Бесплатная электронная книга

# УЧУСЬ

# Erlang Language

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#erlang

# Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: erlang-language

It is an unofficial and free Erlang Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Erlang Language.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# глава 1: Начало работы с языком Erlang

## замечания

«Erlang - это язык программирования, первоначально разработанный в Лаборатории компьютерных наук Эрикссон. OTP (Open Telecom Platform) - это набор промежуточного программного обеспечения и библиотек в Эрланге. Erlang / OTP был протестирован в ряде продуктов Ericsson для создания надежных отказоустойчивых распределенные приложения, например AXD301 (ATM-коммутатор). Erlang / OTP в настоящее время поддерживается модулем Erlang / OTP в Ericsson »( erlang.org )

## Начни здесь

Инструкции по установке см. В разделе « Установка» .

## связи

1. Официальный сайт Erlang: https://www.erlang.org
2. Популярный менеджер пакетов для Erlang и Elixir: http://hex.pm
3. Шаблоны Erlang: http://www.erlangpatterns.org/

## Версии

| Версия | Примечания к выпуску | Дата выхода |
|---|---|---|
| 19,2 | http://erlang.org/download/otp_src_19.2.readme | 2016-12-14 |
| 19,1 | http://erlang.org/download/otp_src_19.1.readme | 2016-09-21 |
| 19,0 | http://erlang.org/download/otp_src_19.0.readme | 2016-06-21 |
| 18,3 | http://erlang.org/download/otp_src_18.3.readme | 2016-03-15 |
| 18.2.1 | http://erlang.org/download/otp_src_18.2.1.readme | 2015-12-18 |
| 18,2 | http://erlang.org/download/otp_src_18.2.readme | 2015-12-16 |
| 18,1 | http://erlang.org/download/otp_src_18.1.readme | 2015-09-22 |
| 18,0 | http://erlang.org/download/otp_src_18.0.readme | 2015-06-24 |
| 17,5 | http://erlang.org/download/otp_src_17.5.readme | 2015-04-01 |
| 17,4 | http://erlang.org/download/otp_src_17.4.readme | 2014-12-10 |

| Версия | Примечания к выпуску | Дата выхода |
|---|---|---|
| +17,3 | http://erlang.org/download/otp_src_17.3.readme | 2014-09-17 |
| 17,1 | http://erlang.org/download/otp_src_17.1.readme | 2014-06-24 |
| 17,0 | http://erlang.org/download/otp_src_17.0.readme | 2014-04-07 |
| R16B03-1 | http://erlang.org/download/otp_src_R16B03-1.readme | 2014-01-23 |
| R16B03 | http://erlang.org/download/otp_src_R16B03.readme | 2013-12-09 |
| R16B02 | http://erlang.org/download/otp_src_R16B02.readme | 2013-09-17 |
| R16B01 | http://erlang.org/download/otp_src_R16B01.readme | 2013-06-18 |
| R16B | http://erlang.org/download/otp_src_R16B.readme | 2013-02-25 |

# Examples

### Привет, мир

Есть две вещи, которые вам нужно знать при написании приложения «hello world» в Erlang:

1. Исходный код написан на *языке программирования erlang* с использованием текстового редактора по вашему выбору
2. Затем приложение выполняется в *виртуальной машине erlang* . В этом примере мы будем взаимодействовать с erlang VM thorugh the erlang shell.

# Сначала исходный код приложения:

Создайте новый файл `hello.erl` содержащий следующее:

```
-module(hello).
-export([hello_world/0]).

hello_world() ->
  io:format("Hello, World!~n", []).
```

Давайте быстро посмотрим, что это значит:

- `-module(hello)` . Все функции erlang существуют внутри *модуля* . Модули затем используются для создания приложений, которые представляют собой набор модулей. Эта первая строка состоит в том, чтобы идентифицировать этот модуль, а именно *hello* . Модули можно сравнить с *пакетами* Java
- `-export([hello_world/0])` . Сообщает компилятору, который функционирует, чтобы сделать «общедоступным» (по сравнению с языками OO) и *арность* соответствующей

функции. Арность - это количество аргументов, которые выполняет функция. Поскольку в erlang функция с 1 аргументом рассматривается как другая функция, чем одна с двумя аргументами, хотя имя может быть точно таким же. Т.е., `hello_world/0` - совершенно другая функция, например, `hello_world/1` .

- `hello_world()` Это имя функции. Параметр `->` указывает на переход к реализации (телу) функции. Это можно прочитать так: «hello_world () определяется как ...». Обратите внимание, что `hello_world()` (без аргументов) идентифицируется `hello_world/0` в VM и `hello_world(Some_Arg)` как `hello_world/1` .
- `io:format("Hello, World!~n", [])` Из модуля `io` вызывается функция `format/2` , которая является функцией стандартного вывода. `~n` - спецификатор формата, который означает печать новой строки. `[]` - список переменных для печати, указанных спецификаторами формата в выходной строке, что в данном случае ничего.
- Все утверждения erlang должны заканчиваться на a `.` (Точка).

В Erlang возвращается результат последнего оператора в функции.

## Теперь давайте запустим наше приложение:

Запустите оболочку `hello.erl` из того же каталога, что и файл `hello.erl` :

```
$ erl
```

Вы должны получить подсказку, которая выглядит примерно так (ваша версия может быть другой):

```
Eshell V8.0  (abort with ^G)
1>
```

Теперь введите следующие команды:

```
1> c(hello).
{ok,hello}
2> hello:hello_world().
Hello, World!
ok
```

Давайте переходим через каждую строку один за другим:

- `c(hello)` - эта команда вызывает функцию `c` на атоме `hello` . Это эффективно говорит Erlang найти файл `hello.erl` , скомпилировать его в модуль (файл с именем `hello.beam` будет сгенерирован в каталоге) и загрузить его в среду.
- `{ok, hello}` - это результат вызова функции `c` выше. Это кортеж, содержащий атом `ok` и атомный `hello` . Обычно функции Erlang возвращают либо `{ok, Something}` либо `{error, Reason}` .
- `hello:hello_world()` - это `hello_world()` функции `hello_world()` из модуля `hello` .
- `Hello, World!` - это то, что печатает наша функция.

- `ok` - это то, что вернула наша функция. Поскольку Erlang является функциональным языком программирования, каждая функция возвращает *что-то* . В нашем случае, хотя мы не вернули ничего в `hello_world()` , последний вызов этой функции был в `io:format(...)` и эта функция вернулась в `ok` , что в свою очередь возвращает нашу функцию.

## Модули

Модуль erlang - это файл с несколькими функциями, сгруппированными вместе. Обычно этот файл имеет расширение `.erl` .

Модуль «Hello World» с именем `hello.erl` показан ниже

```
-module(hello).
-export([hello_world/0]).

hello_world() ->
  io:format("Hello, World!~n", []).
```

В файле требуется объявить имя модуля. Как показано выше в строке 1. Имя модуля и имя файла перед расширением `.erl` должны быть одинаковыми.

## функция

Функция представляет собой набор команд, которые сгруппированы вместе. Эти сгруппированные инструкции вместе выполняют определенную задачу. В erlang все функции возвращают значение при их вызове.

Ниже приведен пример функции, которая добавляет два числа

```
add(X, Y)-> X + Y.
```

Эта функция выполняет операцию добавления с значениями X и Y и возвращает результат. Функция может использоваться как ниже

```
add(2,5).
```

Объявления функций могут состоять из нескольких предложений, разделенных точкой с запятой. Аргументы в каждом из этих предложений оцениваются путем сопоставления шаблонов. Следующая функция вернет «кортеж», если Аргумент является кортежем в форме: {test, X}, где X может быть любым значением. Он вернет «список», если Аргумент - это список длины 2 в форме [«test», X], и он вернет «{ошибка,« Причина »} в любом другом случае:

```
function({test, X}) -> tuple;
function(["test", X]) -> list;
```

```
function(_) -> {error, "Reason"}.
```

Если аргумент не является кортежем, будет оценено второе предложение. Если аргумент не является списком, будет оценено третье предложение.

Объявление функций может состоять из так называемых «гвардейских» или «гвардейских последовательностей». Эти гвардейцы являются выражениями, которые ограничивают оценку функции. Функция с гвардейцами выполняется только тогда, когда все выражения Guard выражают истинное значение. Несколько гвардейцев могут быть разделены точкой с запятой.

```
function_name(Argument) when Guard1; Guard2; ... GuardN -> (...).
```

Функция «имя_функции» будет оцениваться только тогда, когда Guard Sequence верна. Функция follwing вернет true, только если аргумент `X` находится в правильном диапазоне (0..15):

```
in_range(X) when X>=0; X<16 -> true;
in_range(_) -> false.
```

## Учет списка

Перечисления списков представляют собой синтаксическую конструкцию для создания списка на основе существующих списков.
В erlang понимание списка имеет форму `[Expr || Qualifier1, ..., QualifierN]` .
Если квалификаторы являются либо генераторами `Pattern <- ListExpr` либо фильтруют как `integer(X)` оценивая либо `true` либо `false` .

В следующем примере показано понимание списка с одним генератором и двумя фильтрами.

```
[X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].
```

Результатом является список, содержащий только целые числа больше 3.

```
[4,5,6]
```

## Запуск и остановка оболочки Erlang

### Запуск оболочки Erlang

В системе UNIX вы запускаете оболочку Erlang из командной строки с помощью команды `erl`

Пример:

```
$ erl
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.0  (abort with ^G)
1>
```

Текст, который показывает, когда вы запускаете оболочку, сообщает вам информацию о том, какую версию Erlang вы используете, а также другую полезную информацию о системе erlang.

Чтобы запустить оболочку в Windows, щелкните значок Erlang в меню запуска Windows.

**Остановка оболочки Эрланг**

Для управляемого выхода оболочки erlang вы вводите:

```
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.0  (abort with ^G)
1> q().
```

Вы также можете выйти из оболочки Erlang, нажав Ctrl + C в системах UNIX или Ctrl + Break в Windows, в результате чего вы получите следующую подсказку:

```
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.0  (abort with ^G)
1>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

Если вы затем нажмете (для отмены), вы сразу же выйдете из оболочки.

Другие способы выхода из оболочки erlang: `init:stop()` что делает то же самое, что и `q()` или `erlang:halt()` .

## Соответствие шаблону

Одной из наиболее распространенных операций в erlang является сопоставление образцов. Он используется при назначении значения переменной, в объявлениях функций и в структурах потока управления, таких как `case` and `receive` statements. Для операции сопоставления шаблонов требуется как минимум 2 части: шаблон и термин, в котором сопоставляется шаблон.

Назначение переменной в erlang выглядит следующим образом:

```
X = 2.
```

В большинстве языков программирования семантика этой операции проста: привяжите значение ( `2` ) к названию по вашему выбору (переменная - в этом случае `X` ). Эрланг имеет несколько иной подход: сопоставьте шаблон с левой стороны ( `X` ) с термином справа ( `2` ). В этом случае эффект тот же: переменная `X` теперь привязана к значению `2` . Однако при сопоставлении с образцами вы можете выполнять более структурированные задания.

```
{Type, Meta, Doc} = {document, {author, "Alice"}, {text, "Lorem Ipsum"}}.
```

Эта операция сопоставления выполняется, анализируя структуру члена правой стороны и применяя все переменные в левой части к соответствующим значениям термина, так что левая сторона равна правой стороне. В этом примере `Type` привязан к термину: `document` , `Meta` to `{author, "Alice"}` и `Doc` to `{text, "Lorem Ipsum"}` . В этом конкретном примере переменные: `Type` , `Meta` и `Doc` считаются *несвязанными* , поэтому каждая переменная может использоваться.

Шаблонные сопоставления также могут быть построены с использованием связанных переменных.

```
Identifier = error.
```

`Identifier` переменной теперь привязан к `error` значения. Следующая операция сопоставления шаблонов работает, поскольку структура соответствует, а связанная переменная `Identifier` имеет то же значение, что и соответствующая правая часть термина.

```
{Identifier, Reason} = {error, "Database connection timed out."}.
```

Операция сопоставления с образцом завершается с ошибкой, когда есть несоответствие между правой стороной и левым рисунком стороны. Следующее совпадение не будет выполнено, потому что `Identifier` привязан к `error` значения, у которой нет соответствующего выражения с правой стороны.

