eBook Gratuit

APPRENEZ Erlang Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#erlang

Table des matières

À propos 1
Chapitre 1: Démarrer avec Erlang Language 2
Remarques 2
Commencer ici 2
Liens 2
Versions 2
Examples 3
Bonjour le monde 3
D'abord le code source de l'application: 3
Maintenant, exécutons notre application: 4
Modules 5
Fonction 5
Compréhension de liste 6
Démarrer et arrêter la coque Erlang 6
Correspondance de motif 7
Chapitre 2: Bit Syntax: Defaults 9
Introduction 9
Examples 9
Les défauts expliqués 9
Chapitre 3: Bit Syntax: Defaults 10
Introduction 10
Examples 10
Réécriture des documents 10
Chapitre 4: Boucle et récursivité 12
Syntaxe 12
Remarques 12
Pourquoi des fonctions récursives? 12
Examples 12
liste 12
Boucle récursive avec des actions IO 13

Liste récursive sur la liste renvoyant la liste modifiée 13
Iolist et Bitstring 14
Fonction récursive sur la taille binaire variable 14
fonction récursive sur la taille binaire variable avec des actions 15
Fonction récursive sur la chaîne de bits renvoyant la chaîne de bits modifiée 16
Carte 17
Etat de gestion 17
Fonction anonyme 18
Chapitre 5: Comportement de gen_server 19
Remarques 19
Examples 19
Greeter Service 19
Utilisation du comportement de gen_server 20
Comportement de gen_server 22
start_link / 0 22
start_link / 3,4 22
init / 1 22
handle_call / 3 22
handle_cast / 2 23
handle_info / 2 23
terminer / 2 24
code_change / 3 24
Commencer ce processus 24
Base de données simple clé / valeur 25
Utiliser notre serveur de cache 27
Chapitre 6: Comportements 28
Examples 28
Utiliser un comportement 28
Définir un comportement 28
Rappels facultatifs dans un comportement personnalisé 29
Chapitre 7: Fichier I / O 30

Examples 30
Lecture d'un fichier 30
Lire l'intégralité du fichier à la fois 30
Lire une ligne à la fois 30
Lire avec l'accès aléatoire 30
Ecrire dans un fichier 31
Ecrire une ligne à la fois 31
Écrivez le fichier entier en une fois 31
Ecrire avec un accès aléatoire 31
Chapitre 8: Format de terme externe 33
Introduction 33
Examples 33
Utiliser ETF avec Erlang 33
Utiliser ETF avec C 33
Initialisation de la structure de données 33
Numéro de codage 33
Atome de codage 33
Tuple de codage 33
Liste de codage 33
Carte d'encodage 33
Chapitre 9: Format des chaînes 35
Syntaxe 35
Examples 35
Séquences de contrôle communes dans les chaînes de format 35
~ s 35
~ w 35
~ p 36
Chapitre 10: Installation 37
Examples 37
Construire et installer Erlang / OTP sur Ubuntu 37

Méthode 1 - Paquet binaire pré-construit 37
Méthode 2 - Créer et installer à partir de la source 37
Construire et installer Erlang / OTP sur FreeBSD 38
Méthode 1 - Paquet binaire pré-construit 38
Méthode 2 - Construire et installer à l'aide de la collection de ports (recommandé) 39
Méthode 3 - Construire et installer à partir de l'archive de version 39
Construire et installer en utilisant kerl 40
Autres sorties 41
Référence 41
Construire et installer Erlang / OTP sur OpenBSD 41
Méthode 1 - Paquet binaire pré-construit 41
Méthode 2 - Construire et installer à l'aide de ports 42
Méthode 3 - Construire à partir de la source 43
Les références 43
Chapitre 11: iolists 44
Introduction 44
Syntaxe 44
Remarques 44
Examples 45
Les listes IO sont généralement utilisées pour générer une sortie vers un port, par exempl 45
Ajoutez les types de données autorisés au début d'une liste d'E / S, en en créant une nouv 45
Les données IO peuvent être efficacement ajoutées à la fin d'une liste. 45
Soyez prudent avec les listes inappropriées 45
Obtenir la taille de la liste IO 45
La liste IO peut être convertie en binaire 45
Chapitre 12: Les superviseurs 47
Examples 47
Superviseur de base avec un processus de travail 47
Chapitre 13: NIFs 48
Examples 48
Définition 48

Exemple: heure UNIX actuelle ... 48
Erlang C API (C à Erlang) ... 50
Chapitre 14: Processus ... 51
Examples ... 51
Création de processus ... 51
Message passant ... 51
Envoi de messages ... 51
Recevoir des messages ... 52
Exemple (compteur) ... 52
Enregistrer les processus ... 53
Chapitre 15: réalisateur ... 54
Introduction ... 54
Remarques ... 54
Avertissements ... 54
Examples ... 55
Télécharger ... 55
Compiler ... 55
Comment ça marche ... 55
Puis-je déboguer le directeur? ... 63
Générer la documentation de l'API ... 64
Chapitre 16: Rebar3 ... 66
Examples ... 66
Définition ... 66
Installer Rebar3 ... 66
Installation à partir du code source ... 66
Amorcer un nouveau projet Erlang ... 67
Chapitre 17: Types de données ... 68
Remarques ... 68
Examples ... 68
Nombres ... 68
Les atomes ... 68
Exemples ... 69

Atomes utilisés dans la plupart des programmes Erlang 69
Utiliser comme balises 69
Espace de rangement 70
Binaires et Bitstrings 70
Tuples 70
Des listes 70
Ajouter un élément à une liste 71
Listes concaténantes 71
Cordes 71
Identificateurs de processus (Pid) 72
Funs 72
Plans 73
Bit Syntax: Defaults 73
Crédits 75

À propos

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

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

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Chapitre 1: Démarrer avec Erlang Language

Remarques

"Erlang est un langage de programmation développé à l'origine au laboratoire d'informatique d'Ericsson. OTP (Open Telecom Platform) est un ensemble de middleware et de bibliothèques à Erlang. Erlang / OTP a été testé sur de nombreux produits Ericsson applications distribuées, par exemple AXD301 (commutateur ATM). Erlang / OTP est actuellement géré par l'unité Erlang / OTP chez Ericsson "(erlang.org)

Commencer ici

Pour les instructions d'installation, voir la rubrique Installation .

Liens

1. Site officiel d'Erlang: https://www.erlang.org
2. Gestionnaire de paquets populaire pour Erlang et Elixir: http://hex.pm
3. Modèles Erlang: http://www.erlangpatterns.org/

Versions

Version	Notes de version	Date de sortie
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	2015-04-02
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

Version	Notes de version	Date de sortie
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

Bonjour le monde

Il y a deux choses que vous devez savoir lorsque vous écrivez une application "hello world" dans Erlang:

1. Le code source est écrit dans le *langage de programmation erlang en* utilisant l'éditeur de texte de votre choix
2. L'application est ensuite exécutée dans la *machine virtuelle erlang* . Dans cet exemple, nous allons interagir avec la machine virtuelle erlang via le shell erlang.

D'abord le code source de l'application:

Créez un nouveau fichier `hello.erl` contenant les éléments suivants:

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

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

Jetons un coup d'œil à ce que cela signifie:

- `-module(hello)` . Toutes les fonctions erlang existent dans un *module* . Les modules sont ensuite utilisés pour créer des applications, qui constituent un ensemble de modules. Cette première ligne consiste à identifier ce module, à savoir *bonjour* . Les modules peuvent être comparés aux *packages* Java
- `-export([hello_world/0])` . Indique au compilateur qui fonctionne pour rendre "public" (par rapport aux langages OO) et à l' *arité* de la fonction concernée. L'arité est le nombre d'arguments pris par la fonction. Puisque dans erlang une fonction avec 1 argument est vue comme une fonction différente de celle avec 2 arguments même si le nom peut être

exactement le même. C'est-à-dire que `hello_world/0` est une fonction complètement différente de `hello_world/1` par exemple.
- `hello_world()` C'est le nom de la fonction. Le `->` indique la transition vers l'implémentation (corps) de la fonction. Cela peut être lu comme "hello_world () est défini comme ...". Notez que `hello_world()` (aucun argument) est identifié par `hello_world/0` dans la machine virtuelle et `hello_world(Some_Arg)` par `hello_world/1` .
- `io:format("Hello, World!~n", [])` Depuis le module `io` , la fonction `format/2` est appelée, qui est la fonction pour la sortie standard. `~n` est un spécificateur de format qui signifie imprimer une nouvelle ligne. Le `[]` est une liste de variables à imprimer indiquée par des spécificateurs de format dans la chaîne de sortie, ce qui dans ce cas n'est rien.
- Toutes les déclarations erlang doivent se terminer par un `.` (point).

