

# **Erlang/OTP R11B 文档**

## **Erlang/OTP R11B documentation**

**v0.1.5**

Erlang/OTP 官方 著

D. Wu 译

Email: [linux@sun7.org](mailto:linux@sun7.org)

[linux@263.net](mailto:linux@263.net)

## 译者 注

本译文仅供对 Erlang 感兴趣的朋友学习使用，并非用于商业目的。无论如何请尊重本人的劳动成果。

由于原文档很多，我会随翻译的进度更新本译文的版本。而不是全部翻译完之后直接发布最终版本。由于是第一次翻译文档，所以译文中可能出现少量的错误，还希望得到你的谅解和指正，我会在后续的版本中纠正。

相关网站：

<http://wiki.sun7.cn/ErlangDoc>

## 其它译者

# 目录

译者注.....	I
其它译者.....	II
第一部分 入门.....	1
1. 简介.....	1
1.1 简介.....	1
1.2 其它方面.....	1
2. 顺序编程.....	2
2.1 Erlang Shell.....	2
2.2 模块和函数.....	3
2.3 元子(Atoms).....	6
2.4 元组.....	7
2.5 列表.....	8
2.6 标准模块及用户手册.....	11
2.7 将输出写到终端上.....	11
2.8 一个更大的例子.....	12
2.9 变量的匹配、guard 和作用域.....	14
2.10 更多关于列表.....	16
2.11 If 和 Case.....	21
2.12 内建函数(BIFs).....	25
2.13 复杂函数.....	27
3. 并行编程.....	30
3.1 进程.....	30
3.2 信息传递.....	31
3.3 进程名称注册.....	35
3.4 分布式编程.....	37
3.5 一个更大的例子.....	40
4. 健壮性(Robustness 鲁棒性).....	49
4.1 超时(Timeouts).....	49
4.2 错误处理.....	51
4.3 增强健壮性之后的大型例子.....	54
5. 记录和宏(Records and Macros).....	59
5.1 将大型的例子分割在多个文件中.....	59
5.2 头文件(Header Files).....	64
5.3 记录(Records).....	64

5.4 宏(Macros).....	65
6. 实例.....	65
第二部分 OTP 设计原则.....	68
1. 概述.....	68
1.1 监督树.....	68
1.2 Behaviour.....	68
1.3 应用.....	72
1.4 发布.....	73
1.5 发布控制.....	73
2. Gen_Server Behaviour(文档缺失).....	73
3. Gen_Fsm Behaviour.....	73
3.1 有限状态机.....	73
3.2 实例.....	74
3.3 启动一个 Gen_Fsm.....	75
3.4 事情通知.....	76
3.5 超时.....	76
3.6 All 状态事件.....	77
3.7 停止函数.....	77
3.7.1 在监督树中.....	77
3.7.2 独立 Gen_Fsm.....	78
3.8 处理其它消息.....	78
4. Gen_Event Behaviour.....	79
4.1 事件处理原则.....	79
4.2 实例.....	79
4.3 启动一个事件管理器.....	80
4.4 加入一个事件处理器.....	80
4.5 事件通知.....	81
4.6 删除一个事件处理函数.....	82
4.7 停止.....	82
4.7.1 在监督树中.....	82
4.7.2 独占式事件管理器.....	82

# 第一部分 入门

## 1. 简介

### 1.1 简介

这是一篇入门文章，教你如何开始使用 Erlang。在这里，我将只告诉你一些最简单的语法，而不是所有技术细节。有很多地方我会写上\*manual\*，表示这里有很多信息，你可以在 Erlang 相关的书籍中找到或是在<Erlang 参考手册>中找得到。

我假设这不是第一次接触计算机，而且你了解编程的最基本的思想。不过不要着急，我并不会假定你是一个有经验的开发人员。

### 1.2 其它方面

文章省略了以下几个方面：

- 参考
- 本地错误处理(cache/throw)
- 单向连接(显示器)
- 二进制数据处理
- 列表相关
- 与外界如何通信，以及/或者 port 其它语言开发的软件。当然，有一些向导中会单独讲解这个问题。<互操作向导>
- 涉及到的极少数 Erlang 库(如，文件处理)
- 关于 OTP 的问题完全被跳过，关于 Mnesia 数据库的信息在结论中也被省略。
- Erlang 中的哈杀表。
- 运行时改变代码。

## 2. 顺序编程

### 2.1 Erlang Shell

大多数的操作系统都有一个命令行解释器或一个Shell。Unix 和 Linux 有很多种，Windows 下也有一个命令提示行。当然了，Erlang 也有它自己的Shell，你可以直接写一些小程序或 Erlang 代码并评估(运行)它，以查看发生了什么。(see shell(3))。你可以通过打开一个命令行解释器，并输入 erl，以启动 Erlang Shell(在 Linux 或 Unix 中)。

```
% erl
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2 (abort with ^G)
1>
```

现在在 Shell 中输入 "2 + 5."，运行结果正如你下面看到的。

```
1> 2 + 5.
7
2>
```

在 Windows 中，Shell 可以通过双击 Erlang Shell 的图标打开。

你可能发现 Erlang shell 有很多行号(如 1> 2>)。它们可以准确的告诉你 2 + 5 是 7! 同时也提示你，你应该通过一个终止符 "." 和一个回车来结束你的输入。如果你在 shell 中输入了错误的东西，在多数的 shell 中，你可以通过输入退格键(backspace) 来删除他们。同时在 shell 中也有很多编译命令的方式。(See the chapter "tty - A command line interface" 在 ERTS 用户向导)。

(注意：你会发现现在本文中以及代码测试中，很多行号并不是连续的)。

现在我们试着做更复杂的计算。

```
2> (42 + 77) * 66 / 3.
2618.00
```

在这里，你可以看到括号与乘法(\*)、除法(/)的用法，就像在正常算数中的用法一样(See

the chapter "Arithmetic Expressions" in the Erlang Reference Manual)。

退出 Erlang 系统和 Erlang shell 请输入 Control-C。你将看到如下的输出：

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
        (v)ersion (k)ill (D)b-tables (d)istribution
a
%
```

输入"a"以退出 Erlang 系统。

另一个退出 Erlang 系统的方法是输入 halt()：

```
3> halt().
%
```

## 2.2 模块和函数

这里是一个 Erlang 小程序。主用一个合适的编辑器把它输入一个名为 tut.erl 的文件 (文件名 tut.erl 这一点很重要, 同时需要确定文件与 erl 在同一个目录下)。如果走运的话, 你的编辑器可能支持 Erlang 语法, 这可以让你的开发容易一些, 它可以格式化你的代码, 让他们看起来更漂亮些。(See the chapter "The Erlang mode for Emacs" in Tools User's Guide), 不过没有它们, 你仍然也可以将你的代码做得很完美。下面是你将要输入的代码：

```
-module(tut).
-export([double/1]).

double(X) ->
    2 * X.
```

不难猜出这段代码是用来求一个数的倍数的。一会儿我们再回头来说关于前两行的事情。我们先编译这个程序。可以通过在你的 Erlang shell 下输入下面代码来实现：

```
3> c(tut).
{ok,tut}
```

{ok,tut}告诉你编译成功。如果它提示"error", 你可能在输入文本的时候犯错误了。这

里的错误信息可能会给你一些有关于如何纠正错误的思路，依此你可以改变你的代码，重新再试。

现在让我们运行这个程序。

```
4> tut:double(10).  
20
```

10 的倍数是 20，很正确。

现在我们回过头来看程序的前两行。Erlang 程序写在文件中。每个文件都包含一个 Erlang 模块 (Module)。在文件中的第一行，就告诉我们模块的名称 (See the chapter "Modules" in the Erlang Reference Manual)。

```
-module(tut).
```

这告诉我们模块的名称是 `tut`。注意本行结尾的 `.`。存放模块代码的文件名，也必须和模块同名并以 `.erl` 做为扩展名。在我们的例子中，文件名为 `tut.erl`。当我们使用另一个模块的函数，我们使用语法，`模块名:函数名(参数)`。所以

```
4> tut:double(10).
```

意味着我们调用 `tut` 模块中的 `double` 函数，并使用 `"10"` 做为参数。

第二行：

```
-export([double/1]).
```

说明 `tut` 模块包含一个名称为 `double` 的函数，并且带有一个参数 (在我们的例子中为 `X`)，这个函数可以在 `tut` 模块以外被调用。更多关于这个问题的说明，我们稍后讨论。应该再次注意这一行后面的 `.`。

下面给出一个更复杂的例子，一个数字的阶乘 (如：4 的阶乘是  $4*3*2*1$ )。在名为 `tut1.erl` 中输入下面的代码。

```
-module(tut1).  
-export([fac/1]).
```

```
fac(1) ->
    1;
fac(N) ->
    N * fac(N - 1).
```

编译这个文件

```
5> c(tut1).
{ok,tut1}
```

现在计算 4 的阶乘。

```
6> tut1:fac(4).
24
```

第一部分:

```
fac(1) ->
    1;
```

说明 1 的阶乘是 1。注意我们以一个 ";" 结束这一部分，这说明这个函数没有结束。第二部分:

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

说明 N 的阶乘是: N 与 N - 1 的阶乘的乘积 ( $N! = N * (N - 1)!$ )。注意这部分以 "." 结束，以说明本函数没有其它部分了。

一个函数可以有多个参数。让我们扩展 tut1 模块，让他计算两数的乘积。

```
-module(tut1).
-export([fac/1, mult/2]).

fac(1) ->
    1;
fac(N) ->
    N * fac(N - 1).
```

```
mult(X, Y) ->
    X * Y.
```

注意，我们同时也扩展了-export 行，并给于它关于另一个带有两个参数的函数信息。

编译:

```
7> c(tut1).
{ok,tut1}
```

试验:

```
8> tut1:mult(3, 4).
12
```

上面的例子中的数值是整数，在函数功能代码中的参数 N, X, Y 被叫做变量。变量首字母必须是大写(see the chapter "Variables" in the Erlang Reference Manual)。例如变量可以是 Number, ShoeSize, Age 等等。

### 2.3 元子(Atoms)

元子是 Erlang 中的另一个数据类型。元子以小写字母开头(See the chapter "Atom" in the Erlang Reference Manual)，例如: charles, centimeter, inch。元子只是一个简单的名字，其它什么都不是。他们不像变量可以带有一个值。

来看下一个程序(文件: tut2.erl)用来把英寸转换成厘米或反过来。

```
-module(tut2).
-export([convert/2]).

convert(M, inch) ->
    M / 2.54;

convert(N, centimeter) ->
    N * 2.54.
```

编译:

```
9> c(tut2).
{ok,tut2}
```

```
10> tut2:convert(3, inch).
1.18110
11> tut2:convert(7, centimeter).
17.7800
```

对于小数我不再多做解释了，我猜你可以理解。

看看如果我们输入 centimeter 和 inch 以外的参数给 convert 函数后会发生什么。

```
13> tut2:convert(3, miles).

=ERROR REPORT==== 28-May-2003::18:36:27 ===
Error in process <0.25.0> with exit value: {function_clause, [{tut2,convert,
[3,miles]},{erl_eval,expr,3},{erl_eval,exprs,4},{shell,eval_loop,2}]}
** exited: {function_clause, [{tut2,convert,[3,miles]},
                                {erl_eval,expr,3},
                                {erl_eval,exprs,4},
                                {shell,eval_loop,2}]} **
```

convert 函数的两个部分都调用了它的句块。在这里我们看到"miles"不在句块中。Erlang 系统无法匹配(match)任一个句块，所以我们得到一个错误信息 function\_clause。上面的输出看起来很乱，不过稍微练习一下，你就可以看出程序究竟在代码的哪个地方出错的。

## 2.4 元组

现在 tut2 程序还不是一个很好的程序风格。考虑：

```
tut2:convert(3, inch).
```

它说明 3 是以英寸为单位的？或是 3 是厘米，我们想把他转换成英寸？所以 Erlang 有一个方法把他们归到一起，让他们更容易理解一些。这个东东叫元组(tuples)。元组以 "{"和"}"括起来的。

所以我们可以写 {inch, 3} 来表示 3 英寸，并以 {centimeter, 5} 来表示 5 厘米。现在让我们写一个新的程序，将厘米转换成英寸，反之亦然(文件名: tut3.erl)。

```
-module(tut3).
-export([convert_length/1]).
```

```
convert_length({centimeter, X}) ->
    {inch, X / 2.54};
convert_length({inch, Y}) ->
    {centimeter, Y * 2.54}.
```

编译并测试:

```
14> c(tut3).
{ok,tut3}
15> tut3:convert_length({inch, 5}).
{centimeter,12.7000}
16> tut3:convert_length(tut3:convert_length({inch, 5})).
{inch,5.00000}
```

注意 16 行。我们将 5 英寸转换为厘米，并把它转换回原来的值。此例说明，函数的参数也可以是另一个函数的返回值。稍停片刻，来想一下 16 行的代码是如何工作的。我们给函数的参数 {inch, 5}，是第一个与 convert\_length 第一句块匹配的参数。如：convert\_length({centimeter, X}) 我们认为 {centimeter, X} 与 {inch, 5} 不匹配，我们找下一个句块。如，convert\_length({inch, Y})，Y 得到 5 以后，就匹配了。

我们看到上面的元组有两部分组成，不过元组可以有很多部分，我们想要多少都可以。如，提供一个世界上不同城市温度值，我们可以这么写。

```
{moscow, {c, -10}}
{cape_town, {f, 70}}
{paris, {f, 28}}
```

一组元组有一个固定的大小。我们称元组中的东西为‘元素’。所以在元组 {moscow, {c, -10}} 中，元素 1 是 moscow，元素 2 是 {c, -10}。我已经把 c 的意思确定为摄氏度 (Centigrade or Celsius)，f 确定为华氏度 (Fahrenheit)。

## 2.5 列表

鉴于元组把东西组到一起，我们也希望提供一个列表一样的东西。列表在 Erlang 中被括在 "[ " 和 "]" 里。如一个不同城市温度的列表可能是这样的：

```
[{moscow, {c, -10}}, {cape_town, {f, 70}}, {stockholm, {c, -4}},
 {paris, {f, 28}}, {london, {f, 36}}]
```

注意，这个列表很长，无法写在同一行上。没关系，Erlang 可允许分成多行，不过，不能在元子或整数中间的某部分来分。

一个很有用的遍历列表的方法是使用"`|`"。这里最好用一个 shell 中的例子来说明。