```
{Identifier, Reason} = {fail, "Database connection timed out."}.
> ** exception error: no match of right hand side value {fail,"Database ..."}
```

Прочитайте Начало работы с языком Erlang онлайн: https://riptutorial.com/ru/erlang/topic/825/начало-работы-с-языком-erlang

# глава 2: iolists

## Вступление

В то время как строка Erlang представляет собой список целых чисел, «iolist» представляет собой список, чьи элементы являются целыми, двоичными или другими iolists, например `["foo", $b, $a, $r, <<"baz">>]` . Этот иолит представляет собой строку `"foobarbaz"` .

Хотя вы можете преобразовать iolist в двоичный файл с помощью `iolist_to_binary/1` , вам часто не нужно, поскольку функции библиотеки Erlang, такие как `file:write_file/2` и `gen_tcp:send/2` принимают iolists, а также строки и двоичные файлы.

## Синтаксис

- -type iolist () :: maybe_improper_list (byte () | binary () | iolist (), binary () | []).

## замечания

#### Что такое иолит?

*Это любой двоичный файл. Или любой список, содержащий целые числа от 0 до 255. Или любой произвольно вложенный список, содержащий любую из этих двух вещей.*

#### Оригинальная статья

Используйте глубоко вложенные списки целых чисел и двоичных файлов для представления данных ввода-вывода, чтобы избежать копирования при конкатенации строк или двоичных файлов.

Они эффективны даже при объединении большого количества данных. Например, объединение двух пятибуквенных двоичных файлов с использованием бинарного синтаксиса `<<B1/binary, B2/binary>>` обычно требует перераспределения обоих в новый 100 kb двоичный файл. Использование списков ввода-вывода `[B1, B2]` присваивает только список, в данном случае три слова. В списке используется одно слово и другое слово для каждого элемента, см. Здесь для получения дополнительной информации.

Использование оператора `++` создало бы целый новый список, а не только новый список из двух элементов. Воспроизведение списков для добавления элементов в конец может стать дорогостоящим, когда список длинный.

В случаях, когда двоичные данные малы, распределение списков ввода-вывода может быть больше, чем добавление двоичных файлов. Если двоичные данные могут быть либо небольшими, либо большими, часто лучше принимать согласованную стоимость списков

ввода-вывода.

Обратите внимание, что добавляемые двоичные файлы оптимизируются, как описано здесь . Короче говоря, двоичный файл может иметь дополнительное, скрытое пространство. Это будет заполнено, если к нему добавится другой бинар, который помещается в свободное пространство. Это означает, что не каждый бинарный append вызывает полную копию обоих двоичных файлов.

# Examples

### Списки IO обычно используются для создания вывода на порт, например, в файле или в сетевом сокете.

```
file:write_file("myfile.txt", ["Hi " [<<"there">>], $\n]).
```

### Добавьте разрешенные типы данных в начало списка ввода-вывода, создав новый.

```
["Guten Tag " | [<<"Hello">>]].
[<<"Guten Tag ">> | [<<"Hello">>]].
[$G, $u, $t, $e, $n , $T, $a, $g | [<<"Hello">>]].
[71,117,116,101,110,84,97,103,<<"Hello">>].
```

### Данные ввода-вывода могут быть эффективно добавлены в конец списка.

```
Data_1 = [<<"Hello">>].
Data_2 = [Data_1,<<" Guten Tag ">>].
```

### Будьте осторожны с неправильными списками

```
["Guten tag " | <<"Hello">>].
```

В оболочке это будет напечатано как `["Guten tag "|<<"Hello">>]` вместо `["Guten tag ",<<"Hello">>]` . Оператор трубы создаст неправильный список, если последний элемент справа не является списком. Хотя неправильный список, чей «хвост» является двоичным, по-прежнему является достоверным iolist, неправильные списки могут вызывать проблемы, потому что многие рекурсивные функции ожидают, что пустой список будет последним элементом, а не, как в этом случае двоичным.

### Получить размер списка ввода-вывода

```
Data = ["Guten tag ",<<"Hello">>],
```

```
Len = iolist_size(Data),
[<<Len:32>> | Data].
```

Размер iolist может быть рассчитан с использованием `iolist_size/1` . Этот фрагмент вычисляет размер сообщения и создает и добавляет его к фронту в виде четырехбайтового двоичного кода. Это типичная операция в протоколах обмена сообщениями.

## Список IO может быть преобразован в двоичный

```
<<"Guten tag, Hello">> = iolist_to_binary(["Guten tag, ",<<"Hello">>]).
```

Список IO может быть преобразован в двоичный файл с помощью функции `iolist_to_binary/1` . Если данные будут храниться в течение длительного периода или отправляться в виде сообщений другим процессам, тогда имеет смысл преобразовать их в двоичный файл. Единственная стоимость конвертации в двоичную систему может быть дешевле, чем копирование списка IO много раз, в сборке мусора одного процесса или в передаче сообщений другим.

Прочитайте iolists онлайн: https://riptutorial.com/ru/erlang/topic/5677/iolists

# глава 3: Rebar3

## Examples

### Определение

**Официальная страница** : https://www.rebar3.org/

**Исходный код** : https://github.com/erlang/rebar3

Rebar3 - это, главным образом, менеджер зависимостей проектов Erlang и Elixir, но он также предлагает несколько других функций, таких как проекты начальной загрузки (в соответствии с несколькими шаблонами, следуя принципам OTP), исполнителем задач, инструментом сборки, тестовым runner и расширяемым с помощью плагинов.

### Установка Rebar3

Rebar3 написан в Erlang, поэтому вам нужно, чтобы Erlang запускал его. Он доступен как двоичный файл, который можно загрузить и запустить. Просто приготовьте ночную сборку и дайте ей разрешения на выполнение:

```
$ wget https://s3.amazonaws.com/rebar3/rebar3 && chmod +x rebar3
```

Поместите этот двоичный файл в удобное место и добавьте его на свой путь. Например, в каталоге bin в вашем доме:

```
$ mkdir ~/bin && mv rebar3 ~/bin
$ export PATH=~/bin:$PATH
```

Эта последняя строка должна быть помещена в ваш `.bashrc` . В качестве альтернативы можно также связать двоичный каталог `/usr/local/bin` , сделав его доступным как обычная команда.

```
$ sudo ln -s /path/to/your/rebar3 /usr/local/bin
```

### Установка из исходного кода

Поскольку Rebar3 является бесплатным, открытым исходным кодом и написан в Erlang, его можно просто клонировать и строить из исходного кода.

```
$ git clone https://github.com/erlang/rebar3.git
$ cd rebar3
$ ./bootstrap
```

Это создаст скрипт rebar3, который вы можете нанести на свой PATH или ссылку на `/usr/local/bin` как описано в разделе «Установка Rebar3» выше.

### Загрузите новый проект Erlang

Чтобы загрузить новый проект Erlang, просто выберите шаблон, который вы хотите использовать из списка. Доступные шаблоны можно получить с помощью следующей команды:

```
$ rebar3 new

app (built-in): Complete OTP Application structure
cmake (built-in): Standalone Makefile for building C/C++ in c_src
escript (built-in): Complete escriptized application structure
lib (built-in): Complete OTP Library application (no processes) structure
plugin (built-in): Rebar3 plugin project structure
release (built-in): OTP Release structure for executable programs
```

После того, как вы выбрали соответствующий шаблон, загрузите его с помощью следующей команды (rebar3 создаст новый каталог для вашего проекта):

```
$ rebar3 new lib libname

===> Writing libname/src/libname.erl
===> Writing libname/src/libname.app.src
===> Writing libname/rebar.config
===> Writing libname/.gitignore
===> Writing libname/LICENSE
===> Writing libname/README.md
```

OBS: Хотя вы *можете* запустить `rebar3 new <template> .` для создания нового проекта в текущем каталоге это не рекомендуется, так как будут загружены загрузочные файлы `.` (точка) в качестве имен приложений и модулей, а также в `rebar.config` , что вызовет проблемы с синтаксисом.

Прочитайте Rebar3 онлайн: https://riptutorial.com/ru/erlang/topic/4480/rebar3

# глава 4: Внешний формат

## Вступление

Формат внешнего термина - это двоичный формат, используемый для общения с внешним миром. Вы можете использовать его на любом языке через порты, драйверы или NIF. BERT (Binary ERlang Term) может использоваться на других языках.

## Examples

### Использование ETF с Erlang

```
binary_to_term().
```

### Использование ETF с C

# Инициализация структуры данных

```
#include <stdio.h>
#include <string.h>
#include <ei.h>
```

# Номер кодировки

```
#include <stdio.h>
#include <ei.h>

int
main() {
}
```

# Кодирующий атом

# Кодирование кортежа

# Список кодировок

# Кодирующая карта

Прочитайте Внешний формат онлайн: https://riptutorial.com/ru/erlang/topic/10731/внешний-формат

# глава 5: директор

## Вступление

Гибкая, быстрая и мощная библиотека супервизора для процессов Erlang.

## замечания

## Предупреждения

- Не используйте `'count'=>infinity` и `restart` элемента в вашем плане.
  лайк:

```
Childspec = #{id => foo
              ,start => {bar, baz, [arg1, arg2]}
              ,plan => [restart]
              ,count => infinity}.
```

Если ваш процесс не начался после сбоя, **директор** заблокирует и повторит попытку перезапуска вашего `infinity` времени! Если вы используете `infinity` для `'count'` , всегда используйте `{restart, MiliSeconds}` в `'plan'` вместо `restart` .

- Если у вас есть планы:

```
Childspec1 = #{id => foo
               ,start => {bar, baz}
               ,plan => [restart,restart,delete,wait,wait, {restart, 4000}]
               ,count => infinity}.

Childspec2 = #{id => foo
               ,start => {bar, baz}
               ,plan => [restart,restart,stop,wait, {restart, 20000}, restart]
               ,count => infinity}.

Childspec3 = #{id => foo
               ,start => {bar, baz}
               ,plan => [restart,restart,stop,wait, {restart, 20000}, restart]
               ,count => 0}.

Childspec4 = #{id => foo
               ,start => {bar, baz}
               ,plan => []
               ,count => infinity}.
```

Остальная часть элемента `delete` в `Childspec1` и остальном элементе `stop` в `Childspec2` никогда не будет оцениваться!
В `Childspec3` вы хотите запустить свой план 0 раз!
В `ChildSpec4` вас нет плана запуска `infinity` раз!

- Когда вы `release_handler` выпуск с помощью `release_handler` , `release_handler` вызывает `supervisor:get_callback_module/1` для извлечения его модуля обратного вызова.
  В OTP <19 `get_callback_module/1` использует запись внутреннего состояния супервизора для предоставления своего модуля обратного вызова. Наш **директор** не знает о государственной записи `supervisor:get_callback_module/1` , тогда `supervisor:get_callback_module/1` не работает с **директором** s.
  Хорошая новость заключается в том, что в OTP> = 19 `supervisor:get_callback_module/1` отлично работает с **директором** s :).

```
1> foo:start_link().
{ok,<0.105.0>}

2> supervisor:get_callback_module(foo_sup).
foo

3>
```

# Examples

## Скачать

```
Pouriya@Jahanbakhsh ~ $ git clone https://github.com/Pouriya-Jahanbakhsh/director.git
```

## компилировать

Обратите внимание, что требуется **OTP> = 19** (если вы хотите обновить его с помощью `release_handler` ).
Пойдите к `director` и используйте `rebar` или `rebar3` .

```
Pouriya@Jahanbakhsh ~ $ cd director
```

арматура

```
Pouriya@Jahanbakhsh ~/director $ rebar compile
==> director_test (compile)
Compiled src/director.erl
Pouriya@Jahanbakhsh ~/director $
```

rebar3

```
Pouriya@Jahanbakhsh ~/director $ rebar3 compile
===> Verifying dependencies...
===> Compiling director
Pouriya@Jahanbakhsh ~/director $
```

## Как это устроено

**директору** нужен модуль обратного вызова (например, супервизор OTP).
В модуле обратного вызова вы должны экспортировать функцию `init/1` .
Какой `init/1` должен вернуться? подождите, я объясню шаг за шагом.

```
-module(foo).
-export([init/1]).

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

Сохраните код в `foo.erl` в **директории** директории и перейдите в оболочку Erlang.
Используйте `erl -pa ./ebin` если вы использовали `rebar` для ее компиляции и использовали `rebar3 shell` если вы использовали `rebar3` .

```
Erlang/OTP 19 [erts-8.3] [source-d5c06c6] [64-bit] [smp:8:8] [async-threads:0] [hipe] [kernel-
poll:false]

Eshell V8.3  (abort with ^G)
1> c(foo).
{ok,foo}

2> Mod = foo.
foo

3> InitArg = undefined. %% i don't need it yet.
undefined

4> {ok, Pid} = director:start_link(Mod, InitArg).
{ok,<0.112.0>}

5>
```

Теперь у нас есть руководитель без детей.
Хорошая новость заключается в том, что **директор** поставляется с полным API OTP / supervisor и имеет свои расширенные функции и особый подход.

```
5> director:which_children(Pid). %% You can use supervisor:which_children(Pid) too :)
[]

6> director:count_children(Pid). %% You can use supervisor:count_children(Pid) too :)
[{specs,0},{active,0},{supervisors,0},{workers,0}]

7> director:get_pids(Pid). %% You can NOT use supervisor:get_pids(Pid) because it hasn't :D
[]
```

Хорошо, я сделаю простой `gen_server` и передам его нашему **режиссеру** .

```
-module(bar).
-behaviour(gen_server).
-export([start_link/0
        ,init/1
        ,terminate/2]). %% i am not going to use handle_call, handle_cast ,etc.
```

```
start_link() ->
    gen_server:start_link(?MODULE, null, []).

init(_GenServerInitArg) ->
    {ok, state}.

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

Сохраните код выше в `bar.erl` и вернитесь в оболочку.

```
8> c(bar).
bar.erl:2: Warning: undefined callback function code_change/3 (behaviour 'gen_server')
bar.erl:2: Warning: undefined callback function handle_call/3 (behaviour 'gen_server')
bar.erl:2: Warning: undefined callback function handle_cast/2 (behaviour 'gen_server')
bar.erl:2: Warning: undefined callback function handle_info/2 (behaviour 'gen_server')
{ok,bar}

%% You should define unique id for your process.
9> Id = bar_id.
bar_id

%% You should tell diector about start module and function for your process.
%% Should be tuple {Module, Function, Args}.
%% If your start function doesn't need arguments (like our example)
%% just use {Module, function}.
10> start = {bar, start_link}.
{bar,start_link}

%% What is your plan for your process?
%% I asked you some questions at the first of this README file.
%% Plan should be an empty list or list with n elemenst.
%% Every element can be one of
%% 'restart'
%% 'delete'
%% 'stop'
%% {'stop', Reason::term()}
%% {'restart', Time::pos_integer()}
%% for example my plan is:
%% [restart, {restart, 5000}, delete]
%% In first crash director will restart my process,
%% after next crash director will restart it after 5000 mili-seconds
%% and after third crash director will not restart it and will delete it
11> Plan = [restart, {restart, 5000}, delete].
[restart,{restart,5000},delete]

%% What if i want to restart my process 500 times?
%% Do i need a list with 500 'restart's?
%% No, you just need a list with one element, I'll explain it later.

