concurrent - параллельный

pattern — образец

guard — условие

statement — утверждение оператор

shell — шелл, оболочка

reply, response — ответ (отзыв)

evaluate — вычислять

match — сопоставить, совпасть

bind — привязать

spawn - порождать

Глава 8

Параллельное программирование

В этой главе мы поговорим о процессах. Это маленькие изолированные виртуальные машины, которые могут исполнять функции Эрланга.

Я уверен — вы встречали процессы раньше, но только в контексте операционных систем.

В Эрланге процессы относятся к языку программирования, а НЕ к операционной системе.

В Эрланге:

l создание и разрушение ?уничтожение? процессов очень быстрое;

l посылка сообщений между процессами очень быстрая;

l процессы ведут себя одинаково во всех операционных системах;

l может быть очень большое количество процессов;

l процессы не разделяют память и являются полностью независимыми;

l единственный способ для взаимодействия процессов — это через передачу

сообщений.

По этим причинам Эрланг иногда называют *языком с чистой передачей сообщений*.

Если вы раньше не программировали процессы, то до вас доходили слухи о том, что это достаточно трудно. Возможно, вы слышали ужасные истории о нарушениях памяти (memory violations), race conditions, искажении разделяемой памяти (shared-memory corruption) и тому подобном. В Эрланге программировать процессы легко. Для этого нужно только три примитива: **spawn**, **send** и **receive**.

8.1 Параллельные примитивы

Всё, чему мы научились о последовательном программировании, верно и для параллельного. Единственное, что нам надо сделать — это добавить следующие примитивы:

Pid = spawn(Fun)

Создаёт новый параллельный процесс, который вычисляет (evaluates) Fun. Новый процесс работает параллельно с вызвавшим его. Spawn возвращает Pid (сокращение для *идентификатор процесса*). Вы можете использовать Pid для посылки сообщений процессу.

Pid ! Message

Посылает сообщение Message процессу с идентификатором Pid. Посылка сообщения асинхронна. Отправитель не ждёт, а продолжает делать то, чем занимался. ! называется оператором *send*.

Pid ! M определяется как M — примитив отправки сообщения ! возвращает само сообщение. Поэтому Pid1 ! Pid2 ! ... ! M означает отправку сообщения M всем процессам — Pid1, Pid2 и т. д.

receive ... end

Принимает сообщение, которое было послано процессу. У него следующий синтаксис:

receive

Pattern1 [when Guard1] ->

Expressions1;

Pattern2 [when Guard2] ->

Expressions2;

...

end

Когда сообщение прибывает к процессу система пытается сопоставить его с образцом Pattern1 (возможно с учётом условия Guard1). Если это выполнилось успешно, то она вычисляет выражение Expression1. Если первый образец не совпадает, то она использует Pattern2 и т. д. Если ни один из образцов не соответствует, сообщение сохраняется для последующей обработки, а процесс ожидает следующего сообщения. Это объясняется подробнее в части 8.6 Избирательный приём на стр. 12.

Образцы и условия в операторе приёма имеют точно такую же синтаксическую форму и значение, как образцы и условия, которые мы используем, когда определяем функцию.

8.2 Простой пример

Помните, как мы писали функцию area/1 в части 3.1 *Модули* на стр. ___? Просто, чтобы напомнить вам, код, который определял функцию выглядел вот так:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/geometry.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/geometry.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/geometry.erl"geometryHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/geometry.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/geometry.erl"erl

```
area({rectangle, Width, Ht}) -> Width * Ht;

area({circle, R}) -> 3.14159 * R * R.
```

Теперь перепишем эту же функцию как процесс:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server0.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server0.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server0.erl"areaHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server0.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server0.erl"serverHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server0.erl"0.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server0.erl"erl

```
-module(area_server0).

-export([loop/0]).
```

```
loop() ->

receive

{rectangle, Width, Ht} ->

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

loop();

{circle, R} ->

io:format("Area of circle is \~p\~n" , [3.14159 * R * R]),

loop();

Other ->

io:format("I don't know what the area of a \~p is \~n" ,[Other]),

loop()

end.
```

Мы можем создать процесс, который вычисляет loop/0 в шелле:

```
1> Pid = spawn(fun area_server0:loop/0).

<0.36.0>

2> Pid ! {rectangle, 6, 10}.

Area of rectangle is 60

{rectangle,6,10}

3> Pid ! {circle, 23}.

Area of circle is 1661.90

{circle,23}

4> Pid ! {triangle,2,4,5}.

I don't know what the area of a {triangle,2,4,5} is

{triangle,2,4,5}
```

Что здесь произошло? В строке 1 мы создали новый параллельный процесс.

spawn(Fun) создаёт параллельный процесс, который вычисляет Fun. Он возвращает Pid, который печатается как <0.36.0>.