```
18> [First |TheRest] = [1,2,3,4,5].
[1,2,3,4,5]
19> First.
1
20> TheRest.
[2,3,4,5]
```

我们使用`|`来分隔列表中的第一个元素和后续的元素。(First 得到 1 这个值，TheRest 的值为`[2,3,4,5]`)。

另一个例子：

```
21> [E1, E2 | R] = [1,2,3,4,5,6,7].
[1,2,3,4,5,6,7]
22> E1.
1
23> E2.
2
24> R.
[3,4,5,6,7]
```

这里我们看到`|`的用途是从列表中得到前两个元素。当然，如果我们试着从列表中得到比列表中定义的元素个数更多的元素的话，我们会得到一个错误。也要注意列表中没有元素的特例。

```
25> [A, B | C] = [1, 2].
[1,2]
26> A.
1
27> B.
2
28> C.
[]
```

上面的所有例子，我使用了新的变量名，而不是重新使用旧的变量：Frist，

TheRest, E1, E2, R, A, B, C。原因是变量在它的上下文(作用域)中, 只能赋值一次。我会在下面说明这个问题, 虽然这听上去好像很奇特。

下面的例子告诉我们如何查出列表的长度:

```
-module(tut4).  
-export([list_length/1]).  
  
list_length([]) ->  
    0;  
list_length([First | Rest]) ->  
    1 + list_length(Rest).
```

编译(文件 tut4.erl)并测试:

```
29> c(tut4).  
{ok,tut4}  
30> tut4:list_length([1,2,3,4,5,6,7]).  
7
```

解释:

```
list_length([]) ->  
    0;
```

一个空列表的长度显然应该是 0。

```
list_length([First | Rest]) ->  
    1 + list_length(Rest).
```

列表的长度, 是 First 与其余元素的长度之合, 即 1 + Rest 的长度。

(对高级读者: 这并不是一个尾递归式, 当然还有更好的方法来写此函数)。

我们通常所说的使用元组, 在其它的编程语言中, 我们可能会叫做“记录”或“结构体”。当我们希望提供一个大小可变的東西的时候, 我们需要使用列表(我们在其它编程语言中所说的“链表”)。

Erlang 并没有字符串类型, 取而代之我们可以提供一个由 ASCII 字符组成的列表。所以

列表[97,98,99]相当于"abc"。Erlang shell 很“聪明”，它会推测我们使用列表的本意，并以最接近形式输出，如：

```
31> [97,98,99].  
"abc"
```

## 2.6 标准模块及用户手册

Erlang 有很多标准模块帮助你做一些事情。比如，io 模块会包含很多帮助你格式化输出/输入的函数。查找关于标准模块的信息，可以在操作 shell 或命令行中使用命令 `erl -man` 来做到这一点(在与你启动 `erl` 时，相同的目录中)。尝试在操作系统 shell 输入命令：

```
% erl -man io  
ERLANG MODULE DEFINITION          io(3)  
  
MODULE  
  io - Standard I/O Server Interface Functions  
  
DESCRIPTION  
  This module provides an interface to standard Erlang IO  
  servers. The output functions all return ok if they are suc-  
  ...  
  
If this doesn't work on your system, the documentation is included as HTML  
in the Erlang/OTP release, or you can read the documentation as HTML or  
download it as PDF from either of the sites www.erlang.se (commercial  
Erlang) or www.erlang.org (open source), for example for release R9B:  
  
http://www.erlang.org/doc/r9b/doc/index.html
```

## 2.7 将输出写到终端上

在这些例子中，我们可以很好的按格式输出了。所以，下面的列子，展示一个简单的使用 `io:format` 函数的方法。当然，正如其它所有的输出函数一样，你可以在 shell 中测试 `io:format` 函数：

```
32> io:format("hello world~n", []).  
hello world  
ok  
33> io:format("this outputs one Erlang term: ~w~n", [hello]).
```

```
this outputs one Erlang term: hello
ok
34> io:format("this outputs two Erlang terms: ~w~w~n", [hello, world]).
this outputs two Erlang terms: helloworld
ok
35> io:format("this outputs two Erlang terms: ~w ~w~n", [hello, world]).
this outputs two Erlang terms: hello world
ok
```

函数 `format/2` (`format` 带有 2 个参数) 带有两个列表。第一个列表几乎总是在 "" 中列出。这个列表会按原样输出，除了 `~w` 被使用第二个参数中的对相的元素替换了。每个 `~n` 都是一个新行。`io:format/2` 函数自身返回一个元子 `ok`，如果一切都按计划正确执行的话。正如其它的 Erlang 函数一样，当发生错误的时候，它会崩溃掉。这并不是 Erlang 的问题，这是一种策略。Erlang 有一套强大的机制用于处理错误，我们会在下面展示。接下来，我们要做一个练习，我们试着让 `io:format` 崩溃，这不难。不过注意虽然 `io:format` 会崩溃，Erlang shell 本身并不会崩溃。

## 2.8 一个更大的例子

现在用一个更大的例子来巩固我们前面所学的知识。假定我们有一个来自世界上很多城市的温度列表。有一些是摄氏度，有一些是华氏度。首先我们把他们都转换成摄氏度，然后把数据清楚的输出出来。

```
%% This module is in file tut5.erl

-module(tut5).
-export([format_temps/1]).

%% Only this function is exported
format_temps([])->                                     % No output for an empty list
    ok;
format_temps([City | Rest]) ->
    print_temp(convert_to_celsius(City)),
    format_temps(Rest).

convert_to_celsius({Name, {c, Temp}}) -> % No conversion needed
    {Name, {c, Temp}};
convert_to_celsius({Name, {f, Temp}}) -> % Do the conversion
    {Name, {c, (Temp - 32) * 5 / 9}}.
```

```

print_temp({Name, {c, Temp}}) ->
    io:format("~-15w ~w c~n", [Name, Temp]).

36> c(tut5).
{ok,tut5}
37> tut5:format_temps([moscow, {c, -10}], {cape_town, {f, 70}},
    {stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          -10 c
cape_town       21.1111 c
stockholm       -4 c
paris           -2.22222 c
london          2.22222 c
ok

```

在我们看程序是如何运行的之前。注意一下我们在代码中加入了一些注释。注释以%开始，直到本行结束。同时也要注意-export([format\_temps/1]).一行只包含format\_temps/1函数，另一个函数是局部(local)函数，他们无法被tut5模块以外的东西访问。

注意，与我们在shell中测试程序一样，当一行过长的时候，我们把它们延到两行来写。

当我们第一次调用format\_temps的时候，City取得了值{moscow,{c,-10}}，Rest取得了列表中剩余的元素。于是我们调用函数

```
print_temp(convert_to_celsius({moscow,{c,-10}})).
```

这里我们看到，我们调用convert\_to\_celsius({moscow,{c,-10}})做为函数print\_temp的参数。当我们像这样嵌入(nest)函数的时候，它们是从内向外的执行。先执行convert\_to\_celsius({moscow,{c,-10}})并返回一个已经是摄氏温度的{moscow,{c,-10}}，然后我们执行print\_temp({moscow,{c,-10}})。函数convert\_to\_celsius的工作方式和我们前面例子中提到的convert\_length函数是一样的。

print\_temp简单的调用io:format，和我们上面所描述的一样。注意~-15w说明以15做为栏位长度打印，并左对齐。

现在我们调用format\_temp(Rest)，并以列表其余的部分做为参数。这个式形，就相当于其它语言中的循环结构。(是的，这是一个递归，不过不用担心)。因此，同一个format\_temps函数再次被调用，这一次，City取得{cape\_town,{f,70}}这个值，后

面同理，我们一直这样做，直到列表变成空[]，以导致第一个句块 `format_temp([])` 被匹配。它简单的返回元子 `ok`，程序结束。

## 2.9 变量的匹配、guard 和作用域

在列表中，找出最大的和最小的温度有时是很有用的。在扩展这个程序以做到这一点之前，让我们看看如何在列表中找出最大的元素：

```
-module(tut6).
-export([list_max/1]).

list_max([Head|Rest]) ->
    list_max(Rest, Head).

list_max([], Res) ->
    Res;
list_max([Head|Rest], Result_so_far) when Head > Result_so_far ->
    list_max(Rest, Head);
list_max([Head|Rest], Result_so_far) ->
    list_max(Rest, Result_so_far).

39> c(tut6).
{ok,tut6}
40> tut6:list_max([1,2,3,4,5,7,4,3,2,1]).
7
```

首先注意我们这里有两个同名的函数 `list_max`。虽然每个都带有不同数据的参数。在 Erlang 这些都会被认为是完全不同的函数。我们需要通过“名称/参数数量”区分这些我们写的函数，比如在这里是 `list_max/1` 和 `list_max/2`。

在这里我们遍历一个列表的并取得值，在这里用 `Result_so_far`。`list_max/1` 简单的假设最大值是列表中第一个元素，并且调用 `list_max/2`，把列表中剩余的值以及列表中第一个元素做为参数传给它，上面的例子，是

`list_max([2,3,4,5,7,4,3,2,1],1)`。如果我们试着使用 `list_max/1`，并带有一个空列表的，或试着使它带有一个非列表的东西做为参数，我们可能引发一个错误。注意，Erlang 中是不处理这样函数中的错误的，但是在其它地方可能这样做。我们在后面说明。

在 `list_max/2` 中，遍历列表，当 `Head > Result_so_far` 的时候，使用 `Head` 代替 `Result_so_far`。在 `->` 之前的特定字符，表示 -- 我们只会在这个特定的条件满足的时候，才会执行函数的这个部分。我们叫这种检测，叫 `guard(guard)`。如果 `guard` 不是‘真’

(我们称它为 guard 失败), 我们会尝试执行函数的下一个部分。这种情况下如果 Head 不大于 Result\_so\_far 的话, 它必定是小于等于它, 所以我们在下一部分中, 不需要再使用 guard。

一些 guard 中常见的操作符有 < 小于, > 大于, == 等于, >= 大于等于, <= 小于等于, /= 不等于。(see the chapter "Guard Sequences" in the Erlang Reference Manual)。

把上面的程序换成在列表中找到最小值的程序, 我们只需要把 '<' 替换成 '>'。(不过最好也把函数的名字, 改为 list\_min:-))。

还记得我前面所提到的 -- 一个变量在它的作用域中只能做一次赋值吗? 如上面所看到的, 如: Result\_so\_far 给赋了很多值。那是因为, 我们每次调用 list\_max/2 的时候, 建立了一个新的作用域, Result\_so\_far 在这个作用域中, 都会被认为是完全不同的变量。

另一个建立、或给一个变量赋值的方法是使用 =。如果写 M = 5, 那么一个名为 M 的变量会被创建, 并赋于 5 这个值。如果在同一下作用域下, 当我写 M = 6, 它就会发生错误。。

```
41> M = 5.
5
42> M = 6.
** exited: {{badmatch,6},{erl_eval,expr,3}} **
43> M = M + 1.
** exited: {{badmatch,6},{erl_eval,expr,3}} **
44> N = M + 1.
6
```

匹配操作符在通过一组元素创建新变量的时候很有用。

```
45> {X, Y} = {paris, {f, 28}}.
{paris,{f,28}}
46> X.
paris
47> Y.
{f,28}
```

这里我们看到 X 的值被赋于 paris, 而 Y 是 {f, 28}。

当然, 如果我们试着使用别的城市去做同样的操作, 我们会得到错误:

```
49> {X, Y} = {london, {f, 36}}.  
** exited: {{badmatch,{london,{f,36}}},{erl_eval,expr,3}} **
```

变量也可以用于提高程序的可读性，例如，上面的 `list_max/2` 函数，我们可以这样写：

```
list_max([Head|Rest], Result_so_far) when Head > Result_so_far ->  
    New_result_far = Head,  
    list_max(Rest, New_result_far);
```

这样能更清楚一些。

## 2.10 更多关于列表

记住，`|`可以用来得到列表中的第一个元素。

```
50> [M1|T1] = [paris, london, rome].  
[paris,london,rome]  
51> M1.  
paris  
52> T1.  
[london,rome]
```

`|`也可以用来在列表的头部添加元素：

```
53> L1 = [madrid | T1].  
[madrid,london,rome]  
54> L1.  
[madrid,london,rome]
```

现在举一个使用列表的例子 -- 反转列表：

```
-module(tut8).  
  
-export([reverse/1]).  
  
reverse(List) ->  
    reverse(List, []).  
  
reverse([Head | Rest], Reversed_List) ->  
    reverse(Rest, [Head | Reversed_List]);
```

```
reverse([], Reversed_List) ->
  Reversed_List.
```

```
56> c(tut8).
```

```
{ok,tut8}
```

```
57> tut8:reverse([1,2,3]).
```

```
[3,2,1]
```

考虑反转列表是如何建立的。在 [] 做为开始，我们成功的把头部从列表中取出，反转序放入 Reversed\_List 中，就像下面你所看到的：

```
reverse([1|2,3], []) =>
  reverse([2,3], [1|[]])
```

```
reverse([2|3], [1]) =>
  reverse([3], [2|[1]])
```

```
reverse([3|[]], [2,1]) =>
  reverse([], [3|[2,1]])
```

```
reverse([], [3,2,1]) =>
  [3,2,1]
```

模块 lists 包含了很多函数用于维护列表，例如反转列表。所以在我们写一个列表函数之前，最好先看一下是否在库中已经存在了这样一个函数。

现在让我们回过头来看城市的温度，让我们把它做得更结构化一些。首先让我们把整个列表转换成摄氏温度，并测试这个函数：

```
-module(tut7).
```

```
-export([format_temps/1]).
```

```
format_temps(List_of_cities) ->
  convert_list_to_c(List_of_cities).
```

```
convert_list_to_c([{Name, {f, F}} | Rest]) ->
  Converted_City = {Name, {c, (F - 32) * 5 / 9}},
  [Converted_City | convert_list_to_c(Rest)];
```

```
convert_list_to_c([City | Rest]) ->
```

```
[City | convert_list_to_c(Rest)];  
  
convert_list_to_c([]) ->  
[].
```

```
58> c(tut7).  
{ok, tut7}.  
59> tut7:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},  
  {stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).  
[{moscow, {c, -10}},  
  {cape_town, {c, 21.1111}},  
  {stockholm, {c, -4}},  
  {paris, {c, -2.22222}},  
  {london, {c, 2.22222}}]
```

一点一点的看这里:

```
format_temps(List_of_cities) ->  
  convert_list_to_c(List_of_cities).
```

这里看到的 `format_temps/1` 调用 `convert_list_to_c/1`。 `convert_list_to_c/1` 取出列表 `List_of_cities` 的头部，必要的话，把它转换成摄氏温度。|操作符用于把转换完的数，放到转换结束列表中:

```
[Converted_City | convert_list_to_c(Rest)];
```

或

```
[City | convert_list_to_c(Rest)];
```

我们继续这样的操作，直到达到列表尾部(即，空列表)

```
convert_list_to_c([]) ->  
[].
```

现在我们已经转换完了这个列表，我们加入一个打印函数来打印它:

```
-module(tut7).
```

```

-export([format_temps/1]).

format_temps(List_of_cities) ->
  Converted_List = convert_list_to_c(List_of_cities),
  print_temp(Converted_List).

convert_list_to_c([{Name, {f, F}} | Rest]) ->
  Converted_City = {Name, {c, (F - 32) * 5 / 9}},
  [Converted_City | convert_list_to_c(Rest)];

convert_list_to_c([City | Rest]) ->
  [City | convert_list_to_c(Rest)];

convert_list_to_c([]) ->
  [].

print_temp([{Name, {c, Temp}} | Rest]) ->
  io:format("~-15w ~w c~n", [Name, Temp]),
  print_temp(Rest);
print_temp([]) ->
  ok.