12> Childspec = #{id => Id
                  ,start => Start
                  ,plan => Plan}.
#{id => bar_id,
  plan => [restart,{restart,5000},delete],
  start => {bar,start_link}}

13> director:start_child(Pid, Childspec). %% You can use supervisor:start_child(Pid,
ChildSpec) too :)
{ok,<0.160.0>}
```

```
14>
```

Давайте проверим это

```
14> director:which_children(Pid).
[{bar_id,<0.160.0>,worker,[bar]}]

15> director:count_children(Pid).
[{specs,1},{active,1},{supervisors,0},{workers,1}]

%% What was get_pids/1?
%% It will returns all RUNNING ids with their pids.
16> director:get_pids(Pid).
[{bar_id,<0.160.0>}]

%% We can get Pid for specific RUNNING id too
17> {ok, BarPid1} = director:get_pid(Pid, bar_id).
{ok,<0.160.0>}


%% I want to kill that process
18> erlang:exit(BarPid1, kill).
true

%% Check all running pids again
19> director:get_pids(Pid).
[{bar_id,<0.174.0>}] %% changed (restarted)

%% I want to kill that process again
%% and i will check children before spending time
20> {ok, BarPid2} = director:get_pid(Pid, bar_id), erlang:exit(BarPid2, kill).
true

21> director:get_pids(Pid).
[]

22> director:which_children(Pid).
[{bar_id,restarting,worker,[bar]}] %% restarting

23> director:get_pid(Pid, bare_id).
{error,not_found}

%% after 5000 ms
24> director:get_pids(Pid).
[{bar_id,<0.181.0>}]

25> %% Yoooohooooooo
```

Я упомянул о **расширенных функциях** , каковы они? Давайте посмотрим другие приемлемые ключи для карты `Childspec` .

```
-type childspec() :: #{'id' => id()
                       ,'start' => start()
                       ,'plan' => plan()
                       ,'count' => count()
                       ,'terminate_timeout' => terminate_timeout()
                       ,'type' => type()
```

```
                      ,'modules' => modules()
                      ,'append' => append()}.

%% 'id' is mandatory and can be any Erlang term
-type  id() :: term().

%% Sometimes 'start' is optional ! just wait and read carefully
-type  start() :: {module(), function()} % default Args is []
                | mfa().

%% I explained 'restart', 'delete' and {'restart', MiliSeconds}
%% 'stop': director will crash with reason {stop, [info about process crash]}.
%% {'stop', Reason}: director exactly will crash with reason Reason.
%% 'wait': director will not restart process,
%%  but you can restart it using director:restart_child/2 and you can use
supervisor:restart_child/2 too.
%% fun/2: director will execute fun with 2 arguments.
%%  First argument is crash reason for process and second argument is restart count for
process.
%%  Fun should return terms like other plan elements.
%% Default plan is:
%% [fun
%%      (normal, _RestartCount) ->
%%          delete;
%%      (shutdown, _RestartCount) ->
%%          delete;
%%      ({shutdown, _Reason}, _RestartCount) ->
%%          delete;
%%      (_Reason, _RestartCount) ->
%%          restart
%%  end]
-type  plan() :: [plan_element()] | [].
-type   plan_element() :: 'restart'
                        | {'restart', pos_integer()}
                        | 'wait'
                        | 'stop'
                        | {'stop', Reason::term()}
                        | fun((Reason::term()
                              ,RestartCount::pos_integer()) ->
                                  'restart'
                                | {'restart', pos_integer()}
                                | 'wait'
                                | 'stop'
                                | {'stop', Reason::term()}).

%% How much time you want to run plan?
%% Default value of 'count' is 1.
%% Again, What if i want to restart my process 500 times?
%%  Do i need a list with 500 'restart's?
%%  You just need plan ['restart'] and 'count' 500 :)
-type  count() :: 'infinity' | non_neg_integer().

%% How much time director should wait for process termination?
%% 0 means brutal kill and director will kill your process using erlang:exit(YourProcess,
kill).
%% For workers default value is 1000 mili-seconds and for supervisors default value is
'infinity'.
-type  terminate_timeout() :: 'infinity' | non_neg_integer().

%% default is 'worker'
-type  type() :: 'worker' | 'supervisor'.
```

```
%% Default is first element of 'start' (process start module)
-type  modules() :: [module()] | 'dynamic'.

%% :)
%% Default value is 'false'
%% I'll explan it
-type  append() :: boolean().
```

Изменить модуль `foo` :

```
-module(foo).
-export([start_link/0
        ,init/1]).

start_link() ->
    director:start_link({local, foo_sup}, ?MODULE, null).

init(_InitArg) ->
    Childspec = #{id => bar_id
                 ,plan => [wait]
                 ,start => {bar,start_link}
                 ,count => 1
                 ,terminate_timeout => 2000},
    {ok, [Childspec]}.
```

Снова перейдите в оболочку Erlang:

```
1> c(foo).
{ok,foo}

2> foo:start_link().
{ok,<0.121.0>}

3> director:get_childspec(foo_sup, bar_id).
{ok,#{append => false,count => 1,id => bar_id,
      modules => [bar],
      plan => [wait],
      start => {bar,start_link,[]},
      terminate_timeout => 2000,type => worker}}

4> {ok, Pid} = director:get_pid(foo_sup, bar_id), erlang:exit(Pid, kill).
true

5> director:which_children(foo_sup).
[{bar_id,undefined,worker,[bar]}] %% undefined

6> director:count_children(foo_sup).
[{specs,1},{active,0},{supervisors,0},{workers,1}]

7> director:get_plan(foo_sup, bar_id).
{ok,[wait]}

%% I can change process plan
%% I killed process one time.
%% If i kill it again, entire supervisor will crash with reason {reached_max_restart_plan...
because 'count' is 1
%% But after changing plan, its counter will restart from 0.
```

```
8> director:change_plan(foo_sup, bar_id, [restart]).
ok

9> director:get_childspec(foo_sup, bar_id).
{ok,#{append => false,count => 1,id => bar_id,
      modules => [bar],
      plan => [restart], %% here
      start => {bar,start_link,[]},
      terminate_timeout => 2000,type => worker}}

10> director:get_pids(foo_sup).
[]

11> director:restart_child(foo_sup, bar_id).
{ok,<0.111.0>}

12> {ok, Pid2} = director:get_pid(foo_sup, bar_id), erlang:exit(Pid2, kill).
true

13> director:get_pid(foo_sup, bar_id).
{ok,<0.113.0>}

14> %% Hold on
```

Наконец, что такое ключ `append` ?

на самом деле всегда есть один `DefaultChildspec` .

```
14> director:get_default_childspec(foo_sup).
{ok,#{count => 0,modules => [],plan => [],terminate_timeout => 0}}

15>
```

`DefaultChildspec` похож на обычный childspecs, за исключением того, что он не может принимать ключи `id` и `append` .
Если я изменяю значение `append` в `true` в моем `Childspec` :
Мой `terminate_timeout` будет добавлен в `terminate_timeout` из `DefaultChildspec` .
Мой `count` будет добавлен к `count DefaultChildspec` .
Мои `modules` будут добавлены в `modules DefaultChildspec` .
Мой `plan` будет добавлен в `plan DefaultChildspec` .
И если у меня есть ключ `start` со значением `{ModX, FuncX, ArgsX}` в `DefaultChildspec` и ключ `start` со значением `{ModY, FunY, ArgsY}` в `Childspec` , конечным значением будет `{ModY, FuncY, ArgsX ++ ArgsY}` .
И, наконец, если у меня есть ключ `start` со значением `{Mod, Func, Args}` в `DefaultChildspec` , ключ `start` в `Childspec` является необязательным для меня.
Вы можете вернуть свой собственный `DefaultChildspec` качестве третьего элемента кортежа в `init/1` .
Изменить `foo.erl` :

```
-module(foo).
-behaviour(director). %% Yes, this is a behaviour
-export([start_link/0
```

```
        ,init/1]).

start_link() ->
    director:start_link({local, foo_sup}, ?MODULE, null).

init(_InitArg) ->
    Childspec = #{id => bar_id
                  ,plan => [wait]
                  ,start => {bar,start_link}
                  ,count => 1
                  ,terminate_timeout => 2000},
    DefaultChildspec = #{start => {bar, start_link}
                         ,terminate_timeout => 1000
                         ,plan => [restart]
                         ,count => 5},
    {ok, [Childspec], DefaultChildspec}.
```

Перезапустите оболочку:

```
1> c(foo).
{ok,foo}

2> foo:start_link().
{ok,<0.111.0>}

3> director:get_pids(foo_sup).
[{bar_id,<0.112.0>}]

4> director:get_default_childspec(foo_sup).
{ok,#{count => 5,
      plan => [restart],
      start => {bar,start_link,[]},
      terminate_timeout => 1000}}

5> Childspec1 = #{id => 1, append => true},
%% Default 'plan' is [Fun], so 'plan' will be [restart] ++ [Fun] or [restart, Fun].
%% Default 'count' is 1, so 'count' will be 1 + 5 or 6.
%% Args in above Childspec is [], so Args will be [] ++ [] or [].
%% Default 'terminate_timeout' is 1000, so 'terminate_timeout' will be 1000 + 1000 or 2000.
%% Default 'modules' is [bar], so 'modules' will be [bar] ++ [] or [bar].
5> director:start_child(foo_sup, Childspec1).
{ok,<0.116.0>}

%% Test
6> director:get_childspec(foo_sup, 1).
{ok,#{append => true,
      count => 6,
      id => 1,
      modules => [bar],
      plan => [restart,#Fun<director.default_plan_element_fun.2>],
      start => {bar,start_link,[]},
      terminate_timeout => 2000,
      type => worker}}

7> director:get_pids(foo_sup).
[{bar_id,<0.112.0>},{1,<0.116.0>}]

%% I want to have 9 more children like that
8> [director:start_child(foo_sup
                         ,#{id => Count, append => true})
```

```
   || Count <- lists:seq(2, 10)].
[{ok,<0.126.0>},
 {ok,<0.127.0>},
 {ok,<0.128.0>},
 {ok,<0.129.0>},
 {ok,<0.130.0>},
 {ok,<0.131.0>},
 {ok,<0.132.0>},
 {ok,<0.133.0>},
 {ok,<0.134.0>}]

10> director:count_children(foo_sup).
[{specs,11},{active,11},{supervisors,0},{workers,11}]

11>
```

Динамически можно изменить `defaultChildspec` используя `change_default_childspec/2` !
И вы можете изменить `Childspec` детей динамически и установить их `append` в `true` !
Но, изменив их в разных частях кода, вы **создадите код спагетти**

## Могу ли я отлаживать директора?

Yesssss, **diorector** имеет собственную отладку и принимает стандартный `sys:dbg_opt/0` .
**директор** отправляет действительные журналы в `sasl` и `error_logger` в разных состояниях.

```
1> Name = {local, dname},
   Mod = foo,
   InitArg = undefined,
   DbgOpts = [trace],
   Opts = [{debug, DbgOpts}].
[{debug,[trace]}]

2> director:start_link(Name, Mod, InitArg, Opts).
{ok,<0.106.0>}
3>
3> director:count_children(dname).
*DBG* director "dname" got request "count_children" from "<0.102.0>"
*DBG* director "dname" sent "[{specs,1},
                              {active,1},
                              {supervisors,0},
                              {workers,1}]" to "<0.102.0>"
[{specs,1},{active,1},{supervisors,0},{workers,1}]

4> director:change_plan(dname, bar_id, [{restart, 5000}]).
*DBG* director "dname" got request "{change_plan,bar_id,[{restart,5000}]}" from "<0.102.0>"
*DBG* director "dname" sent "ok" to "<0.102.0>"
ok

5> {ok, Pid} = director:get_pid(dname, bar_id).
*DBG* director "dname" got request "{get_pid,bar_id}" from "<0.102.0>"
*DBG* director "dname" sent "{ok,<0.107.0>}" to "<0.102.0>"
{ok,<0.107.0>}

%% Start SASL
6> application:start(sasl).
ok
... %% Log about starting SASL
```

```
7> erlang:exit(Pid, kill).
*DBG* director "dname" got exit signal for pid "<0.107.0>" with reason "killed"
true

=SUPERVISOR REPORT==== 4-May-2017::12:37:41 ===
     Supervisor: dname
     Context:    child_terminated
     Reason:     killed
     Offender:   [{id,bar_id},
                  {pid,<0.107.0>},
                  {plan,[{restart,5000}]},
                  {count,1},
                  {count2,0},
                  {restart_count,0},
                  {mfargs,{bar,start_link,[]}},
                  {plan_element_index,1},
                  {plan_length,1},
                  {timer_reference,undefined},
                  {terminate_timeout,2000},
                  {extra,undefined},
                  {modules,[bar]},
                  {type,worker},
                  {append,false}]
8>

%% After 5000 mili-seconds
*DBG* director "dname" got timer event for child-id "bar_id" with timer reference
"#Ref<0.0.1.176>"

=PROGRESS REPORT==== 4-May-2017::12:37:46 ===
          supervisor: dname
             started: [{id,bar_id},
                       {pid,<0.122.0>},
                       {plan,[{restart,5000}]},
                       {count,1},
                       {count2,1},
                       {restart_count,1},
                       {mfargs,{bar,start_link,[]}},
                       {plan_element_index,1},
                       {plan_length,1},
                       {timer_reference,#Ref<0.0.1.176>},
                       {terminate_timeout,2000},
                       {extra,undefined},
                       {modules,[bar]},
                       {type,worker},
                       {append,false}]
8>
```

## Создать API-документацию

арматурных:

```
Pouriya@Jahanbakhsh ~/director $ rebar doc
```

rebar3:

```
Pouriya@Jahanbakhsh ~/director $ rebar3 edoc
```

Эрл

```
Pouriya@Jahanbakhsh ~/director $ mkdir -p doc &&
                                 erl -noshell\
                                     -eval "edoc:file(\"./src/director.erl\", [{dir,
\"./doc\"}]),init:stop()."
```

После выполнения одной из приведенных выше команд документация HTML должна быть в каталоге `doc` .

Прочитайте директор онлайн: https://riptutorial.com/ru/erlang/topic/9878/директор

# глава 6: Контролеры

## Examples

### Основной руководитель с одним рабочим процессом

В этом примере используется формат карты, представленный в Erlang / OTP 18.0.