В строке 2 мы посылаем сообщение процессу. Это сообщение совпадает с первым образцом в операторе приёма в loop/0.

```
loop() ->

receive

{rectangle, Width, Ht} ->

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

loop()

...
```

По приёму сообщения, процесс печатает площадь прямоугольника. В конце шелл печатает {rectangle,6,10}. Это потому, что значением Pid ! Msg является Msg. Если мы отправляем процессу сообщение, которое он не понимает, он печатает предупреждение. Это выполняется кодом Other -> ... в операторе приёма **receive**.

8.3 Клиент-сервер - введение

Архитектуры клиент-сервер центральные в Эрланге. По традиции клиент-серверные архитектуры включают сеть, которая отделяет клиента от сервера. Наиболее часто присутствуют несколько экземпляров клиента и один сервер. Слово *сервер* часто вызывает образ некоего достаточно тяжёлого программного обеспечения, работающего на специализированной машине.

В нашем случае предполагается гораздо более легковесный механизм. Клиент и сервер в клиент-серверной архитектуре — это раздельные процессы, и для связи между клиентом и сервером используется обычная передача сообщений Эрланга. Как клиент, так и сервер могут работать на одной и той же машине или на двух разных машинах.

Слова *клиент* и *сервер* ссылаются на роли, которые выполняют эти два процесса. Клиент всегда начинает вычисление отправляя *запрос* к серверу. Сервер вычисляет ответ и отправляет *отзыв* клиенту.

Давайте-ка напишем наше первое клиент-серверное приложение. Начнём вносить небольшие изменения в программу, написанную нами в предыдущей главе.

В предыдущей программе всё, что нам было надо — это послать запрос к процессу, который примет и напечатает этот запрос. Что мы хотим теперь — это послать ответ

процессу, который послал первоначальный запрос. Проблема в том, что мы не знаем кому слать ответ. Чтобы сервер послал ответ, клиент должен включить адрес, на который сервер сможет ответить. Это подобно отправке письма кому-то — если вы хотите получить ответ, вам лучше бы указать в письме ваш адрес!

Итак, отправитель должен включить обратный адрес в сообщение. Этого можно достичь, поменяв это:

```
Pid ! {rectangle, 6, 10}
```

на это:

```
Pid ! {self(),{rectangle, 6, 10}}
```

self() - это PID клиентского процесса.

Для ответа на запрос нам придётся поменять код, принимающий запросы с такого:

```
loop() ->

receive

{rectangle, Width, Ht} ->

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

loop()

...
```

на такой:

```
loop() ->

receive

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

From ! Width * Ht,

loop();

...
```

Заметьте, как теперь мы посылаем результат наших вычислений обратно к процессу, определяемому параметром From. Клиент примет результат, т.к. он устанавливает этот параметр в свой собственный идентификатор процесса.

Процесс, который посылает начальный запрос называется *клиентом*. Процесс, который принимает запрос и отправляет ответ называется *сервером*.

В итоге, мы добавили маленькую полезную функцию, названную rpc (сокращение для *remote procedure call* — удалённый вызов процедуры), которая включает в себя посылку запроса на сервер и ожидание ответа:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"areaHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"serverHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"1.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"erl

```
rpc(Pid, Request) ->

Pid ! {self(), Request},

receive

Response ->

Response

end.
```

Сложив всё это вместе, мы получим следующее:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"areaHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"serverHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"1.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server1.erl"erl

```
-module(area_server1).

-export([loop/0, rpc/2]).

rpc(Pid, Request) ->

Pid ! {self(), Request},
```

```
receive

Response ->

Response

end.

loop() ->

receive

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

From ! Width * Ht,

loop();

{From, {circle, R}} ->

From ! 3.14159 * R * R,

loop();

{From, Other} ->

From ! {error,Other},

loop()

end.
```