60> c(tut7).
{ok,tut7}
61> tut7:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
  {stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          -10 c
cape_town       21.1111 c
stockholm       -4 c
paris           -2.22222 c
london          2.22222 c
ok

```

我们现在需要加入一个函数，以便于找出最高温度和最低温度来。下面的程序遍历列表 4 次，并不是一个很有效的方法。不过我们还是力争让它简单、正确并在真正需要的时候，再让它工作得更有效率。

```

-module(tut7).
-export([format_temps/1]).

```

```

format_temps(List_of_cities) ->
    Converted_List = convert_list_to_c(List_of_cities),
    print_temp(Converted_List),
    {Max_city, Min_city} = find_max_and_min(Converted_List),
    print_max_and_min(Max_city, Min_city).

convert_list_to_c([{Name, {f, Temp}} | Rest]) ->
    Converted_City = {Name, {c, (Temp - 32)* 5 / 9}},
    [Converted_City | convert_list_to_c(Rest)];

convert_list_to_c([City | Rest]) ->
    [City | convert_list_to_c(Rest)];

convert_list_to_c([]) ->
    [].

print_temp([{Name, {c, Temp}} | Rest]) ->
    io:format("~-15w ~w c~n", [Name, Temp]),
    print_temp(Rest);
print_temp([]) ->
    ok.

find_max_and_min([City | Rest]) ->
    find_max_and_min(Rest, City, City).

find_max_and_min([{Name, {c, Temp}} | Rest],
    {Max_Name, {c, Max_Temp}},
    {Min_Name, {c, Min_Temp}}) ->
    if
        Temp > Max_Temp ->
            Max_City = {Name, {c, Temp}};           % Change
        true ->
            Max_City = {Max_Name, {c, Max_Temp}} % Unchanged
    end,
    if
        Temp < Min_Temp ->
            Min_City = {Name, {c, Temp}};           % Change
        true ->
            Min_City = {Min_Name, {c, Min_Temp}} % Unchanged
    end,
    find_max_and_min(Rest, Max_City, Min_City);

find_max_and_min([], Max_City, Min_City) ->

```

```

    {Max_City, Min_City}.

print_max_and_min({Max_name, {c, Max_temp}}, {Min_name, {c, Min_temp}}) ->
    io:format("Max temperature was ~w c in ~w~n", [Max_temp, Max_name]),
    io:format("Min temperature was ~w c in ~w~n", [Min_temp, Min_name]).

62> c(tut7).
{ok, tut7}
63> tut7:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
    {stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          -10 c
cape_town       21.1111 c
stockholm       -4 c
paris           -2.22222 c
london          2.22222 c
Max temperature was 21.1111 c in cape_town
Min temperature was -10 c in moscow
ok

```

## 2.11 If 和 Case

函数 `find_max_and_min` 解决了最高和最低温度的问题。我们已经介绍了一个新的结构 `if`。If 这样工作的：

```

if
    Condition 1 ->
        Action 1;
    Condition 2 ->
        Action 2;
    Condition 3 ->
        Action 3;
    Condition 4 ->
        Action 4
end

```

注意，在结尾没有";"! 条件(Conditions)和 `guard` 一样，测试成功或是失败。Erlang 会从最顶上开始向下，直到找到一个成功的条件为止，并执行行条件下面的动作，并且乎略掉其后面它的条件。如果没有条件匹配，会发生一个运行时(run-time)错误。元子 `true` 在条件中表示真，它常常被放在条件层，表示如果没有匹配的条件的话，应该做什么动作。

下面的小程序是一个 if 的例子。

```
-module(tut9).
-export([test_if/2]).

test_if(A, B) ->
  if
    A == 5 ->
      io:format("A = 5~n", []),
      a_equals_5;
    B == 6 ->
      io:format("B = 6~n", []),
      b_equals_6;
    A == 2, B == 3 ->                                     %i.e. A equals 2 and B
equals 3
      io:format("A == 2, B == 3~n", []),
      a_equals_2_b_equals_3;
    A == 1 ; B == 7 ->                                     %i.e. A equals 1 or B equals
7
      io:format("A == 1 ; B == 7~n", []),
      a_equals_1_or_b_equals_7
  end.
```

测试这个程序:

```
64> c(tut9).
{ok,tut9}
65> tut9:test_if(5,33).
A = 5
a_equals_5
66> tut9:test_if(33,6).
B = 6
b_equals_6
67> tut9:test_if(2, 3).
A == 2, B == 3
a_equals_2_b_equals_3
68> tut9:test_if(1, 33).
A == 1 ; B == 7
a_equals_1_or_b_equals_7
```

```

69> tut9:test_if(33, 7).
A == 1 ; B == 7
a_equals_1_or_b_equals_7
70> tut9:test_if(33, 33).

=ERROR REPORT==== 11-Jun-2003::14:03:43 ===
Error in process <0.85.0> with exit value:
{if_clause,[{tut9,test_if,2},{erl_eval,exprs,4},{shell,eval_loop,2}]}
** exited: {if_clause,[{tut9,test_if,2},
                        {erl_eval,exprs,4},
                        {shell,eval_loop,2}]} **

```

注意, `tut9:test_if(33,33)` 不会引发任何的条件判断为‘真’, 所以我们得到一个 `if_clause` 错误。(See the chapter "Guard Sequences" in the Erlang Reference Manual for details of the many guard tests available)。 `case` 是另一个 Erlang 结构。重新唤回我们写的 `convert_length` 函数:

```

convert_length({centimeter, X}) ->
    {inch, X / 2.54};
convert_length({inch, Y}) ->
    {centimeter, Y * 2.54}.

```

我们也可以这样写:

```

-module(tut10).
-export([convert_length/1]).

convert_length(Length) ->
    case Length of
        {centimeter, X} ->
            {inch, X / 2.54};
        {inch, Y} ->
            {centimeter, Y * 2.54}
    end.

71> c(tut10).
{ok,tut10}
72> tut10:convert_length({inch, 6}).
{centimeter,15.2400}
73> tut10:convert_length({centimeter, 2.5}).

```

```
{inch,0.98425}
```

注意，case 和 if 都有返回值，即，在上面的例子中 case 返回{inch,X/2.54}或{centimeter,Y\*2.54}。case 的行为也可以使用 guard 来代替。可能需要一个例子来澄清这一点。下面的例子告诉我们一个给定年份的某月份的长度。我们当然需要知道年份。而且要知道在闰年中，2 月有 29 天。

```
-module(tut11).
-export([month_length/2]).

month_length(Year, Month) ->
  %% All years divisible by 400 are leap
  %% Years divisible by 100 are not leap (except the 400 rule above)
  %% Years divisible by 4 are leap (except the 100 rule above)
  Leap = if
    trunc(Year / 400) * 400 == Year ->
      leap;
    trunc(Year / 100) * 100 == Year ->
      not_leap;
    trunc(Year / 4) * 4 == Year ->
      leap;
    true ->
      not_leap
  end,
  case Month of
    sep -> 30;
    apr -> 30;
    jun -> 30;
    nov -> 30;
    feb when Leap == leap -> 29;
    feb -> 28;
    jan -> 31;
    mar -> 31;
    may -> 31;
    jul -> 31;
    aug -> 31;
    oct -> 31;
    dec -> 31
  end.

74> c(tut11).
```

```
{ok,tut11}
75> tut11:month_length(2004, feb).
29
76> tut11:month_length(2003, feb).
28
77> tut11:month_length(1947, aug).
31
```

## 2.12 内建函数(BIFs)

内建函数(BIFs)是一个因为某原因被内嵌在 Erlang 虚拟机中的函数。BIFs 经常是一些在 Erlang 上无法实现的功能，或在 Erlang 下实现起来效率不高的东西。有些 BIFs 可以直接使用函数的名字，他们默认是属于 erlang 模块下的函数。比如 trunc 函数，和 erlang:trunc 调用是等同的。

正如你所见，我们首先查出一个年份是不是闰年。如果可以被 400 速除，它就是闰年。所以我们首先除 400，并使用内建函数 trunc，去掉小数部分。我们接着乘以 400，看看与原值是否相同。如，2004 年

```
2004 / 400 = 5.01
trunc(5.01) = 5
5 * 400 = 2000
```

我们可以看出，我们得到 2000 这个结果，与 2004 不相等，所以 2004 不可以被 400 整除。2000 年的话：

```
2000 / 400 = 5.0
trunc(5.0) = 5
5 * 400 = 2000
```

所以我们得到一个闰年。下面两个例子，年份以同样的方法除以 100 或 4。第一个 if 返回 leap 或 net\_leap 并提并给 Leap 变量。我们在 guard 中使用这个变量的值 feb 提交给下面的 case，它会告诉我们月份的长度。

这个例子展示了 trunc 的用法，另一个更简单的方法是使用 Erlang 操作符 rem，它将给出两数相除的余数。如：

```
2> 2004 rem 400.
```

我们可以改写:

```
trunc(Year / 400) * 400 == Year ->
    leap;
```

改写成:

```
Year rem 400 == 0 ->
    leap;
```

还有很多像 `trunc` 这样的内建函数。只有很少一部分的内建函数，可以用于 `guard`，你也无法在 `guard` 中使用你自己定义的函数。(see the chapter "Guard Sequences" in the Erlang Reference Manual)。我们在 shell 中试一下这一小部分的函数:

```
78> trunc(5.6).
5
79> round(5.6).
6
80> length([a,b,c,d]).
4
81> float(5).
5.00000
82> is_atom(hello).
true
83> is_atom("hello").
false
84> is_tuple({paris, {c, 30}}).
true
85> is_tuple([paris, {c, 30}]).
false
```

所有这些可以用于 `guard`。现在我们看看不能用于 `guard` 的函数:

```
87> atom_to_list(hello).
"hello"
88> list_to_atom("goodbye").
goodbye
89> integer_to_list(22).
"22"
```

这3个内建函数，很难(不可能)在Erlang中实现。

## 2.13 复杂函数

Erlang，像很多主流的函数程序语言一样，有一个复杂函数定义。我们以一个shell例子来说明：

```
90> Xf = fun(X) -> X * 2 end.  
#Fun<erl_eval.5.123085357>  
91> Xf(5).  
10
```

当我们定义了一个求一个数倍数的函数，并把这个函数赋给了一个变量。这个Xf(5)返回值10。下面定义了两个有用的操作列表的函数foreach和map：

```
foreach(Fun, [First|Rest]) ->  
    Fun(First),  
    foreach(Fun, Rest);  
foreach(Fun, []) ->  
    ok.  
  
map(Fun, [First|Rest]) ->  
    [Fun(First)|map(Fun,Rest)];  
map(Fun, []) ->  
    [].
```

这两个函数在标准模块list中有提供。foreach带入一下列表，并把每一个元素，作用于一个函数fun中，map通过一个函数fun来创立出一个新的列表。我们回过头来看shell，我们从使用map和一个fun函数来添加3到每一个元素中开始：

```
92> Add_3 = fun(X) -> X + 3 end.  
#Fun<erl_eval.5.123085357>  
93> lists:map(Add_3, [1,2,3]).  
[4,5,6]
```

现在让我们打印出一个关于城市温度的列表：

```
95> Print_City = fun({City, {X, Temp}}) -> io:format("~-15w ~w ~w~n",
```

```
[City, X, Temp]) end.
#Fun<erl_eval.5.123085357>
96> lists:foreach(Print_City, [{moscow, {c, -10}}, {cape_town, {f, 70}},
  {stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          c -10
cape_town       f 70
stockholm       c -4
paris           f 28
london          f 36
ok
```

我们现在定义一个 fun 函数，用来遍历城市温度这个列表，并把他们都转换成摄氏温度。

```
-module(tut13).
-export([convert_list_to_c/1]).

convert_to_c({Name, {f, Temp}}) ->
  {Name, {c, trunc((Temp - 32) * 5 / 9)}};
convert_to_c({Name, {c, Temp}}) ->
  {Name, {c, Temp}}.

convert_list_to_c(List) ->
  lists:map(fun convert_to_c/1, List).

98> tut13:convert_list_to_c([{moscow, {c, -10}}, {cape_town, {f, 70}},
  {stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
[{moscow,{c,-10}},
 {cape_town,{c,21}},
 {stockholm,{c,-4}},
 {paris,{c,-2}},
 {london,{c,2}}]
```

convert\_to\_c 函数，和以前定义的一样，不过我们把它做为一个带入函数来用：

```
lists:map(fun convert_to_c/1, List)
```

当我们使用一个要别处定义的 fun 函数的话，我们可以能过函数名/参数个数，来区别。所以在 map 调用中，我们写 lists:map(fun convert\_to\_c/1, List)。如你所见 convert\_list\_to\_c 变得更简洁和容易理解了。

标准模块 lists 也包含一个函数 sort(Fun, List)，Fun 是一个带入函数，并带有两个参数。如果第一个参数比第二个参数小，这个函数应该返回 true，反之返回 false。我们

在 `convert_list_to_c` 中加入排序:

```
-module(tut13).