Dans Erlang, le résultat de la dernière instruction d'une fonction est renvoyé.

Maintenant, exécutons notre application:

Démarrez le shell erlang à partir du même répertoire que le fichier `hello.erl` :

```
$ erl
```

Vous devriez recevoir une invite ressemblant à ceci (votre version peut être différente):

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

Maintenant, entrez les commandes suivantes:

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

Parcourons chaque ligne une par une:

- `c(hello)` - cette commande appelle la fonction `c` sur un atome `hello` . Cela dit à Erlang de trouver le fichier `hello.erl` , de le compiler dans un module (un fichier nommé `hello.beam` sera généré dans le répertoire) et de le charger dans l'environnement.
- `{ok, hello}` - c'est le résultat de l'appel de la fonction `c` ci-dessus. C'est un tuple contenant un atome `ok` et un atome `hello` . Les fonctions Erlang renvoient généralement soit `{ok, Something}` ou `{error, Reason}` .
- `hello:hello_world()` - cela appelle une fonction `hello_world()` du module `hello` .
- `Hello, World!` - c'est ce que notre fonction imprime.
- `ok` - c'est ce que notre fonction est revenue. Comme Erlang est un langage de programmation fonctionnel, chaque fonction retourne *quelque chose* . Dans notre cas, même si nous n'avons rien `hello_world()` dans `hello_world()` , le dernier appel de cette fonction était sur `io:format(...)` et cette fonction renvoyait `ok` , ce qui correspond à notre fonction.

Modules

Un module erlang est un fichier avec quelques fonctions regroupées. Ce fichier a généralement l'extension `.erl` .

Un module "Hello World" nommé `hello.erl` est présenté ci-dessous

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

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

Dans le fichier, il est nécessaire de déclarer le nom du module. Comme indiqué précédemment à la ligne 1. Le nom du module et le nom du fichier avant l'extension `.erl` doivent être identiques.

Fonction

La fonction est un ensemble d'instructions regroupées. Ces instructions groupées effectuent ensemble certaines tâches. Dans erlang, toutes les fonctions renverront une valeur lorsqu'elles seront appelées.

Voici un exemple d'une fonction qui ajoute deux nombres

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

Cette fonction effectue une opération d'ajout avec les valeurs X et Y et renvoie le résultat. Fonction peut être utilisé comme ci-dessous

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

Les déclarations de fonction peuvent être composées de plusieurs clauses séparées par un point-virgule. Les arguments de chacune de ces clauses sont évalués par correspondance de modèle. La fonction suivante renverra 'tuple' si l'argument est un tuple dans la forme: {test, X} où X peut être n'importe quelle valeur. Il retournera "liste", si l'argument est une liste de la longueur 2 sous la forme ["test", X], et il renverra '{erreur, "raison"} dans tout autre cas:

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

Si l'argument n'est pas un tuple, la deuxième clause sera évaluée. Si l'argument n'est pas une liste, la troisième clause sera évaluée.

Les déclarations de fonction peuvent consister en des «gardes» ou des «séquences de garde». Ces gardes sont des expressions qui limitent l'évaluation d'une fonction. Une fonction avec des gardes n'est exécutée que lorsque toutes les expressions de garde donnent une valeur vraie. Plusieurs gardes peuvent être séparés par un point-virgule.

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

La fonction 'nom_fonction' ne sera évaluée que lorsque la séquence de garde est vraie. La fonction suivante renvoie true uniquement si l'argument `X` est dans la plage correcte (0..15):

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

Compréhension de liste

Les compréhensions de liste sont une construction syntaxique permettant de créer une liste basée sur des listes existantes.
Dans erlang une compréhension de liste a la forme `[Expr || Qualifier1, ..., QualifierN]` .
Lorsque les qualificateurs sont des générateurs `Pattern <- ListExpr` ou un filtre comme un `integer(X)` évalués sur `true` ou sur `false` .

L'exemple suivant montre une compréhension de liste avec un générateur et deux filtres.

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

Le résultat est une liste contenant uniquement des entiers supérieurs à 3.

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

Démarrer et arrêter la coque Erlang

Commencer la coquille Erlang

Sur un système UNIX, vous démarrez le shell Erlang à partir d'une invite de commande avec la commande `erl`

Exemple:

```
$ 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>
```

Le texte qui s'affiche lorsque vous démarrez le shell vous indique quelles sont les versions d'Erlang que vous utilisez, ainsi que d'autres informations utiles sur le système erlang.

Pour démarrer le shell sous Windows, cliquez sur l'icône Erlang dans le menu Démarrer de Windows.

Arrêter la coquille d'Erlang

Pour une sortie contrôlée de la coque erlang, tapez:

```
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().
```

Vous pouvez également quitter le shell Erlang en appuyant sur Ctrl + C sur les systèmes UNIX ou sur Ctrl + Break sous Windows, ce qui vous amène à l'invite suivante:

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

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

Si vous appuyez ensuite sur (pour annuler), vous quitterez directement le shell.

Voici d'autres façons de quitter le shell erlang: `init:stop()` qui fait la même chose que `q()` ou `erlang:halt()` .

Correspondance de motif

L'une des opérations les plus courantes dans erlang est la correspondance de modèle. Il est utilisé lors de l'affectation d'une valeur à une variable, dans des déclarations de fonctions et dans des structures de contrôle-flux comme les instructions `case` et `receive` . Une opération d'appariement de motifs nécessite au moins 2 parties: un motif et un terme correspondant au motif.

Une affectation de variable dans erlang ressemble à ceci:

```
X = 2.
```

Dans la plupart des langages de programmation, la sémantique de cette opération est simple: liez une valeur (`2`) à un nom de votre choix (la variable - dans ce cas `X`). Erlang a une approche légèrement différente: faites correspondre le motif du côté gauche (`X`) au terme du côté droit (`2`). Dans ce cas, l'effet est le même: la variable `X` est maintenant liée à la valeur `2` . Cependant, avec la correspondance de motif, vous pouvez effectuer des affectations plus structurées.

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

Cette opération de correspondance est effectuée en analysant la structure du terme du côté droit et en appliquant toutes les variables du côté gauche aux valeurs appropriées du terme, de sorte que le côté gauche soit égal au côté droit. Dans cet exemple, `Type` est lié au terme: `document` , `Meta` à `{author, "Alice"}` et `Doc` à `{text, "Lorem Ipsum"}` . Dans cet exemple particulier, les variables: `Type` , `Meta` et `Doc` sont supposées être *non liées* , de sorte que chaque variable puisse être utilisée.

Les correspondances de modèles peuvent également être construites à l'aide de variables liées.

```
Identifier = error.
```

La variable `Identifier` est maintenant liée à l' `error` valeur. L'opération de correspondance de modèle suivante fonctionne, car la structure correspond et la variable liée `Identifier` a la même valeur que la partie droite appropriée du terme.

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

Une opération de correspondance de modèle échoue lorsqu'il y a un décalage entre le terme du côté droit et le motif du côté gauche. La correspondance suivante échouera, car `Identifier` est lié à l' `error` valeur, qui n'a pas d'expression appropriée du terme du côté droit.

