EBook Gratuito

APPRENDIMENTO
Erlang Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#erlang

Sommario

Di ... 1

Capitolo 1: Iniziare con Erlang Language ... 2

Osservazioni ... 2

Comincià qui ... 2

link ... 2

Versioni ... 2

Examples ... 3

Ciao mondo ... 3

Prima il codice sorgente dell'applicazione: ... 3

Ora, eseguiamo la nostra applicazione: ... 4

moduli ... 5

Funzione ... 5

Comprensione delle liste ... 6

Avvio e arresto della shell di Erlang ... 6

Pattern Matching ... 7

Capitolo 2: Bit Sintassi: valori predefiniti ... 9

introduzione ... 9

Examples ... 9

Le impostazioni predefinite sono state spiegate ... 9

Capitolo 3: Bit Sintassi: valori predefiniti ... 10

introduzione ... 10

Examples ... 10

Riscrivi i documenti ... 10

Capitolo 4: comportamenti ... 12

Examples ... 12

Usando un comportamento ... 12

Definire un comportamento ... 12

Richiami opzionali in un comportamento personalizzato ... 13

Capitolo 5: comportamento di gen_server ... 14

Osservazioni ... 14

Examples ... 14
Servizio di accoglienza ... 14
Utilizzando il comportamento di gen_server ... 15
comportamento di gen_server ... 17
start_link / 0 ... 17
start_link / 3,4 ... 17
init / 1 ... 17
handle_call / 3 ... 18
handle_cast / 2 ... 18
handle_info / 2 ... 18
interrompere / 2 ... 19
code_change / 3 ... 19
Avvio di questo processo ... 19
Semplice database di chiavi / valori ... 20
Usando il nostro server cache ... 22
Capitolo 6: direttore ... 23
introduzione ... 23
Osservazioni ... 23
Avvertenze ... 23
Examples ... 24
Scaricare ... 24
Compilare ... 24
Come funziona ... 24
Posso eseguire il debug di director? ... 32
Genera documentazione API ... 33
Capitolo 7: File I / O ... 34
Examples ... 34
Lettura da un file ... 34
Leggi l'intero file alla volta ... 34
Leggi una riga alla volta ... 34
Leggi con l'accesso casuale ... 34

Scrivere su un file ... 35
Scrivi una riga alla volta ... 35
Scrivi l'intero file in una sola volta ... 35
Scrivi con accesso casuale ... 35
Capitolo 8: Formato stringhe ... 37
Sintassi ... 37
Examples ... 37
Sequenze di controllo comuni nelle stringhe di formato ... 37
~ s ... 37
~ w ... 37
~ p ... 38
Capitolo 9: Formato termine esterno ... 39
introduzione ... 39
Examples ... 39
Usare ETF con Erlang ... 39
Usare ETF con C ... 39
Inizializzazione della struttura dei dati ... 39
Numero di codifica ... 39
Atomo di codifica ... 39
Tupla di codifica ... 39
Lista di codifica ... 39
Mappa di codifica ... 39
Capitolo 10: Installazione ... 41
Examples ... 41
Costruisci e installa Erlang / OTP su Ubuntu ... 41
Metodo 1 - Pacchetto binario precostruito ... 41
Metodo 2: compilazione e installazione dalla sorgente ... 41
Costruisci e installa Erlang / OTP su FreeBSD ... 42
Metodo 1 - Pacchetto binario precostruito ... 42
Metodo 2 - Costruire e installare usando la collezione di porte (raccomandata) ... 43

Metodo 3: crea e installa dal tarball di rilascio 43
Costruisci e installa usando kerl 44
Altre versioni 45
Riferimento 45
Costruisci e installa Erlang / OTP su OpenBSD 45
Metodo 1 - Pacchetto binario precostruito 45
Metodo 2: creare e installare utilizzando le porte 46
Metodo 3 - Costruisci dalla fonte 47
Riferimenti 47
Capitolo 11: iolists 48
introduzione 48
Sintassi 48
Osservazioni 48
Examples 49
Gli elenchi di I / O vengono in genere utilizzati per creare output su una porta, ad esemp 49
Aggiungi i tipi di dati consentiti all'inizio di un elenco IO, creando uno nuovo. 49
I dati I / O possono essere aggiunti in modo efficiente alla fine di un elenco. 49
Fai attenzione agli elenchi impropri 49
Ottieni dimensioni elenco IO 49
Lista I / O può essere convertita in un file binario 49
Capitolo 12: Le autorità di vigilanza 51
Examples 51
Supervisore di base con un processo di lavoro 51
Capitolo 13: Loop e ricorsione 52
Sintassi 52
Osservazioni 52
Perché le funzioni ricorsive? 52
Examples 52
Elenco 52
Ciclo ricorsivo con azioni IO 53
Ciclo ricorsivo su lista che restituisce lista modificata 53

Iolist e Bitstring ... 54
Funzione ricorsiva su dimensione binaria variabile ... 54
funzione ricorsiva su dimensione binaria variabile con azioni ... 55
Funzione ricorsiva su bitstring che restituisce bitstring modificato ... 56
Carta geografica ... 57
Stato di gestione ... 57
Funzione anonima ... 57
Capitolo 14: NIF ... 59
Examples ... 59
Definizione ... 59
Esempio: tempo UNIX corrente ... 59
Erlang C API (C a Erlang) ... 61
Capitolo 15: Processi ... 62
Examples ... 62
Creazione di processi ... 62
Messaggio in corso ... 62
Invio di messaggi ... 62
Ricevere messaggi ... 63
Esempio (contatore) ... 63
Registra i processi ... 64
Capitolo 16: Rebar3 ... 65
Examples ... 65
Definizione ... 65
Installare Rebar3 ... 65
Installazione dal codice sorgente ... 65
Avvio automatico di un nuovo progetto Erlang ... 66
Capitolo 17: Tipi di dati ... 67
Osservazioni ... 67
Examples ... 67
Numeri ... 67
atomi ... 67

Esempi 68
Atomi che sono usati nella maggior parte dei programmi di Erlang 68
Usa come tag 68
Conservazione 69
Binari e Bitstring 69
Le tuple 69
elenchi 69
Prependendo un elemento a un elenco 70
Elenchi concatenanti 70
stringhe 70
Identificatori di processi (Pid) 71
funs 71
Mappe 72
Bit Sintassi: valori predefiniti 72
Titoli di coda 74

Di

Capitolo 1: Iniziare con Erlang Language

Osservazioni

"Erlang è un linguaggio di programmazione originariamente sviluppato presso l'Ericsson Computer Science Laboratory.OTP (Open Telecom Platform) è una raccolta di middleware e librerie in Erlang.Ellang / OTP è stato testato in battaglia in un certo numero di prodotti Ericsson per la costruzione di robusti fault-tolerant applicazioni distribuite, ad esempio AXD301 (switch ATM). Erlang / OTP è attualmente gestito dall'unità Erlang / OTP di Ericsson "(erlang.org)

Comincià qui

Per le istruzioni di installazione, consultare la sezione sull'installazione .

link

1. Sito ufficiale di Erlang: https://www.erlang.org
2. Popolare gestore di pacchetti per Erlang ed Elixir: http://hex.pm
3. Modelli di Erlang: http://www.erlangpatterns.org/

Versioni

Versione	Note di rilascio	Data di rilascio
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

Versione	Note di rilascio	Data di rilascio
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

Ciao mondo

Ci sono due cose che devi sapere quando scrivi un'applicazione "ciao mondo" in Erlang:

1. Il codice sorgente è scritto nel *linguaggio di programmazione di erlang* usando l'editor di testo di tua scelta
2. L'applicazione viene quindi eseguita nella *macchina virtuale di erlang* . In questo esempio interagiremo con l'erlang VM attraverso la shell di erlang.