-export([convert_list_to_c/1]).

convert_to_c({Name, {f, Temp}}) ->
    {Name, {c, trunc((Temp - 32) * 5 / 9)}};
convert_to_c({Name, {c, Temp}}) ->
    {Name, {c, Temp}}.

convert_list_to_c(List) ->
    New_list = lists:map(fun convert_to_c/1, List),
    lists:sort(fun({_, {c, Temp1}}, {_, {c, Temp2}}) ->
                Temp1 < Temp2 end, New_list).

99> c(tut13).
{ok,tut13}
100> tut13:convert_list_to_c([{moscow, {c, -10}}, {cape_town, {f, 70}},
    {stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
[  
  {moscow,{c,-10}},  
  {stockholm,{c,-4}},  
  {paris,{c,-2}},  
  {london,{c,2}},  
  {cape_town,{c,21}}]
```

在 `sort` 函数中, 我们使用的 `fun` 函数:

```
fun({_, {c, Temp1}}, {_, {c, Temp2}}) -> Temp1 < Temp2 end,
```

这里, 我们介绍匿名变量“\_”。这是一个简单化的, 用于一个将得到值的变量, 并且我们还需要丢弃这个值的情况下。它可以用在任何合适的场合中, 不只在 `fun` 中使用。 `Temp1 < Temp2` 返回 `true` 如果 `Temp1` 小于 `Temp2`。

## 3. 并行编程

### 3.1 进程

一个使用 Erlang 来代替其它面向函数的语言的主要原因是，Erlang 可以处理并发和分布式编程。并发，是指一种可以‘若干线程同时执行’这样的程序。比如，现在的操作系统都允许你同时使文字处理，表格处理，或邮箱客户端以及打印任务，并而让他们同一时刻运行。当然，每个处理器可能只能处理一个任务，不过它可以快速不停的在任务间切换，这给了我们一种错觉，认为他们在同时运行。在 Erlang 中，可以很容易建立出并发的线程，并允许这些线程间互相通信。在 Erlang 中，我们称每个线程为一个进程(process)。

(“进程”经常指线程间没有共享数据，“线程”指他们之间，有共享的数据或共享内存区)

Erlang 内建函数 spawn 被用于建立一个新的进程：spawn(模块，导出的函数，参数列表)。考虑下面的模块：

```
-module(tut14).
-export([start/0, say_something/2]).

say_something(What, 0) ->
    done;
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(tut14, say_something, [hello, 3]),
    spawn(tut14, say_something, [goodbye, 3]).

5> c(tut14).
{ok,tut14}
6> tut14:say_something(hello, 3).
hello
hello
hello
done
```

我们可以看到函数 say\_something 的功能是输出它的第一个参数，次数是由第二个参数指定的。现在看函数 start。它启动两个 Erlang 进程，一个输出 3 次“hello”，另一个输出 3 次“goodbye”。这两个进程使用函数 say\_something。注意一个函数要想用于 spawn 启动一个进程的话，必须先从模块中导出(export)才行。

```
9> tut14:start().
hello
goodbye
<0.63.0>
hello
goodbye
hello
goodbye
```

注意，它并没有输出“hello”3次，然后再输出3次“goodbye”，而是第一个进程输出“hello”，然后第二个进程输出“goodbye”，然后第一个进程再输出“hello”，依次类推。不过哪里出来的<0.63.0>？一个函数的输出当然是内部最后一个东西的输出。在start函数中，最后一个东西是spawn(tut14, say\_something, [goodbye, 3])。

spawn返回一个进程ID，或pid，即一个进程的唯一标识。所以<0.63.0>是spawn函数的pid，即进程ID。我们将在下面的例子中看到如何使用这个pid。

注意，我们在io:format中使用~p代替了~w。这里引用一下用户手册：“~p和~w一样输出数据，和~w的语法也是一样的，不过它会把一行很长的数据，分成若干行，以使所有数据可见。它也会检测列表中的可打印字符，并把他们以字符串形式输出”。

### 3.2 信息传递

下面的例子中，我们建立了两个互相传递信息的两个进程。

```
-module(tut15).

-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).
```

```

pong() ->
  receive
    finished ->
      io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
      io:format("Pong received ping~n", []),
      Ping_PID ! pong,
      pong()
  end.

start() ->
  Pong_PID = spawn(tut15, pong, []),
  spawn(tut15, ping, [3, Pong_PID]).

1> c(tut15).
{ok,tut15}
2> tut15:start().
<0.36.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
Pong finished

```

函数 start 建立了一个进程，我们叫它“pong”：

```
Pong_PID = spawn(tut15, pong, [])
```

进程执行 tut15:pong()。Pong\_PID 是“pong”的进程 ID。然后函数 start 又建立了另一个进程“ping”。

```
spawn(tut15, ping, [3, Pong_PID]),
```

这个进程执行：

```
tut15:ping(3, Pong_PID)
```

<0.36.0>是 start 函数的返回值。

现在“pong”执行：

```
receive
  finished ->
    io:format("Pong finished~n", []);
  {ping, Ping_PID} ->
    io:format("Pong received ping~n", []),
    Ping_PID ! pong,
    pong()
end.
```

receive 构造用于允许进程等待来自其它进程的消息。它的格式是：

```
receive
  pattern1 ->
    actions1;
  pattern2 ->
    actions2;
  ....
  patternN
    actionsN
end.
```

注意，在结尾没有“;”。

在 Erlang 进程间的消息是一个有效的 Erlang 类型，即，它们可以是列表、元组、整数、元子、pid 等等。

每个进程对于它收到的消息，有一个自己的输入队列。新的消息会放在队列的尾部。当一个进程执行了一个 receive，队列中第一个与匹配条件一致的消息就会从队列中移除，并且匹配的动作会执行。

如果第一个匹配式没有被匹配的话，第二个就会被检验，如果与队列中条件匹配了的话，则第二个式子中的动作会被执行。第三个也依次类推。如果没有匹配的话，那么第一个消息就会被放回队列中，并用第二个消息来做对应的操作。如果第二个消息匹配的话，相应的动作就会被执行，并把第二个消息从队列中移除（并保留第一个消息和它的消息在队列中）。如果第二个消息也不匹配，我们会试着用第三个消息，直到达到队列尾。如果到达到队列尾

部，进程就会阻塞(中止执行)并等待，直到一个新的消息被收到，上面的过程重复。

当然，Erlang 实现得很“聪明”，它会保证最少的对比次数。

现在我们回过头来看 ping pong 的例子。

“Pong”正在等待消息。如果元子 finished 被收到，“pong”输出“Pong finished”，并不做其它别的事，终止掉了。如果它收到一个下面格式的消息：

```
{ping, Ping_PID}
```

它会输出“Pong received ping”并向进程“ping”发送 pong 消息：

```
Ping_PID ! pong
```

注意，“!”是如何发送消息的，以及“!”的句法：

```
Pid ! Message
```

即，Message(任何东西)被发到给了标识为 Pid 的进程。

向进程“ping”发送 pong 之后，“pong”再次调用 pong 函数，使程序再次返回 receive 并等待再次等待消息发来。现在让我们看看进程“ping”。它以执行下面的函数开始：

```
tut15:ping(3, Pong_PID)
```

我们看函数 ping/2，我们看到 ping/2 的第二个句话为 3 的时候执行。

第二个句话发送一个消息给“pong”：

```
Pong_PID ! {ping, self()},
```

self()返回当前自在运行的进程 ID，在这里是“ping”的进程 ID。

“ping”现在等待“pong”的回应：

```
receive
```

```
pong ->
    io:format("Ping received pong~n", [])
end,
```

当回复收到的时候，输出“Ping received pong”，之后“ping”再次调用 ping 函数。

```
ping(N - 1, Pong_PID)
```

N-1 导致第一个参数递减，直到它变成 0。当这件事发生的时候，ping/2 的第一句话就会被执行。

```
ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);
```

元子 finished 被发送给了“pong”（导致它像下面所说的那个被终止）并而“ping finished”被输出。“Ping”会在无事可做的时候终止。

### 3.3 进程名称注册

在上面的例子中，我们首先创建了“pong”，所以我们可以给它一个名字“pong”，然后启动“ping”。即，ping 必须知道“pong”的标识，以使其可以向它发送消息。有些时候进程可能需要知道每一个与它不相关的启动的进程的标识。Erlang 在此提供一个机制，以此可以给进程一个名字，而不是 pid。这是由 register 这个内建函数完成的：

```
register(some_atom, Pid)
```

我们现在重写 ping pong 这个例子，使用这个函数，给 pong 进程一个名字“pong”：

```
-module(tut16).
-export([start/0, ping/1, pong/0]).

ping(0) ->
    pong ! finished,
    io:format("ping finished~n", []);

ping(N) ->
    pong ! {ping, self()},
    receive
        pong ->
```

```

        io:format("Ping received pong~n", [])
    end,
    ping(N - 1).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    register(pong, spawn(tut16, pong, [])),
    spawn(tut16, ping, [3]).

2> c(tut16).
{ok, tut16}
3> tut16:start().
<0.38.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
Pong finished

```

在 start/0 函数中,

```
regist(pong, spawn(tut16, pong, [])),
```

启动“pong”这个进程的同时，我们给予它了一些名字 pong。在“ping”进程中，我们现在可以这样向 pong 进程发送消息了：

```
pong ! {ping, self()},
```

所以 ping/2 现在变成了 ping/1，我们不再需要第一个参数 Pong\_PID 了。

### 3.4 分布式编程

现在我们重写 ping pong 这个程序，把“ping”和“pong”放不同的计算机上。在我们做这个工作之前，我们需要做一些设置，以使其工作。分布式 Erlang 的实现，提供了一个基本的安全机制以防止来自其它计算机的非授权的访问(\*manual\*)。Erlang 系统要想互相通信，必需要相同的 magic cookie。最简单的获取它的方法是在你的每台需要 Erlang 通信的机器中的 home 文件夹中建立一个 .erlang.cookie 的文件(在 Windows 系统中，home 文件夹由 \$HOME 环境变量指定 - 你可能需要首先设定它。在 Linux 或 Unix 系统中，你可以忽略这一步，只需要简单的在你用户 home 文件夹中，建立 .erlang.cookie 就可以了)。.erlang.cookie 文件中需要包含相同的元子字符串。如 Linux 或 Unix 系统中：

```
$ cd
$ cat > .erlang.cookie
this_is_very_secret
$ chmod 400 .erlang.cookie
```

上面的 chmod 使 .erlang.cookie 文件只可以被它的所有者访问。这是必需的。

当你启动一个需要与其它机器上的 Erlang 系统交互的一个 Erlang 系统时，你必须给它一个名字，如：

```
erl -sname my_name
```

我们后面将会看到更多的细节(\*manual\*)。如果你希望体验一下分布式 Erlang，可是你只有一机计算机的话，你只需要启动两个独立的 Erlang 系统在同一台机器上，并给他们不同的名字就可以了。每一个运行于计算机上的 Erlang 系统，被称为一个 Erlang 节点。

(注意：erl -sname 假设所有节点都在同一个 IP 域下，如果我们想要使用其它 IP 域上的节点，我们使用 -name 代替，并需要给出完全的 IP 地址(\*manual\*)。)

这里修改 ping pong，让他们运行于两个独立的节点中：

```
$ cd
$ cat > .erlang.cookie
this_is_very_secret
$ chmod 400 .erlang.cookie
```

让我们假设我们有两个计算机，分别叫 gollum 和 kosken。我们将在 kosken 上启动一个

节点，调用 ping，然后在 gollum 上调用 pong。

在 konsken 上(在 Linux/Unix 系统中):

```
kosken> erl -sname ping
Erlang (BEAM) emulator version 5.2.3.7 [hipe] [threads:0]

Eshell V5.2.3.7 (abort with ^G)
(ping@kosken)1>

On gollum:
gollum> erl -sname pong
Erlang (BEAM) emulator version 5.2.3.7 [hipe] [threads:0]

Eshell V5.2.3.7 (abort with ^G)
(pong@gollum)1>
```

现在我们在 gollum 上启动“pong”进程:

```
(pong@gollum)1> tut17:start_pong().
true
```

然后启动 konsken 上的“ping”进程(在上面的例子中，我们可以看出 start\_ping 的一个参数，是运行“pong”的节点的名字)。

```
(ping@kosken)1> tut17:start_ping(pong@gollum).
<0.37.0>
Ping received pong
Ping received pong
Ping received pong
ping finished
```

这里我们看到，ping pong 程序已经运行，在 pong 的一端，我们看到:

```
(pong@gollum)2>
Pong received ping
Pong received ping
Pong received ping
Pong finished
(pong@gollum)2>
```

再看 tut17 的代码，pong 的函数代码本身并没有改变：

```
{ping, Ping_PID} ->
  io:format("Pong received ping~n", []),
  Ping_PID ! pong,
```

他们以相同的方式工作，而不管“ping”进程是在哪个节点上执行。Erlang 的 pid 已经包含它了有关于进程运行的相关信息，所以你只要知道 pid，“!”就可以用于发送消息，无论目的地是在同一个节点上，还是其它节点上。

不同的是，我们如何把消息发送到另一个节点上注册的进程中：

```
{pong, Pong_Node} ! {ping, self()},
```

我们使用元组{register\_name, node\_name}代替 注册名。

在前面的例子中，我们从两个不同的节点上启动“ping”和“pong”。spawn 也可以用于在其它的节点上创建新进程。下一个例子还是 ping pong 程序，不过这次我们在另一个节点上启动“ping”：

```
-module(tut18).
-export([start/1, ping/2, pong/0]).

ping(0, Pong_Node) ->
  {pong, Pong_Node} ! finished,
  io:format("ping finished~n", []);

ping(N, Pong_Node) ->
  {pong, Pong_Node} ! {ping, self()},
  receive
    pong ->
      io:format("Ping received pong~n", [])
  end,
  ping(N - 1, Pong_Node).

pong() ->
  receive
    finished ->
      io:format("Pong finished~n", []);
```

```

    {ping, Ping_PID} ->
        io:format("Pong received ping~n", []),
        Ping_PID ! pong,
        pong()
    end.

start(Ping_Node) ->
    register(pong, spawn(tut18, pong, [])),
    spawn(Ping_Node, tut18, ping, [3, node()]).

```

假定已经在 kosken 节点上的 Erlang 系统中调用了 ping(不是“ping”进程), 在 gollum 上我们这样做:

```

(pong@gollum)1> tut18:start(ping@kosken).
<3934.39.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong finished
ping finished

```

注意, 我们中在 gollum 上得到的输出。这是因为 IO 系统可以查出, 我们在哪里调用的 spawn, 并把输出输信息出到那里。

### 3.5 一个更大的例子

现在我们写一个更大的例子。我们将做一个极其简单的聊天工具。

我们将允许客户端程序登入一个中心服务器, 并告知它在什么地方。即, 一个用户不需要知道消息来源的节点名字。

文件 messenger.erl:

```

%%% Message passing utility.
%%% User interface:
%%% logon(Name)
%%%     One user at a time can log in from each Erlang node in the
%%%     system messenger: and choose a suitable Name. If the Name

```

```

%%%   is already logged in at another node or if someone else is
%%%   already logged in at the same node, login will be rejected
%%%   with a suitable error message.
%%% logoff()
%%%   Logs off anybody at at node
%%% message(ToName, Message)
%%%   sends Message to ToName. Error messages if the user of this
%%%   function is not logged on or if ToName is not logged on at
%%%   any node.
%%%
%%% One node in the network of Erlang nodes runs a server which maintains
%%% data about the logged on users. The server is registered as "messenger"
%%% Each node where there is a user logged on runs a client process
registered
%%% as "mess_client"
%%%
%%% Protocol between the client processes and the server
%%% -----
%%%
%%% To server: {ClientPid, logon, UserName}
%%% Reply {messenger, stop, user_exists_at_other_node} stops the client
%%% Reply {messenger, logged_on} logon was successful
%%%
%%% To server: {ClientPid, logoff}
%%% Reply: {messenger, logged_off}
%%%
%%% To server: {ClientPid, logoff}
%%% Reply: no reply
%%%
%%% To server: {ClientPid, message_to, ToName, Message} send a message
%%% Reply: {messenger, stop, you_are_not_logged_on} stops the client
%%% Reply: {messenger, receiver_not_found} no user with this name logged on
%%% Reply: {messenger, sent} Message has been sent (but no guarantee)
%%%
%%% To client: {message_from, Name, Message},
%%%
%%% Protocol between the "commands" and the client
%%% -----
%%%
%%% Started: messenger:client(Server_Node, Name)
%%% To client: logoff
%%% To client: {message_to, ToName, Message}
%%%

```

```

%%% Configuration: change the server_node() function to return the
%%% name of the node where the messenger server runs

-module(messenger).
-export([start_server/0, server/1, logon/1, logoff/0, message/2,
client/2]).

%%% Change the function below to return the name of the node where the
%%% messenger server runs
server_node() ->
    messenger@bill.

%%% This is the server process for the "messenger"
%%% the user list has the format [{ClientPid1, Name1},{ClientPid22,
Name2},...]
server(User_List) ->
    receive
        {From, logon, Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
        {From, logoff} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        {From, message_to, To, Message} ->
            server_transfer(From, To, Message, User_List),
            io:format("list is now: ~p~n", [User_List]),
            server(User_List)
    end.

%%% Start the server
start_server() ->
    register(messenger, spawn(messenger, server, [[]])).

%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! {messenger, stop, user_exists_at_other_node}, %reject
logon
            User_List;
        false ->

```

```

        From ! {messenger, logged_on},
        [{From, Name} | User_List]          %add user to the list
    end.

%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).

%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
        false ->
            From ! {messenger, stop, you_are_not_logged_on};
            {value, {From, Name}} ->
                server_transfer(From, Name, To, Message, User_List)
    end.

%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
    case lists:keysearch(To, 2, User_List) of
        false ->
            From ! {messenger, receiver_not_found};
            {value, {ToPid, To}} ->
                ToPid ! {message_from, Name, Message},
                From ! {messenger, sent}
    end.

%%% User Commands
logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                spawn(messenger, client, [server_node(), Name]));
        _ -> already_logged_on
    end.

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->

```

```

case whereis(mess_client) of % Test if the client is running
    undefined ->
        not_logged_on;
    _ -> mess_client ! {message_to, ToName, Message},
        ok
end.

%%% The client process which runs on each server node
client(Server_Node, Name) ->
    {messenger, Server_Node} ! {self(), logon, Name},
    await_result(),
    client(Server_Node).

client(Server_Node) ->
    receive
        logoff ->
            {messenger, Server_Node} ! {self(), logoff},
            exit(normal);
        {message_to, ToName, Message} ->
            {messenger, Server_Node} ! {self(), message_to, ToName,
Message},
            await_result();
        {message_from, FromName, Message} ->
            io:format("Message from ~p: ~p~n", [FromName, Message])
    end,
    client(Server_Node).

%%% wait for a response from the server
await_result() ->
    receive
        {messenger, stop, Why} -> % Stop the client
            io:format("~p~n", [Why]),
            exit(normal);
        {messenger, What} -> % Normal response
            io:format("~p~n", [What])
    end.

```

为了使用这个程序，你需要：

- 配置 `server_node()` 函数
- 复制编译后的代码 (`messenger.beam`) 到每一台运行 Erlang 的计算机。

在下面的例子中，我在四个不同的计算机中启动了节点，不过如果你没有这个网络条件，你可以在同一台机器中运行它们。

我们启动 4 个 Erlang 节点，messenger@super，c1@bilbo，c2@kosken，c3@gollum。

我们分别启动它们并测试：

```
(messenger@super)1> messenger:start_server().
true

(c1@bilbo)1> messenger:logon(peter).
true
logged_on

(c2@kosken)1> messenger:logon(james).
true
logged_on

(c3@gollum)1> messenger:logon(fred).
true
logged_on

(c1@bilbo)2> messenger:message(fred, "hello").
ok
sent
```

Fred 收到消息，并发送一个消息给 Peter 并登出：

```
(c3@gollum)2> messenger:message(peter, "go away, I'm busy").
ok
sent
(c3@gollum)3> messenger:logoff().
logoff
```

James 一在试着发送一个消息给 Fred：

```
(c2@kosken)2> messenger:message(fred, "peter doesn't like you").
ok
```

receiver\_not\_found

不过失败了，因为 Fred 已经登出。

首先让我们看一些新概念。

server\_transfer 函数有两个版本，一个有 4 个参数(server\_transfer/4)，另一个有 5 个(server\_transfer/5)。它们被认为是两个不同的函数。

注意，我写了一个自我调用的服务器函数 server(User\_List)并构建起了一个循环。Erlang 编译器很聪明，并把他们实际编译成一个循环。不过这只适用于调用后面没有其它代码的情况下。这会造成进程越来越大。

我们使用了 lists 模块中的函数。这是一个非常有用的模块(erl -man lists)。lists:keymember(Key, Position, Lists)遍历列表，并看是否找到相同的键值的元组。第一个元组的位置为 1。如果它找到一个与键值相符的元组，它会返回 true。否则返回 false。

```
3> lists:keymember(a, 2, [{x,y,z},{b,b,b},{b,a,c},{q,r,s}]).
true
4> lists:keymember(p, 2, [{x,y,z},{b,b,b},{b,a,c},{q,r,s}]).
false
```

lists:keydelete 以相同的方式运行，但删掉第一个匹配的元组，并返回剩余的列表：

```
5> lists:keydelete(a, 2, [{x,y,z},{b,b,b},{b,a,c},{q,r,s}]).
[{x,y,z},{b,b,b},{q,r,s}]
```

lists:keysearch 和 lists:keymember 很像，不过它返回{value, Tuple\_Found} 或元子 false。

在 lists 模块中，有很多有用的函数。

一个 Erlang 进程理论上将一直运行，直到收到不需要的信息。之所以我说“理论上”因为 Erlang 系统与活动进程运享 CPU 时间。

当没有事可做的时候，一个进程会终止，即，最后一个函数被调用，并返回一个值，而且不再调用其它函数了。另一个中止进程的方法是使用 exit/1。exit/1 的参数有特定的含意，

我们后面再说。在这个例子中，我们使用了 `exit(normal)`。

内建函数 `whereis(RegisteredName)` 检测如果一个注册的进程名称叫 `RegisteredName` 存在，如果存在，返回 `pid`。否则返回 `undefined`。

你现在应该了解了上面代码的大体意思了。所以我将只说明一点：一个消息是如果从一个用户发送给另一个用户的。

每一个用户是这样发送消息的：

```
messenger:message(fred, "hello")
```

之后，测试客户端进程是否存在：

```
whereis(mess_client)
```

一个消息被发送给 `mess_client`：

```
mess_client ! {message_to, fred, "hello"}
```

客户端这样向服务器发消息：

```
{messenger, messenger@super} ! {self(), message_to, fred, "hello"},
```

等待一个服务端发来的回复。

服务端收到消息并调用：

```
server_stransfer(From, fred, "hello", User_List),
```

它将检测 `pid` 是否在用户列表中：

```
lists:keysearch(From, 1, User_List)
```

如果 `keysearch` 返回元子 `false`，会出现一些错误，服务端会回复消息：

```
From ! {messenger, stop, you_are_not_logged_on}
```

客户端就会收到这个消息，并调用 `exit(normal)` 并终止。如果 `keysearch` 返回 `{value, {From, Name}}` 我们就知道用户登录了，它的用户名 (`peter`) 是变量 `Name` 的值。现在我们调用：

```
server_transfer(From, peter, fred, "hello", User_List)
```

注意，这里的 `server_transfer/5` 和前面的函数 `server_transfer/4` 不是一个函数。我们做了另一个 `keysearch` 以找了客户端的 `pid`。

```
lists:keysearch(fred, 2, User_List)
```

这次，我们使用参数 2，即在元组中的第二个元素。如果返回 `false`，我们知道 `fred` 没有登录，然后我们发送消息：

```
From ! {messenger, receiver_not_found};
```

如果 `keysearch` 返回 `{value, {ToPid, fred}}` 的话，发送消息：

```
ToPid ! {message_from, peter, "hello"},
```

发送到 `fred` 的客户端，并且发：

```
From ! {messenger, sent}
```

到 `peter` 的客户端。

`Fred` 的客户端收到消息，打印：

```
{message_from, peter, "hello"} ->  
io:format("Message from ~p: ~p~n", [peter, "hello"])
```

`peter` 的客户端在 `await_result` 函数中收到消息。

## 4. 健壮性(Robustness 鲁棒性)

(译者注：不知道最早是什么人把Robustness音译成鲁棒，在我的观点中，类似的科技术语翻译要么不译保持英文，要么用合适的方式意译，类似鲁棒这样拙劣的音译是最不可取的。)

在上一章的 messenger 例子中，存在很多不合理的部分。例如，如果一个包含有已登录的用户的节点未正常退出登录就被关闭。用户的信息将保留在服务器的 User\_List 中。虽然客户端已经不存在了，服务器却仍然认为用户已经登录，这使得用户无法再次登录到服务器。

我们再看看另一种情况，如果服务器在消息发送的过程中退出，发送的客户端将挂起在 await\_result 函数中。

### 4.1 超时(Timeouts)

在进一步完善 messenger 程序之前，我们先看看一些我们应该遵循的原则，这里我们使用 ping pong 程序作为例子。回忆一下，当"ping"结束的时候，它将通过发送一个 atom finished 消息给"pong"，告诉"pong"它的工作已经完成，然后"pong"结束。另一种让"pong"结束的方式是让"pong"在一段时间内没有接受到来自 ping 的消息就退出，这可以通过在 pong 中引入超时机制来实现，如下所示：

```
-module(tut19).  
  
-export([start_ping/1, start_pong/0, ping/2, pong/0]).  
  
ping(0, Pong_Node) ->  
    io:format("ping finished~n", []);  
  
ping(N, Pong_Node) ->  
    {pong, Pong_Node} ! {ping, self()},  
    receive  
        pong ->  
            io:format("Ping received pong~n", [])  
    end,  
    ping(N - 1, Pong_Node).  
  
pong() ->  
    receive  
        {ping, Ping_PID} ->  
            io:format("Pong received ping~n", []),
```

```

        Ping_PID ! pong,
        pong()
    after 5000 ->
        io:format("Pong timed out~n", [])
    end.

start_pong() ->
    register(pong, spawn(tut19, pong, [])).

start_ping(Pong_Node) ->
    spawn(tut19, ping, [3, Pong_Node]).

```

在编译并且复制 tut19.beam 文件到合适的目录之后：  
在(pong@kosken)上：

```

(pong@kosken)1> tut19:start_pong().
true
Pong received ping
Pong received ping
Pong received ping
Pong timed out

```

在(ping@gollum)上：

```

(ping@gollum)1> tut19:start_ping(pong@kosken).
<0.36.0>
Ping received pong
Ping received pong
Ping received pong
ping finished

```

超时设置在这段代码中：

```

pong() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    after 5000 ->
        io:format("Pong timed out~n", [])
    end.

```

当我们进入 receive 代码段时，启动了超时机制(after 5000) 如果接受到了消息

{ping, Ping\_PID} 超时将被取消，如果没有收到该消息，在 5000 毫秒之后，超时代码段的操作将被执行。after 语句必须是 receive 代码段的最后一个匹配条件。就是说，让 receive 代码段中的其他匹配定义优先处理。当然，我们也可以使用一个返回一个整数的函数来作为超时的设定值：

```
after pong_timeout() ->
```

通常，在分布式 Erlang 系统的监控部分使用超时机制是一种很好的处理方式。超时非常适合在监控外部事件的场合被使用，比方说你希望在一个指定的时间内从某个外部系统接收到一个消息。例如，我们可以使用超时机制来使得一个十分钟内都没有访问系统的用户退出登录。

## 4.2 错误处理

在我们进一步了解 Erlang 系统的监控和错误处理方式之前，我们首先看看 Erlang 进程是如何退出的，这在 Erlang 的术语中被称为 exit。

一个进程如果使用 exit(normal) 退出或者运行完所有的代码然后退出称为正常 (normal) 退出。

一个进程如果发生了运行时错误(例如 除以 0，错误的匹配，试图调用一个不存在的函数等) 将因为错误而结束，也就是异常 (abnormal) 退出。一个进程执行 exit(Reason) (\*manual\*)，Reason 为一个除开 atom normal 之外的 term，也被看作一个异常退出。

一个 Erlang 的进程可以设置到其他进程的链接 (link)。进程通过调用 link(Other\_Pid) (\*manual\*) 来建立自身和 Other\_Pid 进程的双向链接。当一个进程中止的时候，将给每一个与它建立了链接的进程发送一个信号 (signal)。

这个信号包含了发送者的 pid 和进程结束的原因。

缺省情况下一个进程在将忽略接收到的正常退出信号。

而对于另外两种情况(异常退出)，缺省处理将忽略所有传递给接收进程(接收到异常退出信号的进程)的消息并且杀死(kill)该进程，并且将同样的错误信号传播给所有的和接收进程链接的进程。在这种方式下，你可以将一个事务(transaction)所涉及到的所有进程连接在一起，如果其中的一个进程异常退出，所有的该事务中的进程将被全部杀死。很多时候我们需要在创建一个进程的同时建立链接，内嵌函数 spawn\_link 在完成 spawn 函数功能的同时建立了一个与新创建的进程的链接(\*manual\*)。

下面是一个通过链接来结束"pong"的 ping pong 例子：

```
-module(tut20).  
  