Мы можем поэкспериментировать с этим в шелле:

```
1> Pid = spawn(fun area_server1:loop/0).

<0.36.0>

2> area_server1:rpc(Pid, {rectangle,6,8}).

48

3> area_server1:rpc(Pid, {circle,6}).

113.097

4> area_server1:rpc(Pid, socks).

{error,socks}
```

С этим кодом есть небольшая проблема. В функции rpc/2 мы посылаем запрос к серверу и ждём ответа. *Но мы ждём не ответа от сервера*, мы ждём любое сообщение. Если какой-нибудь другой процесс пошлёт клиенту сообщение, в то время как он ждёт ответа от сервера, он (клиент) ошибочно истолкует это сообщение как ответ от сервера. Мы можем исправить это, поменяв вид операторе приёма на такой:

```
loop() ->

receive

{From, ...} ->

From ! {self(), ...}

loop()

...
```

и поменяв rpc на следующее:

```
rpc(Pid, Request) ->

Pid ! {self(), Request},

receive

{Pid, Response} ->

Response

end.
```

Как это работает? Когда мы выполняем функцию rpc, Pid уже связан с каким-то значением, так что в образце {Pid, Response} Pid привязан к какому-то значению, а Response нет. Этот образец совпадёт только с сообщением, состоящим из двухэлементного кортежа, первый элемент которого Pid. Все другие сообщения будут поставлены в очередь. (**receive** обеспечивает то, что называется 8.6 Избирательный приём, который я опишу после этой главы).

С этим изменением мы получим следующее:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/area_server2.erl"DownloadHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/area_server2.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/area_server2.erl"areaHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/area_server2.erl"_HYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/area_server2.erl"serverHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server2.erl"2.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server2.erl"erl

```
-module(area_server2).

-export([loop/0, rpc/2]).

rpc(Pid, Request) ->

Pid ! {self(), Request},

receive

{Pid, Response} ->

Response

end.

loop() ->

receive

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

From ! {self(), Width * Ht},

loop();

{From, {circle, R}} ->

From ! {self(), 3.14159 * R * R},

loop();

{From, Other} ->

From ! {self(), {error,Other}},

loop()

end.
```

Это работает как и ожидается:

```
1> Pid = spawn(fun area_server2:loop/0).

<0.37.0>
```

3> area_server2:rpc(Pid, {circle, 5}).

78.5397

Есть одно финальное улучшение, которое мы можем сделать. Мы можем *скрыть* spawn и rpc *внутри* модуля. Это хорошая практика, т.к. мы сможем менять внутренние детали сервера без изменения кода клиента. В конце мы получаем это:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"areaHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"*HYPERLINK*
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"serverHYPERLINK
*"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"*HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"finalHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"erl

```
-module(area_server_final).

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

start() -> spawn(fun loop/0).

area(Pid, What) ->

rpc(Pid, What).

rpc(Pid, Request) ->

Pid ! {self(), Request},

receive

{Pid, Response} ->

Response

end.

loop() ->

receive

{From, {rectangle, Width, Ht}} ->
```

```
From ! {self(), Width * Ht},

loop();

{From, {circle, R}} ->

From ! {self(), 3.14159 * R * R},

loop();

{From, Other} ->

From ! {self(), {error,Other}},

loop()

end.
```

Для запуска этого мы вызываем функции start/0 и area/2 (где раньше мы вызывали spawn и rpc). Имена лучше те, которые более точно описывают то, что делает сервер:

```
1> Pid = area_server_final:start().

<0.36.0>

2> area_server_final:area(Pid, {rectangle, 10, 8}).

80

4> area_server_final:area(Pid, {circle, 4}).

50.2654
```

8.4 Как долго занимает создать процесс?

В этом месте вы можете начать волноваться о производительности. В конце концов, если мы создаём сотни или тысячи эрланговых процессов, мы должны как-то расплачиваться за это. Давайте поищем — как.