Prima il codice sorgente dell'applicazione:

Crea un nuovo file `hello.erl` contenente quanto segue:

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

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

Diamo una rapida occhiata a cosa significa:

- `-module(hello)` . Tutte le funzioni di erlang esistono all'interno di un *modulo* . I moduli vengono quindi utilizzati per creare applicazioni, che sono una raccolta di moduli. Questa prima linea è identificare questo modulo, cioè *ciao* . I moduli possono essere confrontati con i *pacchetti* di Java
- `-export([hello_world/0])` . Indica al compilatore quali funzioni rendere "pubbliche" (rispetto alle lingue OO) e l' *arità* della funzione pertinente. L'arità è il numero di argomenti che la funzione assume. Dato che in erlang una funzione con 1 argomento è vista come una funzione diversa da una con 2 argomenti anche se il nome potrebbe essere esattamente lo stesso. Cioè, `hello_world/0` è una funzione completamente diversa da `hello_world/1` per

esempio.

- `hello_world()` Questo è il nome della funzione. `->` indica la transizione all'attuazione (corpo) della funzione. Questo può essere letto come "hello_world () è definito come ...". Prendi nota che `hello_world()` (nessun argomento) è identificato da `hello_world/0` nella VM e `hello_world(Some_Arg)` come `hello_world/1` .
- `io:format("Hello, World!~n", [])` Dal modulo `io` , viene chiamata la funzione `format/2` function, che è la funzione per l'output standard. `~n` è un identificatore di formato che significa stampare una nuova riga. `[]` È un elenco di variabili da stampare indicato dagli specificatori di formato nella stringa di output, che in questo caso non è nulla.
- Tutte le dichiarazioni di erlang devono terminare con a `.` (punto).

In Erlang, viene restituito il risultato dell'ultima istruzione in una funzione.

Ora, eseguiamo la nostra applicazione:

Avvia la shell di erlang dalla stessa directory del file `hello.erl` file:

```
$ erl
```

Dovresti ricevere una richiesta simile a questa (la tua versione potrebbe essere diversa):

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

Ora inserisci i seguenti comandi:

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

Passiamo attraverso ciascuna riga una ad una:

- `c(hello)` - questo comando chiama la funzione `c` su un atomo `hello` . Questo in effetti dice a Erlang di trovare il file `hello.erl` , compilarlo in un modulo (un file denominato `hello.beam` verrà generato nella directory) e caricarlo nell'ambiente.
- `{ok, hello}` - questo è il risultato di chiamare la funzione `c` sopra. È una tupla contenente un atom `ok` e un atomo `hello` . Le funzioni di Erlang solitamente restituiscono `{ok, Something}` o `{error, Reason}` .
- `hello:hello_world()` - questo chiama una funzione `hello_world()` dal modulo `hello` .
- `Hello, World!` - questo è ciò che la nostra funzione stampa.
- `ok` - questo è ciò che la nostra funzione ha restituito. Dal momento che Erlang è un linguaggio di programmazione funzionale, ogni funzione restituisce *qualcosa* . Nel nostro caso, anche se non abbiamo restituito nulla in `hello_world()` , l'ultima chiamata in quella funzione era in `io:format(...)` e quella funzione ha restituito `ok` , che è a sua volta ciò che la nostra funzione ha restituito.

moduli

Un modulo di erlang è un file con un paio di funzioni raggruppate insieme. Questo file di solito ha estensione `.erl` .

Un modulo "Hello World" con nome `hello.erl` è mostrato sotto

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

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

Nel file, è necessario dichiarare il nome del modulo. Come mostrato prima nella riga 1. Il nome del modulo e il nome del file prima dell'estensione `.erl` devono essere uguali.

Funzione

La funzione è un insieme di istruzioni, che sono raggruppate insieme. Queste istruzioni raggruppate eseguono determinati compiti. In erlang, tutte le funzioni restituiscono un valore quando vengono chiamate.

Di seguito è riportato un esempio di una funzione che aggiunge due numeri

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

Questa funzione esegue un'operazione di aggiunta con i valori X e Y e restituisce il risultato. La funzione può essere utilizzata come di seguito

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

Le dichiarazioni di funzione possono essere costituite da più clausole, separate da un punto e virgola. Gli argomenti in ciascuna di queste clausole sono valutati dalla corrispondenza del modello. La seguente funzione restituirà 'tuple' se l'Argomento è una tupla nel Form: {test, X} dove X può essere qualsiasi valore. Restituirà 'lista', se l'argomento è una lista della lunghezza 2 nella forma ["test", X], e restituirà '{errore,' Motivo '}' in qualsiasi altro caso:

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

Se l'argomento non è una tupla, verrà valutata la seconda clausola. Se l'argomento non è un elenco, verrà valutata la terza clausola.

Le dichiarazioni di funzione possono consistere in cosiddette "guardie" o "sequenze di guardie". Queste guardie sono espressioni che limitano la valutazione di una funzione. Una funzione con le guardie viene eseguita solo quando tutte le espressioni di guardia forniscono un valore reale. Le guardie multiple possono essere separate da un punto e virgola.

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

La funzione 'nome_funzione' sarà valutata solo quando la sequenza di guardia è vera. La funzione follwing restituirà true solo se l'argomento `X` è nell'intervallo corretto (0..15):

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

Comprensione delle liste

La comprensione delle liste è un costrutto sintattico per creare una lista basata su liste esistenti. In erlang una comprensione delle liste ha la forma `[Expr || Qualifier1, ..., QualifierN]` . Dove qualificatori sono o generatori `Pattern <- ListExpr` o filtro come `integer(X)` valuta sia `true` che `false` .

Il seguente esempio mostra una comprensione di lista con un generatore e due filtri.

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

Il risultato è una lista contenente solo numeri interi maggiori di 3.

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

Avvio e arresto della shell di Erlang

Avvio della shell di Erlang

Su un sistema UNIX si avvia la shell di Erlang da un prompt dei comandi con il comando `erl`

Esempio:

```
$ 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>
```

Il testo che mostra quando si avvia la shell fornisce informazioni su quale versione di Erlang è in esecuzione e altre informazioni utili sul sistema di erlang.

Per avviare la shell su Windows fai clic sull'icona di Erlang nel menu Start di Windows.

Fermare la shell di Erlang

Per un'uscita controllata della shell di erlang digiti:

```
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().
```

Puoi anche uscire dalla shell di Erlang premendo Ctrl + C sui sistemi UNIX o Ctrl + Break su Windows, che ti porta al seguente prompt:

```
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
```

Se premi quindi a (per interrompere) uscirai direttamente dalla shell.

Altri modi di uscire dalla shell di erlang sono: `init:stop()` che fa la stessa cosa di `q()` o di `erlang:halt()` .

Pattern Matching

Una delle operazioni più comuni in erlang è la corrispondenza del modello. Viene utilizzato quando si assegna un valore a una variabile, in dichiarazioni di funzione e in strutture di flusso di controllo come `case` e `receive` statements. Un'operazione di corrispondenza del modello richiede almeno 2 parti: un modello e un termine contro il quale il modello è abbinato.

Un'assegnazione variabile in erlang assomiglia a questo:

```
X = 2.
```

Nella maggior parte del linguaggio di programmazione la semantica di questa operazione è semplice: associa un valore (`2`) a un nome di tua scelta (la variabile - in questo caso `X`). Erlang ha un approccio leggermente diverso: abbina il modello sul lato sinistro (`X`) al termine sul lato destro (`2`). In questo caso, l'effetto è lo stesso: la variabile `X` è ora associata al valore `2` . Tuttavia, con la corrispondenza dei modelli è possibile eseguire compiti più strutturati.

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

Questa operazione di corrispondenza viene eseguita analizzando la struttura del termine del lato destro e applicando tutte le variabili sul lato sinistro ai valori appropriati del termine, in modo che il lato sinistro sia uguale al lato destro. In questo esempio, `Type` è associato al termine: `document` , `Meta` a `{author, "Alice"}` e `Doc` a `{text, "Lorem Ipsum"}` . In questo particolare esempio si presume che le variabili: `Type` , `Meta` e `Doc` *non* siano *associate* , in modo che ciascuna variabile possa essere utilizzata.

È inoltre possibile creare abbinamenti di modelli, utilizzando variabili associate.

```
Identifier = error.
```

La variabile `Identifier` è ora associata `error` valore. La seguente operazione di corrispondenza del modello funziona, poiché la struttura corrisponde e la variabile associata `Identifier` ha lo stesso valore della parte destra appropriata del termine.

