EBook Gratis

APRENDIZAJE

Erlang Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#erlang

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con Erlang Language

Observaciones

"Erlang es un lenguaje de programación desarrollado originalmente en Ericsson Computer Science Laboratory. OTP (Open Telecom Platform) es una colección de middleware y bibliotecas en Erlang. Erlang / OTP ha sido probado en una serie de productos de Ericsson para crear robustos tolerantes a fallas aplicaciones distribuidas, por ejemplo AXD301 (conmutador ATM). Erlang / OTP se mantiene actualmente en la unidad Erlang / OTP en Ericsson "(erlang.org)

Empieza aqui

Para las instrucciones de instalación, vea el tema de instalación .

Campo de golf

1. Sitio oficial de Erlang: https://www.erlang.org
2. Gestor de paquetes popular para Erlang y Elixir: http://hex.pm
3. Patrones de Erlang: http://www.erlangpatterns.org/

Versiones

Versión	Notas de lanzamiento	Fecha de lanzamiento
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
17.3	http://erlang.org/download/otp_src_17.3.readme	2014-09-17

Versión	Notas de lanzamiento	Fecha de lanzamiento
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

Hola Mundo

Hay dos cosas que debe saber al escribir una aplicación "hola mundo" en Erlang:

1. El código fuente está escrito en el *lenguaje de programación erlang* usando el editor de texto de su elección
2. La aplicación se ejecuta en la *máquina virtual erlang* . En este ejemplo, interactuaremos con el erlang VM a través del shell erlang.

Primero el código fuente de la aplicación:

Cree un nuevo archivo `hello.erl` contenga lo siguiente:

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

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

Echemos un vistazo rápido a lo que esto significa:

- `-module(hello)` . Todas las funciones erlang existen dentro de un *módulo* . Los módulos se utilizan para crear aplicaciones, que son una colección de módulos. Esta primera línea es para identificar este módulo, a saber, *hola* . Los módulos se pueden comparar a los *paquetes* de Java
- `-export([hello_world/0])` . Indica al compilador qué funciones hacer "públicas" (en comparación con los idiomas OO) y la *aridad* de la función relevante. La aridad es el número de argumentos que toma la función. Dado que en erlang, una función con 1 argumento se ve como una función diferente a una con 2 argumentos, aunque el nombre puede ser exactamente el mismo. Es decir, `hello_world/0` es una función completamente diferente a `hello_world/1`

por ejemplo.

- `hello_world()` Este es el nombre de la función. El `->` indica la transición a la implementación (cuerpo) de la función. Esto se puede leer como "hello_world () se define como ...". Tenga en cuenta que `hello_world()` (sin argumentos) se identifica por `hello_world/0` en la máquina virtual, y `hello_world(Some_Arg)` como `hello_world/1` .
- `io:format("Hello, World!~n", [])` Desde el módulo `io` , se llama a la función `format/2` function, que es la función para la salida estándar. `~n` es un especificador de formato que significa imprimir una nueva línea. La `[]` es una lista de variables para imprimir indicadas por especificadores de formato en la cadena de salida, que en este caso no es nada.
- Todas las declaraciones erlang deben terminar con a `.` (punto).

En Erlang, se devuelve el resultado de la última instrucción en una función.

Ahora, vamos a ejecutar nuestra aplicación:

Inicie el shell erlang desde el mismo directorio que el archivo `hello.erl` :

```
$ erl
```

Debería recibir un mensaje que se parece a esto (su versión puede ser diferente):

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

Ahora ingrese los siguientes comandos:

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

Vayamos a través de cada línea una por una:

- `c(hello)` : este comando llama a la función `c` en un átomo `hello` . Esto le indica a Erlang que busque el archivo `hello.erl` , lo compile en un módulo (se generará un archivo llamado `hello.beam` en el directorio) y lo carga en el entorno.
- `{ok, hello}` - este es el resultado de llamar a la función `c` arriba. Es una tupla que contiene un átomo `ok` y un átomo `hello` . Las funciones de Erlang generalmente devuelven `{ok, Something}` o `{error, Reason}` .
- `hello:hello_world()` : esto llama a una función `hello_world()` desde el módulo `hello` .
- `Hello, World!` - Esto es lo que imprime nuestra función.
- `ok` - esto es lo que devolvió nuestra función. Dado que Erlang es un lenguaje de programación funcional, cada función devuelve *algo* . En nuestro caso, aunque no `hello_world()` nada en `hello_world()` , la última llamada en esa función fue `io:format(...)` y esa función devolvió `ok` , que a su vez es lo que nuestra función devolvió.

Módulos

Un módulo erlang es un archivo con un par de funciones agrupadas. Este archivo usualmente tiene la extensión `.erl` .

A `hello.erl` se muestra un módulo "Hello World" con el nombre `hello.erl`

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

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

En el archivo, se requiere declarar el nombre del módulo. Como se muestra antes en la línea 1. El nombre del módulo y el nombre del archivo antes de la extensión `.erl` deben ser iguales.

Función

La función es un conjunto de instrucciones, que se agrupan. Estas instrucciones agrupadas realizan ciertas tareas. En erlang, todas las funciones devolverán un valor cuando sean llamadas.

A continuación se muestra un ejemplo de una función que suma dos números.

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

Esta función realiza una operación de adición con valores X e Y y devuelve el resultado. La función se puede utilizar como a continuación

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

Las declaraciones de funciones pueden constar de varias cláusulas, separadas por un punto y coma. Los argumentos en cada una de estas cláusulas se evalúan mediante la coincidencia de patrones. La siguiente función devolverá 'tupla' si el Argumento es una tupla en el Formulario: {prueba, X} donde X puede ser cualquier valor. Devolverá 'lista', si el Argumento es una lista de la longitud 2 en la forma ["prueba", X], y devolverá '{error, "Razón"}' en cualquier otro caso:

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

Si el argumento no es una tupla, se evaluará la segunda cláusula. Si el argumento no es una lista, la tercera cláusula será evaluada.

Las declaraciones de funciones pueden consistir en los llamados 'Guardias' o 'Secuencias de Guardias'. Estos guardias son expresiones que limitan la evaluación de una función. Una función con Guardias solo se ejecuta cuando todas las Expresiones de Guard producen un valor verdadero. Los guardias múltiples se pueden separar por un punto y coma.

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

La función 'function_name' solo se evaluará cuando la secuencia de guarda sea verdadera. La

función de seguimiento devolverá verdadero solo si el argumento `X` está en el rango apropiado (0..15):

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

Comprensión de lista

Las comprensiones de listas son una construcción sintáctica para crear una lista basada en listas existentes.
En erlang una lista de comprensión tiene la forma `[Expr || Qualifier1, ..., QualifierN]` .
Donde los calificadores son generadores `Pattern <- ListExpr` o filtro como `integer(X)` evaluándose como `true` o `false` .

El siguiente ejemplo muestra una lista de comprensión con un generador y dos filtros.

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

El resultado es una lista que contiene sólo números enteros mayores que 3.