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

Lire Démarrer avec Erlang Language en ligne: https://riptutorial.com/fr/erlang/topic/825/demarrer-avec-erlang-language

Chapitre 2: Bit Syntax: Defaults

Introduction

Le document Erlang intitulé Bit Syntax a une section appelée Defaults qui contient une erreur et le document est confus dans son ensemble. Je l'ai réécrit

Examples

Les défauts expliqués

> 4.4 Valeurs par défaut
> ...
> ...
> La taille par défaut dépend du type. Pour un entier, il est de 8. Pour float, il est de 64. Pour le binaire, c'est la taille du binaire spécifié:
>
> ```
> 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
> ```
>
> Lors de la correspondance, un segment binaire sans Taille n'est autorisé que pour t

Lire Bit Syntax: Defaults en ligne: https://riptutorial.com/fr/erlang/topic/10050/bit-syntax--defaults

Chapitre 3: Bit Syntax: Defaults

Introduction

Blah bla bla.

Examples

Réécriture des documents.

4.4 Valeurs par défaut

[Début omis: << 3.14 >> n'est même pas une syntaxe légale.]

La taille par défaut dépend du type. Pour entier, il est de 8. Pour float, il est de 64. Pour binaire, c'est la taille réelle du binaire spécifié:

```
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
```

Lors de la mise en correspondance, un segment binaire sans Taille est uniquement autorisé à la fin du modèle, et la Taille par défaut correspond au reste du fichier binaire sur le côté droit de la correspondance:

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

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

27> X.
97

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

Tous les autres segments avec un type binaire dans un modèle doivent spécifier une taille:

```
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
```

Lire Bit Syntax: Defaults en ligne: https://riptutorial.com/fr/erlang/topic/10051/bit-syntax--defaults

Chapitre 4: Boucle et récursivité

Syntaxe

- fonction (liste | iolist | tuple) -> fonction (queue).

Remarques

Pourquoi des fonctions récursives?

Erlang est un langage de programmation fonctionnel et ne comporte aucun type de structure en boucle. Tout dans la programmation fonctionnelle est basé sur des données, des types et des fonctions. Si vous voulez une boucle, vous devez créer une fonction qui s'appelle elle-même.

Les `while` traditionnels ou `for` boucle dans un langage impératif et orienté objet peuvent être représentés comme cela dans Erlang:

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

Une bonne méthode pour comprendre ce concept consiste à développer tous les appels de fonctions. Nous verrons cela sur d'autres exemples.

Examples

liste

Voici la fonction récursive la plus simple sur le type de liste . Cette fonction ne fait que naviguer dans une liste depuis son début jusqu'à sa fin et ne fait plus rien.

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

Vous pouvez l'appeler comme ceci:

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

Voici l'expansion de la fonction récursive:

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

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

Boucle récursive avec des actions IO

Le code précédent ne fait rien et est plutôt inutile. Nous allons donc créer maintenant une fonction récursive qui exécute certaines actions. Ce code est similaire aux `lists:foreach/2` .

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

Vous pouvez l'appeler comme ceci:

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

Voici l'expansion de la fonction récursive:

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

Vous pouvez comparer avec des `lists:foreach/2` sortie:

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

Liste récursive sur la liste renvoyant la liste modifiée

Un autre exemple utile, similaire aux `lists:map/2` . Cette fonction prendra une liste et une fonction anonyme. Chaque fois qu'une valeur dans la liste est appariée, nous lui appliquons une fonction.

```
-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).
```

Vous pouvez l'appeler comme ceci:

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

Voici l'expansion de la fonction récursive:

```
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].
```

Cette fonction est également appelée "fonction récursive de la queue", car nous utilisons une variable comme un accumulateur pour transmettre des données modifiées sur plusieurs contextes d'exécution.

Iolist et Bitstring

Comme la liste, la fonction la plus simple sur iolist et bitstring est:

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

Vous pouvez l'appeler comme ceci:

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

Voici l'expansion de la fonction récursive:

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

Fonction récursive sur la taille binaire variable

Ce code prend la chaîne de bits et définit dynamiquement sa taille binaire. Donc, si nous définissons une taille de `4` , tous les `4` bits, une donnée sera mise en correspondance. Cette boucle ne fait rien d'intéressant, c'est juste notre pilier.

```
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.
```

Vous pouvez l'appeler comme ceci:

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

Voici l'expansion de la fonction récursive:

```
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.
```

Notre chaîne de bits est divisée en 7 motifs. Pourquoi? Parce que par défaut, Erlang utilise une taille binaire de `8` bits, si nous le divisons en deux, nous avons `4` bits. Notre chaîne est `8*3=24` bits. `24/4=6` motifs. Le dernier modèle est `<<>>` . `loop/2` est appelée 7 fois.

fonction récursive sur la taille binaire variable avec des actions

Maintenant, nous pouvons faire quelque chose de plus intéressant. Cette fonction prend un argument de plus, une fonction anonyme. Chaque fois que nous faisons correspondre un motif, celui-ci lui sera transmis.

```
-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.
```

Vous pouvez l'appeler comme ceci:

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

Voici l'expansion de la fonction récursive:

```
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.
```

Fonction récursive sur la chaîne de bits renvoyant la chaîne de bits modifiée

Celui-ci est similaire aux `lists:map/2` mais pour bitstring et 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.
```

Ce code semble plus complexe. Deux fonctions ont été ajoutées: `loop/2` et `loop/3` . Ces deux fonctions sont une interface simple à `loop/4` .

Vous pouvez l'exécuter comme ceci:

```
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>>
```

Carte

Map in Erlang est équivalent aux hashes dans Perl ou aux dictionnaires dans Python, c'est un magasin de clé / valeur. Pour répertorier toutes les valeurs stockées, vous pouvez répertorier chaque clé et renvoyer une paire clé / valeur. Cette première boucle vous donne une idée:

```
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).
```

Vous pouvez l'exécuter comme ça:

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

Etat de gestion

La fonction récursive utilise leurs états pour boucler. Lorsque vous créez un nouveau processus, ce processus sera simplement une boucle avec un état défini.

Fonction anonyme

Voici 2 exemples de fonctions anonymes récursives basées sur l'exemple précédent. Tout d'abord, une boucle infinie simple:

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

Deuxièmement, la fonction anonyme fait la liste en boucle:

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

Ces deux fonctions peuvent être réécrites comme suit:

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

Dans ce cas, `loop/0` est une référence à `loop/0` des remarques. Deuxièmement, avec un peu plus complexe:

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

Ici, `loop/2` est une référence à `loop/2` de la liste exemple. Ces deux notations sont le sucre syntaxique.

Lire Boucle et récursivité en ligne: https://riptutorial.com/fr/erlang/topic/10720/boucle-et-recursivite

Chapitre 5: Comportement de gen_server

Remarques

`gen_server` est une fonctionnalité importante d'Erlang, et nécessite des prérequis pour comprendre chaque aspect de cette fonctionnalité:

- Boucle, récursion et état
- Processus de frai
- Message passant
- Principes du Bureau du Procureur

Un bon moyen d'en savoir plus sur une fonctionnalité d'Erlang est de lire directement le code source depuis le dépôt officiel de github . `gen_server` comportement de `gen_server` nombreuses informations utiles et une structure intéressante dans son cœur.

`gen_server` est défini dans `gen_server.erl` et sa documentation associée peut être trouvée dans la documentation `stdlib` Erlang . `gen_server` est une fonctionnalité OTP et vous trouverez plus d'informations dans les principes de conception d'OTP et le guide de l'utilisateur .

Frank Hebert vous propose également une autre bonne introduction à `gen_server` partir de son livre en ligne gratuit Learn You Some Erlang!

Documentation officielle pour le rappel de `gen_server` :

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

Examples

Greeter Service

Voici un exemple de service qui accueille les utilisateurs par leur nom et suit le nombre d'utilisateurs rencontrés. Voir utilisation ci-dessous.