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

Un'operazione di corrispondenza del modello non riesce, quando c'è una mancata corrispondenza tra il termine del lato destro e il modello del lato sinistro. La seguente corrispondenza avrà esito negativo, poiché l' `Identifier` è associato `error` valore, che non ha un'espressione appropriata sul termine del lato destro.

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

Leggi Iniziare con Erlang Language online: https://riptutorial.com/it/erlang/topic/825/iniziare-con-erlang-language

Capitolo 2: Bit Sintassi: valori predefiniti

introduzione

Il documento Erlang intitolato Bit Syntax ha una sezione chiamata Defaults che contiene un errore e il documento è confuso nel suo insieme. L'ho riscritto.

Examples

Le impostazioni predefinite sono state spiegate

4.4 Predefiniti
...
...
La dimensione predefinita dipende dal tipo. Per numero intero è 8. Per float è 64. Per binario è la dimensione del binario specificato:

```
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
```

In corrispondenza, un segmento binario senza dimensione è consentito solo a t

Leggi Bit Sintassi: valori predefiniti online: https://riptutorial.com/it/erlang/topic/10050/bit-sintassi--valori-predefiniti

Capitolo 3: Bit Sintassi: valori predefiniti

introduzione

Blah blah blah.

Examples

Riscrivi i documenti.

4.4 Predefiniti

[Inizio omesso: << 3.14 >> non è nemmeno una sintassi legale.]

La dimensione predefinita dipende dal tipo. Per numero intero è 8. Per float è 64. Per binario è la dimensione effettiva del binario specificato:

```
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
```

Nell'abbinamento, un segmento binario senza dimensione è consentito solo alla fine del pattern e la dimensione predefinita è il resto del file binario sul lato destro della corrispondenza:

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

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

27> X.
97

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

Tutti gli altri segmenti con tipo binario in un modello devono specificare una dimensione:

```
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
```

Leggi Bit Sintassi: valori predefiniti online: https://riptutorial.com/it/erlang/topic/10051/bit-sintassi--valori-predefiniti

Capitolo 4: comportamenti

Examples

Usando un comportamento

Aggiungi una direttiva `-behaviour` al tuo modulo per indicare che segue un comportamento:

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

L'ortografia americana è anche accettata:

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

Ora il compilatore ti avviserà se hai dimenticato di implementare ed esportare le funzioni richieste dal comportamento, ad esempio:

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

Definire un comportamento

Puoi definire il tuo comportamento aggiungendo `-callback` direttive `-callback` nel tuo modulo. Ad esempio, se i moduli che implementano il tuo comportamento devono avere una funzione `foo` che prende un intero e restituisce un atomo:

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

Se si utilizza questo comportamento in un altro modulo, il compilatore avviserà se non esporta `foo/1` e Dialyzer avviserà se i tipi non sono corretti. Con questo modulo:

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

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

e in esecuzione `dialyzer --src bar.erl my_behaviour.erl` , ottieni questi avvertimenti:

```
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
```

Richiami opzionali in un comportamento personalizzato

18,0

Per impostazione predefinita, qualsiasi funzione specificata in una direttiva `-callback` in un modulo comportamentale deve essere esportata da un modulo che implementa tale comportamento. Altrimenti, riceverai un avvertimento sul compilatore.

A volte, si desidera che una funzione di callback sia facoltativa: il comportamento lo userebbe se presente ed esportato, e in caso contrario ricadrebbe su un'implementazione predefinita. Per fare ciò, scrivere la direttiva `-callback` come al solito, quindi elencare la funzione di callback in una direttiva `-optional_callbacks` :

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

Se il modulo esporta la `bar/0` , Dialyzer controllerà comunque le specifiche del tipo, ma se la funzione è assente, non verrà visualizzato un avviso del compilatore.

In Erlang / OTP sé, questo è fatto per il `format_status` funzione di richiamata nei `gen_server` , `gen_fsm` e `gen_event` comportamenti.

Leggi comportamenti online: https://riptutorial.com/it/erlang/topic/7004/comportamenti

Capitolo 5: comportamento di gen_server

Osservazioni

`gen_server` è una caratteristica importante di Erlang e richiede alcuni prerequisiti per comprendere ogni aspetto di questa funzionalità:

- Loop, ricorsione e stato
- Processi di deposizione delle uova
- Messaggio che passa
- Principi OTP

Un buon modo per saperne di più su una funzione in Erlang è leggere direttamente il codice sorgente dal repository github ufficiale . `gen_server` comportamento di `gen_server` molte informazioni utili e una struttura interessante nel suo nucleo.

`gen_server` è definito in `gen_server.erl` e la sua documentazione associata può essere trovata nella documentazione di `stdlib` Erlang . `gen_server` è una funzionalità OTP e ulteriori informazioni possono essere trovate anche in OTP Design Principles e nella Guida per l'utente .

Frank Hebert ti offre anche un'altra buona introduzione a `gen_server` dal suo libro online gratuito Learn You Some Erlang per il grande bene!

Documentazione ufficiale per il callback `gen_server` :

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

Examples

Servizio di accoglienza

Ecco un esempio di un servizio che saluta le persone con il nome specificato e tiene traccia del numero di utenti che ha incontrato. Vedi l'utilizzo di seguito.

```
%% 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}.
```

Ecco un esempio di utilizzo di questo servizio nella shell di 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}
```

Utilizzando il comportamento di gen_server

Un `gen_server` è una macchina a stati finiti specifica che funziona come un server. `gen_server` può gestire diversi tipi di eventi:

- richiesta sincrona con `handle_call`
- richiesta asincrona con `handle_cast`
- altro messaggio (non definito nelle specifiche OTP) con `handle_info`

I messaggi sincroni e asincroni sono specificati in OTP e sono tuple con tag semplici con qualsiasi tipo di dati su di esso.

Un semplice `gen_server` è definito in questo modo:

```
-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.
```

Questo codice è semplice: ogni messaggio ricevuto viene stampato sullo standard output.

comportamento di gen_server

Per definire un `gen_server` , è necessario dichiararlo esplicitamente nel codice sorgente con `-behaviour(gen_server)` . Nota, il `behaviour` può essere scritto negli USA (comportamento) o nel Regno Unito (comportamento).

start_link / 0

Questa funzione è una semplice scorciatoia per chiamare un'altra funzione: `gen_server:start_link/3,4` .

start_link / 3,4

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

Questa funzione viene chiamata quando si desidera avviare il server collegato a un `supervisor` oa un altro processo. `start_link/3,4` può registrare automaticamente il tuo processo (se ritieni che il tuo processo debba essere unico) o può semplicemente generare come un semplice processo. Quando chiamato, questa funzione esegue `init/1` .

Questa funzione può restituire questi valori di definizione:

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

init / 1

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

`init/1` è la prima funzione eseguita all'avvio del server. Questo inizializza tutti i prerequisiti dell'applicazione e restituisce lo stato al processo appena creato.

Questa funzione può restituire solo questi valori definiti:

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

`State` variabile di `State` può essere tutto (ad es. Lista, tupla, proplists, map, record) e rimanere accessibile a tutte le funzioni all'interno del processo generato.

handle_call / 3

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

`gen_server:call/2` esegue questo callback. Il primo argomento è il tuo messaggio (`_Request`), il secondo è l'origine della richiesta (`_From`) e l'ultimo è lo stato corrente (`State`) del tuo comportamento in esecuzione gen_server.

Se vuoi una risposta al chiamante, `handle_call/3` deve restituire una di queste strutture dati:

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

Se non si desidera rispondere al chiamante, `handle_call/3` deve restituire una di queste strutture dati:

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

Se vuoi interrompere l'attuale esecuzione del tuo attuale gen_server, `handle_call/3` deve restituire una di queste strutture dati:

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

handle_cast / 2

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

`gen_server:cast/2` esegue questo callback. Il primo argomento è il tuo messaggio (`_Msg`), e il secondo lo stato corrente del tuo comportamento in esecuzione gen_server.

Per impostazione predefinita, questa funzione non può fornire dati al chiamante, quindi, hai solo due scelte, continua l'esecuzione corrente:

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

Oppure interrompi la tua attuale procedura `gen_server` :

- `{stop,Reason,NewState}`

handle_info / 2

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

`handle_info/2` viene eseguito quando il messaggio OTP non standard proviene da un mondo esterno. Questo non può rispondere e, come `handle_cast/2` può fare solo 2 azioni, continuando l'esecuzione corrente:

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

Oppure interrompi il processo `gen_server` corrente:

- `{stop,Reason,NewState}`

interrompere / 2

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

`terminate/2` viene chiamato quando si verifica un errore o quando si desidera `gen_server` processo `gen_server` .

code_change / 3

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

`code_change/3` funzione `code_change/3` viene chiamata quando si desidera aggiornare il codice in esecuzione.

Questa funzione può restituire solo questi valori definiti:

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

Avvio di questo processo

Puoi compilare il tuo codice e avviare `simple_gen_server` :

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

Se si desidera inviare un messaggio al server, è possibile utilizzare queste funzioni:

```
% 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).
```

Semplice database di chiavi / valori

Questo codice sorgente crea un semplice servizio di archiviazione chiavi / valore basato sulla struttura di dati di Erlang della `map` . In primo luogo, dobbiamo definire tutte le informazioni riguardanti il nostro `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 il nostro server cache

Ora possiamo compilare il nostro codice e iniziare 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().
```

Leggi comportamento di gen_server online:
https://riptutorial.com/it/erlang/topic/7481/comportamento-di-gen-server

Capitolo 6: direttore

introduzione

Libreria di supervisione flessibile, veloce e potente per i processi di Erlang.

Osservazioni

Avvertenze

- Non utilizzare `'count'=>infinity` e element `restart` nel tuo piano.
  piace:

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

Se il processo non è iniziato dopo l'arresto, **Director** bloccherà e riproverà per riavviare i tempi `infinity` processo! Se stai usando `infinity` per `'count'` , usa sempre `{restart, MiliSeconds}` in `'plan'` invece di `restart` .

- Se hai piani come:

```
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}.
```

Il resto dell'elemento `delete` in `Childspec1` e il resto dell'elemento `stop` in `Childspec2` non saranno mai valutati!
In `Childspec3` vuoi eseguire il tuo piano 0 volte!
In `ChildSpec4` non hai piani per eseguire `infinity` volte!

- Quando si aggiorna una versione utilizzando `release_handler` , `release_handler` chiama `supervisor:get_callback_module/1` per il recupero del suo modulo di callback.
  In OTP <19 `get_callback_module/1` usa il record dello stato interno del supervisore per fornire il suo modulo di callback. Il nostro **direttore** non conosce il record dello stato interno del supervisore, quindi il `supervisor:get_callback_module/1` non funziona con i **director** s.
  La buona notizia è che in OTP> = 19 `supervisor:get_callback_module/1` funziona perfettamente con **director** s :).

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

2> supervisor:get_callback_module(foo_sup).
foo

3>
```

Examples

Scaricare

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

Compilare

Nota che **OTP> = 19** richiesto (se vuoi aggiornarlo usando `release_handler`).
Vai a `director` e utilizza `rebar` o `rebar3` .

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

tondo per cemento armato

```
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 $
```

Come funziona

direttore ha bisogno di un modulo di callback (come supervisore OTP).
Nel modulo di callback è necessario esportare la funzione `init/1` .
Quale `init/1` dovrebbe tornare? aspetta, ti spiego passo dopo passo.

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

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

Salva il codice sopra in `foo.erl` nella **directory** director e vai alla shell di Erlang.
Usa `erl -pa ./ebin` se hai usato il `rebar` per compilarlo e usa la `rebar3 shell` se hai usato `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>
```

Ora abbiamo un supervisore senza figli.
La buona notizia è che il **regista è** dotato di API OTP / supervisore completo e ha anche le sue funzionalità avanzate e un approccio specifico.

```
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, renderò semplice `gen_server` e lo darò al nostro **direttore** .

```
-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.
```

Salva il codice sopra in `bar.erl` e torna alla 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>
```

Consente di controllarlo

```
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
```

Ho menzionato **le funzionalità avanzate** , quali sono? Vediamo altre chiavi accettabili per la mappa di `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().
```

Modifica il modulo `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]}.
```

Vai di nuovo alla shell di Erlang:

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

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

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

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

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

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

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

%% I can change process plan
%% I killed process one time.
%% If i kill it again, entire supervisor will crash with reason {reached_max_restart_plan...
because 'count' is 1
%% But after changing plan, its counter will restart from 0.
8> director:change_plan(foo_sup, bar_id, [restart]).
ok

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

```
10> director:get_pids(foo_sup).
[]

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

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

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

14> %% Hold on
```

Finalmente cos'è la chiave `append` ?

in realtà abbiamo sempre uno `DefaultChildspec` .

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

15>
```

`DefaultChildspec` è come una normale spettrografia, tranne per il fatto che non può accettare `id` e `append` chiavi.
Se cambio il valore di `append` a `true` in `Childspec` :
Il mio `terminate_timeout` verrà aggiunto a `terminate_timeout` di `DefaultChildspec` .
Il mio `count` verrà aggiunto al `count` di `DefaultChildspec` .
I miei `modules` verranno aggiunti ai `modules` di `DefaultChildspec` .
Il mio `plan` verrà aggiunto al `plan` di `DefaultChildspec` .
E se ho la chiave di `start` con valore `{ModX, FuncX, ArgsX}` in `DefaultChildspec` e la chiave di `start` con valore `{ModY, FunY, ArgsY}` in `Childspec` , il valore finale sarà `{ModY, FuncY, ArgsX ++ ArgsY}` .
E infine se ho la chiave di `start` con valore `{Mod, Func, Args}` in `DefaultChildspec` , la chiave di `start` in `Childspec` è facoltativa per me.
Puoi restituire il tuo `DefaultChildspec` come terzo elemento di tupla in `init/1` .
Modifica `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}.
```

Riavvia la 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>
```

Puoi cambiare `defaultChildspec` modo dinamico usando `change_default_childspec/2` !
Ed è possibile modificare anche `Childspec` dei bambini in modo dinamico e impostare la loro `append` su `true` !
Ma cambiandoli in diverse parti del codice, farai il **codice spaghetti**

Posso eseguire il debug di director?

Yessssss, **diorector** ha il proprio debug e accetta `sys:dbg_opt/0` standard `sys:dbg_opt/0` .
director invia registri validi a `sasl` e `error_logger` in stati diversi.

```
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>
```

Genera documentazione API

tondo per cemento armato:

```
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()."
```

Dopo aver eseguito uno dei comandi sopra, la documentazione HTML dovrebbe essere nella directory `doc` .

Leggi direttore online: https://riptutorial.com/it/erlang/topic/9878/direttore

Capitolo 7: File I / O

Examples

Lettura da un file

Supponiamo che tu abbia un file **text.txt** che contiene i seguenti dati:

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

Leggi l'intero file alla volta

Usando il `file:read_file(File)` , puoi leggere l'intero file. È un'operazione atomica:

```
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">>}
```

Leggi una riga alla volta

`io:get_line` legge il testo fino alla fine riga o alla fine del file.

```
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
```

Leggi con l'accesso casuale

`file:pread(IoDevice, Start, Len)` legge da `Start` tanto quanto `Len` da `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"}
```

Scrivere su un file

Scrivi una riga alla volta

Apri un file con la modalità di `write` e usa `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>
```

Il risultato sarà un file denominato **fruit_count.txt** con i seguenti contenuti:

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

Si noti che l'apertura di un file in modalità di scrittura verrà creata, se non già esistente nel file system.

Nota anche che usando l'opzione di `write` con `file:open/2` **troncherà** il file (anche se non scrivi nulla in esso). Per evitare ciò, aprire il file in modalità `[read,write]` o `[append]` .

Scrivi l'intero file in una sola volta

`file:write_file(Filename, IO)` è la funzione più semplice per scrivere un file in una volta. Se il file esiste già, verrà sovrascritto, altrimenti verrà creato.