Чтобы исследовать это мы измерим время, нужное для порождения большого количества процессов. Вот программа:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/processes.erl"DownloadHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/processes.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/processes.erl"processesHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/processes.erl".HYPERLINK

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

```
-module(processes).
-export([max/1]).
%% max(N)
%% Create N processes then destroy them
%% See how much time this takes
max(N) ->
Max = erlang:system_info(process_limit),
io:format("Maximum allowed processes:\~p\~n" ,[Max]),
statistics(runtime),
statistics(wall_clock),
L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
{_, Time1} = statistics(runtime),
{_, Time2} = statistics(wall_clock),
lists:foreach(fun(Pid) -> Pid ! die end, L),
U1 = Time1 * 1000 / N,
U2 = Time2 * 1000 / N,
io:format("Process spawn time=\~p (\~p) microseconds\~n" ,
[U1, U2]).
wait() ->
receive
die -> void
end.
for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

Вот результаты, которые я получил на компьютере, который я использовал для написания этой книги - 2.40GHz Intel Celeron с 512 МБ ОЗУ под управлением Ubuntu Linux:

1> processes:max(20000).

Maximum allowed processes:32768

Process spawn time=3.50000 (9.20000) microseconds

ok

2> processes:max(40000).

Maximum allowed processes:32768

=ERROR REPORT==== 26-Nov-2006::14:47:24 ===

Too many processes

...

Порождение 20,000 процессов заняло в среднем 3,5 мкс/процесс процессорного времени и 9,2 мкс прошедшего (по часам) времени.

Заметьте, что я использовал встроенную функцию (BIF) erlang:system_info(process_limit) для нахождения максимального разрешенного количества процессов. Заметьте, что некоторые из них зарезервированы, так что ваша программа не может на самом деле использовать это количество. Когда мы превышаем системный лимит система рушится с сообщением об ошибке (команда 2).

Системный лимит установлен в 32,767 процессов. Чтобы превысить этот лимит вам придётся запустить эмулятор Эрланга с параметром +P как здесь:

$ erl +P 500000

1> processes:max(50000).

Maximum allowed processes:500000

Process spawn time=4.60000 (10.8200) microseconds

ok

2> processes:max(200000).

Maximum allowed processes:500000

Process spawn time=4.10000 (10.2150) microseconds

3> processes:max(300000).

Maximum allowed processes:500000

Process spawn time=4.13333 (73.6533) microseconds

В предыдущем примере я установил системный лимит в полмиллиона процессов. Мы можем видеть, что время порождения процесса по существу постоянно между 50,000 и 200,000 процессов. При 300,000 процессов процессорное время порождения остаётся постоянным, но прошедшее время увеличивается в 7 раз. Также я слышу, как вибрирует мой диск. Это верный знак того, что система выполняет подкачку и у меня недостаточно физической памяти для работы с 300,000 процессов.

8.5 Приём с таймаутом

Иногда оператор приёма может вечно ждать сообщения, которое так никогда и не придёт. Для этого может быть несколько причин. Например, может быть логическая ошибка в нашей программе или процесс, который собирался отправить нам сообщение рухнул до отправки.

Чтобы избежать проблем мы может добавить таймаут в оператор приёма. Он устанавливает максимально время, которое процесс будет ждать при получении сообщения. Синтаксис этого следующий:

receive

Pattern1 [when Guard1] ->

Expressions1;

Pattern2 [when Guard2] ->

Expressions2;

...

after Time ->

Expressions

end

Если в течение Time миллисекунд в оператор приёма не придёт ни одного совпадающего сообщения, то процесс перестаёт ждать сообщения и вычисляет Expressions.

Приём только с таймаутом

Вы можете написать **receive**, содержащий только таймаут. Используя это, мы можем определить функцию sleep(T), которая останавливает текущий процесс на время T миллисекунд.

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

```
sleep(T) ->

receive

after T ->

true

end.
```

Приём с нулевым таймаутом

Значение таймаута 0 приводит к немедленному срабатыванию таймаута, но перед тем, как это случится, система пытается сопоставить хоть какой-нибудь образец из почтового ящика. Мы можем использовать это для определения функции flush_buffer, которая полностью опустошает почтовый ящик процесса:

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