```
%% 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}.
```

Voici un exemple d'utilisation de ce service dans le shell erlang:

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

Utilisation du comportement de gen_server

Un `gen_server` est une machine à états finis spécifique fonctionnant comme un serveur. `gen_server` peut gérer différents types d'événements:

- requête synchrone avec `handle_call`
- demande asynchrone avec `handle_cast`
- autre message (non défini dans la spécification OTP) avec `handle_info`

Les messages synchrones et asynchrones sont spécifiés dans OTP et sont de simples tuples marqués avec n'importe quel type de données.

Un simple `gen_server` est défini comme ceci:

```
-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.
```

Ce code est simple: chaque message reçu est imprimé sur une sortie standard.

Comportement de gen_server

Pour définir un `gen_server` , vous devez le déclarer explicitement dans votre code source avec `-behaviour(gen_server)` . Notez que le `behaviour` peut être écrit aux États-Unis (comportement) ou au Royaume-Uni (comportement).

start_link / 0

Cette fonction est un simple raccourci pour appeler une autre fonction: `gen_server:start_link/3,4` .

start_link / 3,4

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

Cette fonction est appelée lorsque vous souhaitez démarrer votre serveur lié à un `supervisor` ou à un autre processus. `start_link/3,4` peut enregistrer automatiquement votre processus (si vous pensez que votre processus doit être unique) ou peut simplement le générer comme un processus simple. Lorsqu'elle est appelée, cette fonction exécute `init/1` .

Cette fonction peut renvoyer ces valeurs de définition:

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

init / 1

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

`init/1` est la première fonction exécutée lorsque votre serveur sera lancé. Celui-ci initialise tous les prérequis de votre application et renvoie l'état au processus nouvellement créé.

Cette fonction ne peut renvoyer que les valeurs définies:

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

`State` variable d' `State` peut être tout (par exemple liste, tuple, listes de propriétés, carte, enregistrement) et rester accessible à toutes les fonctions à l'intérieur du processus généré.

handle_call / 3

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

`gen_server:call/2` exécute ce rappel. Le premier argument est votre message (`_Request`), le second est l'origine de la requête (`_From`) et le dernier est l'état actuel (`State`) de votre comportement gen_server en cours d'exécution.

Si vous souhaitez une réponse à l'appelant, `handle_call/3` doit renvoyer l'une de ces structures de données:

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

Si vous ne souhaitez pas répondre à l'appelant, `handle_call/3` doit renvoyer l'une de ces structures de données:

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

Si vous souhaitez arrêter l'exécution en cours de votre gen_server actuel, `handle_call/3` doit renvoyer l'une de ces structures de données:

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

handle_cast / 2

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

`gen_server:cast/2` exécute ce rappel. Le premier argument est votre message (`_Msg`) et le second l'état actuel de votre comportement gen_server en cours d'exécution.

Par défaut, cette fonction ne peut pas transmettre de données à l'appelant. Vous n'avez donc que deux choix, continuez l'exécution en cours:

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

Ou arrêtez votre processus `gen_server` actuel:

- `{stop,Reason,NewState}`

handle_info / 2

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

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

`handle_info/2` est exécuté lorsque le message OTP non standard provient du monde extérieur. Celui-ci ne peut pas répondre et, comme `handle_cast/2` ne peut faire que 2 actions, continue l'exécution en cours:

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

Ou arrêtez le processus en cours de `gen_server` cours d' `gen_server` :

- `{stop,Reason,NewState}`

terminer / 2

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

`terminate/2` est appelée lorsqu'une erreur survient ou lorsque vous souhaitez arrêter votre processus `gen_server` .

code_change / 3

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

`code_change/3` fonction `code_change/3` est appelée lorsque vous souhaitez mettre à niveau votre code en cours d'exécution.

Cette fonction ne peut renvoyer que les valeurs définies:

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

Commencer ce processus

Vous pouvez compiler votre code et démarrer `simple_gen_server` :

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

Si vous souhaitez envoyer un message à votre serveur, vous pouvez utiliser ces fonctions:

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

% will use handle_cast as callback and print:
```

```
%   handle_cast: mymessage
gen_server:cast(simple_gen_server, mymessage).

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

Base de données simple clé / valeur

Ce code source crée un service de banque de clés / valeur simple basé sur la structure de données de `map` Erlang. Tout d'abord, nous devons définir toutes les informations concernant notre `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}.
```

Utiliser notre serveur de cache

Nous pouvons maintenant compiler notre code et commencer à l'utiliser avec `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().
```

Lire Comportement de gen_server en ligne:
https://riptutorial.com/fr/erlang/topic/7481/comportement-de-gen-server

Chapitre 6: Comportements

Examples

Utiliser un comportement

Ajoutez une directive `-behaviour` à votre module pour indiquer qu'il suit un comportement:

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

L'orthographe américaine est également acceptée:

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

Maintenant, le compilateur vous avertira si vous avez oublié d'implémenter et d'exporter l'une des fonctions requises par le comportement, par exemple:

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

Définir un comportement

Vous pouvez définir votre propre comportement en ajoutant des directives `-callback` dans votre module. Par exemple, si les modules implémentant votre comportement doivent avoir une fonction `foo` qui prend un entier et renvoie un atome:

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

Si vous utilisez ce comportement dans un autre module, le compilateur avertira s'il n'exporte pas `foo/1` et Dialyzer avertit si les types ne sont pas corrects. Avec ce module:

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

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

et en exécutant `dialyzer --src bar.erl my_behaviour.erl` , vous obtenez ces avertissements:

```
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
```

Rappels facultatifs dans un comportement personnalisé

18,0

Par défaut, toute fonction spécifiée dans une directive `-callback` dans un module de comportement doit être exportée par un module qui implémente ce comportement. Sinon, vous obtiendrez un avertissement du compilateur.

Parfois, vous souhaitez qu'une fonction de rappel soit facultative: le comportement l'utilise si elle est présente et exportée et, autrement, elle utilise une implémentation par défaut. Pour ce faire, écrivez la directive `-callback` comme d'habitude, puis répertoriez la fonction de rappel dans une directive `-optional_callbacks` :

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

Si le module exporte `bar/0` , Dialyzer vérifiera toujours la spécification de type, mais si la fonction est absente, vous ne recevrez pas d'avertissement du compilateur.

Dans Erlang / OTP lui-même, cela est fait pour la fonction de rappel `format_status` dans les `gen_server` , `gen_fsm` et `gen_event` .

Lire Comportements en ligne: https://riptutorial.com/fr/erlang/topic/7004/comportements

Chapitre 7: Fichier I / O

Examples

Lecture d'un fichier

Supposons que vous avez un fichier **lyrics.txt** qui contient les données suivantes:

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

Lire l'intégralité du fichier à la fois

En utilisant `file:read_file(File)` , vous pouvez lire le fichier entier. C'est une opération atomique:

```
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">>}
```

Lire une ligne à la fois

`io:get_line` lit le texte jusqu'à la nouvelle ligne ou la fin du fichier.

```
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
```

Lire avec l'accès aléatoire

`file:pread(IoDevice, Start, Len)` lit à partir de `Start` , autant que `Len` de `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"}
```

Ecrire dans un fichier

Ecrire une ligne à la fois

Ouvrez un fichier en mode `write` et utilisez `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>
```

Le résultat sera un fichier nommé **fruit_count.txt** avec le contenu suivant:

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

Notez que l'ouverture d'un fichier en mode écriture le créera, s'il n'existe pas déjà dans le système de fichiers.

Notez également que l'utilisation de l'option `write` avec `file:open/2` **tronquera** le fichier (même si vous n'y écrivez rien). Pour éviter cela, ouvrez le fichier en mode `[read,write]` ou `[append]` .

Écrivez le fichier entier en une fois

`file:write_file(Filename, IO)` est la fonction la plus simple pour écrire un fichier à la fois. Si le fichier existe déjà, il sera écrasé, sinon il sera créé.

```
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>
```

Ecrire avec un accès aléatoire

Pour l'écriture à accès aléatoire, le `file:pwrite(IoDevice, Location, Bytes)` est utilisé. Si vous souhaitez remplacer une chaîne dans le fichier, cette méthode est utile.

Supposons que vous souhaitiez remplacer "Olive 12" par "Apple 15" dans le fichier créé ci-dessus.

