Trap — ловить, перехватывать

exit signal — сигнал выхода

link — связь

monitoring — мониторинг, слежение

exit handler — обработчик выхода

die — умирать

kill — убить, принудительно завершить

keep-alive — постоянный

race conditions

Глава 9

Ошибки в параллельных программах

Ранее мы видели как ловить ошибки в последовательных программах. В этой главе мы расширим механизм обработки ошибок на параллельные программы.

Это второй и последний этап в понимании того, как Эрланг обрабатывает ошибки. Чтобы понять это нам надо ввести три новые концепции: *связи (links), сигналы выхода (exit signals)* и идею *системного процесса (system process)*.

9.1 Связанные процессы

Если один процесс как-то зависит от другого, то он, вполне возможно, захочет присмотреть за его здоровьем. Один из способов это сделать - посредством встроенной функции Эрланга link. (Другой способ сделать это — используя мониторинг, который описан в руководстве по эрлангу).

Рисунок 9.1: Сигналы выхода и связи

На Рис. 9.1 показаны процессы A и B. Они связаны между собой (показано прерывистой линией). Связь была установлена, когда один из процессов вызвал встроенную функцию link(P) с параметром P — идентификатором другого процесса. После установления связи оба процесса неявно следят друг за другом. Если умрёт A, то B получит *сигнал выхода (exit signal)*. И наоборот — если умрёт B, то такой сигнал получит A.

Механизмы, описанные в этой главе, совершенно общие. Они работают на и на одном-единственном узле системы и на нескольких узлах в распределенной эрланговой системе. Как мы увидим в Главе 10 *«Распределённое программирование»* на стр. ____ мы можем порождать процессы на удалённых узлах так же легко, как и на текущем узле. Все механизмы для связи, которые мы обсуждаем в этой главе, работают так же хорошо и в распределённой системе.

Что происходит, когда процесс получает сигнал выхода? Если не принять специальных мер, то процесс, получивший сигнал выхода, завершится. Однако процесс может перехватывать такие сигналы выхода. В этом случае он называется *системным процессом (system process)*. Если процесс, связанный с системным процессом, завершается по каким-либо причинам, то системный процесс не завершается автоматически. Вместо этого он получает сигнал выхода, который можно перехватить и обработать.

Часть (a) на Рис. 9.1 показывает связанные процессы. A — это системный процесс (показан в двойном кольце). В части (b) B умирает, и в части (c) сигнал выхода посылается к A.

Позже в этой части мы рассмотрим со всеми подробностями то, что происходит, когда процессу приходит полный сигнал выхода. Но до этого мы начнём с небольшого примера, который покажет — как использовать этот механизм для написания простого обработчика выхода. Обработчик выхода — это процесс, который исполняет заданную функцию, когда какой-либо другой процесс завершается аварийно. Обработчик выхода — это полезный строительный блок для создания более развитых абстракций.

9.2 Обработчик on_exit

Мы хотим выполнить некое действие, когда процесс завершается. Можно написать функцию on_exit(Pid,Fun), которая устанавливает связь с процессом Pid. Если Pid умирает с причиной Why, то вычисляется функция Fun(Why).

Вот эта программа:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"libHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"miscHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"erl

```
Line 1 on_exit(Pid, Fun) ->
```

- spawn(fun() ->
- process_flag(trap_exit, true),
- link(Pid),

5 receive

- {'EXIT', Pid, Why} ->
- Fun(Why)
- end
- end).

В строке 3 выражение process_flag(trap_exit, true) превращает порождённый процесс в системный. link(Pid) в строке 4 связывает вновь созданный процесс с Pid. В конце, когда процесс умирает, принимается (строка 6) и обрабатывается (строка 7) сигнал выхода.

Замечание: когда вы прочитаете код этой программы, вы увидите, что мы используем Pid повсеместно. Это Pid связанного процесса. Нежелательно использовать имя переменной вроде LinkedPid, т.к. до вызова link(Pid) связь с этим процессом ещё не установлена. Когда вы видите сообщение типа такого {'EXIT', Pid, _} вы должны понять, что Pid — это связанный процесс и он только что умер.

Чтобы проверить всё это, определим функцию F, которая ждёт единственное сообщение X и затем вычисляет list_to_atom(X):

1> F = fun() ->

receive

X -> list_to_atom(X)

end

end.