```
flush_buffer() ->

receive
```

```
_Any ->

flush_buffer()

after 0 ->

true

end.
```

Без оператора таймаута функция flush_buffer остановилась бы навечно и не вернула бы ничего, если бы почтовый ящик был пуст. Мы также можем использовать нулевой таймаут для создания некоей формы «приоритетного приёма», как в следующем:

```
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

priority_receive() ->

receive

{alarm, X} ->

{alarm, X}

after 0 ->

receive

Any ->

Any

end

end.
```

Если в почтовом ящике есть сообщение, не соответствующее образцу {alarm, X}, то priority_receive примет первое сообщение из почтового ящика. Если же никаких сообщений нет, то приём приостановится на внутреннем операторе receive до прихода любого сообщения. Если есть сообщение, соответствующее {alarm, X}, то это

сообщение будет немедленно возвращено как результат. Помните, что секция after проверяется только после проверки на соответствие шаблону всех сообщений из почтового ящика.

Без оператора 'after 0' сообщение alarm не сработало бы первым.

Замечание: использование больших почтовых ящиков совместно с приоритетным приёмом достаточно неэффективно, так что, если вы собираетесь использовать эту технику, убедитесь, что ваши почтовые ящики не слишком большие.

Приём с бесконечным таймаутом

Если значением таймаута в операторе приёма является атом infinity, то таймаут *никогда* не сработает. Это может быть полезным для программ, в которых значение таймаута вычисляется вне оператора receive. Иногда вычисление может захотеть вернуть какое-то конкретное значение, а иногда оно может захотеть, чтобы receive ждал вечно.

Организация таймера

Мы можем организовать простой таймер, используя таймауты в приёме.

Функция stimer:start(Time, Fun) вычислит Fun (функцию без аргументов) после ожидания Time миллисекунд. Она возвращает обработчик (который на самом деле PID), который может использоваться для отмены таймера при необходимости.

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/stimer.erl"DownloadHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/stimer.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/stimer.erl"stimerHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/stimer.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/stimer.erl"erl

```
-module(stimer).

-export([start/2, cancel/1]).

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

cancel(Pid) -> Pid ! cancel.

