# Обработка ошибок Erlang

## Парадигмы работы с ошибками

При написании любого программного кода условно существует один «счастливый путь» — алгоритм отработал так, как было запланировано и множество исключений — любой нестандартный исход.

Под обработкой исключений/ошибок понимается попытка их исправления или их логирование.

Всего выделяется два подхода к обработке исключений приложения.

* Защитное программирование — программист старается предусмотреть любые ошибочные пути и либо обработать в месте их возникновения, либо отдать ошибки выше по вызову, где там они точно будут обработаны
* Let it crash — сосредоточение только на счастливом пути. Исключение автоматически будет поймано вышестоящей сущностью и как-нибудь обработано.

## Локальная обработка ошибок Erlang

Локальная обработка – обработка на уровне текущего процесса. Методика обработки напрямую зависит от способа генерации ошибки

#### Возврат кортежа-кода ошибки

В Erlang функции как правило возвращают такие кортежи:

* Счастливый путь: {ok} | {ok,Value};
* Ошибка: {error, Reason}.

Обработка состоит в сопоставлении с образцом вернувшегося кода.

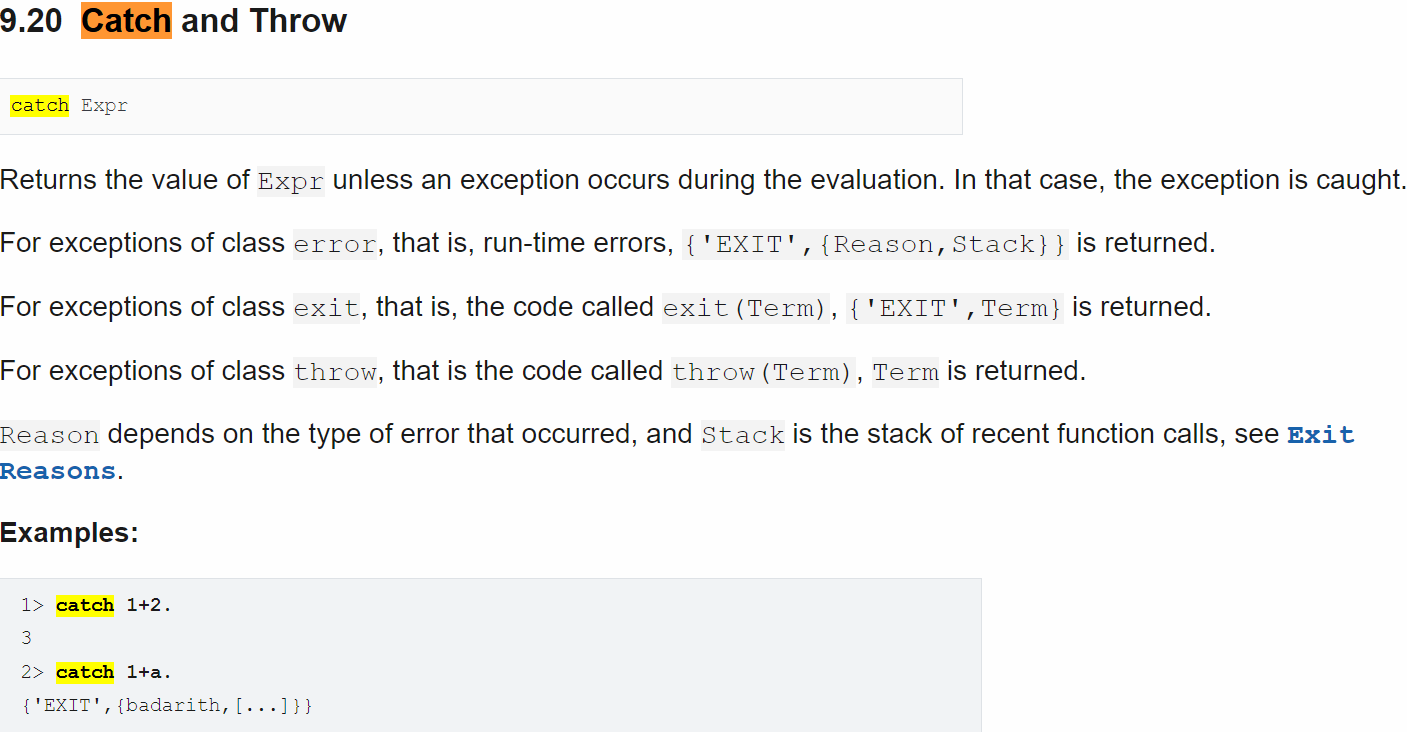
#### Выбрасывание ошибки

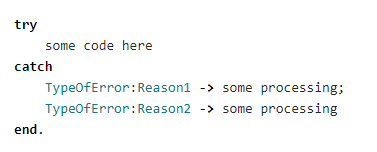
Существует три типа исключений:

Таблица — типы исключений в Erlang

|  |  |
| --- | --- |
| **Class** | **Origin** |
| error | Run-time error, for example, 1+a, or the process called error/1,2 |
| exit | The process called exit/1 |
| throw | The process called throw/1 |

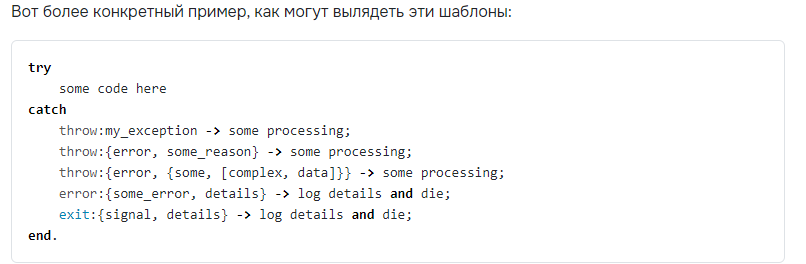
Синтаксис отлова ошибок:





Где **TypeOfError** -- это тип исключения (*throw*, *error* или *exit*)

Принцип работы похож на сопоставление с образцом.



При помощи функции

erlang:get\_stacktrace()

можно получить стэк вызовов во время обработки исключения.

*« В эрланг, и вообще в функциональном программировании, предпочитают использовать специальные значения, а исключениями применяют редко.»*

*«Я могу дать такую рекомендацию: в большинстве случаев использовать специальные типы****Option****и****Result****. Исключения использовать редко, только если они дают более простой и понятный код, чем код со специальными типами.»*

## Глобальная обработка ошибок Erlang

Итак, представим, что в коде процесса не было предпринято ни одного способа обработки ошибки. Что тогда произойдёт?

Тогда процесс завершается с причиной Reason. Сигнал о завершении посылается всем связанным процессам. Подробнее см. Таблица 2 — соответствие между термом Reason и фактическим типом ошибки.

Про работу супервизоров см. Методы отслеживания состояния потоков Супервизор.

# Дерево процессов Erlang

В Erlang код состоит в иерархии:

Модуль -> Функция

Приложением считается совокупность модулей, в которой один является стартовым и служащих одной цели.

Программы на Erlang описывают поведение как минимум одного процесса. Процессы принято организовывать в структурированные деревья вида:

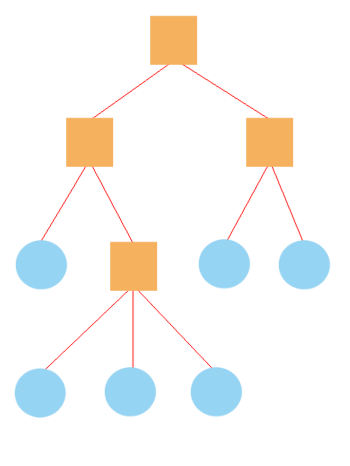


Рисунок — дерево процессов приложения. Квадратами обозначены процессы-наблюдатели, кругами — рабочие процессы

Деревья процессов существуют для:

* Упорядочивания порождаемых приложением процессов;
* Унификации процедуры генерации новых процессов приложения;
* Унификации обработки ошибок.

Дерево супервизоров разворачивается на старте системы

## Что такое процесс

В Erlang процесс является виртуальным объектом машины. Процесс является элементарной единицей исполнения в Erlang.

Подробнее о сущности процессов писал в «Общая структура Erlang VM».

## Процесс-супервизор

Процесс-супервизор — процесс Erlang, выполняющий функции:

* Генерации всех дочерних процессов: как наблюдателей, так и рабочих во время инициализации или динамически после;
* Обработки ошибок на глобальном уровне;
* Перехвата сигналов выхода дочерних процессов, перезапуск на основе полученного идентификатора.

Существует несколько реализаций процессов-супервизоров, см.