#Fun<erl_eval.20.69967518>

Создадим процесс:

2> Pid = spawn(F).

<0.61.0>

И установим обработчик on_exit для мониторинга этого процесса:

```
3> lib_misc:on_exit(Pid,

fun(Why) ->

io:format(" \~p died with:\~p\~n",[Pid, Why])

end).

<0.63.0>
```

Если теперь мы отправим атом к Pid, то процесс Pid умрёт (т.к. он попытается выполнить list_to_atom, а входные данные у него — не список) и тогда вызовется обработчик on_exit:

```
4> Pid ! hello.

hello

<0.61.0> died with:{badarg,[{erlang,list_to_atom,[hello]}]}
```

Функция, которая вызывается при умирании процесса может, разумеется, делать всё, что ей угодно — она может проигнорировать ошибку, зарегистрировать (log) ошибку или перезапустить приложение. Выбор зависит от программиста.

9.3 Дистанционная обработка ошибок

Давайте остановимся и немного подумаем над предыдущим примером. Он показывает исключительно важную часть философии Эрланга, называемую *дистанционной обработкой ошибок*.

Поскольку эрланговая система состоит из множества параллельных процессов, то нам не приходится иметь дело с ошибками прямо в том процессе, где они происходят — мы можем работать с ними в отдельном процессе. Процесс, который имеет дело с ошибкой *даже не обязан находиться на той же самой машине*. В распределённом Эрланге, описанном в следующей главе, мы увидим, что этот простой механизм работает даже между разными машинами. Это очень важно, т.к. если вся машина выходит из строя, то программа, которая исправляет такую ошибку должна находиться на какой-нибудь другой машине.

9.4 Детали обработки ошибок

Давайте снова глянем на те три концепции, которые лежат в основе эрланговой обработки ошибок:

Связи

Связь — это нечто, что определяет путь распространения ошибок между двумя процессами. Если два процесса связаны вместе и один из них умирает, то другому процессу посылается *сигнал выхода*.

Сигналы выхода

Сигнал выхода — это нечто, что создаётся процессом, когда тот умирает. Этот сигнал рассылается всем процессам, которые связаны с умершим. Сигнал выхода содержит информацию о том, почему умер процесс. Причина может быть любым элементом данных Эрланга. Причина может быть явно указана посредством вызова exit(Reason) или неявно при возникновении ошибки. Например, если программа выполняет деление числа на ноль, то причина ошибки устанавливается в атом badarith.

Когда процесс завершает выполнять функцию, с которой он был вызван, причина выхода устанавливается в normal.

Дополнительно процесс Pid1 может послать сигнал выхода X процессу Pid2, выполнив функцию exit(Pid2, X). Процесс, который посылает сигнал выхода не умирает. Он продолжает выполнение после отправки сигнала. Pid2 получит сообщение {'EXIT', Pid1, X} (это если он перехватывает сигналы выхода, т.е. является системным процессом) в точности, как если бы исходный процесс умер. Используя этот механизм, процесс Pid1 может «подделать» собственную смерть (умышленно).

Системные процессы

Когда процесс получает ненормальный сигнал выхода, он тоже завершается, если только это не специальный процесс, называемый *системным процессом*. Когда системный процесс получает сигнал выхода Why от процесса Pid, этот сигнал преобразуется в сообщение {'EXIT', Pid, Why} и добавляется в почтовый ящик системного процесса.

Вызов встроенной функции process_flag(trap_exit, true) превращает обычный процесс в системный, который может перехватывать сигналы выхода.

Когда процесс получает *сигнал выхода* может произойти несколько вещей. Что именно произойдёт, зависит от состояния процесса и значения сигнала выхода и определяется следующей таблицей:

trap_exit

Сигнал выхода

Действие

true

kill

Умереть: рассылка сигнала выхода killed по всем связям

true

X

Добавить {'EXIT', Pid, X} в почтовый ящик

false

normal

Продолжить: ничего не делающий сигнал удаляется

false

kill

Умереть: рассылка сигнала выхода killed по всем связям

false

X

Умереть: рассылка сигнала выхода X по всем связям

Если указана причина kill, то рассылается *неперехватываемый сигнал выхода. Такой сигнал всегда убивает процесс, даже если это системный процесс*. Это используется в OTP процессом-супервизором для принудительного завершения сбойных процессов. Когда процесс получает сигнал kill, он умирает и сигналы killed рассылаются по всем его связям. Это является мерой предосторожности, чтобы случайно не убить больше от системы, чем необходимо.