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

Comenzando y deteniendo la cáscara de Erlang

Comenzando la cáscara de Erlang

En un sistema UNIX, inicia el shell Erlang desde un símbolo del sistema con el comando `erl`

Ejemplo:

```
$ 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>
```

El texto que se muestra cuando inicia el shell le brinda información sobre la versión de Erlang que está ejecutando, así como otra información útil sobre el sistema erlang.

Para iniciar el shell en Windows, haga clic en el ícono Erlang en el menú de inicio de Windows.

Deteniendo la concha de Erlang

Para una salida controlada del shell erlang escribe:

```
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().
```

También puede salir del shell Erlang presionando Ctrl + C en sistemas UNIX o Ctrl + Break en Windows, lo que le lleva al siguiente indicador:

```
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
```

Si luego presiona a (para abortar), saldrá de la shell directamente.

Otras formas de salir del shell erlang son: `init:stop()` que hace lo mismo que `q()` o `erlang:halt()` .

La coincidencia de patrones

Una de las operaciones más comunes en erlang es la coincidencia de patrones. Se usa cuando se asigna un valor a una variable, en declaraciones de funciones y en estructuras de flujo de control como las declaraciones de `case` y `receive` . Una operación de coincidencia de patrones necesita al menos 2 partes: un patrón y un término con el cual se empareja el patrón.

Una asignación de variable en erlang se ve así:

```
X = 2.
```

En la mayoría de los lenguajes de programación, la semántica de esta operación es sencilla: vincule un valor (`2`) al nombre que elija (la variable, en este caso `X`). Erlang tiene un enfoque ligeramente diferente: haga coincidir el patrón en el lado izquierdo (`X`) con el término en el lado derecho (`2`). En este caso, el efecto es el mismo: la variable `X` ahora está vinculada al valor `2` . Sin embargo, con la coincidencia de patrones puede realizar tareas más estructuradas.

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

Esta operación de coincidencia se realiza analizando la estructura del término del lado derecho y aplicando todas las variables del lado izquierdo a los valores apropiados del término, de modo que el lado izquierdo sea igual al lado derecho. En este ejemplo, `Type` está vinculado al término: `document` , `Meta` a `{author, "Alice"}` y `Doc` a `{text, "Lorem Ipsum"}` . En este ejemplo particular, se asume que las variables: `Type` , `Meta` y `Doc` están *unidas* , por lo que se puede usar cada variable.

Los emparejamientos de patrones también se pueden construir, utilizando variables encuadernadas.

```
Identifier = error.
```

El `Identifier` variable ahora está vinculado al `error` valor. La siguiente operación de coincidencia

de patrones funciona, porque la estructura coincide y el `Identifier` variable enlazada tiene el mismo valor que la parte derecha apropiada del término.

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

Una operación de coincidencia de patrón falla, cuando hay una discrepancia entre el término del lado derecho y el patrón del lado izquierdo. La siguiente coincidencia fallará, porque el `Identifier` está vinculado al `error` valor, que no tiene una expresión adecuada en el término del lado derecho.

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

Lea Empezando con Erlang Language en línea:
https://riptutorial.com/es/erlang/topic/825/empezando-con-erlang-language

Capítulo 2: Archivo I / O

Examples

Leyendo de un archivo

Supongamos que tiene un archivo **lyrics.txt** que contiene los siguientes datos:

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

Lee el archivo completo a la vez

Al usar `file:read_file(File)` , puedes leer el archivo completo. Es una operación atómica:

```
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">>}
```

Lee una línea a la vez

`io:get_line` lee el texto hasta la nueva línea o el final del archivo.

```
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
```

Leer con el acceso aleatorio

`file:pread(IoDevice, Start, Len)` lee desde `Start` tanto como `Len` desde `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"}
```

Escribiendo en un archivo

Escribe una línea a la vez

Abra un archivo con el modo de `write` y use `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>
```

El resultado será un archivo llamado **fruit_count.txt** con el siguiente contenido:

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

Tenga en cuenta que abrir un archivo en modo de escritura lo creará, si aún no existe en el sistema de archivos.

Tenga en cuenta también que el uso de la opción de `write` con `file:open/2` **truncará** el archivo (incluso si no escribe nada en él). Para evitar esto, abra el archivo en modo `[read,write]` o `[append]` .

Escribe todo el archivo a la vez

`file:write_file(Filename, IO)` es la función más simple para escribir un archivo a la vez. Si el archivo ya existe, se sobrescribirá, de lo contrario se creará.

```
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>
```

Escribir con acceso aleatorio

Para escritura de acceso aleatorio, se `file:pwrite(IoDevice, Location, Bytes)` . Si desea

reemplazar alguna cadena en el archivo, este método es útil.

Supongamos que desea cambiar "Olive 12" a "Apple 15" en el archivo creado anteriormente.

```
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>
```

Lea Archivo I / O en línea: https://riptutorial.com/es/erlang/topic/5232/archivo-i---o

Capítulo 3: Bucle y recursion

Sintaxis

- función (lista | iolist | tupla) -> función (cola).

Observaciones

¿Por qué las funciones recursivas?

Erlang es un lenguaje de programación funcional y no tiene ningún tipo de estructura de bucle. Todo en la programación funcional se basa en datos, tipo y funciones. Si desea un bucle, necesita crear una función que se llame a sí misma.

El `while` tradicional o `for` bucle en lenguaje imperativo y orientado a objetos se puede representar así en Erlang:

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

Un buen método para entender este concepto es expandir todas las llamadas de función. Lo veremos en otros ejemplos.

Examples

Lista

Aquí la función recursiva más simple sobre el tipo de lista . Esta función solo navega en una lista desde el principio hasta el final y no hace nada más.

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

Puedes llamarlo así:

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

Aquí la expansión de la función recursiva:

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

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

Bucle recursivo con acciones IO.

El código anterior no hace nada, y es bastante inútil. Entonces, crearemos ahora una función recursiva que ejecutará algunas acciones. Este código es similar a las `lists:foreach/2` .

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

Puedes llamarlo así:

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

Aquí la expansión de la función recursiva:

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

Puedes comparar con `lists:foreach/2` output:

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

Bucle recursivo sobre lista que devuelve lista modificada

Otro ejemplo útil, similar a las `lists:map/2` . Esta función tomará una lista y una función anónima. Cada vez que un valor en la lista coincide, aplicamos la función en él.

```
-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).
```

Puedes llamarlo así:

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

Aquí la expansión de la función recursiva:

```
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].
```

Esta función también se denomina "función recursiva de cola", porque usamos una variable como un acumulador para pasar datos modificados sobre un contexto de ejecución múltiple.

Iolist y Bitstring

Lista de Me gusta, la función más simple sobre iolist y la cadena de bits es:

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

Puedes llamarlo así:

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

Aquí la expansión de la función recursiva:

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

Función recursiva sobre tamaño binario

variable.

Este código toma la cadena de bits y define dinámicamente su tamaño binario. Así que si, si establecemos un tamaño de `4` , cada `4` bits, se emparejará un dato. Este bucle no hace nada interesante, es solo nuestro pilar.

```
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.
```

Puedes llamarlo así:

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

Aquí la expansión de la función recursiva:

```
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.
```

Nuestra cadena de bits está dividida en 7 patrones. ¿Por qué? Porque por defecto, Erlang usa un tamaño binario de `8` bits, si lo dividimos en dos, tenemos `4` bits. Nuestra cadena es `8*3=24` bits. `24/4=6` patrones. El último patrón es `<<>>` . `loop/2` se llama 7 veces.

