Глава 4. Исключения

4.1 Исключения

сообщения Эрланга об ошибках и реакции на них. До того как мы глубже погрузимся в последовательное программирование, давайте кратко рассмотрим эту тему. Это может показаться неожиданным отклонением от темы, но, если наша цель - писать надежные и распределенные выражения, то понимание того, как обрабатываются ошибки становится просто необходимым.

Всякий раз когда мы вызываем функцию в Эрланге, происходит одно из двух: либо функция возвращает нам значение, или что-то идет не так. Мы видели такие примеры в предыдущей главе. Помните функцию cost?

```
cost(oranges) -> 5;

cost(newspaper) -> 8;

cost(apples) -> 2;

cost(pears) -> 9;

cost(milk) -> 7.
```

И вот что происходит при ее работе:

```
1> shop:cost(apples).

2

2> shop:cost(socks).

=ERROR REPORT==== 30-Oct-2006::20:45:10 ===

Error in process <0.34.0> with exit value:

{function_clause,[{shop,cost,[socks]},

{erl_eval,do_apply,5},

{shell,exprs,6},

{shell,eval_loop,3}]}
```

Когда мы вызвали cost(socks) функция обвалилась (crashed). Это произошло потому что ни один из вариантов исполнения функции ("клозов") не подошел к данному

аргументу функции.

Вызов cost(socks) -это полная бессмыслица. Функция не сможет вернуть нам никакого значения в ответ, поскольку цена на носки (socks) в ней просто не определена. В этом случае, вместо возврата значения, система *запускает исключение* - так, на техническом языке, называется "падение" программы.

Мы не пытаемся исправить эту ошибку, потому что это не возможно. Мы не знаем стоимость носков, поэтому мы не можем вернуть никакое значение. Теперь это дело того кто вызвал так функцию (cost(socks)) решать, что-же теперь делать дальше, когда функция "повалилась".

Исключения запускаются системой, когда происходят внутренние ошибки или в самом коде, когда вызываются throw(Exception), exit(Exception). или erlang:error(Exception).

В Эрланге есть два метода *перехвата* исключений. Один из них - заключение вызова функции, которая может запустить исключение внутрь **try...catch** выражения. Второй - это заключить такой вызов в **catch** выражение.

4.2. Запуск Исключения

Исключения запускаются автоматически, когда система обнаруживает какую-либо ошибку. Типичные ошибки - это не-соответствие образцу (включая отсутствие подходящих способов обработки аргументов функции), либо вызов стандартных BIF-функций с неправильным типом аргументов (например, вызов atom_to_list с целочисленным аргументом).

Кроме того, мы можем сами, непосредственно сгенерировать ошибку, вызвав одну из порождающих исключение BIF-функций:

exit(Why)

Она используется когда вы действительно хотите терминировать данный процесс. Если соответствующее исключение не будет перехвачено, то сообщение вида {'EXIT',Pid,Why} будет послано всем процессам, которые связанны с данным. Подробнее мы поговорим об этом в разделе 9.1 Связанные процессы, поэтому здесь я не буду останавливаться на деталях.

throw(Why)

Эта функция используется для запуска исключения, которое вызывающий, возможно захочет перехватить. В этом случае мы должны указать в документации к нашей функции , что она может запускать исключение. У пользователя такой функции будет две альтернативы: либо программировать как обычно и просто слепо игнорировать это исключение, либо можно заключить вызов этой функции внутри try...catch выражения и

обработать его.

erlang:error(Why)

Она используется чтобы обозначить "критическую" ошибку в программе. Так иногда бывает лучше, когда происходит что-то совсем нехорошее, с чем очень трудно справиться. Это эквивалентно сгенерированной внутренней ошибке.

Теперь давайте попробуем перехватить эти исключения.

4.3. try...catch

Если вы знакомы с языком Джава, тогда выражение try...catch будет для вас весьма знакомо. Джава перехватывает исключения с помощью следующего синтаксиса:

try {

block

} catch (exception type identifier) {

block

} catch (exception type identifier) {

block

} ...

finally {

block

}

В Эрланге эта конструкция исключительно похожа:

try FuncOrExpressionSequence of

Pattern1 [when Guard1] -> Expressions1;

Pattern2 [when Guard2] -> Expressions2;

...

catch

ExceptionType: ExPattern1 [when ExGuard1] -> ExExpressions1;

ExceptionType: ExPattern2 [when ExGuard2] -> ExExpressions2;

...

after

AfterExpressions

end

Заметьте схожесть между try...catch выражением и case выражением:

case Expression of

Pattern1 [when Guard1] -> Expressions1;

Pattern2 [when Guard2] -> Expressions2;

...

end

Выражение try...catch это как case выражение выращенное на стероидах. Оно очень похоже на case выражение, но с блоками catch и after в своем начале и конце.