Сигнал kill предназначен для убийства сбойных процессов. Хорошенько подумайте прежде чем использовать его.

Особенности программирования перехвата выхода

Перехват выхода обычно гораздо легче, чем вы могли бы подумать, прочитав предыдущие главы. И хотя механизмы выхода и его перехвата можно использовать рядом хитроумных способов, большинство программ используют три простых подхода.

Подход 1: Меня не волнует, если процесс, который я создал, падает

Процесс создаёт другой процесс, используя функцию spawn:

Pid = spawn(fun() -> ... end)

Ничего более. Если порождённый процесс падает, то текущий процесс продолжает работать.

Подход 2: Я хочу умереть, если процесс, который я создал, падает

Если быть точным, мы должны бы сказать «Если процесс, который я создал, падает по причине, отличной от normal». Чтобы достичь этого исходный процесс использует функцию spawn_link и не должен заранее готовиться к перехвату выхода. Можно просто написать так:

Pid = spawn_link(fun() -> ... end)

Теперь, если порождённый процесс падает по причине, отличной от normal, текущий процесс также падает.

Подход 3: Я хочу обработать ошибки, если процесс, который я создал, падает

Здесь мы используем spawn_link и trap_exit. Код будет таким:

...

process_flag(trap_exit, true),

Pid = spawn_link(fun() -> ... end),

...

loop(...).

loop(State) ->

receive

{'EXIT', SomePid, Reason} ->

%% do something with the error

loop(State1);

...

end

Теперь процесс, вычисляющий loop, перехватывает выход и не умрёт, если упадёт

связанный с ним другой процесс. Он увидит все сигналы выхода (преобразованные в сообщения) от умирающего процесса и сможет предпринять все необходимые действия, когда обнаружит сбой.

Перехват сигналов выхода (дальнейшее развитие)

Вы можете пропустить эту часть, если читаете в первый раз. Большинство из того, что вы захотите сделать, может быть обработано тремя подходами из предыдущей части. Если же вы хотите знать, как это работает на самом деле — читайте далее. Но учтите — вас предупреждали. Детали этих механизмов возможно будет трудно понять. В большинстве случаев вам не понадобится вникать в этот механизм, особенно если вы используете один из общих подходов (из предыдущей части) или библиотеки OTP — система всё сделает за вас правильным образом.

Для действительного понимания подробностей обработки ошибок мы напишем небольшую программу. Она покажет как взаимодействуют обработка ошибок и связи. Программа начинается вот так:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"edemoHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"1.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"erl

```
-module(edemo1).

-export([start/2]).

start(Bool, M) ->

A = spawn(fun() -> a() end),

B = spawn(fun() -> b(A, Bool) end),

C = spawn(fun() -> c(B, M) end),

sleep(1000),

status(b, B),

status(c, C).
```

Она запускает три процесса: A, B, C. Мысль в том, что A будет связан с B, а B будет связан с C. A будет перехватывать выход и наблюдать за выходом B. B будет перехватывать выход, если Bool будет true. A C умрёт с причиной M.

(Вы, возможно, удивитесь по поводу sleep(1000). Это для того, чтобы сообщения, приходящие при смерти C, вывелись перед проверкой состояния процессов. Это не меняет логику программы, но влияет на порядок вывода.)

Здесь код для всех трёх процессов:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"edemoHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"1.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"erl

```
a() ->

process_flag(trap_exit, true),

wait(a).

b(A, Bool) ->

process_flag(trap_exit, Bool),

link(A),

wait(b).

c(B, M) ->

link(B),

case M of

{die, Reason} ->

exit(Reason);

{divide, N} ->

1/N,

wait(c);

normal ->

true

end.
```

wait/1 всего лишь печатает сообщение, которое принимает:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"edemoHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"1.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"erl

```
wait(Prog) ->

receive

Any ->

io:format("Process \~p received \~p\~n" ,[Prog, Any]),

wait(Prog)

end.
```