timer(Time, Fun) ->

receive

cancel ->
```

void

after Time ->

Fun()

end.

Мы можем проверить это следующим образом:

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

<0.42.0>

timer event

Здесь я ждал больше пяти секунд, чтобы сработал таймер. Сейчас я запущу таймер и отменю его до того, как выйдет таймерное время:

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

<0.49.0>

3> stimer:cancel(Pid1).

cancel

8.6 Избирательный приём

До сих пор мы проходили по верхушкам того, как действительно работают **send** и **receive**. **Send** на самом деле не отправляет сообщения процессу. В место того, **send** отправляет сообщение в почтовый ящик процесса, а **receive** пытается удалить сообщения из почтового ящика.

Каждый процесс в Эрланге имеет свой собственный почтовый ящик. Когда вы посылаете сообщение процессу, это сообщение помещается в почтовый ящик. Почтовый ящик проверяется только тогда, когда программа вычисляет оператор receive:

receive

Pattern1 [when Guard1] ->

Expressions1;

Pattern2 [when Guard1] ->

Expressions1;

...

after Time ->

ExpressionTimeout

end

receive работает следующим образом:

Когда мы входим в оператор **receive**, мы запускаем таймер (но только, если в выражении присутствует секция **after**).

Взять первое сообщение из почтового ящика и попытаться соотнести его с образцами Pattern1, Pattern2 и т.д. Если соответствие успешно, то сообщение удаляется из почтового ящика и вычисляется выражение, следующее за образцом.

Если ни один из образцов в операторе **receive** не соответствует первому сообщению из почтового ящика, то первое сообщение удаляется из ящика и помещается в «отложенную очередь» (save queue). Затем так же проверяется второе сообщение. Эта процедура повторяется до тех пор, пока не будет найдено совпадающее сообщение, либо не будут проверены все сообщения из почтового ящика.

Если ни одно сообщение из почтового ящика не соответствует, процесс приостанавливается и ждёт до тех пор, пока новое сообщение не будет помещено в почтовый ящик. Заметьте, что когда новое сообщение прибывает, сообщения из отложенной очереди не проверяются заново на соответствие образцам. Проверяется только новое сообщение.

Как только сообщение совпало с образцом, сразу после этого все сообщения из отложенной очереди помещаются обратно в почтовый ящик в том же порядке, в каком они прибыли к процессу. Если был установлен таймер, то он очищается.

Если истёк таймер, пока мы ждали сообщение, то вычислить выражения ExpressionTimeout и поместить все отложенные сообщения обратно в почтовый ящик в том же порядке, в каком они прибыли к процессу.

8.7 Зарегистрированные процессы

Если мы хотим послать сообщение процессу, нам надо знать его PID. Это часто не удобно, т.к. PID надо передать всем процессам в системе, желающим взаимодействовать с данным процессом. С другой стороны, это очень *безопасно*. Если вы не раскрываете PID процесса, другие процессы не могут взаимодействовать с ним никаким образом.

У Эрланга есть метод *публикации* идентификатора процесса, так что любой процесс в системе может общаться с этим процессом. Такой процесс называется *зарегистрированным процессом*. Есть четыре встроенные функции (BIF) для управления зарегистрированными процессами:

register(AnAtom, Pid)

зарегистрировать процесс Pid с именем AnAtom. Регистрация не успешна, если AnAtom уже был использован для регистрации процесса.

unregister(AnAtom)

удалить любые регистрации, связанные с AnAtom.

Замечание: если зарегистрированный процесс умирает, он автоматически разрегистрируется

whereis(AnAtom) -> Pid l undefined

найти, где зарегистрирован AnAtom. Возвращает идентификатор процесса Pid, либо возвращает атом undefined, если никакой процесс не связан с AnAtom.

registered() -> [AnAtom::atom()]

возвращает список зарегистрированных процессов в системе.

Используя register, мы можем пересмотреть пример из части 8.2 Простой пример на стр. 2 и можем попытаться зарегистрировать имя процесса, который мы создали:

1> Pid = spawn(fun area_server0:loop/0).

<0.51.0>

2> register(area, Pid).

true

Как только имя зарегистрировано, мы может отправить ему сообщение подобно этому:

3> area ! {rectangle, 4, 5}.

Area of rectangle is 20

{rectangle,4,5}

Часы

Мы можем использовать регистрацию при создании процесса, который представляет

из себя часы:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/clock.erl"DownloadHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/clock.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/clock.erl"clockHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/clock.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/clock.erl"erl

```
-module(clock).

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

start(Time, Fun) ->

register(clock, spawn(fun() -> tick(Time, Fun) end)).

stop() -> clock ! stop.

tick(Time, Fun) ->

receive

stop ->

void

after Time ->

Fun(),

tick(Time, Fun)

end.
```

Часы будут радостно отстукивать, пока вы не остановите их:

```
3> clock:start(5000, fun() -> io:format("TICK \~p\~n",[erlang:now()]) end).

true

TICK {1164,553538,392266}

TICK {1164,553543,393084}

TICK {1164,553548,394083}

TICK {1164,553553,395064}

4> clock:stop().
```

stop

8.8 Как нам писать параллельную программу?

Когда я пишу параллельную программу, то почти всегда я начинаю с чего-то подобного:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/ctemplate.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/ctemplate.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/ctemplate.erl"ctemplateHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/ctemplate.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/ctemplate.erl"erl

-module(ctemplate).

-compile(export_all).

start() ->

spawn(fun() -> loop([]) end).

rpc(Pid, Request) ->

Pid ! {self(), Request},

receive

{Pid, Response} ->

Response

end.

loop(X) ->

receive

Any ->

io:format("Received:\~p\~n" ,[Any]),

loop(X)

end.

Цикл приёма — это просто пустой цикл, который принимает и печатает все сообщения, которые я посылаю ему. По мере разработки программы я начинаю посылать

сообщения процессу. Т.к. я начинаю цикл приёма вообще без образцов, которые соответствуют сообщениям, то получу распечатку из кода в конце оператора приёма. Когда это происходит, я добавляю образец соответствия в цикл приёма и перезапускаю программу. Эта техника в значительной степени определяет порядок, в котором я пишу программы — я начинаю с небольшой программы, постепенно увеличиваю её, тестируя по мере написания.

8.9 Слово о хвостовой рекурсии

Взгляните на цикл приёма в сервере вычисления площади, который мы писали ранее:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"areaHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"*HYPERLINK*
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"serverHYPERLINK
*"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"*HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"finalHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/area_server_final.erl"erl