У выражения try...catch есть значение

Вы не забыли? В Эрланге все является выражением и все выражения имеют значение. А значит и выражение try...catch тоже имеет свое значение. То есть мы можем написать что-то вроде:

f(...) ->

...

X = try ... end,

Y = g(X),

...

Но, обычно, нам не надо значение выражения try...catch . Поэтому мы обычно пишем просто:

f(...) ->

...

try ... end,

...

...

Выражение try...catch работает следующим образом: Первым делом вычисляется FuncOrExpressionSeq. Если его вычисление заканчивается без запуска исключения, тогда возвращенное оттуда значение проверяется на соответствие образцу Pattern1 (с охранником Guard1 если он присутствует), потом с образцом Pattern2 и так далее, пока не будет найдено соответствие, и тогда, значением всего выражения try...catch будет значение вычисления выражения, следующего за подходящим образцом.

Если внутри FuncOrExpressionSeq будет запущено исключение, тогда на соответствие ему будут проявляться образцы ExPattern1 и так далее, пока не будет найдено соответствие и ее последовательность выражений для вычисления. ExeptionType - это атом (один из throw, exit или error), который говорит нам, как это исключение было сгенерировано. Если ExceptionType отсутствует, то, по-умолчанию, считается, что он - throw.

Замечание: внутренние ошибки, обнаруженные системой исполнения приложений Эрланга, всегда имеют метку error.

Код, следующий за ключевым словом after, используется, обычно, для уборки после выполнения FuncOrExpressionSeq. Этот код гарантированно будет выполнен, даже если будет запущено исключение. Код в секции after будет запущен сразу после исполнения кода в секциях try или catch. Возвращаемое значение AfterExpressions будет утеряно.

Если вы пришли сюда со знанием языка Руби, то все это должно быть также весьма знакомо для вас - в Руби мы используем следующий тип выражения:

begin

...

rescue

...

ensure

...

end

И хотя ключевые слова отличаются, но общее поведение - очень похоже. (Хотя в

Эрланге нет выражения retry !)

Сокращения

Некоторые части выражения try...catch могут быть опущены. Следующая запись

try F

catch

...

end

означает тоже самое что и:

try F of

Val -> Val

catch

...

end

Аналогично, и раздел after может быть пропущен.

Программирование Идиом с try...catch

Когда мы разрабатываем приложение, мы часто хотим, чтобы код, перехватывающий ошибки, мог перехватить все ошибки, которые функция может сгенерировать.

Вот пара функций для иллюстрации этого. Первая функция генерирует все возможные типы исключений:

/файл try_test.erl/

generate_exception(1) -> a;

generate_exception(2) -> throw(a);

generate_exception(3) -> exit(a);

generate_exception(4) -> {'EXIT', a};

generate_exception(5) -> erlang:error(a).

А теперь мы напишем вызывающую ее функцию внутри выражения try...catch.

/файл try_test.erl/

```
demo1() ->

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

catcher(N) ->

try generate_exception(N) of

Val -> {N, normal, Val}

catch

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

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

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

end.
```

Запустив ее мы увидим следующее:

```
try_test:demo1().

[{1,normal,a},

{2,caught,thrown,a},

{3,caught,exited,a},

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

{5,caught,error,a}]
```

Получается, что мы можем перехватить и обработать все формы исключений, которые может сгенерировать нам функция.

4.4 catch

Другим способом перехватить исключение является использование примитива catch. Когда вы так перехватываете исключение, оно конвертируется в тьюпл, который описывает случившуюся ошибку. Чтобы продемонстрировать это, мы можем вызвать generate_exception внутри catch выражения:

```
demo2() ->

[{I, (catch generate_exception(I))} || I <- [1,2,3,4,5]].
```

Запустив эту функцию, мы получим следующее:

2> **try_test:demo2().**

[{1,a},

{2,a},

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

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

{5,{'EXIT',{a,[{try_test,generate_exception,1},

{try_test,'-demo2/0-fun-0-',1},

{lists,map,2},

{lists,map,2},

{erl_eval,do_apply,5},

{shell,exprs,6},

{shell,eval_loop,3}]}}}]

Если вы сравните это с результатом работы try...catch , то увидите, что мы утратили много информации для анализа причин возникшей проблемы.