```
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>
```

Lire Fichier I / O en ligne: https://riptutorial.com/fr/erlang/topic/5232/fichier-i---o

Chapitre 8: Format de terme externe

Introduction

External Term Format est un format binaire utilisé pour communiquer avec le monde extérieur. Vous pouvez l'utiliser avec n'importe quelle langue via les ports, les pilotes ou le NIF. BERT (Binary Erlang terme) peut être utilisé dans d' autres langues.

Examples

Utiliser ETF avec Erlang

```
binary_to_term().
```

Utiliser ETF avec C

Initialisation de la structure de données

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

Numéro de codage

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

int
main() {
}
```

Atome de codage

Tuple de codage

Liste de codage

Carte d'encodage

Lire Format de terme externe en ligne: https://riptutorial.com/fr/erlang/topic/10731/format-de-terme-externe

Chapitre 9: Format des chaînes

Syntaxe

- io: format (FormatString, Args)% écriture en sortie standard
- io: format (standard_error, FormatString, Args)% écriture en erreur standard
- io: format (F, FormatString, Args)% écriture en fichier ouvert
- io_lib: format (FormatString, Args)% renvoie un iolist

Examples

Séquences de contrôle communes dans les chaînes de format

Bien qu'il existe de nombreuses séquences de contrôle différentes pour `io:format` et `io_lib:format` , la plupart du temps, vous n'en utiliserez que trois différentes: `~s` , `~p` et `~w` .

~ s

Le `~s` est pour les *chaînes* .

Il imprime des chaînes, des binaires et des atomes. (Tout le reste provoquera une erreur de `badarg` .) Il ne cite ni n'échappe à rien; il imprime juste la chaîne elle-même:

```
%% 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

Le `~w` sert à *écrire avec une syntaxe standard* .

Il peut imprimer n'importe quel terme Erlang. La sortie peut être analysée pour renvoyer le terme Erlang d'origine, à moins qu'elle ne contienne des termes sans représentation écrite analysable, c'est-à-dire des pids, des ports et des références. Il n'insère pas de nouvelles lignes ni d'indentation, et les chaînes sont toujours interprétées comme des listes:

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

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

~ p

Le `~p` est pour la *jolie impression* .

Il peut imprimer n'importe quel terme Erlang. La sortie diffère de `~w` de la manière suivante:

- Les nouvelles lignes sont insérées si la ligne est trop longue.
- Lorsque des nouvelles lignes sont insérées, la ligne suivante est en retrait pour s'aligner sur un terme précédent du même niveau.
- Si une liste d'entiers ressemble à une chaîne imprimable, elle est interprétée comme une chaîne.

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

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

Si vous ne voulez pas que les listes d'entiers soient imprimées en tant que chaînes, vous pouvez utiliser la séquence `~lp` (insérer une lettre minuscule L avant `p`):

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

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

Lire Format des chaînes en ligne: https://riptutorial.com/fr/erlang/topic/3722/format-des-chaines

Chapitre 10: Installation

Examples

Construire et installer Erlang / OTP sur Ubuntu

Les exemples suivants montrent deux méthodes principales pour installer Erlang / OTP sur Ubuntu.

Méthode 1 - Paquet binaire pré-construit

Il suffit de lancer cette commande pour télécharger et installer la dernière version stable d'Erlang d' Erlang Solutions .

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

Méthode 2 - Créer et installer à partir de la source

Téléchargez le fichier tar:

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

Extrayez le fichier tar:

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

Entrez le répertoire extrait et définissez `ERL_TOP` comme chemin actuel:

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

Maintenant, avant de configurer la construction, vous voulez vous assurer que vous avez toutes les dépendances dont vous avez besoin pour installer Erlang:

Dépendances de base:

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

Autres dépendances d'applications

Application	Installation de dépendance
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`
Documentation	`$ sudo apt-get install fop xsltproc`
Orber et autres projets C ++	`$ sudo apt-get install g++`
jinterface	`$ sudo apt-get install default-jdk`

Configurer et construire:
Vous pouvez définir vos propres options ou le laisser vide pour exécuter la configuration par défaut. Configuration et construction avancées pour Erlang / OTP .

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

Test de la construction:

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

Après avoir exécuté ces commandes, ouvrez `$ERL_TOP/release/tests/test_server/index.html` avec votre navigateur Web et vérifiez que vous n'avez aucun échec. Si tous les tests réussissent, nous pouvons continuer l'installation.

Installation:

```
$ make install
```

Construire et installer Erlang / OTP sur FreeBSD

Les exemples suivants montrent 3 méthodes principales pour installer Erlang / OTP sur FreeBSD.

Méthode 1 - Paquet binaire pré-construit

Utilisez pkg pour installer le paquet binaire pré-construit:

```
$ pkg install erlang
```

Testez votre nouvelle installation:

```
$ 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)
```

Méthode 2 - Construire et installer à l'aide de la collection de ports (recommandé)

Construisez et installez le port comme d'habitude:

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

Testez votre nouvelle installation:

```
$ 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)
```

Cela va récupérer l'archive de la version du site officiel, appliquer quelques correctifs si nécessaire, construire la version et l'installer. Évidemment, cela prendra du temps.

Méthode 3 - Construire et installer à partir de l'archive de version

Remarque: la construction manuelle de la version fonctionne, mais il est préférable d'utiliser les deux méthodes ci-dessus, car la collection de ports incorpore des correctifs qui rendent la version plus conviviale pour FreeBSD.

Téléchargez le fichier de version:

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

Vérifiez que sa somme MD5 est correcte:

```
$ 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
```

Extraire l'archive:

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

Configurez:

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

Si vous souhaitez créer Erlang avec HiPe, vous devrez appliquer les correctifs de la collection de ports.

Construire:

```
$ gmake
```

Installer:

```
$ gmake install
```

Testez votre nouvelle installation:

```
$ 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)
```

Construire et installer en utilisant kerl

kerl est un outil qui vous aide à créer et à installer les versions Erlang / OTP.

Installez curl:

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

Télécharger kerl:

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

Mettre à jour la liste des versions disponibles:

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

Générez la version requise:

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

Vérifiez que la construction est présente dans la liste de construction:

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

Installez la construction quelque part:

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

Source le fichier d' `activate` si vous utilisez bash ou le shell de poisson. Si vous exécutez une cshell, ajoutez le répertoire build bin au PATH:

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

Testez votre nouvelle installation:

```
$ 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)
```

Autres sorties

Si vous souhaitez créer une autre version d'Erlang / OTP, recherchez les autres ports de la collection:

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

Référence

- Manuel FreeBSD -> Chapitre 4. Installation des applications: packages et ports
- Erlang sur FreshPorts
- Documentation Kerl sur GitHub

Construire et installer Erlang / OTP sur OpenBSD

Erlang sur OpenBSD est actuellement en panne sur `alpha` architectures `alpha` , `sparc` et `hppa` .

Méthode 1 - Paquet binaire pré-construit

OpenBSD vous permet de choisir la version que vous souhaitez installer sur votre système:

```
#####################################################################
# 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 peut prendre en charge plusieurs versions d'Erlang. Pour faciliter l'utilisation, chaque fichier binaire est installé dans sa version Erlang en son nom. Donc, si vous avez installé `erlang-19.0v0` , votre `erl` binaire `erl` sera `erl19` .

Si vous voulez utiliser `erl` , vous pouvez créer un lien symbolique:

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

ou créez un alias dans votre fichier de configuration shell ou dans le fichier `.profile` :

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

Vous pouvez maintenant lancer `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)
```

Méthode 2 - Construire et installer à l'aide de ports

```
 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
```

Méthode 3 - Construire à partir de la source

Construire à partir de la source nécessite des paquets supplémentaires:

- `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
```

Les références

- 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

Lire Installation en ligne: https://riptutorial.com/fr/erlang/topic/4483/installation

Chapitre 11: iolists

Introduction

Alors qu'une chaîne Erlang est une liste d'entiers, un "iolist" est une liste dont les éléments sont soit des entiers, des binaires ou d'autres iolists, par exemple `["foo", $b, $a, $r, <<"baz">>]` . Cet iolist représente la chaîne `"foobarbaz"` .

Bien que vous puissiez convertir un iolist en binaire avec `iolist_to_binary/1` , vous n'avez pas souvent besoin, puisque les fonctions de la bibliothèque Erlang telles que `file:write_file/2` et `gen_tcp:send/2` acceptent les iolists ainsi que les chaînes et les binaires.

Syntaxe

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

Remarques

Qu'est-ce qu'un iolist?

