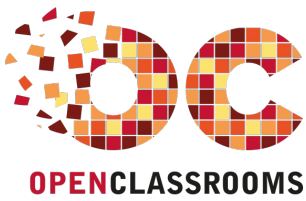
Hot Code Swapping

Par rks`



www.openclassrooms.com

Sommaire

Hot Code Swapping

Par rks`

Mise à jour : 01/01/1970

Dans ce tutoriel, vous allez découvrir le ***hot code swapping*** ainsi que sa mise en application dans le langage de programmation Erlang.
Naturellement, cela va demander un minimum de connaissances en Erlang pour la mise en application de ce concept ; si vous ne les possédez pas, vous pouvez néanmoins vous contenter de la partie théorique.

Sommaire du tutoriel :

Présentation

I have a dream...

Je suppose que chacun d'entre vous a joué ou tout du moins entendu parler de MMORPG, et il est aussi probable que quelques-uns parmi vous ont déjà rêvé d'en créer un, ou tout du moins, un jeu multijoueur en ligne (sans toutefois qu'il soit massivement multijoueur).

Eh bien admettons que ce rêve se réalise ! Vous êtes aujourd'hui le créateur d'un MMORPG très connu, vous connaissez la gloire et le succès. Sans cesse en quête d'améliorations pour votre jeu, vous développez régulièrement de nouvelles *features* qu'aucun autre jeu ne possède. Une fois cette nouveauté codée et testée en local vous pouvez la dévoiler au reste du monde et gagner de nouveaux utilisateurs !

Seulement voilà, pour rendre cette nouvelle fonctionnalité disponible, vous allez devoir redémarrer votre serveur. Ce *reboot* entraînera inévitablement la déconnexion de tous vos utilisateurs. Alors ça peut passer une fois, pour une mise à jour assez révolutionnaire, mais comme je l'ai dit, vous êtes inventif, et des nouvelles fonctionnalités, vous en sortez tous les 3 jours. Cela est bien pour votre jeu, mais pourra vite être relativement agaçant pour vos utilisateurs, qui au bout de quelques-uns de vos redémarrages vont se mettre à abandonner votre jeu...

Que dois-je faire ? Attendre 6 mois avant de faire une mise à jour ? Pour avoir plein de nouveaux trucs à leur montrer ?

Eh bien à vrai dire... vous pourriez faire ça, mais ce serait relativement idiot et pas très motivant de coder des trucs que vous savez géniaux mais que vous ne pouvez pas mettre à disposition de ceux à qui ils sont destinés. Heureusement pour vous, il y a une autre solution...

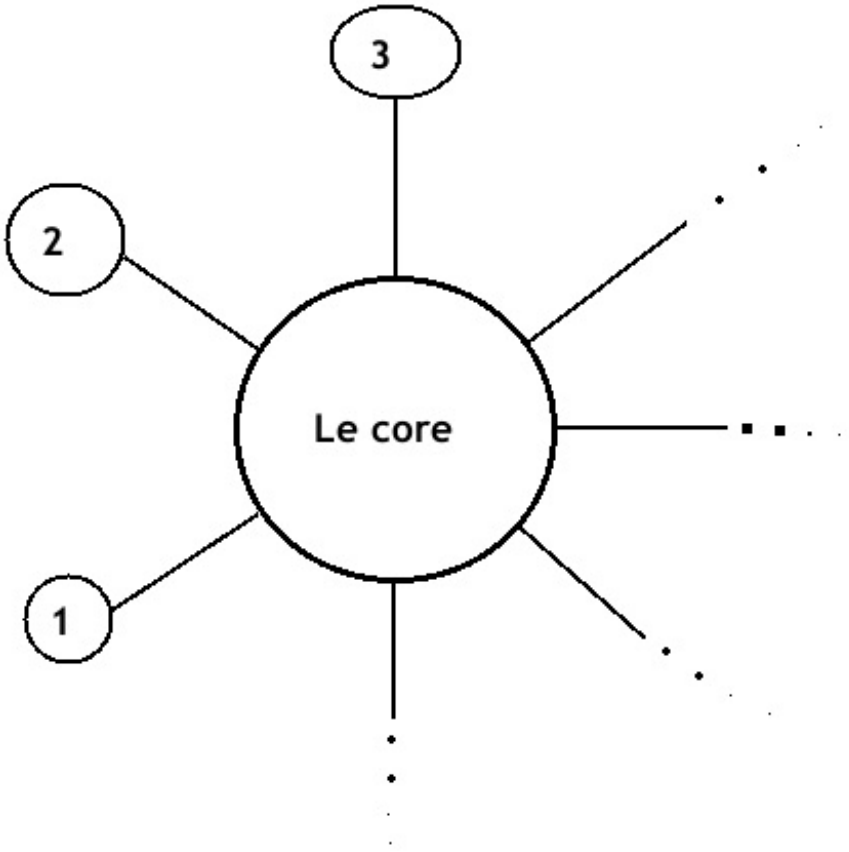
... le hot code swapping !

Citation : Wikipedia

> Hot swapping can also refer to the ability to alter the running code of a program without having to interrupt its execution.

Pour ceux ayant un peu de mal avec l'anglais, le *hot code swapping* fait référence à la capacité de modifier le code d'un programme en cours d'exécution sans avoir à l'interrompre.
Cette possibilité est offerte par peu de langages, parmi lesquels on peut noter : Erlang, Lisp, Haskell et Smalltalk.