Остаток программы такой:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"edemoHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"1.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo1.erl"erl

```
sleep(T) ->

receive

after T -> true

end.

status(Name, Pid) ->

case erlang:is_process_alive(Pid) of

true ->

io:format("process \~p (\~p) is alive\~n" , [Name, Pid]);

false ->
```

```
io:format("process \~p (\~p) is dead\~n" , [Name,Pid])

end.
```

Рисунок 9.2: Перехват сигналов выхода

Теперь запустим программу и будем генерировать различные сигналы выхода в C и наблюдать, как это скажется на B. При запуске программы смотрите на Рис. 9.2. На нём показано что происходит, когда сигнал выхода приходит от C, какие процессы существуют, какие есть связи между ними. Диаграммы состоят из двух частей — часть «до» (верхняя часть) показывает состояние процессов до прихода сигнала выхода и часть «после» (нижняя часть) показывает процессы после того, как средний процесс получил сигнал выхода.

Сначала предположим, что B — это обычный процесс (т.е. который не делал process_flag(trap_exit, true)):

```
1> edemo1:start(false, {die, abc}).

Process a received {'EXIT',<0.44.0>,abc}

process b (<0.44.0>) is dead

process c (<0.45.0>) is dead

ok
```

Когда C выполняет exit(abc) процесс B умирает (потому что он не перехватывает выход). При выходе B рассылает полученный сигнал выхода по всем процессам, с которыми он связан. A (который перехватывает выход) получает сигнал выхода и превращает его в сообщение об ошибке {'EXIT',<0.44.0>,abc}. (Заметьте, что процесс <0.44.0> — это процесс B, который умирает).

Теперь попробуем другой сценарий. Мы скажем процессу C умереть с причиной normal.

```
2> edemo1:start(false, {die, normal}).

process b (<0.48.0>) is alive

process c (<0.49.0>) is dead

ok
```

Процесс B не умирает, т.к. он получает сигнал выхода normal.

Теперь пусть C выполнит арифметическую ошибку:

3> edemo1:start(false, {divide,0}).

=ERROR REPORT==== 8-Dec-2006::11:12:47 ===

Error in process <0.53.0> with exit value: {badarith,[{edemo1,c,2}]}

Process a received {'EXIT',<0.52.0>,{badarith,[{edemo1,c,2}]}}

process b (<0.52.0>) is dead

process c (<0.53.0>) is dead

ok

Когда C пытается делить на ноль происходит ошибка и процесс умирает с ошибкой {badarith, ..}. Процесс B принимает ошибку и тоже умирает, так что ошибка доходит до A.

В конце мы заставляем C завершиться по причине kill:

4> edemo1:start(false, {die,kill}).

Process a received {'EXIT',<0.56.0>,killed} <-- **замена killed**

process b (<0.56.0>) is dead

process c (<0.57.0>) is dead

ok

Причина выхода kill заставляет B умереть и ошибка распространяется по всем связям B как killed. Поведение в этих случаях показано на рисунке в частях (a) и (b).

Мы можем повторить эти тесты для случая, когда B перехватывает выход. Эта ситуация показана на рисунке в части (c).

5> edemo1:start(true, {die, abc}).

Process b received {'EXIT',<0.61.0>,abc}

process b (<0.60.0>) is alive

process c (<0.61.0>) is dead

ok

6> edemo1:start(true, {die, normal}).

Process b received {'EXIT',<0.65.0>,normal}

process b (<0.64.0>) is alive

process c (<0.65.0>) is dead

ok

7> edemo1:start(true, normal).

Process b received {'EXIT',<0.69.0>,normal}

process b (<0.68.0>) is alive

process c (<0.69.0>) is dead

8> edemo1:start(true, {die,kill}).

Process b received {'EXIT',<0.73.0>,kill}

process b (<0.72.0>) is alive

process c (<0.73.0>) is dead

ok

Во всех этих случаях B перехватывает ошибку. Процесс B работает как некий фильтр, ловя все ошибки от C и не допуская их к A. Мы можем проверить exit/2 с code/edemo2.erl. Эта программа похожа на edemo1, отличие только в функции c/2, которая вызывает exit/2. Вот, как это выглядит:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo2.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo2.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo2.erl"edemoHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo2.erl"2.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/edemo2.erl"erl

c(B, M) ->

process_flag(trap_exit, true),

link(B),

exit(B, M),

wait(c).