Función recursiva sobre tamaño binario variable con acciones.

Ahora, podemos hacer algo más interesante. Esta función toma un argumento más, una función anónima. Cada vez que coincidimos con un patrón, éste será pasado a él.

```
-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.
```

Puedes llamarlo así:

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

Aquí la expansión de la función recursiva:

```
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.
```

Función recursiva sobre la cadena de bits que devuelve la cadena de bits modificada

Éste es similar a las `lists:map/2` pero para bitstring e 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.
```

Este código parece más complejo. Se agregaron dos funciones: `loop/2` y `loop/3` . Estas dos funciones son interfaz simple a `loop/4` .

Puedes ejecutarlo así:

```
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>>
```

Mapa

El mapa en Erlang es equivalente a hashes en Perl o diccionarios en Python, es un almacén de clave / valor. Para enumerar cada valor almacenado en, puede enumerar cada clave y devolver el par clave / valor. Este primer bucle te dará una idea:

```
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).
```

Puedes ejecutarlo así:

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

Estado gestor

La función recursiva usa sus estados para hacer un bucle. Cuando genera un nuevo proceso, este proceso será simplemente un bucle con un estado definido.

Función anónima

Aquí 2 ejemplos de funciones anónimas recursivas basadas en el ejemplo anterior. En primer lugar, simple bucle infinito:

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

En segundo lugar, la función anónima haciendo loop over list:

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

Estas dos funciones pueden ser reescritas como:

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

En este caso, `loop/0` es una referencia al `loop/0` de los comentarios. En segundo lugar, con poco más complejo:

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

Aquí, `loop/2` es una referencia a `loop/2` del ejemplo de lista. Estas dos notaciones son azúcares sintácticos.

Lea Bucle y recursion en línea: https://riptutorial.com/es/erlang/topic/10720/bucle-y-recursion

Capítulo 4: comportamiento gen_server

Observaciones

`gen_server` es una característica importante de Erlang, y requiere algún requisito previo para comprender cada aspecto de esta funcionalidad:

- Loop, recursion y estado.
- Procesos de desove.
- Paso de mensajes
- Principios de la Fiscalía

Una buena manera de aprender más sobre una característica en Erlang es leer directamente el código fuente del repositorio oficial de github . `gen_server` comportamiento de `gen_server` mucha información útil y una estructura interesante en su núcleo.

`gen_server` se define en `gen_server.erl` y su documentación asociada se puede encontrar en la documentación de `stdlib` Erlang . `gen_server` es una característica de OTP y puede encontrar más información en la Guía del usuario y principios de diseño de OTP .

¡Frank Hebert también le brinda otra buena introducción a `gen_server` de su libro en línea gratuito Learn You Some Erlang para un gran bien!

Documentación oficial para `gen_server` llamada `gen_server` :

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

Examples

Servicio de Greeter

Aquí hay un ejemplo de un servicio que saluda a las personas por el nombre dado, y hace un seguimiento de cuántos usuarios se encontraron. Ver uso a continuación.

```
%% 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}.
```

Aquí hay una muestra de uso de este servicio en el shell 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}
```

Usando el comportamiento gen_server

Un `gen_server` es una máquina de estados finitos específica que funciona como un servidor. `gen_server` puede manejar diferentes tipos de eventos:

- solicitud síncrona con `handle_call`
- Solicitud asincrona con `handle_cast`
- otro mensaje (no definido en la especificación OTP) con `handle_info`

Los mensajes síncronos y asíncronos se especifican en OTP y son simples tuplas etiquetadas con cualquier tipo de datos.

Un `gen_server` simple se define así:

```
-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.
```

Este código es simple: cada mensaje recibido se imprime en una salida estándar.

comportamiento gen_server

Para definir un `gen_server` , debe declararlo explícitamente en su código fuente con `-behaviour(gen_server)` . Tenga en cuenta que el `behaviour` puede escribirse en EE. UU. (Comportamiento) o Reino Unido (comportamiento).

start_link / 0

Esta función es un atajo simple para llamar a otra función: `gen_server:start_link/3,4` .

start_link / 3,4

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

Se llama a esta función cuando desea iniciar su servidor vinculado a un `supervisor` u otro proceso. `start_link/3,4` puede registrar automáticamente su proceso (si cree que su proceso debe ser único) o simplemente puede generarlo como un proceso simple. Cuando se llama, esta función ejecuta `init/1` .

Esta función puede devolver estos valores definidos:

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

init / 1

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

`init/1` es la primera función ejecutada cuando se iniciará su servidor. Este inicializa todos los requisitos previos de su aplicación y devuelve el estado al proceso recién creado.

Esta función puede devolver solo estos valores definidos:

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

`State` variable de `State` puede ser todo (p. Ej., Lista, tupla, listas de distribución, mapa, registro) y permanecer accesible para todas las funciones dentro del proceso generado.

handle_call / 3

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

`gen_server:call/2` ejecuta esta devolución de llamada. El primer argumento es su mensaje (`_Request`), el segundo es el origen de la solicitud (`_From`) y el último es el estado actual (`State`) de su comportamiento gen_server en ejecución.

Si desea una respuesta a la persona que llama, `handle_call/3` necesita devolver una de estas estructuras de datos:

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

Si no desea responder al llamante, `handle_call/3` necesita devolver una de estas estructuras de datos:

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

Si desea detener la ejecución actual de su gen_server actual, `handle_call/3` necesita devolver una de estas estructuras de datos:

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

handle_cast / 2

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

`gen_server:cast/2` ejecuta esta devolución de llamada. El primer argumento es su mensaje (`_Msg`), y el segundo es el estado actual de su comportamiento gen_server en ejecución.

De forma predeterminada, esta función no puede proporcionar datos a la persona que llama, por lo tanto, solo tiene dos opciones, continuar la ejecución actual:

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

O `gen_server` proceso `gen_server` actual:

- `{stop,Reason,NewState}`

handle_info / 2

```
handle_info(_Info, State) ->
```

```
    {noreply, State}.
```

`handle_info/2` se ejecuta cuando un mensaje OTP no estándar proviene del mundo exterior. Este no puede responder y, al igual que `handle_cast/2` , solo puede realizar 2 acciones, continuando con la ejecución actual:

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

O detenga el proceso `gen_server` ejecución actual:

- `{stop,Reason,NewState}`

terminar / 2

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

`terminate/2` se llama cuando ocurre un error o cuando quiere cerrar su proceso `gen_server` .

code_change / 3

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

`code_change/3` función `code_change/3` cuando desea actualizar su código de ejecución.