```
%% A module implementing a supervisor usually has a name ending with `_sup`.
-module(my_sup).

-behaviour(supervisor).

%% API exports
-export([start_link/0]).

%% Behaviour exports
-export([init/1]).

start_link() ->
    %% If needed, we can pass an argument to the init callback function.
    Args = [],
    supervisor:start_link({local, ?MODULE}, ?MODULE, Args).

%% The init callback function is called by the 'supervisor' module.
init(_Args) ->
    %% Configuration options common to all children.
    %% If a child process crashes, restart only that one (one_for_one).
    %% If there is more than 1 crash ('intensity') in
    %% 5 seconds ('period'), the entire supervisor crashes
    %% with all its children.
    SupFlags = #{strategy => one_for_one,
                 intensity => 1,
                 period => 5},

    %% Specify a child process, including a start function.
    %% Normally the module my_worker would be a gen_server
    %% or a gen_fsm.
    Child = #{id => my_worker,
              start => {my_worker, start_link, []}},

    %% In this case, there is only one child.
    Children = [Child],

    %% Return the supervisor flags and the child specifications
    %% to the 'supervisor' module.
    {ok, {SupFlags, Children}}.
```

Прочитайте Контролеры онлайн: https://riptutorial.com/ru/erlang/topic/6996/контролеры

# глава 7: Монтаж

## Examples

### Создайте и установите Erlang / OTP на Ubuntu

В следующих примерах показаны два основных метода установки Erlang / OTP на Ubuntu.

# Метод 1 - Готовый двоичный пакет

Просто запустите эту команду, и она загрузит и установит последний стабильный выпуск Erlang от Erlang Solutions .

```
$ sudo apt-get install erlang
```

# Способ 2 - сборка и установка из источника

Загрузите файл tar:

```
$ wget http://erlang.org/download/otp_src_19.0.tar.gz
```

Извлеките tar-файл:

```
$ tar -zxf otp_src_19.0.tar.gz
```

Введите извлеченный каталог и установите `ERL_TOP` как текущий путь:

```
$ cd otp_src_19.0
$ export ERL_TOP=`pwd`
```

Теперь перед настройкой сборки вы хотите убедиться, что у вас есть все зависимости, необходимые для установки Erlang:

**Основные зависимости:**

```
$ sudo apt-get install autoconf libncurses-dev build-essential
```

**Другие зависимости приложений**

| заявка | Установка зависимостей |
| --- | --- |
| ВДФЭ | `$ sudo apt-get install m4` |
| ODBC | `$ sudo apt-get install unixodbc-dev` |
| OpenSSL | `$ sudo apt-get install libssl-dev` |
| WxWidgets | `$ sudo apt-get install libwxgtk3.0-dev libglu-dev` |
| Документация | `$ sudo apt-get install fop xsltproc` |
| Orber и другие проекты на C ++ | `$ sudo apt-get install g++` |
| jinterface | `$ sudo apt-get install default-jdk` |

Настроить и построить:
Вы можете установить свои собственные параметры или оставить пустым для запуска конфигурации по умолчанию. Расширенная настройка и сборка для Erlang / OTP .

```
$ ./configure [ options ]
$ make
```

Тестирование сборки:

```
$ make release_tests
$ cd release/tests/test_server
$ $ERL_TOP/bin/erl -s ts install -s ts smoke_test batch -s init stop
```

После запуска этих команд откройте `$ERL_TOP/release/tests/test_server/index.html` с вашим веб-браузером и убедитесь, что у вас нет `$ERL_TOP/release/tests/test_server/index.html` . Если все тесты пройдены, мы сможем продолжить установку.

Установка:

```
$ make install
```

### Создайте и установите Erlang / OTP на FreeBSD

Следующие примеры показывают 3 основных метода установки Erlang / OTP на FreeBSD.

# Метод 1 - Готовый двоичный пакет

Используйте pkg для установки готового двоичного пакета:

```
$ pkg install erlang
```

Проверьте свою новую установку:

```
$ erl
Erlang/OTP 18 [erts-7.3.1] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.3.1  (abort with ^G)
```

# Способ 2 - сборка и установка с использованием коллекции портов (рекомендуется)

Создайте и установите порт как обычно:

```
$ make -C /usr/ports/lang/erlang install clean
```

Проверьте свою новую установку:

```
$ erl
Erlang/OTP 18 [erts-7.3.1] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.3.1  (abort with ^G)
```

Это приведет к извлечению tarball с официального сайта, при необходимости примените несколько исправлений, создайте выпуск и установите его. Очевидно, это займет некоторое время.

# Способ 3 - Создайте и установите из релиза tarball

Примечание: создание релиза выполняется вручную, но использование двух вышеуказанных методов должно быть предпочтительным, поскольку коллекция портов включает патчи, которые делают релиз более дружественным к FreeBSD.

Загрузите файл выпуска:

```
$ fetch 'http://erlang.org/download/otp_src_18.3.tar.gz'
```

Убедитесь, что его сумма MD5 верна:

```
$ fetch 'http://erlang.org/download/MD5'
MD5                                           100% of   24 kB  266 kBps 00m00s

$ grep otp_src_18.3.tar.gz MD5
MD5(otp_src_18.3.tar.gz)= 7e4ff32f97c36fb3dab736f8d481830b

$ md5 otp_src_18.3.tar.gz
MD5 (otp_src_18.3.tar.gz) = 7e4ff32f97c36fb3dab736f8d481830b
```

Извлеките архив:

```
$ tar xzf otp_src_18.3.tar.gz
```

Настройка:

```
$ ./configure --disable-hipe
```

Если вы хотите создать Erlang с помощью HiPe, вам нужно будет применить патчи из коллекции портов.

Телосложение:

```
$ gmake
```

Установка:

```
$ gmake install
```

Проверьте свою новую установку:

```
$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:2:2] [async-threads:10] [kernel-poll:false]

Eshell V7.3  (abort with ^G)
```

## Создайте и установите с помощью kerl

kerl - это инструмент, который помогает вам создавать и устанавливать релизы Erlang / OTP.

Установить завиток:

```
$ make -C /usr/ports/ftp/curl install clean
```

Загрузить kerl:

```
$ fetch 'https://raw.githubusercontent.com/kerl/kerl/master/kerl'
$ chmod +x kerl
```

Обновите список доступных выпусков:

```
$ ./kerl update releases
The available releases are:
R10B-0 R10B-10 R10B-1a R10B-2 R10B-3 R10B-4 R10B-5 R10B-6 R10B-7 R10B-8 R10B-9 R11B-0 R11B-1
R11B-2 R11B-3 R11B-4 R11B-5 R12B-0 R12B-1 R12B-2 R12B-3 R12B-4 R12B-5 R13A R13B01 R13B02-1
R13B02 R13B03 R13B04 R13B R14A R14B01 R14B02 R14B03 R14B04 R14B R14B_erts-5.8.1.1 R15B01
R15B02 R15B02_with_MSVCR100_installer_fix R15B03-1 R15B03 R15B R16A_RELEASE_CANDIDATE R16B01
R16B02 R16B03-1 R16B03 R16B 17.0-rc1 17.0-rc2 17.0 17.1 17.3 17.4 17.5 18.0 18.1 18.2 18.2.1
18.3 19.0
```

Создайте требуемый выпуск:

```
$ ./kerl build 18.3 erlang-18.3
```

Убедитесь, что сборка присутствует в списке сборки:

```
$ ./kerl list builds
18.3,erlang-18.3
```

Установите конструкцию где-нибудь:

```
$ ./kerl install erlang-18.3 ./erlang-18.3
```

Источник `activate` файл, если вы используете bash или shell для рыбы. Если вы используете cshell, добавьте каталог bin для сборки в PATH:

```
$ setenv PATH "/some/where/erlang-18.3/bin/:$PATH"
```

Проверьте свою новую установку:

```
$ which erl
/some/where/erlang-18.3/bin//erl

$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.3  (abort with ^G)
```

# Другие релизы

Если вы хотите создать еще одну версию Erlang / OTP, найдите другие порты в коллекции:

- языки / эрланг-runtime15
- языки / эрланг-runtime16
- языки / эрланг-runtime17
- языки / эрланг-runtime18

## Ссылка

- Руководство FreeBSD -> Глава 4. Установка приложений: пакеты и порты
- Эрланг на FreshPorts
- Документация Kerl по GitHub

### Создание и установка Erlang / OTP на OpenBSD

Erlang на OpenBSD в настоящее время разбит на архитектуру `alpha` , `sparc` и `hppa` .

# Метод 1 - Готовый двоичный пакет

OpenBSD позволяет выбрать желаемую версию, которую вы хотите установить в своей системе:

```
######################################################################
# free-choice:
######################################################################
$ pkg_add erlang
# a       0: <None>
#   1: erlang-16b.03p10v0
#   2: erlang-17.5p6v0
#   3: erlang-18.1p1v0
#   4: erlang-19.0v0

######################################################################
# manual-choice:
######################################################################
pkg_add erlang%${version}
# example:
pkg_add erlang%19
```

OpenBSD может поддерживать несколько версий Erlang. Чтобы упростить использование мышью, в каждом бинарнике установлена его версия Erlang. Итак, если вы установили `erlang-19.0v0` , ваш `erl` файл будет `erl19` .

Если вы хотите использовать `erl` , вы можете создать символическую ссылку:

```
ln -s /usr/local/bin/erl19 /usr/local/bin/erl
```

или создать псевдоним в файле конфигурации оболочки или в файле `.profile` :

```
echo 'alias erl="erl19"' >> ~/.profile
# or
echo 'alias erl="erl19"' >> ~/.shrc
```

Теперь вы можете запустить `erl` :

```
erl19
# or if you have an alias or symlink
erl
# Erlang/OTP 19 [erts-8.0] [source] [async-threads:10] [kernel-poll:false]
#
# Eshell V8.0  (abort with ^G)
```

# Способ 2. Создание и установка с использованием портов

```
RELEASE=OPENBSD_$(uname -r  | sed 's/\./_/g')
cd /usr
cvs -qz3 -danoncvs@anoncvs.openbsd.org:/cvs co -r${RELEASE}
cd /usr/ports/lang/erlang
ls -p
# 16/ 17/ 18/ 19/  CVS/  Makefile  Makefile.inc  erlang.port.mk
cd 19
make && make install
```

# Метод 3 - Построить из источника

Для сборки из источника требуются дополнительные пакеты:

- `git`
- `gmake`
- `autoconf-2.59`

```
pkg_add git gmake autoconf%2.59
git clone https://github.com/erlang/otp.git
cd otp
AUTOCONF_VERSION="2.59" ./build_build all
```

# Рекомендации

- http://openports.se/lang/erlang
- http://cvsweb.openbsd.org/cgi-bin/cvsweb/ports/lang/erlang/
- https://www.openbsd.org/faq/faq15.html
- http://man.openbsd.org/OpenBSD-current/man1/pkg_add.1

Прочитайте Монтаж онлайн: https://riptutorial.com/ru/erlang/topic/4483/монтаж

# глава 8: НИФ

## Examples

### Определение

Официальная документация: http://erlang.org/doc/tutorial/nif.html

NIF были введены в Erlang / OTP R13B03 в качестве экспериментальной функции. Цель состоит в том, чтобы разрешить вызов C-кода из кода Erlang.

NIF реализованы в C вместо Erlang, но они отображаются как любые другие функции в области кода Erlang, поскольку они принадлежат модулю, в котором произошло включение. Библиотеки NIF связаны с компиляцией и загружаются во время выполнения.

Поскольку библиотеки NIF динамически связаны с процессом эмулятора, они бывают быстрыми, но также и опасными, потому что сбой в NIF также приводит к тому, что эмулятор тоже работает.

### Пример: текущее время UNIX

Вот очень простой пример, иллюстрирующий, как писать NIF.

Структура каталога:

```
nif_test
├── c_src
│   ├── Makefile
│   └── nif_test.c
├── rebar.config
└── src
    ├── nif_test.app.src
    └── nif_test.erl
```

Эту структуру можно легко инициализировать с помощью Rebar3:

```
$ rebar3 new lib nif_test && cd nif_test && rebar3 new cmake
```

Содержание `nif_test.c` :

```
#include "erl_nif.h"
#include "time.h"

static ERL_NIF_TERM now(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
        return enif_make_int(env, time(0));
}
```

```
static ErlNifFunc nif_funcs[] = {
        {"now", 0, now}
};

ERL_NIF_INIT(nif_test,nif_funcs,NULL,NULL,NULL,NULL);
```

Содержание `nif_test.erl` :

```
-module(nif_test).
-on_load(init/0).
-export([now/0]).