```
loop() ->

receive

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

From ! {self(), Width * Ht},

loop();

{From, {circle, R}} ->

From ! {self(), 3.14159 * R * R},

loop();

{From, Other} ->

From ! {self(), {error,Other}},

loop()

end.
```

Если вы посмотрите внимательно, то увидите, что каждый раз, когда мы принимаем сообщение, мы обрабатываем это сообщение и затем сразу же снова вызываем loop(). Такая процедура называется *хвостовой рекурсией*. Функция с хвостовой рекурсией может быть скомпилирована так, что последний вызов функции в последовательности операторов может заменяться обычным переходом на начало функции, которую вызывали. Это значит, что функция с хвостовой рекурсией может зацикливаться бесконечно без потребления стека.

Допустим, мы написали следующий (неправильный) код:

```
Line 1 loop() ->
    {From, {rectangle, Width, Ht}} ->
    From ! {self(), Width * Ht},
    loop(),
5 someOtherFunc();
    {From, {circle, R}} ->
    From ! {self(), 3.14159 * R * R},
    loop();
    ...
10 end
```

В строке 4 мы вызываем loop(), но компьютер должен сообразить что «после вызова loop(), мне придётся вернуться сюда, так как надо будет вызвать функцию someOtherFunc() в строке 5». Поэтому он сохраняет адрес someOtherFunc в стеке и переходит к началу loop(). Проблема тут в том, что loop() никогда не возвращается. Вместо этого она зацикливается навечно. Так что каждый раз, когда мы проходим строку 4 адрес возврата заносится в стек. И в конце концов у системы заканчивается место.

Избежать этого легко — если вы пишите функцию F, которая никогда не возвращается (такую как loop()), убедитесь, что вы никогда ничего не вызываете *после* вызова F и не используйте F в создании кортежей или списков.

8.10 Порождение с MFA

Большинство программ, которые мы пишем используют spawn(Fun) для создания нового процесса. Это прекрасно до тех пор, пока мы не захотим обновлять наш код на

ходу. Иногда мы хотим написать код, который можно обновлять в то время, как он выполняется. Если мы хотим быть уверенными в том, что наш код может обновляться динамически, то нам надо использовать другую форму spawn.

spawn(Mod, FuncName, Args)

это создаёт новый процесс. Args — это список аргументов вида [Arg1, Arg2, ..., ArgN]. Вновь созданный процесс начинает вычисление Mod:FuncName(Arg1, Arg2, ..., ArgN).

Порождение функции с явным указанием модуля, имени функции и списка аргументов (называемые MFA) — это верный способ быть уверенными в том, что наши процессы будут корректно обновляться новыми версиями кода модуля, если он компилируется и в то же время используется. Механизм динамического обновления кода не работает с порождёнными функциями. Он работает только с явно указанными MFA. За дальнейшими деталями читайте приложение E.4 *Динамическая загрузка кода* на стр. *435*.

8.11 Проблемы

Напишите функцию start(AnAtom, Fun), чтобы зарегистрировать AnAtom как spawn(Fun). Убедитесь, что ваша программа работает корректно в случае, когда два параллельных процесса одновременно вычисляют start/2. В этом случае вы должны гарантировать, что один из этих процессов преуспеет, а другой потерпит неудачу.

Напишите кольцевой тест. Создайте N процессов в кольце. Отправьте сообщение по кольцу M раз так, чтобы было отправлено N * M сообщений. Замерьте время, которое тратится для разных значений N и M.

Напишите подобную программу на каком-нибудь другом языке программирования, известном вам. Сравните результаты. Напишите блог и опубликуйте результаты в Интернете!

Вот и всё — теперь вы можете писать параллельные программы!

Дальше мы рассмотрим восстановление после ошибок и увидим, как мы можем писать параллельные программы, устойчивые к сбоям, используя три дополнительные концепции: линки, сигналы и перехват завершения процессов. Это в следующей главе.