Esta función puede devolver solo estos valores definidos:

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

Iniciando este proceso

Puedes compilar tu código e iniciar `simple_gen_server` :

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

Si desea enviar un mensaje a su servidor, puede usar estas funciones:

```
% 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).
```

Base de datos clave / valor simple

Este código fuente crea un servicio simple de almacenamiento de clave / valor basado en la estructura de datos de Erlang del `map` . En primer lugar, necesitamos definir toda la información relativa a nuestro `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}.
```

Usando nuestro servidor de caché

Ahora podemos compilar nuestro código y comenzar a usarlo con `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().
```

Lea comportamiento gen_server en línea:
https://riptutorial.com/es/erlang/topic/7481/comportamiento-gen-server

Capítulo 5: Comportamientos

Examples

Usando un comportamiento

Agregue una directiva `-behaviour` a su módulo para indicar que sigue un comportamiento:

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

La ortografía americana también es aceptada:

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

Ahora el compilador dará una advertencia si ha olvidado implementar y exportar cualquiera de las funciones requeridas por el comportamiento, por ejemplo:

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

Definiendo un comportamiento

Puede definir su propio comportamiento agregando directivas de `-callback` en su módulo. Por ejemplo, si los módulos que implementan su comportamiento necesitan tener una función `foo` que tome un entero y devuelva un átomo:

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

Si usa este comportamiento en otro módulo, el compilador avisará si no exporta `foo/1` , y Dialyzer avisará si los tipos no son correctos. Con este módulo:

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

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

y ejecutando `dialyzer --src bar.erl my_behaviour.erl` , obtienes estas advertencias:

```
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
```

Retrollamadas opcionales en un comportamiento personalizado

18.0

De forma predeterminada, cualquier función especificada en una directiva `-callback` en un módulo de comportamiento debe ser exportada por un módulo que implemente ese comportamiento. De lo contrario, obtendrá una advertencia del compilador.

A veces, desea que una función de devolución de llamada sea opcional: el comportamiento lo usaría si estuviera presente y exportado, y de lo contrario recurriría a una implementación predeterminada. Para hacerlo, escriba la directiva `-callback` como de costumbre, y luego enumere la función de devolución de llamada en una directiva `-optional_callbacks` :

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

Si el módulo exporta la `bar/0` , Dialyzer seguirá verificando la especificación de tipo, pero si la función está ausente, no recibirá una advertencia del compilador.

En Erlang / OTP, esto se hace para la función de devolución de llamada `format_status` en los `gen_server` , `gen_fsm` y `gen_event` .

Lea Comportamientos en línea: https://riptutorial.com/es/erlang/topic/7004/comportamientos

Capítulo 6: director

Introducción

Biblioteca de supervisión flexible, rápida y potente para los procesos de Erlang.

Observaciones

Advertencias

- No use `'count'=>infinity` y `restart` elementos en su plan.
  me gusta:

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

¡Si su proceso no se inició después de la falla, el **director** se bloqueará y volverá a intentar reiniciar su proceso `infinity` ! Si está usando `infinity` para `'count'` , use siempre `{restart, MiliSeconds}` en `'plan'` lugar de `restart` .

- Si tienes planes como:

```
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}.
```

¡El resto del elemento de `delete` en `Childspec1` y el resto del elemento de `stop` en `Childspec2` nunca se evaluarán!
¡En `Childspec3` quieres ejecutar tu plan 0 veces!
¡En `ChildSpec4` no tienes ningún plan para ejecutar tiempos `infinity` !

- Cuando actualiza una versión utilizando `release_handler` , `release_handler` llama a `supervisor:get_callback_module/1` para obtener su módulo de devolución de llamada.
  En OTP <19, `get_callback_module/1` utiliza el registro de estado interno del supervisor para proporcionar su módulo de devolución de llamada. Nuestro **director** no conoce el registro de estado interno del supervisor, luego el `supervisor:get_callback_module/1` no funciona con los **directores** .
  La buena noticia es que en OTP> = 19 `supervisor:get_callback_module/1` funciona perfectamente con el **director** s :).

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

2> supervisor:get_callback_module(foo_sup).
foo

3>
```

Examples

Descargar

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

Compilar

Tenga en cuenta que se requiere **OTP> = 19** (si desea actualizarlo con `release_handler`).
Ir al `director` y usar `rebar` o `rebar rebar3` .

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

barra de refuerzo

```
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 $
```

Cómo funciona

El director necesita un módulo de devolución de llamada (como supervisor de OTP).
En el módulo de devolución de llamada debe exportar la función `init/1` .

¿Qué `init/1` debería devolver? Espera, te lo explicaré paso a paso.

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

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

Guarde el código anterior en `foo.erl` en el **directorio del** director y vaya al shell de Erlang.
Use `erl -pa ./ebin` si usó `rebar` para compilarlo y use el `rebar3 shell` si usó `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>
```

Ahora tenemos un supervisor sin hijos.
La buena noticia es que el **director** viene con la API completa de OTP / supervisor y también tiene sus características avanzadas y su enfoque específico.

```
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
[]
```

OK, haré simple `gen_server` y se lo daré a nuestro **director** .

```
-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.
```

Guarde el código anterior en `bar.erl` y vuelva al Shell.

```
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>
```

Vamos a comprobarlo

```
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
```

Mencioné **características avanzadas** , ¿cuáles son? Veamos otras claves aceptables para el mapa de `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().
```

Editar módulo `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]}.
```

Ir a la concha de Erlang de nuevo:

```
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
```

Finalmente, ¿cuál es la clave de `append` ?

En realidad siempre tenemos un `DefaultChildspec` .

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

15>
```

`DefaultChildspec` es como los childspecs normales, excepto que no puede aceptar las claves de `id` y `append` .
Si cambio el valor `append` a `true` en mi `Childspec` :
My `terminate_timeout` se agregará a `terminate_timeout` de `DefaultChildspec` .
Mi `count` se agregará a la `count` de `DefaultChildspec` .
Mis `modules` se agregarán a los `modules` de `DefaultChildspec` .
Mi `plan` se agregará al `plan` de `DefaultChildspec` .
Y si tengo la clave de `start` con el valor `{ModX, FuncX, ArgsX}` en `DefaultChildspec` y la clave de `start` con el valor `{ModY, FunY, ArgsY}` en `Childspec` , el valor final será `{ModY, FuncY, ArgsX ++ ArgsY}` .
Y, finalmente, si tengo la clave de `start` con el valor `{Mod, Func, Args}` en `DefaultChildspec` , la clave de `start` en `Childspec` es opcional para mí.
Puede devolver su propio `DefaultChildspec` como tercer elemento de tupla en `init/1` .
Editar `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}.
```

Reinicie el shell:

```
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>
```

¡Puedes cambiar dinámicamente los `defaultChildspec` usando `change_default_childspec/2` !
¡Y también puede cambiar dinámicamente `Childspec` de los niños y establecer su `append` en `true` !
Pero al cambiarlos en diferentes partes del código, harás un **código de espagueti.**

¿Puedo depurar el director?

Sí, el **diorector** tiene su propia depuración y acepta `sys:dbg_opt/0` estándar `sys:dbg_opt/0` .
el director envía registros válidos a `sasl` y `error_logger` en diferentes estados también.

```
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>
```

Generar documentación API

rebar

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

rebar3:

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

erl

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

Después de ejecutar uno de los comandos anteriores, la documentación HTML debe estar en el

directorio `doc` .