Запустив edemo2, мы увидим следующее:

```
1> edemo2:start(false, abc).
Process c received {'EXIT',<0.81.0>,abc}
Process a received {'EXIT',<0.81.0>,abc}
process b (<0.81.0>) is dead
process c (<0.82.0>) is alive
ok
2> edemo2:start(false, normal).
process b (<0.85.0>) is alive
process c (<0.86.0>) is alive
ok
3> edemo2:start(false, kill).
Process c received {'EXIT',<0.97.0>,killed}
Process a received {'EXIT',<0.97.0>,killed}
process b (<0.97.0>) is dead
process c (<0.98.0>) is alive
ok
4> edemo2:start(true, abc).
Process b received {'EXIT',<0.102.0>,abc}
process b (<0.101.0>) is alive
process c (<0.102.0>) is alive
ok
5> edemo2:start(true, normal).
Process b received {'EXIT',<0.106.0>,normal}
process b (<0.105.0>) is alive
process c (<0.106.0>) is alive
```

ok

6> edemo2:start(true, kill).

Process c received {'EXIT',<0.109.0>,killed}

Process a received {'EXIT',<0.109.0>,killed}

process b (<0.109.0>) is dead

process c (<0.110.0>) is alive

ok

9.5 Примитивы для обработки ошибок

Вот наиболее распространённые примитивы для управления связями и для перехвата и отправки сигналов выхода:

@spec spawn_link(Fun) -> Pid

Это в точности как spawn(Fun), но дополнительно создаёт связь между процессами родителя и потомка. (spawn_link — это атомарная операция. Она не эквивалентна последовательным вызовам spawn и link, т.к. в промежутке между этими двумя вызовами процесс может умереть)

@spec process_flag(trap_exit, true)

Превращает текущий процесс в системный процесс. Системный процесс — это процесс, который может принимать и обрабатывать сигналы об ошибках.

Замечание: признак trap_exit можно установить в false после того, как он был установлен в true. Этот примитив должен использоваться *только* для превращения обычного процесса в системный и больше ни для каких других целей.

@spec link(Pid) -> true

Связывает текущий процесс с процессом Pid, если такой связи ещё нет. Связь симметрична. Если процесс A выполняет link(B), то он связывается с B. Итог этого такой же, как если бы B выполнил link(A).

Если процесс Pid не существует, то возникает исключение с выходом (exit exception) noproc.

Если процесс A уже связан с B (или наоборот), то вызов игнорируется.

@spec unlink(Pid) -> true

Удаляет любую связь между текущим процессом и процессом Pid.

@spec exit(Why) -> none()

Завершает текущий процесс с причиной Why. Если выполнение exit происходит вне пределов **catch**, то текущий процесс рассылает сигнал выхода с причиной Why всем процессам, с которыми он связан на текущий момент.

Joe спрашивает...

Как мы можем сделать систему устойчивую к сбоям?

Чтобы сделать что-то устойчивым к сбоям, нам надо, как минимум, два компьютера. Один компьютер будет делать работу, а второй смотреть за первым и быть готовым продолжить работу с момента, когда первый компьютер выйдет из строя.

Именно так и работает восстановление после ошибок в Эрланге. Один процесс делает дело, а второй наблюдает за первым и подхватывает работу, если что-то идёт неправильно. Вот поэтому нам надо мониторить процессы и знать почему что-то идёт неправильно. Примеры этой главы показывают как сделать это.

В распределённом Эрланге процессы, которые делают работу и процессы, которые наблюдают за теми, которые делают работу, могут быть вообще на разных машинах. Используя такую технику, мы можем создавать программы, устойчивые к сбоям.

Это общий шаблон. Мы называем это моделью *рабочего-наблюдателя* (*worker-supervisor*) и целая секция в библиотеках OTP посвящена построению *деревьев наблюдения (supervision trees)*, которые используют эту идею.

Базовый примитив языка, который делает это возможным — это link.

Как только вы поймёте как работает link и организуете себе доступ к двум компьютерам, вы сможете создать вашу первую устойчивую к сбоям систему.

@spec exit(Pid, Why) -> true

Посылает сигнал выхода с причиной Why к процессу Pid.

