error, Why}。这是目前为止最高效的文件读取方式。在大多数的操作中，通常将整个文件一次性读入内存，然后操作内容并一次性保存文件（采用file:write\_file）



（5）通过随机访问读取文件

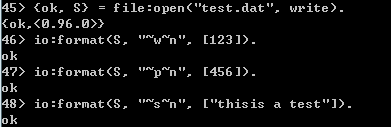
如果要读取的文件非常大，或者它包含牟宗外部定义格式的二进制数据，就可以用raw模式打开这个文件，然后用file:pread读取它的任意部分

File:pread(IoDevice, Start, Len)会从IoDevice读取Len个字节的数据，读取的起点是字节Start处（文件里的字节会被编号，所以文件里的第一个字节的位置是0），它会返回{ok, Bin}或者{error, Why}



**写入文件**

（1）把各行写入文件



（2）一次性写入整个文件

这是最高效的写入文件方式。File:write\_file(File, IO)会把IO里的数据（一个I/O列表）写入File

**目录和文件操作**

file里有三个操作目录的函数

list\_dir(Dir) 生成一个Dir里的文件列表

make\_dir(Dir) 创建一个新目录

del\_dir(Dir) 删除一个目录

**复制和删除文件**

File:copy(Souece, Destination)

File:delete(File)

**套接字编程**

套接字编程主要有两个库：gen\_tcp用于TCP应用，gen\_udp用于UDP应用

-module(socket\_example).

-compile(export\_all).

mano\_get\_url() ->

nano\_get\_url("www.baidu.com").

nano\_get\_url(Host) ->

{ok, Socket} = gen\_tcp:connect(Host, 80, [binary, {packet, 0}]),

ok = gen\_tcp:send(Socket, "GET / HTTP/1.0\r\n\r\n"),

receive\_data(Socket, []).

receive\_data(Socket, SoFar) ->

receive

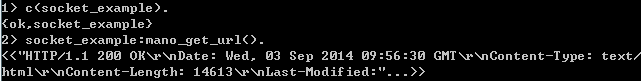
{tcp, Socket, Bin} ->

receive\_data(Socket, [Bin | SoFar]);

{tcp\_closed, Socket} ->

list\_to\_binary(lists:reverse(SoFar))

end.



解释：

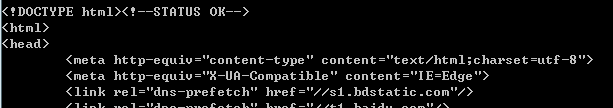
（1）gen\_tcp:connect，打开一个到<http://www.baidu.com> 80端口的TCP套接字。连接调用里的binary参数告诉系统要以“二进制”模式打开套接字，并把所有的数据用二进制型传给应用。{packet, 0}意思是把未经过修改的TCP数据直接传给应用程序

（2）gen\_tcp:send，把消息GET/HTTP/1.0\r\n\r\n发送给套接字，然后等待回复。这个回复并不是放在一个数据包里，而是分成多个片段，一次发送一点。这些片段会被接受成为消息序列，发送给打开（或控制）套接字的进程

（3）收到一个[tcp, Socket, Bin]消息。这个元组的第三个参数是一个二进制型，原因是打开套接字时使用了二进制模式

查看整个输出：





**通用服务器**

其中采用了**热代码交换（即不需要停止服务器可以进行代码升级）**

**server1代码：**

-module(server1).

-export([start/2, rpc/2, swap\_code/2]).

start(Name, Mod) ->

register(Name, spawn(fun() -> loop(Name, Mod, Mod:init()) end)).

loop(Name, Mod, OldState) ->

receive

{From, {swap\_code, NewCallBackMod}} ->

From ! {Name, ok, ack},

loop(Name, NewCallBackMod, OldState);

{From, Request} ->

try Mod:handle(Request, OldState) of

{Response, NewState} ->

From ! {Name, ok, Response},

loop(Name, Mod, NewState)

catch

\_:Why ->

io:format("error ~p~n", [Why]),

From ! {Name, crash},

loop(Name, Mod, OldState)

end

end.

rpc(Name, Request) ->

Name ! {self(), Request},

receive

{Name, ok, Response} -> Response;

{Name, crash} -> exit(rpc)

end.

swap\_code(Name, Mod) -> rpc(Name, {swap\_code, Mod}).

**name\_server代码：**

-module(name\_server).

-compile(export\_all).

-import(server1, [rpc/2]).

add(Name, Place) -> rpc(name\_server, {add, Name, Place}).

find(Name) -> rpc(name\_server, {find, Name}).

%% 回调方法

init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};

handle({find, Name}, Dict) -> {dict:find(Name, Dict), Dict}.

**New\_name\_server代码：（服务器进行代码升级）**

-module(new\_name\_server).

-export([]).

-import(server1, [rpc/2]).

-compile(export\_all).

add(Name, Place) -> rpc(name\_server, {add, Name, Place}).

find(Name) -> rpc(name\_server, {find, Name}).

all\_names() -> rpc(name\_server, allNames).