-define(APPNAME, nif_test).
-define(LIBNAME, nif_test).

%%======================================================================
%% API functions
%%======================================================================

now() -> nif_not_loaded.

%%======================================================================
%% Internal functions
%%======================================================================

init() ->
    SoName = case code:priv_dir(?APPNAME) of
        {error, bad_name} ->
            case filelib:is_dir(filename:join(["..", priv])) of
                true -> filename:join(["..", priv, ?LIBNAME]);
                _ -> filename:join([priv, ?LIBNAME])
            end;
        Dir -> filename:join(Dir, ?LIBNAME)
    end,
    erlang:load_nif(SoName, 0).
```

Содержимое `rebar.config` :

```
{erl_opts, [debug_info]}.
{deps, []}.

{pre_hooks, [
  {"(linux|darwin|solaris)", compile, "make -C c_src"},
  {"(freebsd)", compile, "gmake -C c_src"}
]}.
{post_hooks, [
  {"(linux|darwin|solaris)", clean, "make -C c_src clean"},
  {"(freebsd)", clean, "gmake -C c_src clean"}
]}.
```

Теперь вы можете запустить пример:

```
$ rebar3 shell

===> Verifying dependencies...
===> Compiling nif_test
make: Entering directory '/home/vschroeder/Projects/nif_test/c_src'
```

```
cc -O3 -std=c99 -finline-functions -Wall -Wmissing-prototypes -fPIC -I
/usr/local/lib/erlang/erts-7.3.1/include/ -I /usr/local/lib/erlang/lib/erl_interface-
3.8.2/include  -c -o /home/vschroeder/Projects/nif_test/c_src/nif_test.o
/home/vschroeder/Projects/nif_test/c_src/nif_test.c
cc /home/vschroeder/Projects/nif_test/c_src/nif_test.o -shared -L
/usr/local/lib/erlang/lib/erl_interface-3.8.2/lib -lerl_interface -lei -o
/home/vschroeder/Projects/nif_test/c_src/../priv/nif_test.so
make: Leaving directory '/home/vschroeder/Projects/nif_test/c_src'
Erlang/OTP 18 [erts-7.3.1] [source] [64-bit] [smp:4:4] [async-threads:0] [hipe] [kernel-
poll:false]

Eshell V7.3.1  (abort with ^G)
1> nif_test:now().
1469732239
2> nif_test:now().
1469732264
3>
```

## API Erlang C (от C до Erlang)

**Официальная документация** : http://erlang.org/doc/man/erl_nif.html

Наиболее важными структурами, типами и макросами API Erlang C являются:

- `ERL_NIF_TERM` : тип для терминов Erlang. Это тип возвращаемого значения, которому должны следовать функции NIF.
- `ERL_NIF_INIT(MODULE, ErlNifFunc funcs[], load, reload, upgrade, unload)` : Это макрос, который фактически создает NIF, определенные в определенном файле C. Он должен оцениваться в глобальном масштабе. Обычно это будет последняя строка в файле C.
- `ErlNifFunc` : тип, с которым каждый NIF передается в `ERL_NIF_INIT` для экспорта. Эта структура состоит из имени, arity, poiter для функции C и флагов. Массив этого типа со всеми определениями NIF должен быть создан для передачи в `ERL_NIF_INIT` .
- `ErlNifEnv` : среда Erlang, в которой выполняется NIF. Обязательно передать среду в качестве первого аргумента для каждого NIF. Этот тип непрозрачен и может управляться только с помощью функций, которые предлагает API Erlang C.

Прочитайте НИФ онлайн: https://riptutorial.com/ru/erlang/topic/5274/ниф

# глава 9: Петля и рекурсия

## Синтаксис

- function (list | iolist | tuple) -> function (tail).

## замечания

# Почему рекурсивные функции?

Erlang - это функциональный язык программирования и не имеет никакой структуры циклов. Все функциональное программирование основано на данных, типе и функциях. Если вам нужен цикл, вам нужно создать функцию, которая сама вызывает вызов.

Традиционный `while` или `for` loop в императивном и объектно-ориентированном языке может быть представлен таким же образом в Erlang:

```
loop() ->
  % do something here
  loop().
```

Хорошим методом понимания этой концепции является расширение всех вызовов функций. Мы увидим это на других примерах.

## Examples

### Список

Здесь простейшая рекурсивная функция над типом списка . Эта функция только переходит в список от начала до конца и больше ничего не делает.

```
-spec loop(list()) -> ok.
loop([]) ->
  ok;
loop([H|T]) ->
  loop(T).
```

Вы можете назвать это следующим образом:

```
loop([1,2,3]). % will return ok.
```

Здесь разложение рекурсивной функции:

```
loop([1|2,3]) ->
  loop([2|3]) ->
    loop([3|]) ->
      loop([]) ->
        ok.
```

# Рекурсивный цикл с действиями IO

Предыдущий код ничего не делает и довольно бесполезен. Итак, мы создадим теперь рекурсивную функцию, выполняющую некоторые действия. Этот код похож на `lists:foreach/2` .

```
-spec loop(list(), fun()) -> ok.
loop([], _) ->
  ok;
loop([H|T], Fun)
  when is_function(Fun) ->
    Fun(H),
    loop(T, Fun).
```

Вы можете вызвать его так:

```
Fun = fun(X) -> io:format("~p", [X]) end.
loop([1,2,3]).
```

Здесь разложение рекурсивной функции:

```
loop([1|2,3], Fun(1)) ->
  loop([2|3], Fun(2)) ->
    loop([3|], Fun(3)) ->
      loop([], _) ->
        ok.
```

Вы можете сравнить со `lists:foreach/2` output:

```
lists:foreach(Fun, [1,2,3]).
```

# Рекурсивный цикл над списком, возвращающим измененный список

Другой полезный пример, аналогичный `lists:map/2` . Эта функция будет принимать один список и одну анонимную функцию. Каждый раз, когда значение в списке сопоставляется, мы применяем к нему функцию.

```
-spec loop(A :: list(), fun()) -> list().
```

```
loop(List, Fun)
  when is_list(List), is_function(Fun) ->
    loop(List, Fun, []).

-spec loop(list(), fun(), list()) -> list() | {error, list()}.
loop([], _, Buffer)
  when is_list(Buffer) ->
    lists:reverse(Buffer);
loop([H|T], Fun, Buffer)
  when is_function(Fun), is_list(Buffer) ->
    BufferReturn = [Fun(H)] ++ Buffer,
    loop(T, Fun, BufferReturn).
```

Вы можете назвать это следующим образом:

```
Fun(X) -> X+1 end.
loop([1,2,3], Fun).
```

Здесь разложение рекурсивной функции:

```
loop([1|2,3], Fun(1), [2]) ->
  loop([2|3], Fun(2), [3,2]) ->
    loop([3|], Fun(3), [4,3,2]) ->
      loop([], _, [4,3,2]) ->
        list:reverse([4,3,2]) ->
          [2,3,4].
```

Эта функция также называется «хвостовой рекурсивной функцией», потому что мы используем переменную, подобную аккумулятору, для передачи измененных данных по нескольким контекстам выполнения.

## Iolist и битстрим

Как и список, простейшая функция над iolist и bitstring :

```
-spec loop(iolist()) -> ok | {ok, iolist} .
loop(<<>>) ->
  ok;
loop(<<Head, Tail/bitstring>>) ->
  loop(Tail);
loop(<<Rest/bitstring>>) ->
  {ok, Rest}
```

Вы можете назвать это следующим образом:

```
loop(<<"abc">>).
```

Здесь разложение рекурсивной функции:

```
loop(<<"a"/bitstring, "bc"/bitstring>>) ->
  loop(<<"b"/bitstring, "c"/bitstring>>) ->
    loop(<<"c"/bitstring>>) ->
```

```
      loop(<<>>) ->
        ok.
```

# Рекурсивная функция по переменному двоичному размеру

Этот код принимает bitstring и динамически определяет двоичный размер. Поэтому, если, если мы установим размер `4` , каждые `4` бита, данные будут сопоставлены. Этот цикл не делает ничего интересного, его просто наш столп.

```
loop(Bitstring, Size)
  when is_bitstring(Bitstring), is_integer(Size) ->
    case Bitstring of
      <<>> ->
        ok;
      <<Head:Size/bitstring,Tail/bitstring>> ->
        loop(Tail, Size);
      <<Rest/bitstring>> ->
        {ok, Rest}
    end.
```

Вы можете назвать это следующим образом:

```
loop(<<"abc">>, 4).
```

Здесь разложение рекурсивной функции:

```
loop(<<6:4/bitstring, 22, 38, 3:4>>, 4) ->
  loop(<<1:4/bitstring, "bc">>, 4) ->
    loop(<<6:4/bitstring, 38,3:4>>, 4) ->
      loop(<<2:4/bitstring, "c">>, 4) ->
        loop(<<6:4/bitstring, 3:4>>, 4) ->
          loop(<<3:4/bitstring>>, 4) ->
            loop(<<>>, 4) ->
              ok.
```

Наша битстрима разделена на 7 шаблонов. Зачем? Поскольку по умолчанию Erlang использует двоичный размер `8` бит, если мы разделим его на две части, у нас есть `4` бита. Наша строка `8*3=24` бит. `24/4=6` рисунков. Последний шаблон - `<<>>` . Функция `loop/2` называется 7 раз.

# рекурсивная функция по переменному двоичному размеру с действиями

Теперь мы можем сделать более интересную вещь. Эта функция принимает еще один

аргумент - анонимную функцию. Каждый раз, когда мы сопоставляем шаблон, он будет передан ему.

```
-spec loop(iolist(), integer(), function()) -> ok.
loop(Bitstring, Size, Fun) ->
  when is_bitstring(Bitstring), is_integer(Size), is_function(Fun) ->
        case Bitstring of
      <<>> ->
        ok;
      <<Head:Size/bitstring,Tail/bitstring>> ->
        Fun(Head),
        loop(Tail, Size, Fun);
      <<Rest/bitstring>> ->
        Fun(Rest),
        {ok, Rest}
    end.
```

Вы можете назвать это следующим образом:

```
Fun = fun(X) -> io:format("~p~n", [X]) end.
loop(<<"abc">>, 4, Fun).
```

Здесь разложение рекурсивной функции:

```
loop(<<6:4/bitstring, 22, 38, 3:4>>, 4, Fun(<<6:4>>) ->
  loop(<<1:4/bitstring, "bc">>, 4, Fun(<<1:4>>)) ->
    loop(<<6:4/bitstring, 38,3:4>>, 4, Fun(<<6:4>>)) ->
      loop(<<2:4/bitstring, "c">>, 4, Fun(<<2:4>>)) ->
        loop(<<6:4/bitstring, 3:4>>, 4, Fun(<<6:4>>) ->
          loop(<<3:4/bitstring>>, 4, Fun(<<3:4>>) ->
            loop(<<>>, 4) ->
              ok.
```

# Рекурсивная функция над битовой строкой, возвращающая измененную bitstring

Это похоже на `lists:map/2` но для bitstring и iolist.

```
% public function (interface).
-spec loop(iolist(), fun()) -> iolist() | {iolist(), iolist()}.
loop(Bitstring, Fun) ->
  loop(Bitstring, 8, Fun).