```
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>
```

Scrivi con accesso casuale

Per la scrittura ad accesso casuale, viene utilizzato il `file:pwrite(IoDevice, Location, Bytes)` . Se si desidera sostituire una stringa nel file, questo metodo è utile.

Supponiamo che tu voglia cambiare "Olive 12" in "Apple 15" nel file creato sopra.

```
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>
```

Leggi File I / O online: https://riptutorial.com/it/erlang/topic/5232/file-i---o

Capitolo 8: Formato stringhe

Sintassi

- io: format (FormatString, Args)% scrive su standard output
- io: format (standard_error, FormatString, Args)% scrive su errore standard
- io: format (F, FormatString, Args)% write per aprire il file
- io_lib: format (FormatString, Args)% restituisce un iolist

Examples

Sequenze di controllo comuni nelle stringhe di formato

Mentre ci sono molte diverse sequenze di controllo per `io:format` e `io_lib:format` , la maggior parte delle volte ne userai solo tre differenti: `~s` , `~p` e `~w` .

~ s

Il `~s` è per *archi* .

Stampa stringhe, binari e atomi. (Qualsiasi altra cosa causerà un errore di `badarg`). Non cita né sfugge a nulla; stampa solo la stringa stessa:

```
%% 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

Il `~w` è per la *scrittura con sintassi standard* .

Può stampare qualsiasi termine di Erlang. L'output può essere analizzato per restituire il termine originale di Erlang, a meno che non contenga termini che non hanno una rappresentazione scritta parsabile, cioè pid, porte e riferimenti. Non inserisce alcuna nuova riga o rientro e le stringhe sono sempre interpretate come liste:

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

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

~ p

Il `~p` è per la *stampa carina* .

Può stampare qualsiasi termine di Erlang. L'output differisce da `~w` nei seguenti modi:

- I newline sono inseriti se la linea sarebbe altrimenti troppo lunga.
- Quando vengono inserite nuove righe, la riga successiva viene rientrata per allineare con un termine precedente sullo stesso livello.
- Se un elenco di numeri interi assomiglia a una stringa stampabile, viene interpretato come uno.

```
> 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}
```

Se non vuoi che liste di numeri interi vengano stampate come stringhe, puoi usare la sequenza `~lp` (inserisci una lettera minuscola L prima di `p`):

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

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

Leggi Formato stringhe online: https://riptutorial.com/it/erlang/topic/3722/formato-stringhe

Capitolo 9: Formato termine esterno

introduzione

External Term Format è un formato binario utilizzato per comunicare al mondo esterno. Puoi usarlo con qualsiasi lingua tramite porte, driver o NIF. BERT (binario ERlang Term) può essere utilizzato in altre lingue.

Examples

Usare ETF con Erlang

```
binary_to_term().
```

Usare ETF con C

Inizializzazione della struttura dei dati

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

Numero di codifica

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

int
main() {
}
```

Atomo di codifica

Tupla di codifica

Lista di codifica

Mappa di codifica

Leggi Formato termine esterno online: https://riptutorial.com/it/erlang/topic/10731/formato-termine-esterno

Capitolo 10: Installazione

Examples

Costruisci e installa Erlang / OTP su Ubuntu

I seguenti esempi mostrano due metodi principali per l'installazione di Erlang / OTP su Ubuntu.

Metodo 1 - Pacchetto binario precostruito

Basta eseguire questo comando e scaricherà e installerà l'ultima versione stabile di Erlang da Erlang Solutions .

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

Metodo 2: compilazione e installazione dalla sorgente

Scarica il file tar:

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

Estrai il file tar:

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

Immettere la directory estratta e impostare `ERL_TOP` come il percorso corrente:

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

Ora prima di configurare la build, assicurati di avere tutte le dipendenze necessarie per installare Erlang:

Dipendenze di base:

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

Altre dipendenze delle applicazioni

Applicazione	Installazione delle dipendenze
HIPE	`$ 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`
Documentazione	`$ sudo apt-get install fop xsltproc`
Orber e altri progetti C ++	`$ sudo apt-get install g++`
jinterface	`$ sudo apt-get install default-jdk`

Configura e crea:
È possibile impostare le proprie opzioni o lasciarlo vuoto per eseguire la configurazione predefinita. Configurazione avanzata e build per Erlang / OTP .

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

Test della build:

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

Dopo aver eseguito questi comandi, apri `$ERL_TOP/release/tests/test_server/index.html` con il tuo browser web e verifica che non ci siano `$ERL_TOP/release/tests/test_server/index.html` . Se tutti i test sono passati, possiamo continuare con l'installazione.

Installazione:

```
$ make install
```

Costruisci e installa Erlang / OTP su FreeBSD

I seguenti esempi mostrano 3 metodi principali per l'installazione di Erlang / OTP su FreeBSD.

Metodo 1 - Pacchetto binario precostruito

Usa pkg per installare il pacchetto binario pre-costruito:

```
$ pkg install erlang
```

Metti alla prova la tua nuova installazione:

```
$ 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)
```

Metodo 2 - Costruire e installare usando la collezione di porte (raccomandata)

Costruisci e installa la porta come al solito:

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

Metti alla prova la tua nuova installazione:

```
$ 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)
```

Questo preleverà il rilascio tarball dal sito web ufficiale, applicherà alcune patch se necessario, compilerà il rilascio e lo installerà. Ovviamente, ci vorrà del tempo.

Metodo 3: crea e installa dal tarball di rilascio

Nota: la creazione della versione manuale funziona, ma è preferibile utilizzare i due metodi precedenti, poiché la collezione di porte incorpora le patch che rendono la versione più compatibile con FreeBSD.

Scarica il file di rilascio:

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

Verifica che la sua somma MD5 sia corretta:

```
$ 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
```

Estrai il tarball:

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

Configurazione:

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

Se vuoi costruire Erlang con HiPe, dovrai applicare le patch dalla collezione di porte.

Costruire:

```
$ gmake
```

Installare:

```
$ gmake install
```

Metti alla prova la tua nuova installazione:

```
$ 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)
```

Costruisci e installa usando kerl

kerl è uno strumento che ti aiuta a costruire e installare le versioni di Erlang / OTP.

Installa arricciatura:

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

Scarica kerl:

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

Aggiorna l'elenco delle versioni disponibili:

```
$ ./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
```

Costruisci la versione richiesta:

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

Verifica che la build sia presente nella lista di costruzione:

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

Installa la build da qualche parte:

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

Sorga il file di `activate` se stai eseguendo bash o il guscio di pesce. Se stai usando una cshell, aggiungi la directory bin del build al PATH:

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

Metti alla prova la tua nuova installazione:

```
$ 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)
```

Altre versioni

Se vuoi creare un'altra versione di Erlang / OTP, cerca le altre porte nella raccolta:

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

Riferimento

- Manuale di FreeBSD -> Capitolo 4. Installazione delle applicazioni: pacchetti e porte
- Erlang su FreshPorts
- Documentazione Kerl su GitHub

Costruisci e installa Erlang / OTP su OpenBSD

Erlang su OpenBSD è attualmente rotto su architetture `alpha` , `sparc` e `hppa` .

Metodo 1 - Pacchetto binario precostruito

OpenBSD ti consente di scegliere la versione desiderata che vuoi installare sul tuo 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 può supportare più versioni di Erlang. Per rendere i pensieri più facili da usare, ogni file binario è installato nella versione di Erlang nel suo nome. Quindi, se hai installato `erlang-19.0v0` , il tuo binario `erl` sarà `erl19` .

Se si desidera utilizzare `erl` , è possibile creare un collegamento simbolico:

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

oppure creare un alias nel file di configurazione della shell o nel file `.profile` :

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

Ora puoi eseguire `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)
```

Metodo 2: creare e installare utilizzando le porte

```
 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
```

Metodo 3 - Costruisci dalla fonte

Build from source richiede pacchetti aggiuntivi:

- `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
```

Riferimenti

- 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

Leggi Installazione online: https://riptutorial.com/it/erlang/topic/4483/installazione

Capitolo 11: iolists

introduzione

Mentre una stringa di Erlang è una lista di numeri interi, un "iolist" è una lista i cui elementi sono numeri interi, binari o altri elementi iolistici, ad esempio `["foo", $b, $a, $r, <<"baz">>]` . Quel iolist rappresenta la stringa `"foobarbaz"` .

Mentre è possibile convertire un iolist in un file binario con `iolist_to_binary/1` , spesso non è necessario, poiché le funzioni della libreria di Erlang come `file:write_file/2` e `gen_tcp:send/2` accettano iolist, nonché stringhe e binari.

Sintassi

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

Osservazioni

Cos'è un iolist?

È un qualsiasi binario. O qualsiasi elenco contenente numeri interi compresi tra 0 e 255. O qualsiasi elenco arbitrariamente annidato contenente una di queste due cose.

Articolo originale

Utilizzare elenchi di interi e binari profondamente annidati per rappresentare i dati I / O per evitare la copia durante la concatenazione di stringhe o binari.

Sono efficienti anche quando si combinano grandi quantità di dati. Ad esempio, combinando due binari cinquanta kilobyte usando la sintassi binaria `<<B1/binary, B2/binary>>` genere è necessario riallocarli entrambi in un nuovo binario a 100 kb. Utilizzando gli elenchi di I / O `[B1, B2]` si assegna solo l'elenco, in questo caso tre parole. Una lista usa una parola e un'altra parola per elemento, vedi qui per maggiori informazioni.

L'utilizzo dell'operatore `++` avrebbe creato un elenco completamente nuovo, anziché solo un nuovo elenco di due elementi. Ricreare gli elenchi per aggiungere elementi alla fine può diventare costoso quando l'elenco è lungo.

Nei casi in cui i dati binari sono piccoli, l'assegnazione degli elenchi di I / O può essere maggiore rispetto all'aggiunta dei file binari. Se i dati binari possono essere piccoli o grandi, spesso è meglio accettare il costo coerente degli elenchi di I / O.

Nota che i binari di accodamento sono ottimizzati come descritto qui . In breve, un file binario può disporre di uno spazio nascosto aggiuntivo. Questo sarà riempito se un altro binario viene aggiunto ad esso che si adatta allo spazio libero. Ciò significa che non tutte le append binarie causeranno una copia completa di entrambi i binari.

Examples

Gli elenchi di I / O vengono in genere utilizzati per creare output su una porta, ad esempio un file o un socket di rete.

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

Aggiungi i tipi di dati consentiti all'inizio di un elenco IO, creando uno nuovo.

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

I dati I / O possono essere aggiunti in modo efficiente alla fine di un elenco.

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

Fai attenzione agli elenchi impropri

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

Nella shell questo verrà stampato come `["Guten tag "|<<"Hello">>]` invece di `["Guten tag ",<<"Hello">>]` . L'operatore di pipe creerà un elenco errato se l'ultimo elemento a destra non è un elenco. Mentre un elenco improprio la cui "coda" è un file binario è ancora un iolist valido, elenchi errati possono causare problemi perché molte funzioni ricorsive si aspettano che una lista vuota sia l'ultimo elemento e non, come in questo caso un binario.

Ottieni dimensioni elenco IO

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

La dimensione di un iolist può essere calcolata usando `iolist_size/1` . Questo snippet calcola la dimensione di un messaggio e lo crea e lo aggiunge in primo piano come un binario a quattro byte. Questa è una tipica operazione nei protocolli di messaggistica.

Lista I / O può essere convertita in un file binario

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

Un elenco IO può essere convertito in un binario usando la funzione `iolist_to_binary/1` . Se i dati verranno archiviati per un lungo periodo o inviati come un messaggio ad altri processi, allora

potrebbe avere senso convertirli in un binario. Il costo di una tantum di conversione in un binario può essere più economico rispetto alla copia dell'elenco di IO più volte, nella raccolta di dati inutili di un singolo processo o nel passaggio di messaggi ad altri.

Leggi iolists online: https://riptutorial.com/it/erlang/topic/5677/iolists

Capitolo 12: Le autorità di vigilanza

Examples

Supervisore di base con un processo di lavoro

Questo esempio utilizza il formato di mappa introdotto in 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}}.
```

Leggi Le autorità di vigilanza online: https://riptutorial.com/it/erlang/topic/6996/le-autorita-di-vigilanza

Capitolo 13: Loop e ricorsione

Sintassi

- funzione (lista | iolist | tupla) -> funzione (coda).

Osservazioni

Perché le funzioni ricorsive?

Erlang è un linguaggio di programmazione funzionale e non ha alcun tipo di struttura di loop. Tutto nella programmazione funzionale si basa su dati, tipi e funzioni. Se vuoi un ciclo, devi creare una funzione che si chiama da sola.

Il tradizionale `while` o `for` loop in linguaggio imperativo e orientato agli oggetti può essere rappresentato in questo modo in Erlang:

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

Un buon metodo per capire questo concetto è quello di espandere tutte le chiamate di funzione. Lo vedremo su altri esempi.

Examples

Elenco

Qui la funzione ricorsiva più semplice rispetto al tipo di lista . Questa funzione naviga solo in una lista dall'inizio alla fine e non fa altro.

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

Puoi chiamarlo così:

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

Qui l'espansione della funzione ricorsiva:

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

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

Ciclo ricorsivo con azioni IO

Il codice precedente non fa nulla ed è piuttosto inutile. Quindi, creeremo ora una funzione ricorsiva che esegue alcune azioni. Questo codice è simile agli `lists:foreach/2` .

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

Puoi chiamarlo in questo modo:

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

Qui l'espansione della funzione ricorsiva:

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

Puoi confrontare con gli `lists:foreach/2` output:

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

Ciclo ricorsivo su lista che restituisce lista modificata

Un altro esempio utile, simile agli `lists:map/2` . Questa funzione richiederà una lista e una funzione anonima. Ogni volta che viene abbinato un valore nell'elenco, applichiamo la funzione su di esso.

```
-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).
```

Puoi chiamarlo così:

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

Qui l'espansione della funzione ricorsiva:

```
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].
```

Questa funzione è anche chiamata "funzione ricorsiva di coda", perché usiamo una variabile come un accumulatore per passare dati modificati su più contesti di esecuzione.

Iolist e Bitstring

Come lista, la funzione più semplice su iolist e bitstring è:

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

Puoi chiamarlo così:

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

Qui l'espansione della funzione ricorsiva:

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

Funzione ricorsiva su dimensione binaria

variabile

Questo codice prende bitstring e ne definisce dinamicamente la dimensione binaria. Quindi, se impostiamo una dimensione di `4` , ogni `4` bit, i dati verranno confrontati. Questo loop non fa nulla di interessante, è solo il nostro pilastro.

```
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.
```

Puoi chiamarlo così:

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

Qui l'espansione della funzione ricorsiva:

```
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.
```

Il nostro bitstring è suddiviso su 7 pattern. Perché? Perché per impostazione predefinita, Erlang usa una dimensione binaria di `8` bit, se lo dividiamo in due, abbiamo `4` bit. La nostra stringa è `8*3=24` bit. `24/4=6` modelli. L'ultimo modello è `<<>>` . `loop/2` funzione `loop/2` è chiamata 7 volte.

funzione ricorsiva su dimensione binaria variabile con azioni

Ora possiamo fare qualcosa di più interessante. Questa funzione richiede un altro argomento, una funzione anonima. Ogni volta che abbiniamo un modello, questo verrà passato ad esso.

```
-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.
```

Puoi chiamarlo così:

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

Qui l'espansione della funzione ricorsiva:

```
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.
```

Funzione ricorsiva su bitstring che restituisce bitstring modificato

Questo è simile alle `lists:map/2` ma per 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.
```

Questo codice sembra più complesso. Sono state aggiunte due funzioni: `loop/2` e `loop/3` . Queste due funzioni sono semplici da interfaccia a `loop/4` .

Puoi eseguirlo in questo modo:

```
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>>
```

Carta geografica

La mappa in Erlang è l'equivalente di hash in Perl o dizionari in Python, è un archivio di chiavi / valori. Per elencare ogni valore memorizzato in, è possibile elencare ogni chiave e restituire coppia chiave / valore. Questo primo ciclo ti dà un'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).
```

Puoi eseguirlo in questo modo:

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

Stato di gestione

La funzione ricorsiva usa i loro stati per fare il ciclo. Quando si genera un nuovo processo, questo processo sarà semplicemente un ciclo con uno stato definito.