C'est n'importe quel binaire. Ou toute liste contenant des nombres entiers compris entre 0 et 255. Ou toute liste arbitrairement imbriquée contenant l'une de ces deux choses.

Article original

Utilisez des listes profondément imbriquées d'entiers et de binaires pour représenter les données d'E / S afin d'éviter la copie lors de la concaténation de chaînes ou de fichiers binaires.

Ils sont efficaces même en combinant de grandes quantités de données. Par exemple, combiner deux binaires de cinquante kilo-octets utilisant la syntaxe binaire `<<B1/binary, B2/binary>>` nécessiterait généralement une réallocation des deux fichiers dans un nouveau binaire de 100 kb. L'utilisation des listes d'E / S `[B1, B2]` que la liste, en l'occurrence trois mots. Une liste utilise un mot et un autre mot par élément, voir ici pour plus d'informations.

L'utilisation de l'opérateur `++` aurait créé une toute nouvelle liste au lieu d'une nouvelle liste à deux éléments. Recréer des listes pour ajouter des éléments à la fin peut devenir coûteux lorsque la liste est longue.

Dans les cas où les données binaires sont petites, l'allocation des listes d'E / S peut être supérieure à l'ajout des fichiers binaires. Si les données binaires peuvent être petites ou grandes, il est souvent préférable d'accepter le coût constant des listes d'E / S.

Notez que les fichiers binaires sont optimisés comme décrit ici . En bref, un binaire peut avoir un espace supplémentaire caché. Cela sera rempli si un autre binaire est ajouté à celui-ci qui correspond à l'espace libre. Cela signifie que toutes les extensions binaires ne provoqueront pas une copie complète des deux fichiers.

Examples

Les listes IO sont généralement utilisées pour générer une sortie vers un port, par exemple un fichier ou un socket réseau.

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

Ajoutez les types de données autorisés au début d'une liste d'E / S, en en créant une nouvelle.

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

Les données IO peuvent être efficacement ajoutées à la fin d'une liste.

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

Soyez prudent avec les listes inappropriées

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

Dans le shell, ceci sera imprimé comme `["Guten tag "|<<"Hello">>]` au lieu de `["Guten tag ",<<"Hello">>]` . L'opérateur de canal créera une liste incorrecte si le dernier élément à droite n'est pas une liste. Bien qu'une liste impropre dont la "queue" est un binaire est toujours un iolist valide, des listes incorrectes peuvent causer des problèmes car de nombreuses fonctions récursives s'attendent à ce qu'une liste vide soit le dernier élément, et non un binaire.

Obtenir la taille de la liste IO

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

La taille d'un iolist peut être calculée à l'aide de `iolist_size/1` . Cet extrait de code calcule la taille d'un message et le crée et l'ajoute au recto sous la forme d'un fichier binaire de quatre octets. Ceci est une opération typique dans les protocoles de messagerie.

La liste IO peut être convertie en binaire

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

Une liste IO peut être convertie en un binaire en utilisant la fonction `iolist_to_binary/1` . Si les

données doivent être stockées pendant une longue période ou envoyées sous forme de message à d'autres processus, il peut être judicieux de les convertir en un fichier binaire. Le coût unique de la conversion en fichier binaire peut être moins coûteux que de copier la liste des entrées-sorties à plusieurs reprises, lors de la récupération de la mémoire d'un processus unique ou lors de la transmission de messages à d'autres utilisateurs.

Lire iolists en ligne: https://riptutorial.com/fr/erlang/topic/5677/iolists

Chapitre 12: Les superviseurs

Examples

Superviseur de base avec un processus de travail

Cet exemple utilise le format de carte introduit dans 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}}.
```

Lire Les superviseurs en ligne: https://riptutorial.com/fr/erlang/topic/6996/les-superviseurs

Chapitre 13: NIFs

Examples

Définition

Documentation officielle: http://erlang.org/doc/tutorial/nif.html

Les NIF ont été introduits dans Erlang / OTP R13B03 en tant que fonctionnalité expérimentale. Le but est d'appeler le code C depuis le code Erlang.

Les NIF sont implémentés dans C au lieu d'Erlang, mais ils apparaissent comme toutes les autres fonctions dans le cadre du code Erlang, car ils appartiennent au module où l'inclusion s'est produite. Les bibliothèques NIF sont liées à la compilation et chargées à l'exécution.

Les bibliothèques NIF étant liées dynamiquement au processus de l'émulateur, elles sont rapides, mais également dangereuses, car les pannes dans un fichier NIF entraînent également la perte de l'émulateur.

Exemple: heure UNIX actuelle

Voici un exemple très simple pour illustrer comment écrire un NIF.

Structure du répertoire:

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

Cette structure peut être facilement initialisée avec Rebar3:

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

Contenu de `nif_test.c` :

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

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

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

```
};

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

Contenu de `nif_test.erl` :

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

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

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

now() -> nif_not_loaded.

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

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

Contenu de `rebar.config` :

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

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

Maintenant, vous pouvez exécuter l'exemple:

```
$ 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 à Erlang)

Documentation officielle : http://erlang.org/doc/man/erl_nif.html

Les structures, types et macros les plus importants de l'API Erlang C sont les suivants:

- `ERL_NIF_TERM` : le type des termes Erlang. C'est le type de retour que les fonctions NIF doivent suivre.
- `ERL_NIF_INIT(MODULE, ErlNifFunc funcs[], load, reload, upgrade, unload)` : C'est la macro qui crée réellement les NIF définis dans un certain fichier C. Il doit être évalué dans son ensemble. Normalement, ce sera la dernière ligne du fichier C.
- `ErlNifFunc` : le type avec lequel chaque NIF est passé à `ERL_NIF_INIT` pour être exporté. Cette structure est composée de name, arity, d'un poiter à la fonction C et de flags. Un tableau de ce type avec toutes les définitions NIF doit être créé pour être transmis à `ERL_NIF_INIT` .
- `ErlNifEnv` : l'environnement Erlang dans lequel le NIF est exécuté. Il est obligatoire de passer l'environnement comme premier argument pour chaque NIF. Ce type est opaque et ne peut être manipulé qu'à l'aide des fonctions offertes par l'API Erlang C.

Lire NIFs en ligne: https://riptutorial.com/fr/erlang/topic/5274/nifs

Chapitre 14: Processus

Examples

Création de processus

Nous créons un nouveau processus simultané en appelant la fonction `spawn` . La fonction `spawn` obtiendra en paramètre une fonction `Fun` que le processus évaluera. La valeur de retour de la fonction `spawn` est l'identificateur de processus créé (pid).

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

Vous pouvez également utiliser `spawn/3` pour démarrer un processus qui exécutera une fonction spécifique à partir d'un module: `spawn(Module, Function, Args)` .
Vous pouvez également utiliser `spawn/2` ou `spawn/4` pour démarrer un processus dans un autre nœud: `spawn(Node, Fun)` ou `spawn(Node, Module, Function, Args)` .

Message passant

Deux processus erlang peuvent communiquer entre eux, également connus sous le nom de *passage de messages* .
Cette procédure est *asynchrone* sous la forme que le processus d'envoi ne s'arrêtera pas après l'envoi du message.

Envoi de messages

Cela peut être réalisé avec la construction `Pid ! Message` , où `Pid` est un identificateur de processus valide (pid) et `Message` est une valeur de tout type de données.

Chaque processus a une "boîte aux lettres" qui contient les messages reçus dans l'ordre reçu. Cette "boîte aux lettres" peut être vidée avec la fonction build `flush/0` .

Si un message est envoyé à un processus non existant, le message sera supprimé sans aucune erreur!

Un exemple peut ressembler à ceci, où `self/0` renvoie le pid du processus en cours et `pid/3` crée 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
```

Il est également possible d'envoyer un message à plusieurs processus à la fois, avec `Pid3!Pid2!Pid1!Msg` .

Recevoir des messages

Les messages reçus peuvent être traités avec la construction de `receive` .

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

Le `Pattern` sera comparé aux messages de la "boîte aux lettres" en commençant par le premier et le plus ancien message.
Si un modèle correspond, le message correspondant est supprimé de la "boîte aux lettres" et le corps de la clause est évalué.