% public function (interface).
-spec loop(iolist(), integer(), fun()) -> iolist() | {iolist(), iolist()}.
loop(Bitstring, Size, Fun) ->
  loop(Bitstring, Size, Fun, <<>>)

% private function.
```

```
-spec loop(iolist(), integer(), fun(), iolist()) -> iolist() | {iolist(), iolist()}.
loop(<<>>, _, _, Buffer) ->
  Buffer;
loop(Bitstring, Size, Fun, Buffer) ->
  when is_bitstring(Bitstring), is_integer(Size), is_function(Fun) ->
    case Bitstring of
      <<>> ->
        Buffer;
      <<Head:Size/bitstring,Tail/bitstring>> ->
        Data = Fun(Head),
        BufferReturn = <<Buffer/bitstring, Data/bitstring>>,
        loop(Tail, Size, Fun, BufferReturn);
      <<Rest/bitstring>> ->
        {Buffer, Rest}
    end.
```

Этот код кажется более сложным. Добавлены две функции: `loop/2` и `loop/3` . Эти две функции - простой интерфейс для `loop/4` .

Вы можете выполнить его следующим образом:

```
Fun = fun(<<X>>) -> << (X+1) >> end.
loop(<<"abc">>, Fun).
% will return <<"bcd">>

Fun = fun(<<X:4>>) -> << (X+1) >> end.
loop(<<"abc">>, 4, Fun).
% will return <<7,2,7,3,7,4>>

loop(<<"abc">>, 4, Fun, <<>>).
% will return <<7,2,7,3,7,4>>
```

## карта

Карта в Erlang эквивалентна хэшам в Perl или словарям в Python, его хранилище ключей / значений. Чтобы перечислить каждое сохраненное значение, вы можете перечислить каждый ключ и вернуть пару ключ / значение. Этот первый цикл дает вам представление:

```
loop(Map) when is_map(Map) ->
  Keys = maps:keys(Map),
  loop(Map, Keys).

loop(_ , []) ->
  ok;
loop(Map, [Head|Tail]) ->
  Value = maps:get(Head, Map),
  io:format("~p: ~p~n", [Head, Value]),
  loop(Map, Tail).
```

Вы можете выполнить его так:

```
Map = #{1 => "one", 2 => "two", 3 => "three"}.
loop(Map).
% will return:
```

```
% 1: "one"
% 2: "two"
% 3: "three"
```

### Управляющее государство

Рекурсивная функция использует свои состояния в цикле. Когда вы создаете новый процесс, этот процесс будет просто циклом с некоторым определенным состоянием.

### Анонимная функция

Вот 2 примера рекурсивных анонимных функций, основанных на предыдущем примере. Во-первых, простой бесконечный цикл:

```
InfiniteLoop = fun
  R() ->
    R() end.
```

Во-вторых, анонимная функция делает цикл над списком:

```
LoopOverList = fun
  R([]) -> ok;
  R([H|T]) ->
    R(T) end.
```

Эти две функции можно переписать следующим образом:

```
InfiniteLoop = fun loop/0.
```

В этом случае `loop/0` является ссылкой на `loop/0` из замечаний. Во-вторых, с немного более сложными:

```
LoopOverLlist = fun loop/2.
```

Здесь `loop/2` является ссылкой на `loop/2` из примера списка. Эти две записи являются синтаксическим сахаром.

Прочитайте Петля и рекурсия онлайн: https://riptutorial.com/ru/erlang/topic/10720/петля-и-рекурсия

# глава 10: поведение gen_server

## замечания

`gen_server` является важной особенностью Erlang и требует некоторого предпосылки для понимания каждого аспекта этой функции:

- Петля, рекурсия и состояние
- Нерестовые процессы
- Передача сообщений
- Принципы ОТП

Хороший способ узнать больше об этой функции в Erlang - это прямое чтение исходного кода из официального репозитория github . Поведение `gen_server` внедряет множество полезной информации и интересной структуры в ее ядре.

`gen_server` определен в `gen_server.erl` и соответствующая документации может быть найдена в документации `stdlib` Erlang . `gen_server` - это функция OTP, и дополнительная информация также может быть найдена в OTP Design Principles и User's Guide .

Фрэнк Хеберт дает вам еще одно хорошее введение в `gen_server` из его бесплатной онлайн-книги Learn You Some Erlang для отличного блага!

Официальная документация для обратного вызова `gen_server` :

- `code_change/3`
- `handle_call/3`
- `handle_cast/2`
- `handle_info/2`
- `init/1`
- `terminate/2`

## Examples

### Служба Greeter

Ниже приведен пример службы, которая приветствует людей по данному имени и отслеживает, сколько пользователей оно встретило. См. Использование ниже.

```
%% greeter.erl
%% Greets people and counts number of times it did so.
-module(greeter).
-behaviour(gen_server).
%% Export API Functions
-export([start_link/0, greet/1, get_count/0]).
%% Required gen server callbacks
```

```
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, code_change/3]).

-record(state, {count::integer()}).

%% Public API
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, {}, []).

greet(Name) ->
    gen_server:cast(?MODULE, {greet, Name}).

get_count() ->
    gen_server:call(?MODULE, {get_count}).

%% Private
init({}) ->
    {ok, #state{count=0}}.

handle_cast({greet, Name}, #state{count = Count} = State) ->
    io:format("Greetings ~s!~n", [Name]),
    {noreply, State#state{count = Count + 1}};

handle_cast(Msg, State) ->
    error_logger:warning_msg("Bad message: ~p~n", [Msg]),
    {noreply, State}.

handle_call({get_count}, _From, State) ->
    {reply, {ok, State#state.count}, State};

handle_call(Request, _From, State) ->
    error_logger:warning_msg("Bad message: ~p~n", [Request]),
    {reply, {error, unknown_call}, State}.

%% Other gen_server callbacks
handle_info(Info, State) ->
    error_logger:warning_msg("Bad message: ~p~n", [Info]),
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Вот пример использования этой службы в оболочке erlang:

```
1> c(greeter).
{ok,greeter}
2> greeter:start_link().
{ok,<0.62.0>}
3> greeter:greet("Andriy").
Greetings Andriy!
ok
4> greeter:greet("Mike").
Greetings Mike!
ok
5> greeter:get_count().
{ok,2}
```

## Использование поведения gen_server

`gen_server` - это конечная конечная машина, работающая как сервер. `gen_server` может обрабатывать различные типы событий:

- синхронный запрос с `handle_call`
- асинхронный запрос с `handle_cast`
- другое сообщение (не определено в спецификации OTP) с помощью `handle_info`

Синхронное и асинхронное сообщение указываются в OTP и являются простыми помеченными кортежами с любыми данными на нем.

Простой `gen_server` определяется следующим образом:

```
-module(simple_gen_server).
-behaviour(gen_server).
-export([start_link/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

start_link() ->
    Return = gen_server:start_link({local, ?MODULE}, ?MODULE, [], []),
    io:format("start_link: ~p~n", [Return]),
    Return.

init([]) ->
    State = [],
    Return = {ok, State},
    io:format("init: ~p~n", [State]),
    Return.

handle_call(_Request, _From, State) ->
    Reply = ok,
    Return = {reply, Reply, State},
    io:format("handle_call: ~p~n", [Return]),
    Return.

handle_cast(_Msg, State) ->
    Return = {noreply, State},
    io:format("handle_cast: ~p~n", [Return]),
    Return.

handle_info(_Info, State) ->
    Return = {noreply, State},
    io:format("handle_info: ~p~n", [Return]),
    Return.

terminate(_Reason, _State) ->
    Return = ok,
    io:format("terminate: ~p~n", [Return]),
    ok.

code_change(_OldVsn, State, _Extra) ->
    Return = {ok, State},
    io:format("code_change: ~p~n", [Return]),
    Return.
```

Этот код прост: каждое полученное сообщение печатается на стандартный вывод.

## поведение gen_server

Чтобы определить `gen_server` , вам нужно явно объявить его в исходном коде с помощью `-behaviour(gen_server)` . Обратите внимание, что `behaviour` может быть написано в США (поведение) или Великобритании (поведение).

## start_link / 0

Эта функция представляет собой простой ярлык для вызова другой функции `gen_server:start_link/3,4` .

## start_link / 3,4

```
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
```

Эта функция вызывается, когда вы хотите запустить свой сервер, связанный с `supervisor` или другим процессом. `start_link/3,4` может автоматически регистрировать ваш процесс (если вы считаете, что ваш процесс должен быть уникальным) или может просто порождать его, как простой процесс. При вызове эта функция выполняет `init/1` .

Эта функция может вернуть эти значения:

- `{ok,Pid}`
- `ignore`
- `{error,Error}`

## INIT / 1

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

`init/1` - первая выполняемая функция, когда ваш сервер будет запущен. Это инициализирует все предпосылки вашего приложения и возвращает состояние для вновь созданного процесса.

Эта функция может возвращать только эти определенные значения:

- `{ok,State}`
- `{ok,State,Timeout}`
- `{ok,State,hibernate}`
- `{stop,Reason}`
- `ignore`

`State` переменная может быть все, (например , список, кортеж, proplists, карта, запись) и остаются доступными для всех функций внутри порожденного процесса.

## handle_call / 3

```
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.
```

`gen_server:call/2` выполнить этот обратный вызов. Первым аргументом является ваше сообщение ( `_Request` ), второе - начало запроса ( `_From` ), а последнее - текущее состояние ( `State` ) вашего поведения gen_server.

Если вы хотите получить ответ от вызывающего, `handle_call/3` необходимо вернуть одну из следующих структур данных:

- `{reply,Reply,NewState}`
- `{reply,Reply,NewState,Timeout}`
- `{reply,Reply,NewState,hibernate}`

Если вы не хотите отвечать на вызов, `handle_call/3` необходимо вернуть одну из этих структур данных:

- `{noreply,NewState}`
- `{noreply,NewState,Timeout}`
- `{noreply,NewState,hibernate}`

Если вы хотите остановить текущее выполнение текущего gen_server, `handle_call/3` необходимо вернуть одну из этих структур данных:

- `{stop,Reason,Reply,NewState}`
- `{stop,Reason,NewState}`

## handle_cast / 2

```
handle_cast(_Msg, State) ->
    {noreply, State}.
```

`gen_server:cast/2` выполнить этот обратный вызов. Первым аргументом является ваше сообщение ( `_Msg` ), а второе - текущее состояние вашего поведения gen_server.

По умолчанию эта функция не может передавать данные вызывающему абоненту, поэтому у вас есть только два варианта, продолжить текущее выполнение:

- `{noreply,NewState}`
- `{noreply,NewState,Timeout}`
- `{noreply,NewState,hibernate}`

Или остановите свой текущий процесс `gen_server` :

- `{stop,Reason,NewState}`

## handle_info / 2

```
handle_info(_Info, State) ->
    {noreply, State}.
```

`handle_info/2` выполняется, когда нестандартное сообщение OTP поступает из внешнего мира. Это не может ответить и, подобно `handle_cast/2` может выполнять только 2 действия, продолжая текущее выполнение:

- `{noreply,NewState}`
- `{noreply,NewState,Timeout}`
- `{noreply,NewState,hibernate}`

Или остановите текущий текущий процесс `gen_server` :

- `{stop,Reason,NewState}`

## прекратить / 2

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

`terminate/2` вызывается при возникновении ошибки или когда вы хотите `gen_server` процесс `gen_server` .

## code_change / 3

```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

`code_change/3` вызывается, когда вы хотите обновить `code_change/3` код.

Эта функция может возвращать только эти определенные значения:

- `{ok, NewState}`
- `{error, Reason}`

# Запуск этого процесса

Вы можете скомпилировать свой код и запустить `simple_gen_server` :

```
simple_gen_server:start_link().
```

Если вы хотите отправить сообщение на свой сервер, вы можете использовать следующие функции:

```
% will use handle_call as callback and print:
%   handle_call: mymessage
gen_server:call(simple_gen_server, mymessage).

% will use handle_cast as callback and print:
%   handle_cast: mymessage
gen_server:cast(simple_gen_server, mymessage).

% will use handle_info as callback and print:
%   handle_info: mymessage
erlang:send(whereis(simple_gen_server), mymessage).
```

## Простая база данных ключей / значений

Этот исходный код создает простой ключ / значение магазина обслуживание , основанное на `map` Erlang структура данных. Во-первых, нам нужно определить всю информацию, касающуюся нашего `gen_server` :

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

% our API
-export([start_link/0]).
-export([get/1, put/2, state/0, delete/1, stop/0]).

% our handlers
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

% Defining our function to start `cache` process:

start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

%%%%%%%%%%%%%%
% API

% Key/Value database is a simple store, value indexed by an unique key.
% This implementation is based on map, this datastructure is like hash
# in Perl or dictionaries in Python.

% put/2
% put a value indexed by a key. We assume the link is stable
% and the data will be written, so, we use an asynchronous call with
% gen_server:cast/2.

put(Key, Value) ->
    gen_server:cast(?MODULE, {put, {Key, Value}}).

% get/1
% take one argument, a key and will a return the value indexed
% by this same key. We use a synchronous call with gen_server:call/2.

get(Key) ->
```

```
    gen_server:call(?MODULE, {get, Key}).

% delete/1
% like `put/1`, we assume the data will be removed. So, we use an
% asynchronous call with gen_server:cast/2.

delete(Key) ->
    gen_server:cast(?MODULE, {delete, Key}).

% state/0
% This function will return the current state (here the map who contain all
% indexed values), we need a synchronous call.

state() ->
    gen_server:call(?MODULE, {get_state}).

% stop/0
% This function stop cache server process.

stop() ->
    gen_server:stop(?MODULE).

%%%%%%%%%%%%%%%
% Handlers

% init/1
% Here init/1 will initialize state with simple empty map datastructure.

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

% handle_call/3
% Now, we need to define our handle. In a cache server we need to get our
% value from a key, this feature need to be synchronous, so, using
% handle_call seems a good choice:

handle_call({get, Key}, From, State) ->
    Response = maps:get(Key, State, undefined),
    {reply, Response, State};

% We need to check our current state, like get_fea

handle_call({get_state}, From, State) ->
    Response = {current_state, State},
    {reply, Response, State};

% All other messages will be dropped here.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

% handle_cast/2
% put/2 will execute this function.

handle_cast({put, {Key, Value}}, State) ->
    NewState = maps:put(Key, Value, State),
    {noreply, NewState};

% delete/1 will execute this function.
```

```
handle_cast({delete, Key}, State) ->
    NewState = maps:remove(Key, State),
    {noreply, NewState};

% All other messages are dropped here.

handle_cast(_Msg, State) ->
    {noreply, State}.

%%%%%%%%%%%%%%%%
% other handlers

% We don't need other features, other handlers do nothing.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

## Использование нашего кеш-сервера

Теперь мы можем скомпилировать наш код и начать использовать его с `erl` .

```
% compile cache
c(cache).

% starting cache server
cache:start_link().

% get current store
% will return:
%   #{}
cache:state().

% put some data
cache:put(1, one).
cache:put(hello, bonjour).
cache:put(list, []).

% get current store
% will return:
%   #{1 => one,hello => bonjour,list => []}
cache:state().