Lea director en línea: https://riptutorial.com/es/erlang/topic/9878/director

Capítulo 7: Formato de cadenas

Sintaxis

- io: format (FormatString, Args)% de escritura en salida estándar
- io: format (standard_error, FormatString, Args)% de escritura en error estándar
- io: format (F, FormatString, Args)% escribe en archivo abierto
- io_lib: format (FormatString, Args)% return an iolist

Examples

Secuencias de control comunes en cadenas de formato.

Si bien hay muchas secuencias de control diferentes para `io:format` y `io_lib:format` , la mayoría de las veces solo usará tres diferentes: `~s` , `~p` y `~w` .

~ s

El `~s` es para *cuerdas* .

Imprime cuerdas, binarios y átomos. (Cualquier otra cosa causará un error `badarg`). No cita ni escapa nada; simplemente imprime la cadena en sí:

```
%% 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
```

~ w

El `~w` es para *escribir con sintaxis estándar* .

Puede imprimir cualquier término de Erlang. La salida se puede analizar para devolver el término original de Erlang, a menos que contenga términos que no tengan una representación escrita analizable, es decir, pids, puertos y referencias. No inserta ninguna nueva línea ni sangría, y las cadenas siempre se interpretan como listas:

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

```
[97,98,99]
```

~ p

El `~p` es para *impresión bonita* .

Puede imprimir cualquier término de Erlang. La salida difiere de `~w` de las siguientes maneras:

- Las líneas nuevas se insertan si la línea fuera demasiado larga.
- Cuando se insertan nuevas líneas, la siguiente línea se sangra para alinearse con un término anterior en el mismo nivel.
- Si una lista de enteros se parece a una cadena imprimible, se interpreta como una.

```
> 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}
```

Si no desea que las listas de enteros se impriman como cadenas, puede usar la secuencia `~lp` (inserte una letra L minúscula antes de `p`):

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

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

Lea Formato de cadenas en línea: https://riptutorial.com/es/erlang/topic/3722/formato-de-cadenas

Capítulo 8: Formato de término externo

Introducción

El Formato de término externo es un formato binario utilizado para comunicarse con el mundo exterior. Puedes usarlo con cualquier idioma a través de puertos, drivers o NIF. BERT (Binary ERlang Term) se puede utilizar en otros idiomas.

Examples

Usando ETF con Erlang

```
binary_to_term().
```

Usando ETF con C

Inicializando estructura de datos

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

Número de codificación

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

int
main() {
}
```

Átomo de codificación

Tupla de codificacion

Lista de codificacion

Mapa de codificacion

Lea Formato de término externo en línea: https://riptutorial.com/es/erlang/topic/10731/formato-de-termino-externo

Capítulo 9: Instalación

Examples

Construir e instalar Erlang / OTP en Ubuntu

Los siguientes ejemplos muestran dos métodos principales para instalar Erlang / OTP en Ubuntu.

Método 1 - Paquete binario pre-construido

Simplemente ejecute este comando y se descargará e instalará la última versión estable de Erlang de Erlang Solutions .

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

Método 2 - Construir e instalar desde la fuente

Descarga el archivo tar:

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

Extraer el archivo tar:

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

Ingrese al directorio extraído y configure `ERL_TOP` para que sea la ruta actual:

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

Ahora, antes de configurar la compilación, desea asegurarse de tener todas las dependencias que necesita para instalar Erlang:

Dependencias básicas:

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

Otras dependencias de aplicaciones.

Solicitud	Instalación de dependencia
Hola	`$ 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`
Documentación	`$ sudo apt-get install fop xsltproc`
Orber y otros proyectos de C ++	`$ sudo apt-get install g++`
cara de interfaz	`$ sudo apt-get install default-jdk`

Configurar y construir:
Puede configurar sus propias opciones o dejarlo en blanco para ejecutar la configuración predeterminada. Configuración avanzada y compilación para Erlang / OTP .

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

Probando la construcción:

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

Después de ejecutar estos comandos, abra `$ERL_TOP/release/tests/test_server/index.html` con su navegador web y verifique que no haya errores. Si todas las pruebas pasaron, podemos continuar con la instalación.

Instalación:

```
$ make install
```

Construye e instala Erlang / OTP en FreeBSD

Los siguientes ejemplos muestran 3 métodos principales para instalar Erlang / OTP en FreeBSD.

Método 1 - Paquete binario pre-construido

Usa pkg para instalar el paquete binario pre-construido:

```
$ pkg install erlang
```

Prueba tu nueva instalación:

```
$ 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)
```

Método 2: compilar e instalar utilizando la colección de puertos (recomendado)

Construya e instale el puerto como de costumbre:

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

Prueba tu nueva instalación:

```
$ 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)
```

Esto traerá el lanzamiento del lanzamiento del sitio web oficial, aplicará algunos parches si es necesario, creará el lanzamiento e instalarlo. Obviamente, llevará algún tiempo.

Método 3 - Construir e instalar desde el lanzamiento tarball

Nota: la creación de la versión funciona manualmente, pero se debe preferir el uso de los dos métodos anteriores, ya que la colección de puertos incluye parches que hacen que la versión sea más fácil para FreeBSD.

Descarga el archivo de lanzamiento:

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

Compruebe que su suma MD5 es correcta:

```
$ 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
```

Extraer el tarball:

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

Configurar:

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

Si desea compilar Erlang con HiPe, deberá aplicar los parches de la colección de puertos.

Construir:

```
$ gmake
```

Instalar:

```
$ gmake install
```

Prueba tu nueva instalación:

```
$ 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)
```

Construir e instalar utilizando kerl

kerl es una herramienta que te ayuda a construir e instalar versiones de Erlang / OTP.

Instalar rizo:

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

Descargar kerl:

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

Actualizar la lista de lanzamientos disponibles:

```
$ ./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
```

Construye el lanzamiento requerido:

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

Compruebe que la compilación está presente en la lista de compilación:

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

Instale la compilación en algún lugar:

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

Fuente el archivo `activate` si está ejecutando bash o la cáscara de pescado. Si está ejecutando un cshell, agregue el directorio bin de compilación al PATH:

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

Prueba tu nueva instalación:

```
$ 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)
```

Otros lanzamientos

Si desea compilar otra versión de Erlang / OTP, busque los otros puertos en la colección:

- lang / erlang-runtime15
- lang / erlang-runtime16
- lang / erlang-runtime17
- lang / erlang-runtime18

Referencia

- Manual de FreeBSD -> Capítulo 4. Instalación de aplicaciones: paquetes y puertos
- Erlang en FreshPorts
- Documentación Kerl en GitHub

Construye e instala Erlang / OTP en OpenBSD

Erlang en OpenBSD está actualmente roto en `alpha` arquitecturas `alpha` , `sparc` y `hppa` .

Método 1 - Paquete binario pre-construido

OpenBSD le permite elegir la versión deseada que desea instalar en su sistema:

```
##############################################################################
# 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 puede soportar múltiples versiones de Erlang. Para hacer que el pensamiento sea más fácil de usar, cada uno de los binarios se instala en su versión Erlang. Por lo tanto, si ha instalado `erlang-19.0v0` , su binario `erl` será `erl19` .