## Рабочий процесс

Рабочими процессами называются процессы, содержащие в себе основную логику программы.

## Базовые действия с процессами

Ниже указаны основные манипуляции с процессами Erlang.

### Спавн

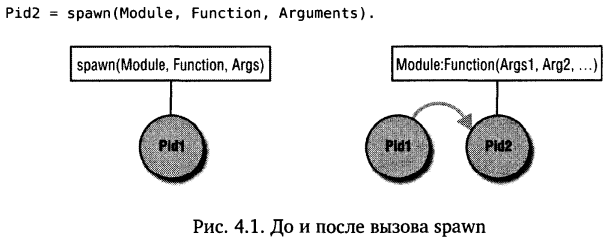


Рисунок — схематическое обозначение генерации новых процессов в Erlang

### Пересылка сообщений

Единственный способ взаимодействия процессов друг с другом — посылка сообщений:

Pid ! Message

Сообщением может быть любой терм Erlang. Сообщения делятся на:

1. Обычные (отправляются через !);
2. Сигналы — необрабатываемые программистом сообщения, которыми обмениваются сообщения в процессе своего жизненного цикла (exit, suspend и т.д.)

Приём обычных сообщений осуществляется так:

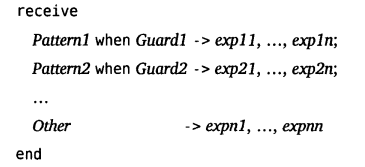


Рисунок — синтаксис receive-выражений

Алгоритм работы:

* Для каждого сообщения:
  + Попытка сопоставления с образцом:
    - Подошёл — выполнение первой ветки с совпадением;
    - Ни один не подошёл — взять из ящика следующее сообщение

Если после перебора ящика так и не нашлось совпадения для последнего сообщения, то процесс попадёт в состояние waiting и выйдет из него либо по таймеру, либо при получении нового.

Также см. Методы отслеживания состояния потоков

### Регистрация

|  |  |
| --- | --- |
| **Функция** | **Действие** |
| Register(Alias,Pid) | Выполняет регистрацию процесса – присваивает ему псевдоним. Псевдоним – атом. |
| Alias ! Message | Любой процесс может посылать сообщение, зная только Alias |
| Unregister(Pid) |  |
| Registered() |  |
| Whereis(Alias) | Возвращает Pid зарегестрированного процесса |

### Методы отслеживания состояния потоков

Связанные процессы — процессы, среди которых хотя бы одна сторона следит за остальными и реагирует на ненормальные завершения.

#### Двусторонняя связь

|  |  |
| --- | --- |
| **Функция** | **Действие** |
| link(Pid) | Создаёт двустороннюю связь между вызвавшим функцию процессом и процессом с номером Pid |
| unlink(Pid) |  |
| spawn\_link(Module,Function,ArgList) |  |

Поскольку соединение двунаправленное, не важно, процесс А был соединён с процессом В или В с А.

Если процесс завершится, то всем связанным с ним процессам будет отправлен сигнал выхода. Остановленные сигналы также передадут по цепочке сигнал выхода всем связанным с ними процессам. Эти процессы называют множеством связывания (link set).

**Информация о падении** передается в **системном сообщении**, но **сигналы**, **не попадают в почтовый ящик**. Сигналы посылаются всем связанным с упавшим процессам.

Возможные причины падения процесса:

1. Функция процесса штатно завершает своё выполнение;
2. Функция процесса завершается ненормально;
3. В коде были принудительно вызваны функции останова:
   1. exit(Reason). — завершает процесс из которого была вызвана функция. После **процесс отправляет сигнал выхода всем связанным с ним процессам.** Если функция вызвана **внутри try…catch или catch-выражения, то сигнал будет перехвачен**;
   2. exit(Pid,Reason) — посылает сигнал выхода процессу с идентификатором Pid;
   3. Функция exit с причиной kill (см. **Ошибка! Источник ссылки не найден.**)

Процесс может конвертировать сигналы выхода в сообщения (перехватывать), установив флаг процесса trap exit вызовом функции process\_flag(trap\_exit, true). Такой процесс называют системным или процессом-перехватчиком. Сообщения имеют формат:

{'EXIT', Pid, Reason}

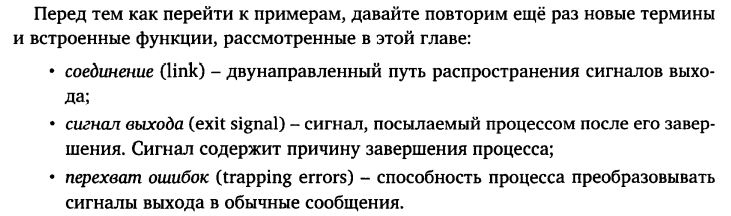
Когда происходит ошибка runtime генерируется исключение типа error. В этом случае:

Reason = {Reason, Stack},

Где Reason – это терм, сообщающий о типе ошибки

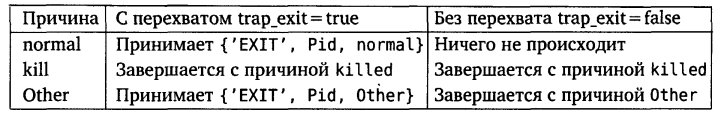
Таблица — соответствие между термом Reason и фактическим типом ошибки

|  |  |
| --- | --- |
| **Reason** | **Type of Error** |
| badarg | Bad argument. The argument is of wrong data type, or is otherwise badly formed. |
| badarith | Bad argument in an arithmetic expression. |
| {badmatch,V} | Evaluation of a match expression failed. The value V did not match. |
| function\_clause | No matching function clause is found when evaluating a function call. |
| {case\_clause,V} | No matching branch is found when evaluating a case expression. The value V did not match. |
| if\_clause | No true branch is found when evaluating an if expression. |
| {try\_clause,V} | No matching branch is found when evaluating the of-section of a try expression. The value V did not match. |
| undef | The function cannot be found when evaluating a function call. |
| {badfun,F} | Something is wrong with a fun F. |
| {badarity,F} | A fun is applied to the wrong number of arguments. F describes the fun and the arguments. |
| timeout\_value | The timeout value in a receive..after expression is evaluated to something else than an integer or infinity. |
| noproc | Trying to link or monitor to a non-existing process or port. |
| noconnection | A link or monitor to a remote process was broken because a connection between the nodes could not be established or was severed. |
| {nocatch,V} | Trying to evaluate a throw outside a catch. V is the thrown term. |
| system\_limit | A system limit has been reached. See [**Efficiency Guide**](https://www.erlang.org/doc/efficiency_guide/advanced.html) for information about system limits. |



Если процесс не является процессом-перехватчиком, любой сигнал выхода, содержащий в причине значение, отличное от атома normal, приведёт к завершению.

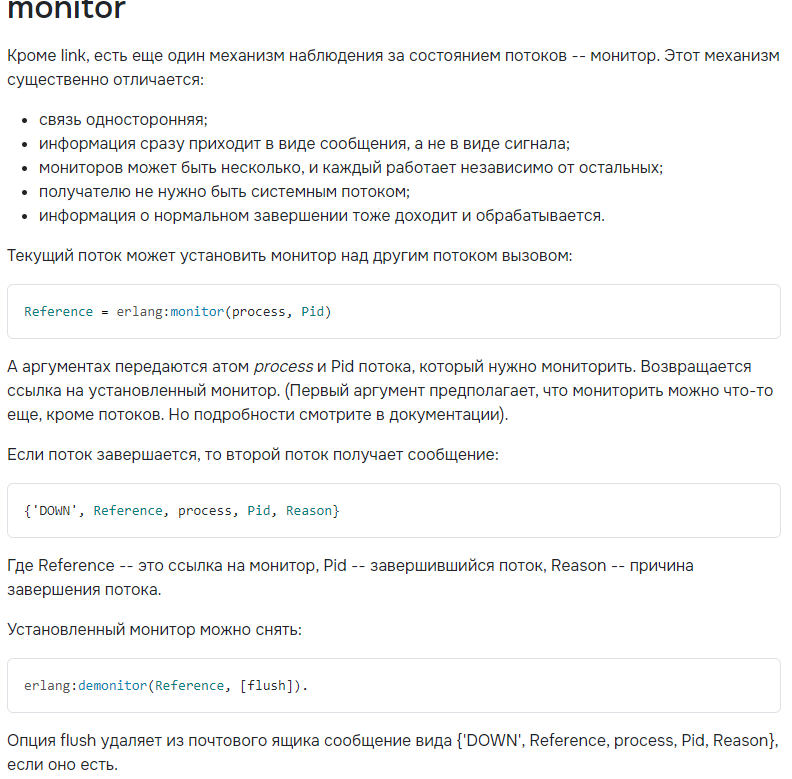
Если процесс получает сигнал выхода с причиной kill процесс будет завершён вне зависимости от его флага. Другим процессам будет передан сигнал выхода с атомом killed и они смогут обработать это исключение.