4.5 Улучшение сообщений об ошибках

Одним из способов использования функции erlang:error является улучшение информативности сообщений об ошибках. Приведем пример. Если мы вызовем math:sqrt(X) с отрицательным аргументом, то мы увидим следующее:

1> **math:sqrt(-1).**

** exited: {badarith,[{math,sqrt,[-1]},

{erl_eval,do_apply,5},

{shell,exprs,6},

{shell,eval_loop,3}]} **

Но мы можем написать обертку для этого, которая улучшит сообщение об ошибке:

sqrt(X) when X < 0 ->

```
erlang:error({squareRootNegativeArgument, X});

sqrt(X) ->

math:sqrt(X).
```

2> **lib_misc:sqrt(-1).**

```
** exited: {{squareRootNegativeArgument,-1},

[{lib_misc,sqrt,1},

{erl_eval,do_apply,5},

{shell,exprs,6},

{shell,eval_loop,3}]} **
```

4.6 Стиль программирования try...catch

Так как-же нам обрабатывать ошибке на практике. Это, естественно, зависит от ситуации....

Код, часто возвращающий error

Если ваша не является гарантированно проходным случаем, то вам, вероятно, следует возвращать что-то вроде {ok, Value} или {error, Reason}, но помните, что это заставит всех вызывающих вашу функцию *что-то* сделать с возвращаемым значением. У вас будут, при этом, две альтернативы: либо вот так:

```
...

case f(X) of

{ok, Val} ->

do_some_thing_with(Val);

{error, Why} ->

%% ... do something with the error ...

end,

...
```

что обработает оба возвращаемых типа значений, или вот так:

```
...
```

{ok, Val} = f(X),

do_some_thing_with(Val);

...

что запустит исключение если функция f(X) вернет {error, ...} .

Код, где ошибки возможны, но редки

В этом случае типично написание кода, который ожидает и обрабатывает ошибки как в нижеследующем примере:

try my_func(X)

catch

throw:{thisError, X} -> ...

throw:{someOtherError, X} -> ...

end

А код, который ловит ошибки, должен при этом иметь соответствующие ветки throw :

my_func(X) ->

case ... of

...

... ->

... throw({thisError, ...})

... ->

... throw({someOtherError, ...})

4.7 Перехват всех возможных исключений

Если мы хотим перехватить все возможные ошибки, мы можем использовать следующий тип выражения:

try Expr

catch

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

end

Если мы опустим тип исключения и напишем вот так:

try Expr

catch

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

end

то мы *НЕ* перехватим все ошибки, потому что в этом случае типом исключений будет только throw, который действует по-умолчанию.

4.8 Обработка ошибок в старом и новом стилях

Этот раздел предназначен только для Эрланг-ветеранов!

try...catch это относительно новая конструкция, которая была введена для исправления дефектов механизма catch...throw. Если вы эрланговец старой закалки, который не читал его последней документации (типа меня), тогда вы автоматически пишете код наподобие вот такого:

case (catch foo(...)) of

{'EXIT', Why} ->

...

Val ->

...

end

Обычно, это тоже корректно работает, но почти всегда лучше написать следующее:

try foo(...) of

Val -> ...

catch

exit: Why ->

...

```
end
```

Так что, вместо написания case (catch ...) of ..., пишите try ... of ....

4.9 Трассировка стека вызовов

Когда исключение перехвачено, мы можем получить текущий стек вызовов с помощью функции erlang:get_stacktrace(). Рассмотрим пример:

```
demo3() ->
try generate_exception(5)
catch
error:X ->
{X, erlang:get_stacktrace()}
end.
```

```
1> try_test:demo3().
{a,[{try_test,generate_exception,1},
{try_test,demo3,0},
{erl_eval,do_apply,5},
{shell,exprs,6},
{shell,eval_loop,3}]}
```

Получаемая трассировка стека содержит список функций из стека, которым данная функция должна вернуть значение, если получится. Он почти совпадает с последовательностью вызовов функций, который привел нас к данной функции, но все вызовы с хвостовой рекурсией будут отсутствовать в этой трассировке (См. раздел 8.9 *Немного о хвостовой рекурсии*).

Сточки зрения исправления ошибок в нашей программе (дебагинга) только первые несколько строчек представляют тут для нас интерес. Начало трассировки стека говорит нам о том система повалилась во время вычисления функции generate_exception (из модуля try_test) с одним аргументом. try_test:generate_exception/1 была вероятно вызвана try_test:demo3() (мы не можем быть в этом абсолютно уверены, потому что try_test:demo3() могла вызвать некоторую другую функцию, которая сделала вызов с хвостовой рекурсией try_test:generate_exception/1, и в этом случае

трассировка стека не покажет нам записей о этой промежуточной функции).