Funzione anonima

Qui 2 esempi di funzioni anonime ricorsive basate sull'esempio precedente. In primo luogo, semplice ciclo infinito:

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

In secondo luogo, la funzione anonima esegue una lista di loop over:

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

Queste due funzioni potrebbero essere riscritte come:

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

In questo caso, il `loop/0` è un riferimento al `loop/0` dalle osservazioni. In secondo luogo, con poco più complesso:

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

Qui, `loop/2` è un riferimento al `loop/2` dell'esempio di lista. Queste due notazioni sono zucchero sintattico.

Leggi Loop e ricorsione online: https://riptutorial.com/it/erlang/topic/10720/loop-e-ricorsione

Capitolo 14: NIF

Examples

Definizione

Documentazione ufficiale: http://erlang.org/doc/tutorial/nif.html

I NIF sono stati introdotti in Erlang / OTP R13B03 come funzione sperimentale. Lo scopo è quello di consentire di chiamare il codice C all'interno del codice Erlang.

I NIF sono implementati in C anziché in Erlang, ma appaiono come qualsiasi altra funzione nell'ambito del codice Erlang in quanto appartengono al modulo in cui è avvenuta l'inclusione. Le librerie NIF sono collegate alla compilazione e caricate in runtime.

Poiché le librerie NIF sono collegate dinamicamente al processo di emulazione, sono veloci, ma anche pericolose, perché l'arresto anomalo in un NIF porta anche l'emulatore verso il basso.

Esempio: tempo UNIX corrente

Ecco un esempio molto semplice per illustrare come scrivere un NIF.

Struttura della directory:

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

Questa struttura può essere facilmente inizializzata usando Rebar3:

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

Contenuto di `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);
```

Contenuto di `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).
```

Contenuto di `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"}
]}.
```

Ora puoi eseguire l'esempio:

```
$ 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)

Documentazione ufficiale : http://erlang.org/doc/man/erl_nif.html

Le strutture, i tipi e le macro più importanti dell'AP Erlang C sono i seguenti:

- `ERL_NIF_TERM` : il tipo per i termini di Erlang. Questo è il tipo di ritorno che le funzioni NIF devono seguire.
- `ERL_NIF_INIT(MODULE, ErlNifFunc funcs[], load, reload, upgrade, unload)` : questa è la macro che crea effettivamente i NIF definiti in un determinato file C. Deve essere valutato nell'ambito globale. Normalmente sarà l'ultima riga nel file C.
- `ErlNifFunc` : il tipo con cui ogni NIF viene passato a `ERL_NIF_INIT` per essere esportato. Questa struct è composta da nome, arity, un poiter alla funzione C e flag. Un array di questo tipo con tutte le definizioni NIF deve essere creato per essere passato a `ERL_NIF_INIT` .
- `ErlNifEnv` : l'ambiente Erlang in cui viene eseguito il NIF. È obbligatorio passare l'ambiente come primo argomento per ogni NIF. Questo tipo è opaco e può essere manipolato solo utilizzando le funzioni che l'API di Erlang C offre.

Leggi NIF online: https://riptutorial.com/it/erlang/topic/5274/nif

Capitolo 15: Processi

Examples

Creazione di processi

Creiamo un nuovo processo concorrente chiamando la funzione `spawn` . La funzione di `spawn` avrà come parametro una funzione `Fun` che il processo valuterà. Il valore di ritorno della funzione `spawn` è l'identificatore del processo creato (pid).

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

Puoi anche usare `spawn/3` per avviare un processo che eseguirà una funzione specifica da un modulo: `spawn(Module, Function, Args)` .
Oppure usa `spawn/2` o `spawn/4` modo simile per avviare un processo in un nodo diverso: `spawn(Node, Fun)` o `spawn(Node, Module, Function, Args)` .

Messaggio in corso

Due processi di erlang possono comunicare tra loro, noto anche come *passaggio di messaggi* . Questa procedura è *asincrona* nella forma in cui il processo di invio non si interrompe dopo l'invio del messaggio.

Invio di messaggi

Questo può essere ottenuto con il costrutto `Pid ! Message` , dove `Pid` è un identificatore di processo valido (pid) e `Message` è un valore di qualsiasi tipo di dati.

Ogni processo ha una "casella di posta" che contiene i messaggi ricevuti nell'ordine ricevuto. Questa "casella di posta" può essere svuotata con la funzione `flush/0` .

Se un messaggio viene inviato a un processo non esistente, il messaggio verrà eliminato senza errori!

Un esempio potrebbe essere simile al seguente, dove `self/0` restituisce il pid del processo corrente e `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
```

È anche possibile inviare un messaggio a più processi contemporaneamente, con `Pid3!Pid2!Pid1!Msg` .

Ricevere messaggi

I messaggi ricevuti possono essere elaborati con il costrutto di `receive` .

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

Il `Pattern` verrà confrontato con i messaggi nella "mailbox" a partire dal primo e dal messaggio meno recente.
Se un modello corrisponde, il messaggio corrispondente viene rimosso dalla "cassetta postale" e il corpo della clausola viene valutato.

È anche possibile definire i timeout con il `after` costrutto.
Un `Timeout` è il tempo di attesa in millisecondi o l' `infinity` dell'atomo.

Il valore di ritorno di `receive` è l'ultimo corpo della clausola valutata.

Esempio (contatore)

Un contatore (molto) semplice con il passaggio del messaggio potrebbe essere simile a quanto segue.

```
-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.
```

Interazione con il contatore.

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

Registra i processi

È possibile registrare un processo (pid) in un alias globale.
Questo può essere ottenuto con la funzione build in `register(Alias, Pid)` , dove `Alias` è l'atomo per accedere al processo e `Pid` è l'id del processo.

L'alias sarà disponibile a livello globale!
È molto facile creare uno stato condiviso, che di solito non è preferibile. (Vedi anche qui)

È possibile `whereis(Alias)` registrazione di un processo con un `whereis(Alias) unregister(Pid)` e ricevere il pid da un alias con `whereis(Alias)` .

Utilizzare `registered()` per un elenco di tutti gli alias registrati.

L'esempio registra Atom foo nel pid del processo corrente e invia un messaggio utilizzando l'Atom registrato.

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

Leggi Processi online: https://riptutorial.com/it/erlang/topic/2285/processi

Capitolo 16: Rebar3

Examples

Definizione

Pagina ufficiale : https://www.rebar3.org/

Codice sorgente : https://github.com/erlang/rebar3

Rebar3 è principalmente un gestore delle dipendenze per i progetti Erlang ed Elixir, ma offre anche molte altre funzionalità, come i progetti di bootstrap (secondo diversi modelli, seguendo i principi OTP), task executor, tool di costruzione, test runner ed è estensibile usando i plugin.

Installare Rebar3

Rebar3 è scritto in Erlang, quindi hai bisogno di Erlang per eseguirlo. È disponibile come file binario che puoi scaricare ed eseguire. Basta recuperare la build notturna e dargli i permessi di esecuzione:

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

Metti questo file binario in un posto conveniente e aggiungilo al tuo percorso. Ad esempio, in una directory bin nella tua casa:

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

Quest'ultima riga deve essere inserita nel tuo `.bashrc` . In alternativa, si può anche collegare il binario alla `/usr/local/bin` , rendendolo disponibile come un normale comando.

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

Installazione dal codice sorgente

Poiché Rebar3 è gratuito, open source e scritto in Erlang, è possibile clonarlo e crearlo semplicemente dal codice sorgente.

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

Questo creerà lo script rebar3, che potrai mettere sul tuo PATH o collegarti a `/usr/local/bin` come spiegato nella sezione "Installare Rebar3" sopra.

Avvio automatico di un nuovo progetto Erlang

Per avviare un nuovo progetto Erlang, è sufficiente scegliere il modello che si desidera utilizzare dall'elenco. I modelli disponibili possono essere recuperati con il seguente 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 volta scelto il modello appropriato, esegui il bootstrap con il seguente comando (rebar3 creerà una nuova directory per il tuo progetto):

```
$ 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: sebbene tu *possa* eseguire `rebar3 new <template> .` per creare il nuovo progetto nella directory corrente, questo non è raccomandato, perché i file bootstrap useranno `.` (punto) come nomi di applicazioni e moduli e anche in `rebar.config` , che causerà problemi di sintassi.