Si quieres usar `erl` , puedes crear un enlace simbólico:

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

o cree un alias en su archivo de configuración de shell o en el archivo `.profile` :

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

Ahora puedes ejecutar `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)
```

Método 2 - Construir e instalar utilizando puertos

```
 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
```

Método 3 - Construir desde la fuente

La compilación desde la fuente requiere paquetes adicionales:

- `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
```

Referencias

- 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

Lea Instalación en línea: https://riptutorial.com/es/erlang/topic/4483/instalacion

Capítulo 10: iolistas

Introducción

Mientras que una cadena Erlang es una lista de enteros, un "iolist" es una lista cuyos elementos son enteros, binarios u otros iolists, por ejemplo, `["foo", $b, $a, $r, <<"baz">>]` . Ese iolist representa la cadena `"foobarbaz"` .

Si bien puede convertir un iolist a un binario con `iolist_to_binary/1` , a menudo no lo necesita, ya que las funciones de la biblioteca Erlang como el `file:write_file/2` y `gen_tcp:send/2` aceptan iolists, así como cadenas y archivos binarios.

Sintaxis

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

Observaciones

¿Qué es un iolist?

Es cualquier binario. O cualquier lista que contenga enteros entre 0 y 255. O cualquier lista anidada arbitrariamente que contenga cualquiera de esas dos cosas.

Artículo original

Utilice listas profundamente anidadas de enteros y binarios para representar datos de IO para evitar copiarlos al concatenar cadenas o binarios.

Son eficientes incluso cuando se combinan grandes cantidades de datos. Por ejemplo, combinar dos binarios de cincuenta kilobytes utilizando la sintaxis binaria `<<B1/binary, B2/binary>>` normalmente requeriría la reasignación de ambos en un nuevo binario de 100kb. El uso de listas de IO `[B1, B2]` solo asigna la lista, en este caso tres palabras. Una lista utiliza una palabra y otra palabra por elemento, consulte aquí para obtener más información.

El uso del operador `++` habría creado una lista completamente nueva, en lugar de una lista de dos elementos nuevos. Volver a crear listas para agregar elementos al final puede resultar costoso cuando la lista es larga.

En los casos en que los datos binarios son pequeños, la asignación de listas de E / S puede ser mayor que agregar los binarios. Si los datos binarios pueden ser pequeños o grandes, a menudo es mejor aceptar el costo constante de las listas de IO.

Tenga en cuenta que los binarios agregados se optimizan como se describe aquí . En resumen, un binario puede tener espacio extra oculto asignado. Se llenará si se le agrega otro binario que se ajuste al espacio libre. Esto significa que no todos los apéndices binarios causarán una copia completa de ambos binarios.

Examples

Las listas de E / S se utilizan normalmente para generar la salida a un puerto, por ejemplo, un archivo o un socket de red.

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

Agregue los tipos de datos permitidos al frente de una lista de IO, creando una nueva.

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

Los datos de IO se pueden agregar de manera eficiente al final de una lista.

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

Tenga cuidado con las listas impropias

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

En el shell se imprimirá como `["Guten tag "|<<"Hello">>]` lugar de `["Guten tag ",<<"Hello">>]` . El operador de tubería creará una lista incorrecta si el último elemento a la derecha no es una lista. Si bien una lista inadecuada cuya "cola" es un binario sigue siendo un iolist válido, las listas inadecuadas pueden causar problemas porque muchas funciones recursivas esperan que una lista vacía sea el último elemento, y no, como en este caso un binario.

Obtener el tamaño de la lista IO

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

El tamaño de un iolist se puede calcular utilizando `iolist_size/1` . Este fragmento calcula el tamaño de un mensaje y lo crea y lo agrega al frente como un binario de cuatro bytes. Esta es una operación típica en los protocolos de mensajería.

La lista IO se puede convertir a un binario

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

Una lista de E / S puede convertirse en un binario utilizando la función `iolist_to_binary/1` . Si los

datos se almacenarán durante un período prolongado o se enviarán como un mensaje a otros procesos, entonces puede tener sentido convertirlos a un binario. El costo único de convertir a un binario puede ser más barato que copiar la lista de IO muchas veces, en la recolección de basura de un solo proceso o en el paso de mensajes a otros.

Lea iolistas en línea: https://riptutorial.com/es/erlang/topic/5677/iolistas

Capítulo 11: NIFs

Examples

Definición

Documentación oficial: http://erlang.org/doc/tutorial/nif.html

Los NIF se introdujeron en Erlang / OTP R13B03 como una característica experimental. El propósito es permitir llamar el código C desde el código de Erlang.

Los NIF se implementan en C en lugar de Erlang, pero aparecen como cualquier otra función en el alcance del código de Erlang, ya que pertenecen al módulo donde ocurrió la inclusión. Las bibliotecas NIF están vinculadas en la compilación y cargadas en tiempo de ejecución.

Debido a que las bibliotecas NIF están vinculadas dinámicamente al proceso del emulador, son rápidas, pero también peligrosas, ya que al bloquearse en un NIF también se desactiva el emulador.

Ejemplo: hora actual de UNIX

Aquí hay un ejemplo muy simple para ilustrar cómo escribir un NIF.

Estructura de directorios:

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

Esta estructura se puede inicializar fácilmente usando Rebar3:

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

Contenido de `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);
```

Contenido de `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).
```

Contenido de `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"}
]}.
```

Ahora puedes ejecutar el ejemplo:

```
$ 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>
```

Erlang C API (C a Erlang)

Documentación oficial : http://erlang.org/doc/man/erl_nif.html

Las estructuras, tipos y macros más importantes de la API de Erlang C son las siguientes:

- `ERL_NIF_TERM` : el tipo para los términos de Erlang. Este es el tipo de retorno que deben seguir las funciones NIF.
- `ERL_NIF_INIT(MODULE, ErlNifFunc funcs[], load, reload, upgrade, unload)` : esta es la macro que realmente crea los NIF definidos en un determinado archivo C. Debe ser evaluado en el ámbito global. Normalmente será la última línea en el archivo C.
- `ErlNifFunc` : el tipo con el cual cada NIF se pasa a `ERL_NIF_INIT` para ser exportado. Esta estructura se compone de un nombre, una aridad, un indicador de la función C y las banderas. Se debe crear una matriz de este tipo con todas las definiciones de NIF para `ERL_NIF_INIT` a `ERL_NIF_INIT` .
- `ErlNifEnv` : el entorno Erlang donde se ejecuta el NIF. Es obligatorio pasar el entorno como el primer argumento para cada NIF. Este tipo es opaco y solo se puede manipular utilizando las funciones que ofrece la API de Erlang C.

Lea NIFs en línea: https://riptutorial.com/es/erlang/topic/5274/nifs

Capítulo 12: Procesos

Examples

Procesos de creación

Creamos un nuevo proceso concurrente llamando a la función `spawn` . El `spawn` función tendrá como parámetro de una función de `Fun` que el proceso va a evaluar. El valor de retorno de la función de `spawn` es el identificador de proceso creado (pid).

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

También puede usar `spawn/3` para iniciar un proceso que ejecutará una función específica desde un módulo: `spawn(Module, Function, Args)` .
O use `spawn/2` o `spawn/4` manera similar para iniciar un proceso en un nodo diferente: `spawn(Node, Fun)` o `spawn(Node, Module, Function, Args)` .

Paso de mensajes

Dos procesos erlang pueden comunicarse entre sí, lo que también se conoce como *paso de mensajes* .
Este procedimiento es *asíncrono* en la forma en que el proceso de envío no se detendrá después de enviar el mensaje.

Enviando mensajes

Esto se puede lograr con el constructo `Pid ! Message` , donde `Pid` es un identificador de proceso válido (pid) y `Message` es un valor de cualquier tipo de datos.

Cada proceso tiene un "buzón" que contiene los mensajes recibidos en el orden recibido. Este "buzón" se puede vaciar con la función de compilación `flush/0` .

Si se envía un mensaje a un proceso no existente, el mensaje se descartará sin ningún error.

Un ejemplo podría ser similar al siguiente, donde `self/0` devuelve el pid del proceso actual y `pid/3` crea un 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
```