-export([start/1, ping/2, pong/0]).  
  
ping(N, Pong_Pid) ->  
    link(Pong_Pid),
```

```

ping1(N, Pong_Pid).

ping1(0, _) ->
    exit(ping);

ping1(N, Pong_Pid) ->
    Pong_Pid ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping1(N - 1, Pong_Pid).

pong() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start(Ping_Node) ->
    PongPID = spawn(tut20, pong, []),
    spawn(Ping_Node, tut20, ping, [3, PongPID]).

(s1@bill)3> tut20:start(s2@kosken).
Pong received ping
<3820.41.0>
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong

```

我们做了一些小的改动，两个进程都在同一个 start/1 函数中被启动，其中 "ping" 线程在另外一个节点启动。注意内嵌函数 spawn\_link 的功能（译者注：上面的代码中我只看见了 link，没有看见 spawn\_link）。最后 "Ping" 调用了 exit(ping)，这使得 "Ping" 结束并且发送一个退出信号给 "pong" 然后 "Pong" 退出。我们可以修改一个进程接收到异常退出信号时的缺省退出行为，这时所有的信号都将被转换成一个格式为 {'EXIT', FromPID, Reason} 的普通消息并且被添加到消息队列中。我们通过以下语句来实现该功能：

```
process_flag(trap_exit, true)
```

还有很多种其他种类的进程标志位(flag)(\*manual\*)。这种改变进程缺省行为的方式在标准的用户程序中并不常见，更多的被用在OTP的监控(supervisory)程序中(在其他指南文档中介绍)。但我们还是将在下面的ping pong程序中演示退出信号(exit)的trapping。

```
-module(tut21).

-export([start/1, ping/2, pong/0]).

ping(N, Pong_Pid) ->
    link(Pong_Pid),
    ping1(N, Pong_Pid).

ping1(0, _) ->
    exit(ping);

ping1(N, Pong_Pid) ->
    Pong_Pid ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping1(N - 1, Pong_Pid).

pong() ->
    process_flag(trap_exit, true),
    pong1().

pong1() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong1();
        {'EXIT', From, Reason} ->
            io:format("pong exiting, got ~p~n", [{'EXIT', From, Reason}])
    end.

start(Ping_Node) ->
    PongPID = spawn(tut21, pong, []),
    spawn(Ping_Node, tut21, ping, [3, PongPID]).
```

```
(s1@bill)1> tut21:start(s2@gollum).
<3820.39.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
pong exiting, got {'EXIT',<3820.39.0>,ping}
```

### 4.3 增强健壮性之后的大型例子

现在，我们回到messenger程序，进行一些修改使之更加健壮：

```
%%% Message passing utility.
%%% User interface:
%%% login(Name)
%%%     One user at a time can log in from each Erlang node in the
%%%     system messenger: and choose a suitable Name. If the Name
%%%     is already logged in at another node or if someone else is
%%%     already logged in at the same node, login will be rejected
%%%     with a suitable error message.
%%% logoff()
%%%     Logs off anybody at at node
%%% message(ToName, Message)
%%%     sends Message to ToName. Error messages if the user of this
%%%     function is not logged on or if ToName is not logged on at
%%%     any node.
%%%
%%% One node in the network of Erlang nodes runs a server which maintains
%%% data about the logged on users. The server is registered as "messenger"
%%% Each node where there is a user logged on runs a client process
registered
%%% as "mess_client"
%%%
%%% Protocol between the client processes and the server
%%% -----
%%%
%%% To server: {ClientPid, logon, UserName}
%%% Reply {messenger, stop, user_exists_at_other_node} stops the client
%%% Reply {messenger, logged_on} logon was successful
%%%
%%% When the client terminates for some reason
```

```

%%% To server: {'EXIT', ClientPid, Reason}
%%%
%%% To server: {ClientPid, message_to, ToName, Message} send a message
%%% Reply: {messenger, stop, you_are_not_logged_on} stops the client
%%% Reply: {messenger, receiver_not_found} no user with this name logged on
%%% Reply: {messenger, sent} Message has been sent (but no guarantee)
%%%
%%% To client: {message_from, Name, Message},
%%%
%%% Protocol between the "commands" and the client
%%% -----
%%%
%%% Started: messenger:client(Server_Node, Name)
%%% To client: logoff
%%% To client: {message_to, ToName, Message}
%%%
%%% Configuration: change the server_node() function to return the
%%% name of the node where the messenger server runs

-module(messenger).
-export([start_server/0, server/0,
        logon/1, logoff/0, message/2, client/2]).

%%% Change the function below to return the name of the node where the
%%% messenger server runs
server_node() ->
    messenger@super.

%%% This is the server process for the "messenger"
%%% the user list has the format [{ClientPid1, Name1},{ClientPid22,
Name2},...]
server() ->
    process_flag(trap_exit, true),
    server([]).

server(User_List) ->
    receive
        {From, logon, Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
        {'EXIT', From, _} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
    end

```

```

        {From, message_to, To, Message} ->
            server_transfer(From, To, Message, User_List),
            io:format("list is now: ~p~n", [User_List]),
            server(User_List)
    end.

%%% Start the server
start_server() ->
    register(messenger, spawn(messenger, server, [])).

%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! {messenger, stop, user_exists_at_other_node}, %reject
logon
            User_List;
        false ->
            From ! {messenger, logged_on},
            link(From),
            [{From, Name} | User_List]           %add user to the list
    end.

%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).

%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
        false ->
            From ! {messenger, stop, you_are_not_logged_on};
        {value, {_, Name}} ->
            server_transfer(From, Name, To, Message, User_List)
    end.

%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
    case lists:keysearch(To, 2, User_List) of

```

```

        false ->
            From ! {messenger, receiver_not_found};
{value, {ToPid, To}} ->
            ToPid ! {message_from, Name, Message},
            From ! {messenger, sent}
end.

%%% User Commands
logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                spawn(messenger, client, [server_node(), Name]));
        _ -> already_logged_on
    end.

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->
    case whereis(mess_client) of % Test if the client is running
        undefined ->
            not_logged_on;
        _ -> mess_client ! {message_to, ToName, Message},
            ok
    end.

%%% The client process which runs on each user node
client(Server_Node, Name) ->
    {messenger, Server_Node} ! {self(), logon, Name},
    await_result(),
    client(Server_Node).

client(Server_Node) ->
    receive
        logoff ->
            exit(normal);
        {message_to, ToName, Message} ->
            {messenger, Server_Node} ! {self(), message_to, ToName,
Message},
            await_result();
        {message_from, FromName, Message} ->
            io:format("Message from ~p: ~p~n", [FromName, Message])
    end.

```

```

end,
client(Server_Node).

%%% wait for a response from the server
await_result() ->
receive
    {messenger, stop, Why} -> % Stop the client
        io:format("~p~n", [Why]),
        exit(normal);
    {messenger, What} -> % Normal response
        io:format("~p~n", [What])
after 5000 ->
    io:format("No response from server~n", []),
    exit(timeout)
end.

```

我们增加了以下的变化：

messenger 服务捕获 (trap) 退出信号。如果其接收到了一个退出信号

{'EXIT', From, Reason}, 这意味着一个客户进程已经中止并且不可到达，因为：

- 用户退出登录(我们去除了"退出登录"消息)，
- 到达该客户端的网络连接已经中端，
- 客户端所在的节点已关闭或者
- 客户进程进行了非法操作。

如果我们接收到了上述的退出信号， 我们使用 server\_logoff 函数从服务器的 User\_List 删除 tuple {From, Name}。 如果服务器所在的节点被关闭， 一个退出信号(系统自动生成) {'EXIT', MessengerPID, noconnection} 将被发送给所有的客户进程， 这将导致所有的客户进程中止。

在 await\_result 函数中， 我们使用了已经介绍过的 5 秒超时机制(5 秒内未接收到服务器的返回信息)来结束客户端。 注意我们仅仅在客户端登录完成之前， 客户端和服务器还未建立起链接的时候需要超时机制。

还有一个比较有意思的地方， 如果客户端在服务器与其建立链接之前中止了， 该异常仍然被有效的处理。 服务器将在试图与一个不存在的进程建立链接时收到一个退出信号 {'EXIT', From, noprocs}, 该信号将导致服务器采用与处理一个刚建立起链接就退出的客户端一样的方式来处理这个在建立链接前就退出的客户端。

## 5. 记录和宏(Records and Macros)

一个大型的程序经常写在一组文件中，这些文件的不同部分包含有各类定义好的接口。

### 5.1 将大型的例子分割在多个文件中

为了演示这一点，我们将前一章 messenger 例子分割在 5 个文件中。

mess\_config.hrl

包含配置信息的头文件

mess\_interface.hrl

定义了客户端和服务端通讯的接口

user\_interface.erl

用户接口函数的定义

mess\_client.erl

messenger 客户端函数的定义

mess\_server.erl

messenger 服务器端函数的定义

在完成该工作的同时，我们清理了 Shell 之间的消息传递接口，客户端和服务端将使用记录(records)来完成接口的定义。同时我们将介绍宏(macros)。

```
%%%----FILE mess_config.hrl----  
  
%%% Configure the location of the server node,  
-define(server_node, messenger@super).  
  
%%%----END FILE----
```

```
%%%---FILE mess_interface.hrl---  
  
%%% Message interface between client and server and client shell for  
%%% messenger program  
  
%%%Messages from Client to server received in server/1 function.  
-record(logon,{client_pid, username}).  
-record(message,{client_pid, to_name, message}).  
%%% {'EXIT', ClientPid, Reason} (client terminated or unreachable.  
  
%%% Messages from Server to Client, received in await_result/0 function  
-record(abort_client,{message}).  
%%% Messages are: user_exists_at_other_node,  
%%%                you_are_not_logged_on  
-record(server_reply,{message}).
```

```

%%% Messages are: logged_on
%%%                 receiver_not_found
%%%                 sent (Message has been sent (no guarantee)
%%% Messages from Server to Client received in client/1 function
-record(message_from,{from_name, message}).

%%% Messages from shell to Client received in client/1 function
%%% spawn(mess_client, client, [server_node(), Name])
-record(message_to,{to_name, message}).
%%% logoff

%%%-----END FILE-----

```

```

%%%-----FILE user_interface.erl-----

%%% User interface to the messenger program
%%% login(Name)
%%%     One user at a time can log in from each Erlang node in the
%%%     system messenger: and choose a suitable Name. If the Name
%%%     is already logged in at another node or if someone else is
%%%     already logged in at the same node, login will be rejected
%%%     with a suitable error message.

%%% logoff()
%%%     Logs off anybody at at node

%%% message(ToName, Message)
%%%     sends Message to ToName. Error messages if the user of this
%%%     function is not logged on or if ToName is not logged on at
%%%     any node.

-module(user_interface).
-export([logon/1, logoff/0, message/2]).
-include("mess_interface.hrl").
-include("mess_config.hrl").

logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                    spawn(mess_client, client, [?server_node, Name]));
        _ -> already_logged_on
    end

```

```

    end.

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->
    case whereis(mess_client) of % Test if the client is running
        undefined ->
            not_logged_on;
        _ -> mess_client ! #message_to{to_name=ToName, message=Message},
            ok
    end.

%%%----END FILE----

```

```

%%%----FILE mess_client.erl----

%%% The client process which runs on each user node

-module(mess_client).
-export([client/2]).
-include("mess_interface.hrl").

client(Server_Node, Name) ->
    {messenger, Server_Node} ! #logon{client_pid=self(), username=Name},
    await_result(),
    client(Server_Node).

client(Server_Node) ->
    receive
        logoff ->
            exit(normal);
        #message_to{to_name=ToName, message=Message} ->
            {messenger, Server_Node} !
                #message{client_pid=self(), to_name=ToName,
message=Message},
            await_result();
        {message_from, FromName, Message} ->
            io:format("Message from ~p: ~p~n", [FromName, Message])
    end,
    client(Server_Node).

```

```

%%% wait for a response from the server
await_result() ->
  receive
    #abort_client{message=Why} ->
      io:format("~p~n", [Why]),
      exit(normal);
    #server_reply{message=What} ->
      io:format("~p~n", [What])
  after 5000 ->
    io:format("No response from server~n", []),
    exit(timeout)
end.

%%%----END FILE----

```

```

%%%----FILE mess_server.erl----

%%% This is the server process of the messneger service

-module(mess_server).
-export([start_server/0, server/0]).
-include("mess_interface.hrl").

server() ->
  process_flag(trap_exit, true),
  server([]).

%%% the user list has the format [{ClientPid1, Name1},{ClientPid22,
Name2},...]
server(User_List) ->
  io:format("User list = ~p~n", [User_List]),
  receive
    #logon{client_pid=From, username=Name} ->
      New_User_List = server_logon(From, Name, User_List),
      server(New_User_List);
    {'EXIT', From, _} ->
      New_User_List = server_logoff(From, User_List),
      server(New_User_List);
    #message{client_pid=From, to_name=To, message=Message} ->
      server_transfer(From, To, Message, User_List),
      server(User_List)
  end.

```

```

%%% Start the server
start_server() ->
    register(messenger, spawn(?MODULE, server, [])).

%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! #abort_client{message=user_exists_at_other_node},
            User_List;
        false ->
            From ! #server_reply{message=logged_on},
            link(From),
            [{From, Name} | User_List]           %add user to the list
    end.