Il est également possible de définir des délais avec la construction `after` .
Un `Timeout` est soit le temps d'attente en millisecondes ou l'atome `infinity` .

La valeur de retour de `receive` est le dernier corps de clause évalué.

Exemple (compteur)

Un compteur (très) simple avec passage de message pourrait ressembler à ce qui suit.

```
-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.
```

Interaction avec le compteur.

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

Enregistrer les processus

Il est possible d'enregistrer un processus (pid) dans un alias global.
Cela peut être réalisé avec la fonction build in `register(Alias, Pid)` , où `Alias` est l'atome pour accéder au processus en tant que et `Pid` est l'ID du processus.

L'alias sera disponible globalement!
Il est très facile de créer un état partagé, ce qui n'est généralement pas préférable. (Voir aussi ici)

Il est possible de désenregistrer un processus avec `unregister(Pid)` et de recevoir le pid d'un alias avec `whereis(Alias)` .

Utilisez `registered()` pour une liste de tous les alias enregistrés.

L'exemple enregistre le totem Atom dans le pid du processus en cours et envoie un message à l'aide de l'atome enregistré.

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

Lire Processus en ligne: https://riptutorial.com/fr/erlang/topic/2285/processus

Chapitre 15: réalisateur

Introduction

Bibliothèque de supervision flexible, rapide et puissante pour les processus Erlang.

Remarques

Avertissements

- N'utilisez pas `'count'=>infinity` et element `restart` dans votre plan.
  comme:

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

Si votre processus n'a pas démarré après une panne, le **directeur** va verrouiller et réessayer de redémarrer les temps d' `infinity` votre processus! Si vous utilisez l' `infinity` pour `'count'` , utilisez toujours `{restart, MiliSeconds}` dans `'plan'` au lieu de `restart` .

- Si vous avez des projets comme:

```
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}.
```

Le reste de l'élément `delete` dans `Childspec1` et le reste de l'élément `stop` dans `Childspec2` ne seront jamais évalués!
Dans `Childspec3` vous voulez exécuter votre plan 0 fois!
Dans `ChildSpec4` vous ne prévoyez pas d'exécuter des temps `infinity` !

- Lorsque vous mettez à niveau une version à l'aide de `release_handler` , `release_handler` appelle `supervisor:get_callback_module/1` pour récupérer son module de rappel.
  Dans OTP, <19 `get_callback_module/1` utilise l'enregistrement d'état interne du superviseur pour donner son module de rappel. Notre **directeur** ne connaît pas l'enregistrement d'état interne du superviseur, alors le `supervisor:get_callback_module/1` ne fonctionne pas avec les **directeurs** .
  La bonne nouvelle est que dans OTP> = 19 `supervisor:get_callback_module/1` fonctionne parfaitement avec les **réalisateurs** :).

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

2> supervisor:get_callback_module(foo_sup).
foo

3>
```

Examples

Télécharger

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

Compiler

Notez que **OTP> = 19 est** requis (si vous souhaitez le mettre à niveau avec `release_handler`).
Accédez au `director` et à l' utilisation des `rebar` d' `rebar3` `rebar` ou `rebar3` .

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

barres d'armature

```
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 $
```

Comment ça marche

le directeur a besoin d'un module de rappel (comme le superviseur OTP).
Dans le module de rappel, vous devez exporter la fonction `init/1` .

Qu'est-ce que `init/1` devrait retourner? Attends, je t'expliquerai pas à pas.

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

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

Enregistrez le code ci-dessus dans `foo.erl` dans le **répertoire** director et accédez au shell Erlang. Utilisez `erl -pa ./ebin` si vous avez utilisé `rebar` pour le compiler et utilisez le `rebar3 shell` si vous avez utilisé `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>
```

Nous avons maintenant un superviseur sans enfants.
La bonne nouvelle est que le **réalisateur** est livré avec une API complète OTP / supervisor et qu'il possède également des fonctionnalités avancées et une approche spécifique.

```
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, je vais faire simple `gen_server` et le donner à notre **directeur** .

```
-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.
```

Enregistrez le code ci-dessus dans le `bar.erl` et revenez au 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>
```

Permet de le vérifier

```
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> %% Yoooohoooooo
```

J'ai mentionné **des fonctionnalités avancées** , qu'est-ce que c'est? Permet de voir d'autres clés acceptables pour la carte `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().
```

Modifier le module `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]}.
```

Allez à nouveau à la coque 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
```

Enfin, quelle est la clé `append` ?

en fait toujours nous avons un `DefaultChildspec` .

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

15>
```

`DefaultChildspec` est comme les enfants normaux, sauf qu'il ne peut pas accepter les clés `id` et `append` .
Si je change `append` valeur à `true` dans ma `Childspec` :
Mon `terminate_timeout` sera ajouté à `terminate_timeout` de `DefaultChildspec` .
Mon `count` sera ajouté au `count` de `DefaultChildspec` .
Mes `modules` seront ajoutés aux `modules` de `DefaultChildspec` .
Mon `plan` sera ajouté au `plan` de `DefaultChildspec` .
Et si j'ai la clé de `start` avec la valeur `{ModX, FuncX, ArgsX}` dans `DefaultChildspec` et la clé de `start` avec la valeur `{ModY, FunY, ArgsY}` dans `Childspec` , la valeur finale sera `{ModY, FuncY, ArgsX ++ ArgsY}` .
Et enfin, si j'ai la clé de `start` avec la valeur `{Mod, Func, Args}` dans `DefaultChildspec` , la clé de `start` dans `Childspec` est facultative pour moi.
Vous pouvez retourner votre propre `DefaultChildspec` tant que troisième élément du tuple dans `init/1` .
Modifier `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}.
```

Redémarrez le 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>
```

Vous pouvez modifier `defaultChildspec` dynamiquement en utilisant `change_default_childspec/2` !
Et vous pouvez également modifier `Childspec` des enfants dynamiquement et définir leur `append` sur `true` !
Mais en les modifiant dans différentes parties du code, vous ferez du **code spaghetti**

Puis-je déboguer le directeur?

Yessssss, **diorector** a son propre debug et accepte le `sys:dbg_opt/0` standard `sys:dbg_opt/0` .
directeur envoie les journaux valides à `sasl` et `error_logger` dans différents états aussi.

```
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>
```

Générer la documentation de l'API

barres d'armature:

```
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()."
```

Après avoir exécuté l'une des commandes ci-dessus, la documentation HTML doit se trouver dans le répertoire `doc` .

Lire réalisateur en ligne: https://riptutorial.com/fr/erlang/topic/9878/realisateur

Chapitre 16: Rebar3

Examples

Définition

Page officielle : https://www.rebar3.org/

Code source : https://github.com/erlang/rebar3

Rebar3 est principalement un gestionnaire de dépendance pour les projets Erlang et Elixir, mais il offre également plusieurs autres fonctionnalités, comme des projets d'amorçage (selon plusieurs modèles, suivant les principes OTP), un exécuteur de tâches, un outil de test et

Installer Rebar3

Rebar3 est écrit en Erlang, il vous faut donc Erlang pour le lancer. Il est disponible sous forme de fichier binaire que vous pouvez télécharger et exécuter. Il suffit de récupérer la construction de nuit et de lui donner des droits d'exécution:

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

Placez ce binaire dans un endroit pratique et ajoutez-le à votre chemin. Par exemple, dans un répertoire bin dans votre maison:

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

Cette dernière ligne doit être placée dans votre `.bashrc` . Comme alternative, on peut également lier le binaire au `/usr/local/bin` , le rendant disponible comme une commande normale.

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

Installation à partir du code source

Comme Rebar3 est gratuit, open source et écrit en Erlang, il est possible de le cloner et de le construire à partir du code source.

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

Cela va créer le script rebar3, que vous pouvez mettre sur votre PATH ou un lien vers `/usr/local/bin` comme expliqué dans la section "Installation de Rebar3" ci-dessus.