También es posible enviar un mensaje a varios procesos a la vez, con `Pid3!Pid2!Pid1!Msg` .

Recibiendo mensajes

Los mensajes recibidos se pueden procesar con la construcción `receive` .

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

El `Pattern` se comparará con los mensajes en el "buzón" comenzando con el primer y más antiguo mensaje.
Si un patrón coincide, el mensaje coincidente se elimina del "buzón" y se evalúa el cuerpo de la cláusula.

También es posible definir tiempos de espera con la construcción `after` .
Un tiempo de `Timeout` es el tiempo de espera en milisegundos o el átomo `infinity` .

El valor de retorno de `receive` es el último cuerpo de la cláusula evaluada.

Ejemplo (Contador)

Un contador (muy) simple con paso de mensajes podría ser similar al siguiente.

```
-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.
```

Interacción con el contador.

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

Procesos de registro

Es posible registrar un proceso (pid) a un alias global.
Esto se puede lograr con la función de creación de `register(Alias, Pid)` , donde `Alias` es el átomo para acceder al proceso y `Pid` es el ID del proceso.

¡El alias estará disponible a nivel mundial!
Es muy fácil crear un estado compartido, que generalmente no es preferible. (Véase también aquí)

Es posible anular el registro de un proceso con `unregister(Pid)` y recibir el pid de un alias con `whereis(Alias)` .

Utilice `registered()` para obtener una lista de todos los alias registrados.

El ejemplo registra el Atom foo al pid del proceso actual y envía un mensaje utilizando el Atom registrado.

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

Lea Procesos en línea: https://riptutorial.com/es/erlang/topic/2285/procesos

Capítulo 13: Rebar3

Examples

Definición

Página oficial : https://www.rebar3.org/

Código fuente : https://github.com/erlang/rebar3

Rebar3 es principalmente un administrador de dependencias para proyectos Erlang y Elixir, pero también ofrece varias otras características, como proyectos de arranque (según varias plantillas, siguiendo los principios de OTP), ejecutor de tareas, herramienta de compilación, corredor de prueba y es extensible mediante el uso de complementos.

Instalar Rebar3

Rebar3 está escrito en Erlang, así que necesitas Erlang para ejecutarlo. Está disponible como un binario que puede descargar y ejecutar. Solo busca la compilación nocturna y dale permisos de ejecución:

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

Coloque este binario en un lugar conveniente y agréguelo a su ruta. Por ejemplo, en un directorio bin en su casa:

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

Esta última línea se debe poner en su `.bashrc` . Como alternativa, también se puede vincular el binario al directorio `/usr/local/bin` , haciéndolo disponible como un comando normal.

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

Instalación desde el código fuente

Como Rebar3 es gratuito, de código abierto y escrito en Erlang, es posible simplemente clonarlo y compilarlo a partir del código fuente.

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

Esto creará la secuencia de comandos rebar3, que puede poner en su PATH o enlace a `/usr/local/bin` como se explica en la sección "Instalar Rebar3" más arriba.

Arrancando un nuevo proyecto de Erlang

Para iniciar un nuevo proyecto de Erlang, simplemente elija la plantilla que desea usar de la lista. Las plantillas disponibles se pueden recuperar con el siguiente comando:

```
$ 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
```

Una vez que haya elegido la plantilla adecuada, inicie con el siguiente comando (rebar3 creará un nuevo directorio para su proyecto):

```
$ 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: Aunque *puede* ejecutar `rebar3 new <template> .` para crear el nuevo proyecto en el directorio actual, esto no se recomienda, ya que se usarán los archivos bootstrapped `.` (punto) como nombres de aplicaciones y módulos, y también en `rebar.config` , lo que le causará problemas de sintaxis.

Lea Rebar3 en línea: https://riptutorial.com/es/erlang/topic/4480/rebar3

Capítulo 14: Sintaxis de bits: valores predeterminados

Introducción

El documento Erlang titulado Bit Syntax tiene una sección llamada Valores predeterminados que contiene un error y el documento es confuso en su totalidad. Lo reescribí.

Examples

Los valores predeterminados explicados

> 4.4 Valores predeterminados
> ...
> ...
> El tamaño predeterminado depende del tipo. Para entero es 8. Para float es 64. Para binario es el tamaño del binario especificado:

```
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
```

> En el emparejamiento, un segmento binario sin un Tamaño solo se permite en t

Lea Sintaxis de bits: valores predeterminados en línea:
https://riptutorial.com/es/erlang/topic/10050/sintaxis-de-bits--valores-predeterminados

Capítulo 15: Sintaxis de bits: valores predeterminados

Introducción

Bla, bla, bla.

Examples

Reescritura de los documentos.

4.4 Valores predeterminados

[Principio omitido: << 3.14 >> ni siquiera es una sintaxis legal.]

El tamaño predeterminado depende del tipo. Para entero es 8. Para float es 64. Para binario es el tamaño real del binario especificado:

```
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
```

En la coincidencia, un segmento binario sin un Tamaño solo se permite al final del patrón, y el Tamaño predeterminado es el resto del binario en el lado derecho de la coincidencia:

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

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

27> X.
97

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

Todos los demás segmentos con tipo binario en un patrón deben especificar un Tamaño:

```
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
```

Lea Sintaxis de bits: valores predeterminados en línea:
https://riptutorial.com/es/erlang/topic/10051/sintaxis-de-bits--valores-predeterminados

Capítulo 16: Supervisores

Examples

Supervisor básico con un proceso de trabajo.

Este ejemplo utiliza el formato de mapa introducido en 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}}.
```

Lea Supervisores en línea: https://riptutorial.com/es/erlang/topic/6996/supervisores

Capítulo 17: Tipos de datos

Observaciones

Cada tipo de datos en erlang se llama término. Es un nombre genérico que significa *cualquier tipo de datos* .

Examples

Números

En Erlang, los números son enteros o flotantes. Erlang utiliza precisión arbitraria para enteros (bignums), por lo que sus valores están limitados solo por el tamaño de la memoria de su sistema.