%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).

%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
        false ->
            From ! #abort_client{message=you_are_not_logged_on};
            {value, {_, Name}} ->
                server_transfer(From, Name, To, Message, User_List)
    end.

%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
    case lists:keysearch(To, 2, User_List) of
        false ->
            From ! #server_reply{message=receiver_not_found};
            {value, {ToPid, To}} ->
                ToPid ! #message_from{from_name=Name, message=Message},
                From ! #server_reply{message=sent}
    end.

%%%----END FILE---

```

## 5.2 头文件(Header Files)

你可以看见一些扩展名为.hrl的文件，这些都是Erlang的头文件，可以被.erl文件通过以下语句：

```
-include("File_Name").
```

包含在代码中，例如：

```
-include("mess_interface.hrl").
```

在这个例子中，头文件从存放其他的messenger例子的文件的目录中获取(\*manual\*)。 .hrl可以包含任何合法的Erlang代码，但在大多数情况下仅仅包含有记录和宏的定义。

## 5.3 记录(Records)

通过如下方式来定义一个记录：

```
-record(name_of_record,{field_name1, field_name2, field_name3, .....}).
```

例如：

```
-record(message_to,{to_name, message}).
```

它严格等效于：

```
{message_to, To_Name, Message}
```

下面的例子更好的演示了创建一个记录的过程：

```
#message_to{message="hello", to_name=fred}
```

这将创建：

```
{message_to, fred, "hello"}
```

注意，我们无需担心创建记录时给记录中的不同部分赋值的顺序。使用记录的好处就是你可以将记录的定义放在头文件中，以便于定义方便更改的接口。例如，如果你想在记录中增加一个新域，你只需要改变代码中使用这个新域的部分而不是修改每一个引用该记录的部分。如果在创建一个记录的时候忽略了某些域，这些域将被赋值为atom undefined。

(\*manual\*)

匹配记录与创建记录十分类似。例如在一个case或者receive代码段中：

```
#message_to{to_name=ToName, message=Message} ->
```

等价于

```
{message_to, ToName, Message}
```

## 5.4 宏(Macros)

另外一个被添加到 messenger 中的部分是宏(macro)。文件 mess\_config.hrl 包含了下面的定义:

```
%%% Configure the location of the server node,  
-define(server_node, messenger@super).
```

我们在 mess\_server.erl 文件中包含了这一头文件:

```
-include("mess_config.hrl").
```

mess\_server.erl 中出现的每一处 ?server\_node 都将被替换成 messenger@super。另外一个用到宏的地方在我们创建服务进程的代码里:

```
spawn(?MODULE, server, [])
```

这是一个标准宏(standard macro)(标准宏指系统定义的宏,而不是用户定义的宏)。?MODULE 每次都将被当前模块的名称(指文件起始处-module 代码定义的模块名称)替换。宏还有一些更高级的使用方式,例如作为参数(parameters)使用(\*manual\*)。

messenger 例子中三个 Erlang 的(.erl)文件将被编译成独立的(.beam)目标代码文件。Erlang 系统在执行时将这些文件装载进入系统并且完成。在这个例子中,我们只需要简单的将这些文件都放在我们的当前工作目录中(也就是你运行 cd 命令到达的目录)。还有一种方式是把所有的 .beam 文件保存在另一个目录中。

在 messenger 例子中,并没有假设被发送的信息有任何特征,该信息可以是任何合法的 Erlang term。

## 6. 实例

下面给出一个 Erlang 建立一个 WEB 服务器的实例,这样可以让我们更好的复习一下和我们学的基础知识,同时也可以扩展一下我们的知识。此代码由 ninhenry@gmail.com 发布于 LUPA 论坛。

```
%%% httpd.erl - MicroHttpd  
-module(httpd).  
-author("ninhenry@gmail.com").  
  
-export([start/0,start/1,start/2,process/2]).  
-import(regexp,[split/2]).
```

```

-define(defPort,6888).
-define(docRoot,"public").

start() -> start(?defPort,?docRoot).
start(Port) -> start(Port,?docRoot).
start(Port,DocRoot) ->
    case gen_tcp:listen(Port, [binary,{packet, 0},{active, false}]) of
        {ok, LSock} -> server_loop(LSock,DocRoot);
        {error, Reason} -> exit({Port,Reason})
    end.

%% main server loop - wait for next connection, spawn child to process it
server_loop(LSock,DocRoot) ->
    case gen_tcp:accept(LSock) of
        {ok, Sock} ->
            spawn(?MODULE,process,[Sock,DocRoot]),
            server_loop(LSock,DocRoot);
        {error, Reason} ->
            exit({accept,Reason})
    end.

%% process current connection
process(Sock,DocRoot) ->
    Req = do_recv(Sock),
    {ok,[Cmd|[Name|[Vers|_]]]} = split(Req,"[ \r\n]"),
    FileName = DocRoot ++ Name,
    LogReq = Cmd ++ " " ++ Name ++ " " ++ Vers,
    Resp = case file:read_file(FileName) of
        {ok, Data} ->
            io:format("~p ~p ok~n",[LogReq,FileName]),
            Data;
        {error, Reason} ->
            io:format("~p ~p failed ~p~n",[LogReq,FileName,Reason]),
            error_response(LogReq,file:format_error(Reason))
    end,
    do_send(Sock,Resp),
    gen_tcp:close(Sock).

%% construct HTML for failure message
error_response(LogReq,Reason) ->
    "<html><head><title>Request Failed</title></head><body>\n" ++
    "<h1>Request Failed</h1>\n" ++ "Your request to " ++ LogReq ++
    " failed due to: " ++ Reason ++ "\n</body></html>\n".

```

```
%% send a line of text to the socket
do_send(Sock,Msg) ->
    case gen_tcp:send(Sock, Msg) of
        ok      -> ok;
        {error, Reason} -> exit(Reason)
    end.

%% receive data from the socket
do_recv(Sock) ->
    case gen_tcp:recv(Sock, 0) of
        {ok, Bin} -> binary_to_list(Bin);
        {error, closed} -> exit(closed);
        {error, Reason} -> exit(Reason)
    end.
```

## 第二部分 OTP 设计原则

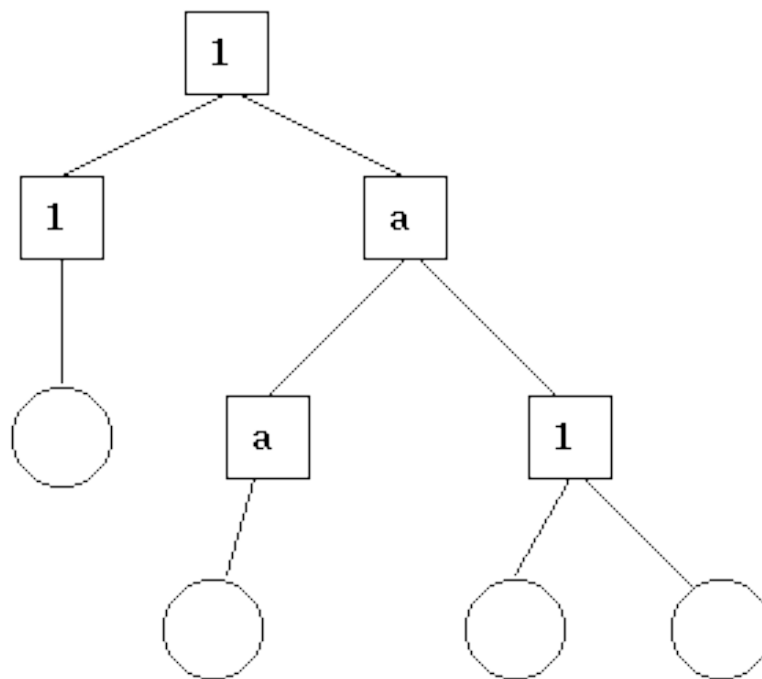
### 1. 概述

OTP 设计原则是一系列原则，用于指定你在一组进程、模块、和目录中构建组织 Erlang 代码。

#### 1.1 监督树

在 Erlang/OTP 中，最基本的概念就是监督树。这是一个基于工作者与监督者思想的进程结构模型。

- 工作者是一个进程，用于计算，这就是它的工作。
- 监督者是一个进程，用于监视工作者的行为。如果工作者出现问题了，一个监督者可以重启这个工作者。
- 监督树是一个代码的分层，它把代码分层为工作者和监督者，使编写高容错的程序成为可能。



监督树

上图中，方框提供监督，圆是工作者。

#### 1.2 Behaviour

在监督树中，很多进程有着相似的职能，他们按相似的原理工作。如，监督者的结构都很相似。唯一的不同是他们所监督的子进程不相同。此外，许多工作者可能是 C/S 架构中的服

务端、有限状态机、或像错误日志那样的事情处理器。

Behaviour，是一些通用的职能的形式化。主要思想是分离进程中的通用过程和特定过程。

Behaviour 模块是 Erlang/OTP 的一部分。为了使一个进程成为监督者，用户只需要实现回调模块，导出一个预定义的函数集合 -- 回调函数。

一个例子可以用来说明代码是如何被分离成为通用部分和特定部分的：考虑下面的代码（纯 Erlang 编写的），一个简单的保持跟踪一系列频道的服务器。另一个进程可以通过 alloc/0 和 free/1 函数，分别分配和释放一个频道。

```
-module(ch1).
-export([start/0]).
-export([alloc/0, free/1]).
-export([init/0]).

start() ->
    spawn(ch1, init, []).

alloc() ->
    ch1 ! {self(), alloc},
    receive
        {ch1, Res} ->
            Res
    end.

free(Ch) ->
    ch1 ! {free, Ch},
    ok.

init() ->
    register(ch1, self()),
    Chs = channels(),
    loop(Chs).

loop(Chs) ->
    receive
        {From, alloc} ->
            {Ch, Chs2} = alloc(Chs),
            From ! {ch1, Ch},
            loop(Chs2);
```

```
    {free, Ch} ->
        Chs2 = free(Ch, Chs),
        loop(Chs2)
end.
```

服务器，可以在 server.erl 中重写：

```
-module(server).
-export([start/1]).
-export([call/2, cast/2]).
-export([init/1]).

start(Mod) ->
    spawn(server, init, [Mod]).

call(Name, Req) ->
    Name ! {call, self(), Req},
    receive
        {Name, Res} ->
            Res
    end.

cast(Name, Req) ->
    Name ! {cast, Req},
    ok.

init(Mod) ->
    register(Mod, self()),
    State = Mod:init(),
    loop(Mod, State).

loop(Mod, State) ->
    receive
        {call, From, Req} ->
            {Res, State2} = Mod:handle_call(Req, State),
            From ! {Mod, Res},
            loop(Mod, State2);
        {cast, Req} ->
            State2 = Mod:handle_cast(Req, State),
            loop(Mod, State2)
    end.
```

还有一个回调模块 ch2.erl：

```

-module(ch2).
-export([start/0]).
-export([alloc/0, free/1]).
-export([init/0, handle_call/2, handle_cast/2]).

start() ->
    server:start(ch2).

alloc() ->
    server:call(ch2, alloc).

free(Ch) ->
    server:cast(ch2, {free, Ch}).

init() ->
    channels().

handle_call(alloc, Chs) ->
    alloc(Chs). % => {Ch,Chs2}

handle_cast({free, Ch}, Chs) ->
    free(Ch, Chs). % => Chs2

```

**注意:**

- server 中的代码，可以重用，用于建立很多不同的服务端。
- 例子中服务器的名字为 ch2，对于客户端的用户而言是隐藏的。这意味着名字可以被改变，而不影响原程序。
- 协议(用于服务器客户端间消息传递)也是隐藏的。这是很好的编程原则，允许我们在不改变主体代码的前提下，改变协议。
- 我们可以扩展服务器的功能，而不用改变 ch2 或任何其它的回调模块。(在上面的 ch1.erl 和 ch2.erl 中的 channels/0、alloc/1、free/2 的实现被有意的搁置了。因为这些功能不是本例所人说明的东西。我们会在下面的例子中完成它。注意，这只是个例子，真实的实现必须处理一些特殊情况)

```

channels() ->
    {_Allocated = [], _Free = lists:seq(1,100)}.

alloc({Allocated, [H|T] = _Free}) ->
    {H, {[H|Allocated], T}}.

free(Ch, {Alloc, Free} = Channels) ->

```

```
case lists:member(Ch, Alloc) of
  true ->
    {lists:delete(Ch, Alloc), [Ch|Free]};
  false ->
    Channels
end.
```

不使用 Behaviour 来写这种代码，可能会更有效率一些。不过提高效率，可能会在通用次上付出代价。统一化、可以管理所有应用程序，在系统中是很重要的。server 模块就显得很简单、一致。标准的 Erlang/OTP Behaviour 是：

gen\_server

提供 C/S 架构中的服务端的实现。

gen\_fsm

提供了有限状态机的实现。

gen\_event

提供事件处理功能。

supervisor

实现一个监督树中的监督者。编译器很清楚模块的性质 - 如果没有指定回调函数，就会给出警告。

如：

```
-module(chs3).
-behaviour(gen_server).
...