Amorcer un nouveau projet Erlang

Pour démarrer un nouveau projet Erlang, choisissez simplement le modèle que vous souhaitez utiliser dans la liste. Les modèles disponibles peuvent être récupérés par la commande suivante:

```
$ 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
```

Une fois que vous avez choisi le modèle approprié, lancez-le avec la commande suivante (rebar3 créera un nouveau répertoire pour votre projet):

```
$ 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: Bien que vous *puissiez* exécuter `rebar3 new <template>` . Pour créer le nouveau projet dans le répertoire en cours, cette opération est déconseillée car les fichiers amorcés seront utilisés `.` (point) en tant que noms d'application et de module, ainsi que dans `rebar.config` , ce qui vous causera des problèmes de syntaxe.

Lire Rebar3 en ligne: https://riptutorial.com/fr/erlang/topic/4480/rebar3

Chapitre 17: Types de données

Remarques

Chaque type de données dans erlang est appelé terme. C'est un nom générique qui signifie *n'importe quel type de données* .

Examples

Nombres

Dans Erlang, les nombres sont soit des entiers, soit des flottants. Erlang utilise une précision arbitraire pour les entiers (bignums), donc leurs valeurs ne sont limitées que par la taille de la mémoire de votre système.

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

Les numéros peuvent être utilisés dans différentes bases:

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

La notation `$` -prefix donne la valeur entière de n'importe quel caractère ASCII / Unicode:

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

Les atomes

Un atome est un objet avec un nom identifié uniquement par le nom lui-même.

Les atomes sont définis dans Erlang en utilisant des littéraux atomiques qui sont soit

- une chaîne non cotée qui commence par une lettre minuscule et ne contient que des lettres, des chiffres, des traits de soulignement ou le caractère `@` , ou
- Une seule chaîne citée

Exemples

```
1> hello.
hello

2> hello_world.
hello_world

3> world_Hello@.
world_Hello@

4> '1234'.
'1234'

5> '!@#$%% ä'.
'!@#$%% ä'
```

Atomes utilisés dans la plupart des programmes Erlang

Certains atomes apparaissent dans presque tous les programmes Erlang, notamment en raison de leur utilisation dans la bibliothèque standard.

- `true` et `false` sont utilisés pour désigner les valeurs booléennes respectives
- `ok` est généralement utilisé comme valeur de retour d'une fonction appelée uniquement pour son effet, ou dans le cadre d'une valeur de retour, dans les deux cas pour indiquer une exécution réussie
- De la même manière, une `error` est utilisée pour indiquer une condition d'erreur qui ne justifie pas un retour rapide des fonctions supérieures.
- `undefined` est généralement utilisé comme espace réservé pour une valeur non spécifiée

Utiliser comme balises

`ok` et `error` sont assez souvent utilisés dans le cadre d'un tuple, dans lequel le premier élément du tuple signale le succès alors que d'autres éléments contiennent la valeur de retour ou la condition d'erreur:

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

```
            {error, wrong_one}
    end.

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

Espace de rangement

Une chose à garder à l'esprit lors de l'utilisation des atomes est qu'ils sont stockés dans leur propre table globale en mémoire et que cette table n'est pas nettoyée, donc la création dynamique d'atomes, en particulier lorsqu'un utilisateur peut influencer le nom de l'atome est fortement déconseillée.

Binaires et Bitstrings

Un binaire est une séquence d'octets de 8 bits non signés.

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

Une chaîne de bits est un binaire généralisé dont la longueur en bits n'est pas nécessairement un multiple de 8.

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

Tuples

Un tuple est une séquence ordonnée de longueur fixe d'autres termes Erlang. Chaque élément du tuple peut être n'importe quel type de terme (n'importe quel type de données).

```
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]}}
```

Des listes

Une liste dans Erlang est une séquence de zéro ou plus de termes Erlang, mis en œuvre sous la forme d'une liste liée par un seul élément. Chaque élément de la liste peut être n'importe quel type de terme (n'importe quel type de données).

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

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

La tête de la liste est le premier élément de la liste.

La queue de la liste est le reste de la liste (sans la tête). C'est aussi une liste.
Vous pouvez utiliser `hd/1` et `tl/1` ou correspondre à `[H|T]` pour obtenir la tête et la queue de la liste.

```
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]
```

Ajouter un élément à une liste

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

Listes concaténantes

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

Cordes

Dans Erlang, les chaînes de caractères ne sont pas un type de données distinct: elles ne sont que des listes d'entiers représentant des points de code ASCII ou Unicode:

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

Lorsque le shell Erlang va imprimer une liste, il essaie de deviner si vous voulez vraiment dire qu'il s'agit d'une chaîne. Vous pouvez désactiver ce comportement en appelant `shell:strings(false)` :

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

Dans l'exemple ci-dessus, l'entier 8 est interprété comme le caractère de contrôle ASCII pour le retour arrière, que le shell considère comme un caractère "valide" dans une chaîne.

Identificateurs de processus (Pid)

Chaque processus dans erlang a un identifiant de processus (`Pid`) dans ce format `<xxx>` , `x` étant un nombre naturel. Voici un exemple de `Pid`

```
<0.1252.0>
```

`Pid` peut être utilisé pour envoyer des messages au processus en utilisant "bang" (`!`), `Pid` peut être limité à une variable

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

En savoir plus sur la création de processus et plus généralement sur les processus dans erlang

Funs

Erlang est un langage de programmation fonctionnel. L'une des fonctionnalités d'un langage de programmation de fonctions est la gestion des fonctions sous forme de données (objets fonctionnels).

- Passer une fonction en argument à une autre fonction.
- Fonction de retour résultant d'une fonction.
- Fonctions de maintien dans certaines structures de données.

Dans Erlang, ces fonctions sont appelées funs. Les funs sont des fonctions anonymes.

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

Les Funs peuvent aussi avoir plusieurs clauses.

```
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
```

Vous pouvez également utiliser les fonctions du module comme des fonctions `fun Module:Function/Arity` avec la syntaxe suivante: `fun Module:Function/Arity` .
Par exemple, prenons la fonction `max` partir du module de `lists` , qui a l'arité 1.

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

Plans

Une carte est un tableau ou un dictionnaire associatif composé de paires (clé, valeur).

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

Pour mettre à jour une carte:

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

Ne mettez à jour que certaines clés existantes:

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

Correspondance de motif:

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

Bit Syntax: Defaults

Clarification du document Erlang sur la syntaxe des bits:

4.4 Valeurs par défaut

[Début omis: << 3.14 >> n'est même pas une syntaxe légale.]

La taille par défaut dépend du type. Pour entier, il est de 8. Pour float, il est de 64. Pour binaire, c'est la taille réelle du binaire spécifié:

```
    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
```

Lors de la mise en correspondance, un segment binaire sans Taille est uniquement autorisé à la fin du modèle, et la Taille par défaut correspond au reste du fichier binaire sur le côté droit de la correspondance:

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

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

27> X.
97

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

Tous les autres segments avec un type binaire dans un modèle doivent spécifier une taille:

```
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
```

Lire Types de données en ligne: https://riptutorial.com/fr/erlang/topic/1128/types-de-donnees

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Erlang Language	A. Sarid, CodeWarrior, Community, drozzy, Efraim Weiss, evnu, Gokul, legoscia, Limmen, maze-le, YsenGrimm, ZeWaren
2	Bit Syntax: Defaults	7stud
3	Boucle et récursivité	M. Kerjouan
4	Comportement de gen_server	drozzy, legoscia, M. Kerjouan, Steve Pallen
5	Comportements	legoscia
6	Fichier I / O	CodeDreamer, drozzy, Victor Schröder
7	Format de terme externe	M. Kerjouan
8	Format des chaînes	drozzy, legoscia
9	Installation	A. Sarid, drozzy, M. Kerjouan, ZeWaren
10	iolists	Dennis Y. Parygin, legoscia
11	Les superviseurs	legoscia
12	NIFs	Victor Schröder
13	Processus	A. Sarid, YsenGrimm
14	réalisateur	Pouriya
15	Rebar3	drozzy, Victor Schröder
16	Types de données	7stud, A. Sarid, Ali Sabil, Atomic_alarm, bigo, drozzy, Eddie Antonio Santos, Evgeny Levenets, filmor, gabriel14, Gokul, Gootik, halfelf, legoscia