% delete a value
cache:delete(1).
cache:state().
%   #{1 => one,hello => bonjour,list => []}

% stopping cache server
cache:stop().
```

Прочитайте поведение gen_server онлайн: https://riptutorial.com/ru/erlang/topic/7481/

поведение-gen-server

# глава 11: поведения

## Examples

### Использование поведения

Добавьте директиву `-behaviour` в свой модуль, чтобы указать, что он следует за поведением:

```
-behaviour(gen_server).
```

Американское правописание также принято:

```
-behavior(gen_server).
```

Теперь компилятор даст предупреждение, если вы забыли реализовать и экспортировать любую из функций, требуемых поведением, например:

```
foo.erl:2: Warning: undefined callback function init/1 (behaviour 'gen_server')
```

### Определение поведения

Вы можете определить свое поведение, добавив в свой модуль директивы `-callback` . Например, если модули, реализующие ваше поведение, должны иметь функцию `foo` которая принимает целое число и возвращает атом:

```
-module(my_behaviour).
-callback foo(integer()) -> atom().
```

Если вы используете это поведение в другом модуле, компилятор будет предупреждать, если он не экспортирует `foo/1` , и Dialyzer будет предупреждать, если типы неверны. С помощью этого модуля:

```
-module(bar).
-behaviour(my_behaviour).
-export([foo/1]).

foo([]) ->
    {}.
```

и работает `dialyzer --src bar.erl my_behaviour.erl` , вы получаете следующие предупреждения:

```
bar.erl:5: The inferred type for the 1st argument of foo/1 ([]) is not a supertype of
integer(), which is expected type for this argument in the callback of the my_behaviour
```

```
behaviour
bar.erl:5: The inferred return type of foo/1 ({}) has nothing in common with atom(), which is
the expected return type for the callback of my_behaviour behaviour
```

## Необязательные обратные вызовы в пользовательском поведении

18,0

По умолчанию любая функция, указанная в директиве `-callback` в модуле поведения, должна быть экспортирована модулем, который реализует это поведение. В противном случае вы получите предупреждение о компиляторе.

Иногда вы хотите, чтобы функция обратного вызова была необязательной: поведение будет использовать ее, если она присутствует и экспортируется, и в противном случае отказываться от реализации по умолчанию. Для этого напишите директиву `-callback` как обычно, а затем перечислите функцию обратного вызова в директиве `-optional_callbacks` :

```
-callback bar() -> ok.
-optional_callbacks([bar/0]).
```

Если модуль экспортирует `bar/0` , Dialyzer по-прежнему будет проверять спецификацию типа, но если функция отсутствует, вы не получите предупреждение о компиляторе.

В самой Erlang / OTP, это делается для `format_status` функции обратного вызова в `gen_server` , `gen_fsm` и `gen_event` поведения.

Прочитайте поведения онлайн: https://riptutorial.com/ru/erlang/topic/7004/поведения

# глава 12: Процессы

## Examples

### Создание процессов

Мы создаем новый параллельный процесс, вызывая функцию `spawn` . Функция `spawn` получит в качестве параметра функцию `Fun` которую процесс будет оценивать. Возвращаемое значение функции `spawn` - это созданный идентификатор процесса (pid).

```
1> Fun = fun() -> 2+2 end.
#Fun<erl_eval.20.52032458>
2> Pid = spawn(Fun).
<0.60.0>
```

Вы также можете использовать `spawn/3` для запуска процесса, который будет выполнять определенную функцию из модуля: `spawn(Module, Function, Args)` .
Или используйте `spawn/2` или `spawn/4` аналогично, чтобы запустить процесс в другом узле: `spawn(Node, Fun)` или `spawn(Node, Module, Function, Args)` .

### Передача сообщений

Два процесса erlang могут взаимодействовать друг с другом, что также известно как *передача сообщений* .
Эта процедура является *асинхронной* в том виде, в котором процесс отправки не будет остановлен после отправки сообщения.

## Отправка сообщений

Это может быть достигнуто с помощью конструкции `Pid ! Message` , где `Pid` является допустимым идентификатором процесса (pid), а `Message` - значением любого типа данных.

Каждый процесс имеет «почтовый ящик», который содержит полученные сообщения в полученном порядке. Этот «почтовый ящик» может быть опустошен с помощью встроенной функции `flush/0` .

Если сообщение отправляется в не существующий процесс, сообщение будет отброшено без каких-либо ошибок!

Пример может выглядеть следующим образом: `self/0` возвращает pid текущего процесса, а `pid/3` создает pid.

```
1> Pidsh = self().
<0.32.0>
```

```
2> Pidsh ! hello.
hello
3> flush().
Shell got hello
ok
4> <0.32.0> ! hello.
* 1: syntax error before: '<'
5> Pidsh2 = pid(0,32,0).
<0.32.0>
6> Pidsh2 ! hello2.
hello2
7> flush().
Shell got hello2
ok
```

Также можно отправить сообщение нескольким процессам сразу, с `Pid3!Pid2!Pid1!Msg` .

## Получение сообщений

Полученные сообщения могут обрабатываться с помощью конструкции `receive` .

```
receive
  Pattern1            -> exp11, .., exp1n1;
  Pattern2 when Guard -> exp21, .., exp2n2;
  ...
  Other               -> exp31, .., exp3n3;
  ...
  after Timeout       -> exp41, .., exp4n4
end
```

`Pattern` будет сравниваться с сообщениями в «почтовом ящике», начиная с первого и самого старого сообщений.
Если шаблон совпадает, совпадающее сообщение удаляется из «почтового ящика» и вычисляется тело предложения.

Также можно определить тайм-ауты с `after` конструкцией.
`Timeout` - это либо время ожидания в миллисекундах, либо `infinity` атома.

Возвращаемое значение `receive` - это последний обработанный объект предложения.

## Пример (счетчик)

А (очень) простой счетчик с передачей сообщений может выглядеть следующим образом.

```
-module(counter0).
-export([start/0,loop/1]).

% Creating the counter process.
start() ->
    spawn(counter0, loop, [0]).

% The counter loop.
```

```
loop(Val) ->
    receive
        increment ->
           loop(Val + 1)
    end.
```

Взаимодействие с прилавком.

```
1> C0 = counter0:start().
<0.39.0>
2> C0!increment.
increment
3> C0!increment.
increment
```

## Регистрация процессов

Можно зарегистрировать процесс (pid) для глобального псевдонима.
Это может быть достигнуто с помощью функции создания в `register(Alias, Pid)` , где `Alias` является атомом для доступа к процессу, а `Pid` - идентификатором процесса.

Псевдоним будет доступен по всему миру!
Очень просто создать общее состояние, которое обычно не является предпочтительным. ( См. Также здесь )

Можно отменить регистрацию процесса с `unregister(Pid)` и получить pid из псевдонима `whereis(Alias)` .

Используйте `registered()` список всех зарегистрированных псевдонимов.

Пример регистрирует Atom foo для pid текущего процесса и отправляет сообщение с использованием зарегистрированного Atom.

```
1> register(foo, self()).
true
2> foo ! 'hello world'.
'hello world'
```

Прочитайте Процессы онлайн: https://riptutorial.com/ru/erlang/topic/2285/процессы

# глава 13: Синтаксис бит: значения по умолчанию

## Вступление

В документе Erlang под названием «Синтаксис бит» есть раздел «По умолчанию», который содержит ошибку, и документ запутан в целом. Я переписал его.

## Examples

### Объяснения по умолчанию

> 4.4 По умолчанию
> ...
> ...
> Размер по умолчанию зависит от типа. Для целого числа оно равно 8. Для float оно равно 64. Для двоичного значения это размер указанного двоичного файла:
>
> ```
> 1> Bin = << 17/integer, 3.2/float, <<97, 98, 99>>/binary >>.
> <<17,64,9,153,153,153,153,153,154,97,98,99>>
>
> 2> size(Bin). % Returns the number of bytes:
> 12            % 8 bits + 64 bits + 3*8 bits = 96 bits => 96/8 = 12 bytes
> ```
>
> При сопоставлении двоичный сегмент без размера допускается только при t

Прочитайте Синтаксис бит: значения по умолчанию онлайн:
https://riptutorial.com/ru/erlang/topic/10050/синтаксис-бит--значения-по-умолчанию

# глава 14: Синтаксис бит: значения по умолчанию

## Вступление

Бла бла бла.

## Examples

### Перепишите документы.

4.4 По умолчанию

[Начало опущено: << 3.14 >> не является даже юридическим синтаксисом.]

Размер по умолчанию зависит от типа. Для целого числа это 8. Для float это 64. Для двоичного значения это фактический размер указанного двоичного файла:

```
1> Bin = << 17/integer, 3.2/float, <<97, 98, 99>>/binary >>.
<<17,64,9,153,153,153,153,153,154,97,98,99>>
  ^ |<------------------------->|<------>|

  |             float=64          binary=24
integer=8
```

```
2> size(Bin). % Returns the number of bytes:
12            % 8 bits + 64 bits + 3*8 bits = 96 bits => 96/8 = 12 bytes
```

При сопоставлении двоичный сегмент без размера допускается только в конце шаблона, а размер по умолчанию - это остальная часть двоичного файла в правой части совпадения:

```
25> Bin = <<97, 98, 99>>.
<<"abc">>

26> << X/integer, Rest/binary >> = Bin.
<<"abc">>

27> X.
97

28> Rest.
<<"bc">>
```

Все остальные сегменты с двоичным типом типа в шаблоне должны указывать

размер:

```
12> Bin = <<97, 98, 99, 100>>.
<<"abcd">>

13> << B:1/binary, X/integer, Rest/binary >> = Bin. %'unit' defaults to 8 for
<<"abcd">>                    %binary type, total segment size is Size * unit

14> B.
<<"a">>

15> X.
98

16> Rest.
<<"cd">>

17> << B2/binary, X2/integer, Rest2/binary >> = Bin.
* 1: a binary field without size is only allowed at the end of a binary pattern
```

Прочитайте Синтаксис бит: значения по умолчанию онлайн:
https://riptutorial.com/ru/erlang/topic/10051/синтаксис-бит--значения-по-умолчанию

# глава 15: Типы данных

## замечания

Каждый тип данных в erlang называется Term. Это общее имя, которое означает *любой тип данных* .

## Examples

### чисел

В Erlang числа являются либо целыми числами, либо поплавками. Erlang использует произвольную точность для целых чисел (bignums), поэтому их значения ограничены только объемом памяти вашей системы.

```
1> 11.
11
2> -44.
-44
3> 0.1.
0.1
4> 5.1e-3.
0.0051
5> 5.2e2.
520.0
```

Числа могут использоваться в различных базах:

```
1> 2#101.
5
2> 16#ffff.
65535
```

`$` -prefix дает целочисленное значение любого символа ASCII / Unicode:

```
3> $a.
97
4> $A.
65
5> $2.
50
6> $□.
129302
```

### атомы

Атом - это объект с именем, который идентифицируется только самим именем.

Атомы определены в Эрланге с использованием атомных литералов, которые либо

- строка без кавычек, которая начинается с строчной буквы и содержит только буквы, цифры, символы подчеркивания или символ `@` или
- Одиночная кавычка

# Примеры

```
1> hello.
hello

2> hello_world.
hello_world

3> world_Hello@.
world_Hello@

4> '1234'.
'1234'

5> '!@#$%% ä'.
'!@#$%% ä'
```

# Атомы, которые используются в большинстве программ Erlang

Есть некоторые атомы, которые появляются почти в каждой программе Erlang, в частности, из-за их использования в стандартной библиотеке.

- `true` и `false` используются для обозначения соответствующих булевых значений
- `ok` обычно используется как возвращаемое значение функции, которая вызывается только для ее эффекта или как часть возвращаемого значения, в обоих случаях означает успешное выполнение
- Точно так же `error` используется для обозначения условия ошибки, которое не гарантирует раннего возврата из верхних функций
- `undefined` обычно используется в качестве заполнителя для неопределенного значения

# Использовать в качестве тегов

`ok` и `error` довольно часто используются как часть кортежа, в котором первый элемент кортежа сигнализирует об успешности, в то время как другие элементы содержат фактическое возвращаемое значение или условие ошибки:

```
func(Input) ->
    case Input of
        magic_value ->
            {ok, got_it};
        _ ->
            {error, wrong_one}
    end.

{ok, _} = func(SomeValue).
```

# Место хранения

Одна вещь, которую следует иметь в виду при использовании атомов, состоит в том, что они хранятся в их собственной глобальной таблице в памяти, и эта таблица не собирает мусор, поэтому динамическое создание атомов, особенно когда пользователь может влиять на имя атома, в значительной степени обескураживается.

## Бинарные и бинты

Двоичная последовательность представляет собой последовательность беззнаковых 8-битных байтов.

```
1> <<1,2,3,255>>.
<<1,2,3,255>>
2> <<256,257,258>>.
<<0,1,2>>
3> <<"hello","world">>.
<<"helloworld">>
```

Битовая строка является обобщенным двоичным кодом, длина которого в битах не обязательно кратна 8.

```
1> <<1:1, 0:2, 1:1>>.
<<9:4>> % 4 bits bitstring
```

## Кортеж

Кортеж представляет собой упорядоченную по порядку последовательность других членов Эрланга. Каждый элемент в кортеже может быть любым типом термина (любой тип данных).

```
1> {1, 2, 3}.
{1,2,3}
2> {one, two, three}.
{one,two,three}
3> {mix, atom, 123, {<<1,2>>, [list]}}.
{mix,atom,123,{<<1,2>>,[list]}}
```

### Списки

Список в Erlang представляет собой последовательность из нуля или более членов Erlang, реализованную как односвязный список. Каждый элемент в списке может быть любым типом термина (любой тип данных).