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

Устанавливает монитор. Item — это PID или зарегистрированное имя процесса. За подробностями обращайтесь к руководству по Эрлангу.

Рисунок 9.3: Перехват сигналов выхода

9.6 Набор связанных процессов

Допустим, у нас есть большой набор параллельных процессов, которые что-то вычисляют и что-то пошло не так. Как найти и убить процессы, которые надо?

Самый лёгкий способ — это убедиться, что все эти процессы связаны и не перехватывают выход. Если какой-либо процесс умирает с причиной, отличной от normal, то умирают и все процессы в группе.

Это поведение показано на Рис. 9.3. В части (a) показан набор из девяти процессов, причём процессы 2, 3, 4, 6 и 7 связаны вместе. Если любой из этих процессов умрёт с ненормальной причиной, то умрёт и вся группа процессов, как показано в части (b).

Наборы связанных процессов используются для структурирования программ при создании устойчивых к сбоям систем. Вы можете сделать это сами, либо вы можете воспользоваться библиотечными функциями, описанными в Главе 18.5 *Дерево наблюдения (Supervision Tree)* на стр. ___

9.7 Мониторы

Программирование связей иногда бывает коварным, т.к. связи являются *двунаправленными*. Если A умирает, то к B будет послан сигнал выхода и наоборот. Чтобы не дать процессу умереть нам приходится делать его системным. Иногда мы не хотим этого делать. В таких случаях мы можем использовать монитор.

Монитор — это однонаправленная связь. Если процесс A мониторит процесс B, и процесс B умирает, то к A будет послан сигнал выхода. Однако, если A умирает, то к B не будет послано никакого сигнала выхода. Полное описание возможностей монитора можно найти в руководстве по Эрлангу.

9.8 Постоянный (keep-alive) процесс

Чтобы подвести итог этой главе, создадим постоянный процесс. Идея в том, чтобы создать зарегистрированный процесс, который будет жив всегда — если он по какой-либо причине умирает, то тут же перезапускается.

Мы можем использовать on_exit, чтобы достичь этого:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"libHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"miscHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl".HYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"erl

keep_alive(Name, Fun) ->

register(Name, Pid = spawn(Fun)),

on_exit(Pid, fun(_Why) -> keep_alive(Name, Fun) end).

Здесь создаётся зарегистрированный под именем Name процесс, который вычисляет spawn(Fun). Если процесс по какой-то причине умирает, то он сразу запускается заново.

В on_exit и keep_alive есть достаточно тонкая ошибка. Хотелось бы знать — заметили ли вы её? Когда мы делаем что-то вроде такого:

Pid = register(...),

on_exit(Pid, fun(X) -> ..),

есть возможность, что процесс умрёт в промежутке *между* этими двумя вызовами. Если процесс умирает перед тем, как выполнится on_exit, то связь не будет создана и on_exit не сработает так, как ожидается. Это может произойти в том случае, если две программы пытаются выполнить keep_alive одновременно с одним и тем же значением Name. Это называется *race conditions*. Два кусочка кода — этот и часть, которая устанавливает связь внутри on_exit, пытаются обогнать друг дружку. Если здесь что-нибудь пойдёт не так, то ваша программа может повести себя непредсказуемо.

Я не буду решать эту проблему здесь — подумайте над этим сами. Когда вы объединяете примитивы spawn, spawn_link, register и т.п., вы должны хорошенько подумать о возможных race conditions. Пишите ваш код так, чтобы race conditions никогда не возникали.

К счастью, в библиотеках OTP есть готовый код для построения серверов, деревьев наблюдения и т.п. Эти библиотеки хорошо протестированы и не должны содержать никаких race conditions. Используйте эти библиотеки для построения своих приложений.

На текущий момент мы прошли все механизмы для обнаружения и перехвата ошибок в эрланговых программах. В следующих главах мы используем эти механизмы для построения надёжных программных систем, которые могут восстанавливаться после сбоев. Мы закончили с программированием, рассчитанным на работу в однопроцессорных системах.

Следующая глава будет рассматривать простые распределённые системы.

Кроме сигнала от exit(Pid, kill)

Использование sleep для синхронизации опасно. В маленьких примерах это допустимо, но в рабочем коде синхронизация должна выполняться явно.

Когда процесс завершается нормально — это то же самое, как если бы он вычислил exit(normal)