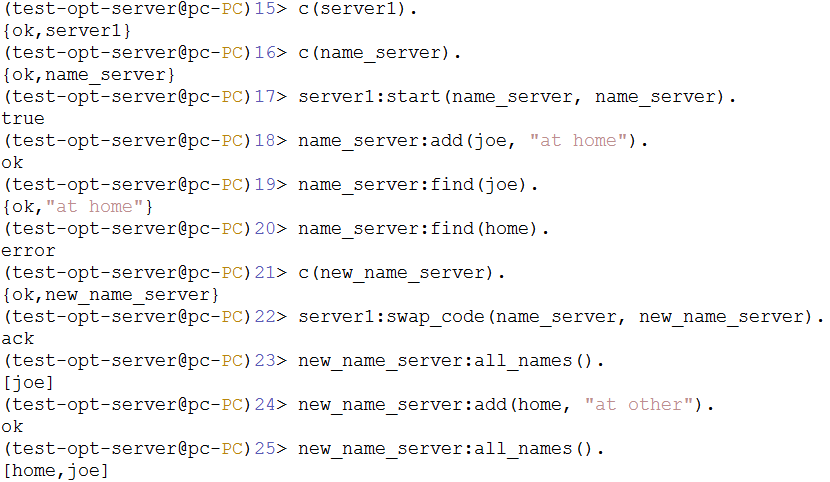
%% 回调函数

init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};

handle({find, Name}, Dict) -> {dict:find(Name, Dict), Dict};

handle(allNames, Dict) -> {dict:fetch\_keys(Dict), Dict}.



从上面的代码中可以看出，实现了从name\_server到new\_name\_server的切换

**gen\_server入门**

编写gen\_server回调模块的简要步骤：

（1）确定回调模块名

（2）编写接口函数

（3）在回调模块里编写六个必要的回调函数

例子（一个简单的gen\_server例子）：

-module(my\_bank).

-behaviour(gen\_server).

-define(SERVER, my\_bank).

-compile(export\_all).

start() -> gen\_server:start\_link({local, ?SERVER}, ?MODULE, [], []).

stop() -> gen\_server:call(?MODULE, stop).

new\_account(Who) -> gen\_server:call(?MODULE, {new, Who}).

deposit(Who, Amount) -> gen\_server:call(?MODULE, {add, Who, Amount}).

withdraw(Who, Amount) -> gen\_server:call(?MODULE, {remove, Who, Amount}).

init([]) -> {ok, ets:new(?MODULE, [])}.

handle\_call({new, Who}, \_From, Tab) ->

Reply = case ets:lookup(Tab, Who) of

[] -> ets:insert\_new(Tab, {Who, 0}),

{welcome, Who};

[\_] -> {Who, you\_already\_are\_a\_customer}

end,

{reply, Reply, Tab};

handle\_call({add, Who, X}, \_From, Tab) ->

Reply = case ets:lookup(Tab, Who) of

[] -> not\_a\_customer;

[{Who, Balance}] ->

NewBalance = Balance + X,

ets:insert(Tab, {Who, NewBalance}),

{thanks, Who, you\_balance\_is, NewBalance}

end,

{reply, Reply, Tab};

handle\_call({remove, Who, X}, \_From, Tab) ->

Reply = case ets:lookup(Tab, Who) of

[] -> not\_a\_customer;

[{Who, Balance}] when X =< Balance ->

NewBalance = Balance - X,

ets:insert(Tab, {Who, NewBalance});

[{Who, Balance}] ->

{sorry, Who, you\_only\_have, Balance, in\_the\_bank}

end,

{reply, Reply, Tab};

handle\_call(stop, \_From, Tab) ->

{stop, normal, stopped, Tab}.

handle\_cast(\_Msg, State) -> {noreply, State}.

handle\_info(\_Info, State) -> {noreply, State}.

terminate(\_Reason, \_State) -> ok.

code\_change(\_OldVsn, State, \_Extra) -> {ok, State}.



**用OPT构建系统**

（1）通用事件处理

**Event\_handler.erl**

make(Name) ->

case register(Name, spawn(fun() -> my\_handler(fun no\_op/1) end)) of

true -> {init\_ok};

false -> {init\_error}

end.

add\_handler(Name, Fun1) -> Name ! {add, Fun1}.

event(Name, Request) -> Name ! {event, Request}.

my\_handler(Fun) ->

receive

{event, Request} ->

(catch Fun(Request)),

my\_handler(Fun);

{add, Fun1} ->

my\_handler(Fun1)

end.

no\_op(\_) -> void.

**Motor\_controller.erl**

add\_event\_handler() ->

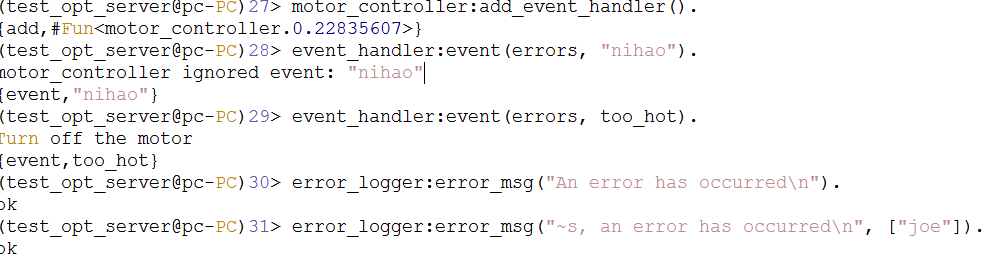
event\_handler:add\_handler(errors, fun controller/1).

controller(too\_hot) ->

io:format("Turn off the motor ~n");

controller(X) ->

io:format("~w ignored event: ~p~n", [?MODULE, X]).



（2）错误记录

通常可以创建一个滚动日志。可以把滚动日志看成一个大型循环缓冲区，内含错误记录生成器的消息。新消息进来后会被附加到日志的末尾，如果日志满了，最早的条目就会被删除

理解erlang中的gen\_server

<http://www.cnblogs.com/me-sa/archive/2011/12/20/erlang0023.html>

OTP behaviour包含gen\_server gen\_event gen\_fsm supervisor.其中绝大多数情况下都是在使用gen\_server,supervisor本身也是使用gen\_server实现的