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

Los números se pueden utilizar en varias bases:

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

La notación de `$` prefijo proporciona el valor entero de cualquier carácter ASCII / Unicode:

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

Los átomos

Un átomo es un objeto con un nombre que se identifica solo por el nombre mismo.

Los átomos se definen en Erlang usando literales atómicos que son

- una cadena sin comillas que comienza con una letra minúscula y solo contiene letras, dígitos, guiones bajos o el carácter `@` , o
- Una sola cadena entre comillas

Ejemplos

```
1> hello.
hello

2> hello_world.
hello_world

3> world_Hello@.
world_Hello@

4> '1234'.
'1234'

5> '!@#$%% ä'.
'!@#$%% ä'
```

Átomos que se utilizan en la mayoría de los programas de Erlang

Hay algunos átomos que aparecen en casi todos los programas de Erlang, en particular debido a su uso en la Biblioteca Estándar.

- `true` y `false` son los utilizados para denotar los valores booleanos respectivos
- `ok` se usa generalmente como un valor de retorno de una función que se llama solo por su efecto, o como parte de un valor de retorno, en ambos casos para significar una ejecución exitosa
- De la misma manera, el `error` se utiliza para indicar una condición de error que no garantiza un retorno anticipado de las funciones superiores
- `undefined` se usa generalmente como un marcador de posición para un valor no especificado

Utilizar como etiquetas

`ok` y `error` se usan con frecuencia como parte de una tupla, en la que el primer elemento de la tupla indica el éxito, mientras que otros elementos contienen el valor de retorno real o la condición de error:

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

```
            {error, wrong_one}
    end.

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

Almacenamiento

Una cosa que se debe tener en cuenta cuando se usan átomos es que se almacenan en su propia tabla global en la memoria y que esta tabla no se recolecta, por lo que se crean átomos dinámicamente, en particular cuando un usuario puede influir en el nombre del átomo.

Binarios y cadenas de bits

Un binario es una secuencia de bytes de 8 bits sin firmar.

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

Una cadena de bits es un binario generalizado cuya longitud en bits no es necesariamente un múltiplo de 8.

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

Tuplas

Una tupla es una secuencia ordenada de longitud fija de otros términos de Erlang. Cada elemento de la tupla puede ser cualquier tipo de término (cualquier tipo de datos).

```
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]}}
```

Liza

Una lista en Erlang es una secuencia de cero o más términos de Erlang, implementada como una lista enlazada individualmente. Cada elemento de la lista puede ser cualquier tipo de término (cualquier tipo de datos).

```
1> [1,2,3].
[1,2,3]
```

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

La cabeza de la lista es el primer elemento de la lista.

La cola de la lista es el resto de la lista (sin la cabeza). También es una lista.
Puede usar `hd/1` y `tl/1` o comparar `[H|T]` para obtener la cabecera y la cola de la lista.

```
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]
```

Anteponer un elemento a una lista

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

Listas de concatenacion

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

Instrumentos de cuerda

En Erlang, las cadenas no son un tipo de datos separado: son solo listas de enteros que representan puntos de código ASCII o Unicode:

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

Cuando el shell de Erlang va a imprimir una lista, intenta adivinar si realmente querías que fuera una cadena. Puedes desactivar ese comportamiento llamando a `shell:strings(false)` :

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

```
true
> [8].
[8]
```

En el ejemplo anterior, el entero 8 se interpreta como el carácter de control ASCII para el retroceso, que el shell considera un carácter "válido" en una cadena.

Identificadores de Procesos (Pid)

Cada proceso en erlang tiene un identificador de proceso (`Pid`) en este formato `<xxx>` , siendo `x` un número natural. A continuación se muestra un ejemplo de un `Pid`

```
<0.1252.0>
```

`Pid` se puede usar para enviar mensajes al proceso usando 'bang' (`!`), También `Pid` se puede enlazar a una variable, ambas se muestran a continuación

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

Lea más sobre la creación de procesos y más en general sobre los procesos en erlang

Diversión

Erlang es un lenguaje de programación funcional. Una de las características de un lenguaje de programación de funciones es manejar las funciones como datos (objetos funcionales).

- Pasa una función como argumento a otra función.
- Función de retorno como resultado de una función.
- Mantener funciones en alguna estructura de datos.

En Erlang esas funciones se llaman funs. Las diversiones son funciones anónimas.

```
1> Fun = fun(X) -> X*X end.
#Fun<erl_eval.6.52032458>
2> Fun(5).
25
```

Las diversiones también pueden tener varias cláusulas.

```
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
```

También puede usar las funciones del módulo como `fun Module:Function/Arity` con la sintaxis: `fun`

`Module:Function/Arity` .

Por ejemplo, tomemos la función `max` del módulo de `lists` , que tiene arity 1.

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

Mapas

Un mapa es una matriz asociativa o diccionario compuesto de pares (clave, valor).

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

Para actualizar un mapa:

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

Solo actualice alguna clave existente:

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

La coincidencia de patrones:

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

Sintaxis de bits: valores predeterminados

Aclaración de Erlang doc en Bit Sintaxis:

> 4.4 Valores predeterminados
>
> [Principio omitido: << 3.14 >> ni siquiera es una sintaxis legal.]

El tamaño predeterminado depende del tipo. Para entero es 8. Para float es 64. Para binario es el tamaño real del binario especificado:

```
  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
```

En la coincidencia, un segmento binario sin un Tamaño solo se permite al final del patrón, y el Tamaño predeterminado es el resto del binario en el lado derecho de la coincidencia:

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

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

27> X.
97

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

Todos los demás segmentos con tipo binario en un patrón deben especificar un Tamaño:

```
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
```

Lea Tipos de datos en línea: https://riptutorial.com/es/erlang/topic/1128/tipos-de-datos

Creditos

S. No	Capítulos	Contributors
1	Empezando con Erlang Language	A. Sarid, CodeWarrior, Community, drozzy, Efraim Weiss, evnu, Gokul, legoscia, Limmen, maze-le, YsenGrimm, ZeWaren
2	Archivo I / O	CodeDreamer, drozzy, Victor Schröder
3	Bucle y recursion	M. Kerjouan
4	comportamiento gen_server	drozzy, legoscia, M. Kerjouan, Steve Pallen
5	Comportamientos	legoscia
6	director	Pouriya
7	Formato de cadenas	drozzy, legoscia
8	Formato de término externo	M. Kerjouan
9	Instalación	A. Sarid, drozzy, M. Kerjouan, ZeWaren
10	iolistas	Dennis Y. Parygin, legoscia
11	NIFs	Victor Schröder
12	Procesos	A. Sarid, YsenGrimm
13	Rebar3	drozzy, Victor Schröder
14	Sintaxis de bits: valores predeterminados	7stud
15	Supervisores	legoscia
16	Tipos de datos	7stud, A. Sarid, Ali Sabil, Atomic_alarm, bigo, drozzy, Eddie Antonio Santos, Evgeny Levenets, filmor, gabriel14, Gokul, Gootik, halfelf, legoscia