3> c(chs3).
./chs3.erl:10: Warning: undefined call-back function handle_call/3
{ok, chs3}
```

### 1.3 应用

Erlang/OTP 提供很多组件，每种组件实现一些特定的功能。组件在 Erlang/OTP 术语中称为“应用”。Erlang/OTP 应用的例子是 Mnesia，具备了所有用于数据库开发所必需的东西，Debugger 是一个用于调试的程序。基于 Erlang/OTP 的最小系统包括应用内核和标准库(STDLIB)。

应用这个概念即对应于程序结构(进程)又对应目录结构(模块)。

一个最简单的应用程序，并不包含任何进程，不过它包含一系列的功能模块。如一种被叫做“库应用”的应用程序。一个库应用的例子就是 STDLIB

一个带有进程的应用程序，是一个监督树中使用标准 Behaviour 的最简单的程序。

在 Application 一节点，描述了如何开发应用。

## 1.4 发布

发布指的是一个完整的系统、一个 Erlang/OTP 应用程序的一个子集，用于特定的用户目标开发的应用程序。

如果发布一个程序，会在 Release 一节中说明。

如果在目标机上安装一个发布的程序，在 Target Systems in System Principles 一节中说明。

## 1.5 发布控制

发布控制是从一个运行中的发行版中升级或降级。如何做到。请看 Release Handling 一节的描述。

## 2. Gen\_Server Behaviour(文档缺失)

## 3. Gen\_Fsm Behaviour

本章需要参考 gen\_fsm(3)，它说详细的说明了 gen\_fsm 的所有接口函数和回调函数。

### 3.1 有限状态机

有限状态机，FSM，可以描述成一种关系形式：

样态(S) x 事情(E) -> 动作(A), 状态(S')

这个关系可以这样理解：如果我们处于状态 S，并且事件 E 发生了，那到我们需要做 A 动作，并且状态 S 迁移至 S'

对于一个使用 gen\_fsm 实现的状态机而言，状态迁移规则被写成下面的转换函数：

```
StateName(Event, StateData) ->
```

```
.. code for actions here ...
{next_state, StateName', StateData'}
```

### 3.2 实例

一个门的密码锁可以看做一个状态机。起初，门是锁着的。每一次按键，就会生成一个事情。依赖于前一次按键，顺序可能是正确的、不完整的、错误的。

如果正确，门在 30 秒内解锁。

如果它是不完整的，我们等待下一次按键操作。

如果它是错误的，我们重新开始，待于下一轮的按键。

使用 gen\_fsm 有限状态机实现一个密码锁：

```
-module(code_lock).
-behaviour(gen_fsm).

-export([start_link/1]).
-export([button/1]).
-export([init/1, locked/2, open/2]).

start_link(Code) ->
    gen_fsm:start_link({local, code_lock}, code_lock, Code, []).

button(Digit) ->
    gen_fsm:send_event(code_lock, {button, Digit}).

init(Code) ->
    {ok, locked, {[], Code}}.

locked({button, Digit}, {SoFar, Code}) ->
    case [Digit|SoFar] of
        Code ->
            do_unlock(),
            {next_state, open, {[], Code}, 3000};
        Incomplete when length(Incomplete)<length(Code) ->
            {next_state, locked, {Incomplete, Code}};
        _Wrong ->
            {next_state, locked, {[], Code}};
    end.

open(timeout, State) ->
```

```
do_lock(),
{next_state, locked, State}.
```

我们在下一节再来解释这个代码。

### 3.3 启动一个 Gen\_Fsm

在前一节例子中，我们调用 `code_lock:start_link(Code)` 来启动 `gen_fsm`：

```
start_link(Code) ->
  gen_fsm:start_link({local, code_lock}, code_lock, Code, []).
```

`start_link` 调用函数 `gen_fsm:start_link/4`。这个函数启动并连接到一个新进程，一个 `gen_fsm`。

- 第一个参数，`{local, code_lock}` 指定名字。在这里，`gen_fsm` 将在本地做为 `code_lock` 来注册。如果名称被省略，`gen_fsm` 将不会被注册，`pid` 将被使用。名字也可以是 `{global, Name}`，这样 `gen_fsm` 可以使用 `global:register_name/2` 来注册。
- 第二个参数，`code_lock`，是回调模块的名称，模块位于回调函数所在的位置。这样，接口函数 (`start_link` 和 `button`)，和回调函数一样被放在与模块同一个地方 (`init`，`locked` 和 `open`)。这是一个很好的编程原则，使代码与模块下的一个进程对应。
- 第三个参数，`Code`，和传给 `init` 函数中的一样。在这里，`init` 取得有效的锁码，做为 `indata`。
- 第四个参数，`[]`，是一个列表选项。详细 `gen_fsm(3)`。

如果一个名字成功的注册了，新的 `gen_fsm` 进程调用回调函数 `code_lock:init(Code)`。这个函数会返回 `{ok, StateName, StateData}`，`StateName` 是 `gen_fsm` 起始状态的名称。`locked`，假设门是锁着的。`StateData` 是 `gen_fsm` 内部状态。(对于 `gen_fsm`，内部状态常常是 `'state data'` 以与状态机区别对待)，在这里，状态数据就是按键顺序的按下，并切为正常的开锁密码。

```
init(Code) ->
  {ok, locked, {[], Code}}.
```

注意，`gen_fsm:start_link` 是同步的。只到 `gen_fsm` 被初始化并且准备好接受通知时，它才返回。

gen\_fsm:start\_link 必须被使用，如果 gen\_fsm 是监督树的一部分。即，它是由监督者启动的。还有一个函数 gen\_fsm:start，它以独占方式启动 gen\_fsm，即，一个不属于监督树的 gen\_fsm。

### 3.4 事情通知

密码锁按钮事件的通知函数使用 gen\_fsm:send\_event/2 来实现的。

```
button(Digit) ->
  gen_fsm:send_event(code_lock, {button, Digit}).
```

code\_lock 是 gen\_fsm 的名字。{button, Digit} 是实际的事情。

事件被做成消息，并且发送到 gen\_fsm。发事件被收到的时候，gen\_fsm 调用 StateName(Event, StateData)，返回一个元组 {next\_state, StateName1, StateData1}。StateName 是当前状态的名字，StateName1 是下一个状态的名字。StateData1 是一个新值，做为 gen\_fsm 的状态数据被使用。

```
locked({button, Digit}, {SoFar, Code}) ->
  case [Digit|SoFar] of
    Code ->
      do_unlock(),
      {next_state, open, {[], Code}, 30000};
    Incomplete when length(Incomplete)<length(Code) ->
      {next_state, locked, {Incomplete, Code}};
    _Wrong ->
      {next_state, locked, {[], Code}};
  end.

open(timeout, State) ->
  do_lock(),
  {next_state, locked, State}.
```

如果门是锁着的，并且按键被按下，完整的按键会与正确的密码对比，以此决定门是解锁页设定 gen\_fsm 状态为‘开’，还是保持关闭的状态。

### 3.5 超时

当正确的密码被给出，门就会打开，下面的元组会被从 locked/2 返回：

```
{next_state, open, {[], Code}, 30000};
```

30000 是一个超时值，以微秒为单位。在 30000ms，即 30 秒后，一个超时事件被触发。然后 StateName(timeout, StateData) 被调用，这样，如果门打开 30 秒后，门会被重新上锁：

```
open(timeout, State) ->
    do_lock(),
    {next_state, locked, State}.
```

### 3.6 All 状态事件

有些时候，一个事件可能携带任何的状态，并传送给 gen\_fsm。代替使用 fen\_fsm:send\_event/2 发送消息的，并写一个句柄，处理事件中每个状态函数，消息可以使用 gen\_fsm:send\_all\_state\_event/2 来发送，并通过 Module:handle\_event/3 处理：

```
-module(code_lock).
...
-export([stop/0]).
...

stop() ->
    gen_fsm:send_all_state_event(code_lock, stop).

...

handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.
```

### 3.7 停止函数

#### 3.7.1 在监督树中

如果 gen\_fsm 是监督树中的一部分，它就不需要 stop 函数。gen\_fsm 会通过它的监督者自动终止。实际上，它是通过在监督树中定义一个关闭策略来实现的。

如果在中止前需要清理一下程序，关闭策略设定为一个超时值，gen\_fsm 也必需在 init 中设定退出信号的处理函数。当需要关闭的时候，gen\_fsm 会调用回调函数

```
terminate(shutdown, StateName, StateData):
```

```

init(Args) ->
    ...,
    process_flag(trap_exit, true),
    ...,
    {ok, StateName, StateData}.

...

terminate(shutdown, StateName, StateData) ->
    ..code for cleaning up here..
    ok.

```

### 3.7.2 独立 Gen\_Fsm

如果 gen\_fsm 不是监督树的一部分，一个 stop 函数必需被设定：

```

...
-export([stop/0]).
...

stop() ->
    gen_fsm:send_all_state_event(code_lock, stop).
...

handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.

...

terminate(normal, _StateName, _StateData) ->
    ok.

```

回调函数处理 stop 事件，返回一个元组 {stop, normal, StateData1}，它通常表示正常终止，StateData1 是一个新值，用做 gen\_fsm 的状态数据。这会导致 gen\_fsm 调用 terminate(normal, StateName, StateData1) 并完美的中止程序。

### 3.8 处理其它消息

如果 gen\_fsm 需要有能力和收到其它消息，而不仅仅是事件的话，回调函数 handle\_info(Info, StateName, StateData) 必须被实现，以处理它们。其它消息的一个例子，就是‘退出’消息，如果 gen\_fsm 被关联到了其它进程，并获取到了‘退出’消

息。

```
handle_info({'EXIT', Pid, Reason}, StateName, StateData) ->
    ..code to handle exits here..
    {next_state, StateName1, StateData1}.
```

## 4. Gen\_Event Behaviour

这一章需要结合参考 `gen_event(3)` 中关于所有接口和回调函数的详细描述。

### 4.1 事件处理原则

在 OTP 中，事情管理器是一个有名称的对象，事件都可以发给它。一个事件可能是，如：一个错误、一个警告、或其它一些需要被记录的信息。

在事件管理器中，0 个、1 个、或若干个事件处理器被设置。当一个事件管理器送达一个关于事件的通知时，事件将被所有这新设定好的事件处理函数来处理。例如，一个事件管理器，可能有一个默认的错误处理函数，用来把错误信息输出到终端上。如果需要在适当的时候把错误写入文件的话，用户就需要另外添加一个事件处理函数来完成这个任务。当我们不再需要把错误写到文件中的时候，我们就把这个处理函数删除。一个事件管理器被做为一个进程来实现，并且每一个事件处理函数被做为一个回调模块来实现的。事件处理器维护着一个 `{Module, State}` 列表，每个模块是一个事情处理者，State 是事件处理者的内部状态。

### 4.2 实例

用于把错误信息输出到终端的事件处理回调模块可能是下面这样的：

```
-module(terminal_logger).
-behaviour(gen_event).

-export([init/1, handle_event/2, terminate/2]).

init(_Args) ->
    {ok, []}.

handle_event(ErrorMessage, State) ->
    io:format("***Error*** ~p~n", [ErrorMessage]),
    {ok, State}.

terminate(_Args, _State) ->
    ok.
```

用于把错误信息写入文件的事件处理回调模块，可能是这样的：

```
-module(file_logger).
-behaviour(gen_event).

-export([init/1, handle_event/2, terminate/2]).

init(File) ->
    {ok, Fd} = file:open(File, read),
    {ok, Fd}.

handle_event(ErrorMsg, Fd) ->
    io:format(Fd, "***Error*** ~p~n", [ErrorMsg]),
    {ok, Fd}.

terminate(_Args, Fd) ->
    file:close(Fd).
```

下一节，我们再来解释这段代码。

### 4.3 启动一个事件管理器

为了启动一个用于处理错误的事件管理器，就像上面的例子所说的那样，调用下面的函数：

```
gen_event:start_link({local, error_man})
```

这个调用启动一个新进程，一个事件管理器。参数{local, error\_man}指定了名字。这样，事件管理器将会在本机注册成为error\_man。如果名字被省略，事件管理器将不会被注册，取而代之的是它的pid将被使用。名字也可以是{global, Name}，这样事件管理器会被使用global:register\_name/2来注册。

gen\_event:start\_link必须被使用，如果事件管理器是监督树的一部分，即，由监督树启动的。还有一个函数gen\_event:start用于启动一个独占的事件管理器，即，一个事件管理器，不属于监督树的一部分。

### 4.4 加入一个事件处理器

这里有一个使用shell启动事件管理器，并加入一个事情处理的例子：

```
1> gen_event:start({local, error_man}).
{ok,<0.31.0>}
```

```
2> gen_event:add_handler(error_man, terminal_logger, []).
ok
```

这个函数发送一个消息给事件管理器，并做为 `error_man` 被注册，并加入一个事件处理函数 `terminal_logger`。事件管理器将调用这个回调函数 `terminal_logger:init([])`，参数 `[]` 是 `add_handler` 的第三个参数。`init` 返回 `{ok, State}`，`State` 是一个事件管理器的内部状态。

```
init(_Args) ->
  {ok, []}.
```

这里，`init` 不需要任何输入数据，所以它呼略掉了它的参数。当然，对于 `terminal_logger` 的内部状态也不会被使用。对于 `file_logger`，内部的状态会被用于存放打开的文件描述。

```
init(_Args) ->
  {ok, Fd} = file:open(File, read),
  {ok, Fd}.
```

#### 4.5 事件通知

```
3> gen_event:notify(error_man, no_reply).
***Error*** no_reply
ok
```

`error_man` 是事件管理器的名字，并且事件没有回应。事件被做成消息并发送到事件管理器。当事件处理器收到了这个消息，事件处理器对于每一个配置了的事件处理函数，会按照它们被加入的顺序调用 `handle_event(Event, State)`。函数返回一个元组 `{ok, State1}`，`State1` 是一个事件管理器的新的状态值。

在 `terminal_logger` 中：

```
handle_event(ErrorMessage, State) ->
  io:format("***Error*** ~p~n", [ErrorMessage]),
  {ok, State}.
```

在 `file_logger` 中：

```
handle_event(ErrorMsg, Fd) ->
  io:format(Fd, "****Error**** ~p~n", [ErrorMsg]),
  {ok, Fd}.
```

## 4.6 删除一个事件处理函数

```
4> gen_event:delete_handler(error_man, terminal_logger, []).
ok
```

函数发送一个消息给 `error_man` 这个事件管理器，告诉它将要删除一个事件处理函数 `terminal_logger`。事件管理器将调用回调函数 `terminal_logger:terminate([], State)`，参数 `[]` 将做为 `delete_handler` 的第三个参数。`terminal` 将相反于 `init` 函数，做一需必要的清理工作。它的返回值将被呼略。

对于 `terminal_logger` 来说，没有什么清理工作需要做的：

```
terminate(_Args, _State) ->
  ok.
```

对于 `file_logger` 来说，`init` 前面打开的文件描述符应该被关闭掉：

```
terminate(_Args, Fd) ->
  file:close(Fd).
```

## 4.7 停止

当一个事件处理器被停止，它将逐一调用所有设置了的事件处理模块中的 `terminate/2`，就像删除一个事件处理函数一样。

### 4.7.1 在监督树中

如果事件管理器是监督树中的一部分，不需要 `stop` 函数。事件管理器将自动的被监督者关闭。实际上，这是由监督者中定义的 `shutdown` 策略来完成的。

### 4.7.2 独占式事件管理器

一个独占式的事件处理器，可以这样关闭：

```
> gen_event:stop(error_man).
ok
```