Преимущество двусторонней связи состоит в том, что если процесс-супервизор упадёт, то он сможет унести за собой все связанные с ним рабочие процессы.

Стандартный супервизор под капотом использует методы link и trap\_exit.

#### Односторонняя связь



## Типовые реализации процессов

В данном разделе речь пойдёт об общих подходах к построению кодов процессов в Erlang.

### Каркас процессов

Любая программа – цикл обработки поступающих сообщений.

Процессы в Erlang — сущности, принимающие сообщения и выполняющие указанный код.

Следовательно, подавляющее большинство процессов должно соответствовать каркасу, изображенному ниже.

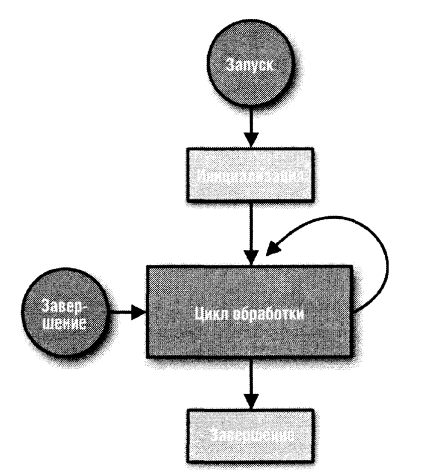


Рисунок — схема алгоритма типичного Erlang-процесса

1. Запуск — функция start с необходимыми для инициализации аргументами (регистрирует процесс если надо);
2. Инициализация – функция init, которая выделяет доп. ресурсы при необходимости;
3. Цикл обработки — функция loop вызывающая себя по хвостовой рекурсии. В теле рекурсии находится блок обработки сообщений (receive). Аргументы, передающиеся по рекурсии называются состоянием процесса;
4. Если в функцию попадает сообщение stop, то процесс освобождает ресурсы и закрывается.

Пример кода:

|  |
| --- |
| -module(gs1).  -export([start/0]).  start() ->  io:format("start ~p~n", [**self**()]),  **spawn**(**fun** loop/0).  loop() ->  io:format("~p enters loop ~n", [**self**()]),  **receive**  stop -> io:format("~p stops now ~n", [**self**()]);  Msg -> io:format("~p receive ~p~n", [**self**(), Msg]),  loop()  **end**. |

### Супервизор

Пример самого простого супервизора, основанного на лвусторонней связи:

|  |
| --- |
| -module(mysupervisor).  -export([start\_link/2]).  -export([init/1]).  %%генерация супервизора с функцией init  %%регистрация супервизора по полученному псевдониму  start\_link(Name, ChildSpecList) ->  **register**(Name, **spawn\_link**(my\_supervisor, init, [ChildSpecList])), ok.  init(ChildSpecList) ->  **process\_flag**(trap\_exit, true),  ChildrenList = start\_children(ChildSpecList),  loop(ChildrenList).  start\_children([]) -> [];  start\_children([{M, F, A} | ChildSpecList]) ->  %%apply/3 - некоторая функция, в которой вызывается spawn\_link/3  %%и которая в случае успеха возвращает {ok, Pid}  **case** (**catch** **apply**(M,F,A)) **of**  {ok, Pid} ->  [{Pid, {M,F,A}}|start\_children(ChildSpecList)];  \_ ->  start\_children(ChildSpecList)  **end**.  restart\_child(Pid, ChildList) ->  {value, {Pid, {M,F,A}}} = lists:keysearch(Pid, 1, ChildList),  {ok, NewPid} = **apply**(M,F,A),  [{NewPid, {M,F,A}} | lists:keydelete(Pid,1,ChildList)].  loop(ChildList) ->  **receive**  {'EXIT', Pid, \_Reason} ->  NewChildList = restart\_child(Pid, ChildList),  loop(NewChildList);  {stop, From}->  From ! {reply, terminate(ChildList)}  **end**.  terminate([{Pid, \_} | ChildList])->  **exit**(Pid,kill),  terminate(ChildList);  terminate(\_ChildList) -> ok. |