```
1> [1,2,3].
[1,2,3]
2> [wow,1,{a,b}].
[wow,1,{a,b}]
```

Головка списка - это первый элемент списка.

Хвост списка - это остальная часть списка (без головы). Это также список.
Вы можете использовать `hd/1` и `tl/1` или соответствовать `[H|T]` чтобы получить заголовок и хвост списка.

```
3> hd([1,2,3]).
1
4> tl([1,2,3]).
[2,3]
5> [H|T] = [1,2,3].
[1,2,3]
6> H.
1
7> T.
[2,3]
```

## Превращение элемента в список

```
8> [new | [1,2,3]].
[new,1,2,3]
```

## Конкатенационные списки

```
9> [concat,this] ++ [to,this].
[concat,this,to,this]
```

## Струны

В Erlang строки не являются отдельным типом данных: это всего лишь списки целых чисел, представляющих кодовые точки ASCII или Unicode:

```
> [97,98,99].
```

```
"abc"
> [97,98,99] =:= "abc".
true
> hd("ABC").
65
```

Когда оболочка Erlang собирается распечатать список, он пытается угадать, действительно ли вы на самом деле означали, что это строка. Вы можете отключить это поведение, вызвав `shell:strings(false)` :

```
> [8].
"\b"
> shell:strings(false).
true
> [8].
[8]
```

В приведенном выше примере целое число 8 интерпретируется как управляющий символ ASCII для backspace, который оболочка считает «допустимым» символом в строке.

## Идентификаторы процессов (Pid)

Каждый процесс в erlang имеет идентификатор процесса ( `Pid` ) в этом формате `<xxx>` , `x` - натуральное число. Ниже приведен пример `Pid`

```
<0.1252.0>
```

`Pid` может использоваться для отправки сообщений в процесс с использованием «bang» ( `!` ), Также `Pid` может быть ограничен переменной, оба показаны ниже

```
MyProcessId = self().
MyProcessId ! {"Say Hello"}.
```

Подробнее о создании процессов и многом другом о процессах в erlang

## Funs

Erlang - это функциональный язык программирования. Одной из особенностей языка программирования функций является обработка функций как данных (функциональных объектов).

- Передайте функцию в качестве аргумента другой функции.
- Функция возврата в результате функции.
- Удерживайте функции в некоторой структуре данных.

В Erlang эти функции называются funs. Развлечения - анонимные функции.

```
1> Fun = fun(X) -> X*X end.
```

```
#Fun<erl_eval.6.52032458>
2> Fun(5).
25
```

У забав также может быть несколько статей.

```
3> AddOrMult = fun(add,X) -> X+X;
3>                (mul,X) -> X*X
3> end.
#Fun<erl_eval.12.52032458>
4> AddOrMult(mul,5).
25
5> AddOrMult(add,5).
10
```

Вы также можете использовать функции модуля в качестве забавы с синтаксисом: `fun Module:Function/Arity`.
Например, давайте возьмем функцию `max` из модуля `lists`, у которого есть arity 1.

```
6> Max = fun lists:max/1.
#Fun<lists.max.1>
7> Max([1,3,5,9,2]).
9
```

## Карты

Карта представляет собой ассоциативный массив или словарь, состоящий из пар (ключ, значение).

```
1> M0 = #{}.
#{}
2> M1 = #{ "name" => "john", "age" => "28" }.
#{"age" => "28","name" => "john"}
3> M2 = #{ a => {M0, M1} }.
#{a => {#{},#{"age" => "28","name" => "john"}}}
```

Чтобы обновить карту:

```
1> M = #{ 1 => x }.
2> M#{ 1 => c }.
#{1 => c}
3> M.
#{1 => x}
```

Обновите только существующий ключ:

```
1> M = #{ 1 => a, 2 => b}.
2> M#{ 1 := c, 2:= d }.
#{1 => c,2 => d}
3> M#{ 3 := c }.
** exception error: {badkey,3}
```

Соответствие шаблону:

```
1> M = #{ name => "john", age => 28 }.
2> #{ name := Name, age := Age } = M.
3> Name.
"john"
4> Age.
28
```

## Синтаксис бит: значения по умолчанию

Разъяснение документа Erlang о бит Синтаксис:

> 4.4 По умолчанию
>
> [Начало опущено: << 3.14 >> не является даже юридическим синтаксисом.]
>
> Размер по умолчанию зависит от типа. Для целого числа это 8. Для float это 64. Для двоичного значения это фактический размер указанного двоичного файла:

```
   1> Bin = << 17/integer, 3.2/float, <<97, 98, 99>>/binary >>.
   <<17,64,9,153,153,153,153,153,154,97,98,99>>
     ^ |<------------------------->|<------>|
     |            float=64           binary=24
 integer=8


   2> size(Bin). % Returns the number of bytes:
   12            % 8 bits + 64 bits + 3*8 bits = 96 bits => 96/8 = 12 bytes
```

> При сопоставлении двоичный сегмент без размера допускается только в конце шаблона, а размер по умолчанию - это остальная часть двоичного файла в правой части совпадения:

```
25> Bin = <<97, 98, 99>>.
<<"abc">>

26> << X/integer, Rest/binary >> = Bin.
<<"abc">>

27> X.
97

28> Rest.
<<"bc">>
```

> Все остальные сегменты с двоичным типом типа в шаблоне должны указывать размер:

```
12> Bin = <<97, 98, 99, 100>>.
<<"abcd">>
```

```
13> << B:1/binary, X/integer, Rest/binary >> = Bin. %'unit' defaults to 8 for
<<"abcd">>                    %binary type, total segment size is Size * unit

14> B.
<<"a">>

15> X.
98

16> Rest.
<<"cd">>

17> << B2/binary, X2/integer, Rest2/binary >> = Bin.
* 1: a binary field without size is only allowed at the end of a binary pattern
```

Прочитайте Типы данных онлайн: https://riptutorial.com/ru/erlang/topic/1128/типы-данных

# глава 16: Файловый ввод-вывод

## Examples

### Чтение из файла

Предположим, у вас есть файл **lyrics.txt,** который содержит следующие данные:

```
summer has come and passed
the innocent can never last
wake me up when september ends
```

# Прочитайте весь файл за раз

Используя `file:read_file(File)` , вы можете прочитать весь файл. Это атомная операция:

```
1> file:read_file("lyrics.txt").
{ok,<<"summer has come and passed\r\nthe innocent can never last\r\nWake me up w
hen september ends\r\n">>}
```

# Прочитайте по одной строке за раз

`io:get_line` считывает текст до строки новой строки или конца файла.

```
1> {ok, S} = file:open("lyrics.txt", read).
{ok,<0.57.0>}
2> io:get_line(S, '').
"summer has come and passed\n"
3> io:get_line(S, '').
"the innocent can never last\n"
4> io:get_line(S, '').
"wake me up when september ends\n"
5> io:get_line(S, '').
eof
6> file:close(S).
ok
```

# Чтение со случайным доступом

`file:pread(IoDevice, Start, Len)` читает от `Start` столько же, сколько `Len` из `IoDevice` .

```
1> {ok, S} = file:open("lyrics.txt", read).
{ok,<0.57.0>}
2> file:pread(S, 0, 6).
```

```
{ok,"summer"}
3> file:pread(S, 7, 3).
{ok,"has"}
```

### Запись в файл

# Напишите по одной строке за раз

Откройте файл с режимом `write` и используйте `io:format/2` :

```
1> {ok, S} = file:open("fruit_count.txt", [write]).
{ok,<0.57.0>}
2> io:format(S, "~s~n", ["Mango 5"]).
ok
3> io:format(S, "~s~n", ["Olive 12"]).
ok
4> io:format(S, "~s~n", ["Watermelon 3"]).
ok
5>
```

Результатом будет файл с именем **fruit_count.txt** со следующим содержимым:

```
Mango 5
Olive 12
Watermelon 3
```

Обратите внимание, что открытие файла в режиме записи создаст его, если он еще не существует в файловой системе.

Также обратите внимание, что использование опции `write` с `file:open/2` приведет к **усечению** файла (даже если вы ничего не напишите в нем). Чтобы предотвратить это, откройте файл в режиме `[read,write]` или `[append]` .

# Напишите весь файл сразу

`file:write_file(Filename, IO)` - самая простая функция для записи файла сразу. Если файл уже существует, он будет перезаписан, иначе он будет создан.

```
1> file:write_file("fruit_count.txt", ["Mango 5\nOlive 12\nWatermelon 3\n"
]).
ok
2> file:read_file("fruit_count.txt").
{ok,<<"Mango 5\nOlive 12\nWatermelon 3\n">>}
3>
```

# Запись со случайным доступом

Для записи произвольного доступа используется `file:pwrite(IoDevice, Location, Bytes)`. Если вы хотите заменить некоторую строку в файле, этот метод полезен.

Предположим, вы хотите изменить «Olive 12» на «Apple 15» в файле, созданном выше.

```
1> {ok, S} = file:open("fruit_count.txt", [read, write]).
{ok,{file_descriptor,prim_file,{#Port<0.412>,676}}}
2> file:pwrite(S, 8, ["Apple 15\n"]).
ok
3> file:read_file("fruit_count.txt").
{ok,<<"Mango 5\nApple 15\nWatermelon 3">>}
4> file:close(S).
ok
5>
```

Прочитайте Файловый ввод-вывод онлайн: https://riptutorial.com/ru/erlang/topic/5232/файловый-ввод-вывод

# глава 17: Форматировать строки

## Синтаксис

- io: format (FormatString, Args)% записать в стандартный вывод
- io: format (standard_error, FormatString, Args)% записать в стандартную ошибку
- io: format (F, FormatString, Args)% записать в открытый файл
- io_lib: format (FormatString, Args)% возвращает iolist

## Examples

### Общие управляющие последовательности в строках формата

Хотя существует множество различных управляющих последовательностей для `io:format` и `io_lib:format` , большую часть времени вы будете использовать только три разных: `~s` , `~p` и `~w` .

## ~ s

`~s` для *строк* .

Он печатает строки, двоичные файлы и атомы. (Все остальное вызовет ошибку `badarg` .) Он ничего не цитирует и не избегает; он просто печатает строку:

```
%% Printing a string:
> io:format("~s\n", ["hello world"]).
hello world

%% Printing a binary:
> io:format("~s\n", [<<"hello world">>]).
hello world

%% Printing an atom:
> io:format("~s\n", ['hello world']).
hello world
```

## ~ ш

`~w` для *написания со стандартным синтаксисом* .

Он может печатать любой термин Эрланга. Вывод может быть проанализирован, чтобы вернуть исходный термин Эрланга, если в нем не содержались термины, которые не имеют синтаксического письменного представления, то есть pids, портов и ссылок. Он не

вставляет новые строки или отступы, а строки всегда интерпретируются как списки:

```
> io:format("~w\n", ["abc"]).
[97,98,99]
```

## ~ p

`~p` - для *довольно-печатной* .

Он может печатать любой термин Эрланга. Выход отличается от `~w` следующими способами:

- Новые строки вставляются, если линия в противном случае была бы слишком длинной.
- Когда новые строки вставлены, следующая строка имеет отступы, чтобы соответствовать предыдущему члену на том же уровне.
- Если список целых чисел выглядит как печатная строка, он интерпретируется как один.

```
> io:format("~p\n",
[{this,is,a,tuple,with,many,elements,'and',a,list,'of',numbers,[97,98,99],that,'end',up,making,the,line

{this,is,a,tuple,with,many,elements,'and',a,list,'of',numbers,"abc",that,
      'end',up,making,the,line,too,long}
```

Если вы не хотите, чтобы списки целых чисел были напечатаны в виде строк, вы можете использовать последовательность `~lp` (вставить строчную букву L перед `p` ):

```
> io:format("~lp\n", [[97,98,99]]).
[97,98,99]

> io:format("~lp\n", ["abc"]).
[97,98,99]
```

Прочитайте Форматировать строки онлайн: https://riptutorial.com/ru/erlang/topic/3722/форматировать-строки

# кредиты

| S. No | Главы | Contributors |
|---|---|---|
| 1 | Начало работы с языком Erlang | A. Sarid, CodeWarrior, Community, drozzy, Efraim Weiss, evnu, Gokul, legoscia, Limmen, maze-le, YsenGrimm, ZeWaren |
| 2 | iolists | Dennis Y. Parygin, legoscia |
| 3 | Rebar3 | drozzy, Victor Schröder |
| 4 | Внешний формат | M. Kerjouan |
| 5 | директор | Pouriya |
| 6 | Контролеры | legoscia |
| 7 | Монтаж | A. Sarid, drozzy, M. Kerjouan, ZeWaren |
| 8 | НИФ | Victor Schröder |
| 9 | Петля и рекурсия | M. Kerjouan |
| 10 | поведение gen_server | drozzy, legoscia, M. Kerjouan, Steve Pallen |
| 11 | поведения | legoscia |
| 12 | Процессы | A. Sarid, YsenGrimm |
| 13 | Синтаксис бит: значения по умолчанию | 7stud |
| 14 | Типы данных | 7stud, A. Sarid, Ali Sabil, Atomic_alarm, bigo, drozzy, Eddie Antonio Santos, Evgeny Levenets, filmor, gabriel14, Gokul, Gootik, halfelf, legoscia |
| 15 | Файловый ввод-вывод | CodeDreamer, drozzy, Victor Schröder |
| 16 | Форматировать строки | drozzy, legoscia |