Leggi Rebar3 online: https://riptutorial.com/it/erlang/topic/4480/rebar3

Capitolo 17: Tipi di dati

Osservazioni

Ogni tipo di dati in erlang è chiamato Term. È un nome generico che indica *qualsiasi tipo di dati* .

Examples

Numeri

In Erlang, i numeri sono interi o galleggianti. Erlang usa la precisione arbitraria per i numeri interi (bignum), quindi i loro valori sono limitati solo dalla dimensione della memoria del tuo sistema.

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

I numeri possono essere utilizzati in varie basi:

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

La notazione `$` -prefix restituisce il valore intero di qualsiasi carattere ASCII / Unicode:

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

atomi

Un atomo è un oggetto con un nome identificato solo dal nome stesso.

Gli atomi sono definiti in Erlang usando letterali atomici che sono entrambi

- una stringa non quotata che inizia con una lettera minuscola e contiene solo lettere, cifre,

caratteri di sottolineatura o il carattere `@` , oppure
- Una singola stringa quotata

Esempi

```
1> hello.
hello

2> hello_world.
hello_world

3> world_Hello@.
world_Hello@

4> '1234'.
'1234'

5> '!@#$%% ä'.
'!@#$%% ä'
```

Atomi che sono usati nella maggior parte dei programmi di Erlang

Ci sono alcuni atomi che appaiono in quasi tutti i programmi di Erlang, in particolare a causa del loro uso nella libreria standard.

- `true` e `false` sono gli abituati a denotare i rispettivi valori booleani
- `ok` è usato solitamente come valore di ritorno di una funzione chiamata solo per il suo effetto, o come parte di un valore di ritorno, in entrambi i casi per indicare un'esecuzione corretta
- Allo stesso modo `error` è usato per indicare una condizione di errore che non garantisce un ritorno anticipato alle funzioni superiori
- `undefined` viene in genere utilizzato come segnaposto per un valore non specificato

Usa come tag

`ok` e `error` sono abbastanza spesso usati come parte di una tupla, in cui il primo elemento della tupla segnala il successo mentre altri elementi contengono il valore di ritorno effettivo o la condizione di errore:

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

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

Conservazione

Una cosa da tenere a mente quando si usano gli atomi è che sono memorizzati nella loro tabella globale nella memoria e questa tabella non è raccolta, quindi creare dinamicamente gli atomi, in particolare quando un utente può influenzare il nome dell'atomo, è fortemente scoraggiato.

Binari e Bitstring

Un binario è una sequenza di byte a 8 bit senza segno.

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

Un bitstring è un binario generalizzato la cui lunghezza in bit non è necessariamente un multiplo di 8.

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

Le tuple

Una tupla è una sequenza ordinata di lunghezza fissa di altri termini di Erlang. Ogni elemento nella tupla può essere qualsiasi tipo di termine (qualsiasi tipo di dati).

```
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]}}
```

elenchi

Una lista in Erlang è una sequenza di zero o più termini di Erlang, implementata come una lista concatenata. Ogni elemento nell'elenco può essere qualsiasi tipo di termine (qualsiasi tipo di dati).

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

La testa della lista è il primo elemento della lista.

La coda della lista è il resto della lista (senza la testa). È anche una lista.
Puoi usare `hd/1` e `tl/1` o confrontare con `[H|T]` per ottenere la testa e la coda dell'elenco.

```
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]
```

Prependendo un elemento a un elenco

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

Elenchi concatenanti

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

stringhe

In Erlang, le stringhe non sono un tipo di dati separato: sono solo elenchi di numeri interi che rappresentano punti di codice ASCII o Unicode:

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

Quando la shell di Erlang stamperà una lista, cercherà di indovinare se si intende effettivamente che si tratti di una stringa. Puoi disattivare questo comportamento chiamando `shell:strings(false)` :

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

Nell'esempio sopra, il numero intero 8 viene interpretato come il carattere di controllo ASCII per il backspace, che la shell considera un carattere "valido" in una stringa.

Identificatori di processi (Pid)

Ogni processo in erlang ha un identificatore di processo (`Pid`) in questo formato `<xxx>` , `x` è un numero naturale. Di seguito è riportato un esempio di `Pid`

```
<0.1252.0>
```

`Pid` può essere usato per inviare messaggi al processo usando 'bang' (`!`), Anche `Pid` può essere limitato a una variabile, entrambi sono mostrati sotto

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

Ulteriori informazioni sulla creazione di processi e più in generale sui processi in erlang

funs

Erlang è un linguaggio di programmazione funzionale. Una delle caratteristiche di un linguaggio di programmazione delle funzioni è la gestione delle funzioni come dati (oggetti funzionali).

- Passa una funzione come argomento a un'altra funzione.
- Funzione di ritorno come risultato di una funzione.
- Mantieni le funzioni in alcune strutture dati.

In Erlang quelle funzioni sono chiamate funs. I divertimenti sono funzioni anonime.

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

I divertimenti possono anche avere diverse clausole.

```
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
```

Puoi anche usare le funzioni del modulo come dei `fun Module:Function/Arity` con la sintassi: `fun Module:Function/Arity` .
Ad esempio, lasciamo la funzione `max` dal modulo `lists` , che ha arity 1.

```
6> Max = fun lists:max/1.
```

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

Mappe

Una mappa è un array o dizionario associativo composto da coppie (chiave, valore).

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

Per aggiornare una mappa:

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

Aggiorna solo alcune chiavi esistenti:

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

Pattern matching:

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

Bit Sintassi: valori predefiniti

Chiarimento di Erlang doc su Bit Sintassi:

> 4.4 Predefiniti
>
> [Inizio omesso: << 3.14 >> non è nemmeno una sintassi legale.]
>
> La dimensione predefinita dipende dal tipo. Per numero intero è 8. Per float è 64. Per binario è la dimensione effettiva del binario specificato:

```
   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
```

Nell'abbinamento, un segmento binario senza dimensione è consentito solo alla fine del pattern e la dimensione predefinita è il resto del file binario sul lato destro della corrispondenza:

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

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

27> X.
97

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

Tutti gli altri segmenti con tipo binario in un modello devono specificare una dimensione:

```
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
```

Leggi Tipi di dati online: https://riptutorial.com/it/erlang/topic/1128/tipi-di-dati

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Erlang Language	A. Sarid, CodeWarrior, Community, drozzy, Efraim Weiss, evnu, Gokul, legoscia, Limmen, maze-le, YsenGrimm, ZeWaren
2	Bit Sintassi: valori predefiniti	7stud
3	comportamenti	legoscia
4	comportamento di gen_server	drozzy, legoscia, M. Kerjouan, Steve Pallen
5	direttore	Pouriya
6	File I / O	CodeDreamer, drozzy, Victor Schröder
7	Formato stringhe	drozzy, legoscia
8	Formato termine esterno	M. Kerjouan
9	Installazione	A. Sarid, drozzy, M. Kerjouan, ZeWaren
10	iolists	Dennis Y. Parygin, legoscia
11	Le autorità di vigilanza	legoscia
12	Loop e ricorsione	M. Kerjouan
13	NIF	Victor Schröder
14	Processi	A. Sarid, YsenGrimm
15	Rebar3	drozzy, Victor Schröder
16	Tipi di dati	7stud, A. Sarid, Ali Sabil, Atomic_alarm, bigo, drozzy, Eddie Antonio Santos, Evgeny Levenets, filmor, gabriel14, Gokul, Gootik, halfelf, legoscia