Concrètement, ça signifie que l'on va pouvoir charger nos nouvelles fonctionnalités au fur et à mesure tout en laissant tourner le serveur. Ça, ça va faire plaisir à vos utilisateurs ! En fait, le principe du hcs est simple : notre application est divisée en plusieurs modules (ou programmes) reliés entre eux par le *core*. C'est le core qui devra charger, recharger ou quitter des modules ainsi que s'occuper de la liaison entre les différents modules.
Ainsi, lorsque vous voudrez ajouter une nouvelle fonctionnalité, il suffira de dire à votre core de la charger, ce ne sera plus la peine de tout relancer !



Un schéma bilan du système

Et en bonus...

Un autre avantage majeur du hcs est qu'il permet une meilleure gestion des crashs. En effet, si un de vos modules plante le problème n'affectera pas tout le système, il suffira de dire à votre core de le relancer. Alors que sans le hcs, si une partie du jeu plante tout votre serveur plante, et il vous faut alors vous dépêcher de tout relancer, faire des excuses à vos utilisateurs, etc.

Ces possibilités devraient en intéresser plus d'un ici (du moins, je l'espère) et actuellement, vous vous demandez probablement comment mettre ça en place. Eh bien c'est exactement ce que vous allez découvrir ici.

Et du côté d'Erlang ?

Cette partie demande un minimum de connaissances du langage Erlang, bien qu'aucun code n'y soit étudié. La lecture du tutoriel de présentation devrait vous apporter les quelques « connaissances » requises.

Si l'on regarde de plus près le principe du *hcs*, on comprend bien vite que notre programme devra posséder une architecture du type : « *core-<module(s)>* ». En effet, il faut à tout moment être en mesure de (re)lancer un module, ou de gérer un de ses crashs, c'est ce dont le *core* devra s'occuper. D'un autre côté, les modules éventuels devront être capables de communiquer avec lui, pour notamment comprendre ses requêtes mais aussi pour pouvoir lui faire passer des messages (qu'il pourra éventuellement transmettre aux autres modules).

Notre core devra donc être sans cesse à l'écoute des différents modules qui auront pu être lancés, ainsi que de l'utilisateur. Mais en plus de cela, il devra être en mesure de fermer, voir recharger, certains modules, cela implique de posséder une liste exhaustive de ces différents modules. Notre boucle d'interaction (j'espère que vous aviez deviné qu'on allait en utiliser une ) possédera donc comme argument la liste des modules lancés.

Les modules quant à eux devront respecter une organisation stricte. Par exemple, la fonction avec laquelle ils seront initialisés devra porter le même nom, quelque soit le module. Une boucle étant à l'écoute des messages envoyés par le core est bien entendu indispensable, ne serait-ce que pour être quittés ou rechargés.

En pratique

Règles

Comme je vous l'ai dit, pour mettre en place un tel système il vous faudra respecter certaines règles, notamment pour la création des fonctions spécifiques ; vous êtes libres d'utiliser vos propres règles, mais je vais vous présenter les miennes ici, afin de clarifier les choses. Celles-ci seront valables pour cette partie et la suivante, c'est pourquoi certaines vous paraîtront inutiles de prime abord. Trève de bavardages, voici les règles.

- Les fonctions d'initialisation de mes modules seront nommées init.
- L'initialisation du core se fera par contre à l'aide d'une fonction nommée start, pour bien marquer le contraste.
- Les fonctions *init*s auront toujours la même arité (= le même nombre d'arguments).
  Pour ma part ce sera 1, une liste, afin de pouvoir passer des arguments ou non, en fonction des besoins du module.
- Les messages envoyés au *core* par les modules seront de la forme {From, Atom, ArgsList}.
- Les messages envoyés au *core* par l'utilisateur seront par contre de la forme {core, Atom, {Args}}.
- Les messages à destination des modules seront eux de la forme : {Atom, ArgsList}, exception faite du premier core, où ce serait inutile, et surtout lourd.

Une application uni-modulaire

Comme énoncé précédemment, le *core* sera formé d'une fonction d'initialisation (start) et d'un boucle d'interaction.

Code : Erlang

```
-module(core).
-export([start/0]).

start() ->
    io:format("Lancement du core ...~n"),
    spawn(fun() -> loop(fun do_nothing/1) end).

do_nothing(X) -> X;

loop(Fun) ->
    receive
        {core, charger, NewFun} ->
            loop(NewFun);

        quit ->
            io:format("Fermeture du programme ...~n");

        Else ->
            Result = Fun(Else),
            io:format("~p~n", [Result]),
            loop(Fun)
    end.
```

On a ici pu regarder un exemple basique d'utilisation du hot code swapping :

1. on lance le module ;
2. on lui passe une fonction ;
3. il attend que :
    - on lui passe une nouvelle fonction,
      retour à l'étape 3,
    - on lui dise de quitter,
      passage à l'étape 4,
    - on lui passe « quelque chose d'autre »,
      il applique la fonction à ce quelque chose,
      retour à l'étape 3 ;
4. on quitte le programme.

Une application plus élaborée

Une connaissance de l'Erlang est désormais indispensable : en effet, cette partie du tutoriel est en majorité dédiée à la mise en place du système de hot code swapping.

Notre application va cette fois être un peu plus élaborée. En effet, on va utiliser le hot code swapping pour charger, recharger, et quitter des modules.

Le core

Voyons d'abord le code :

Code : Erlang

```
-module(core).
-export([start/0]).

start() ->
    io:format("Lancement du core ...~n"),
    register(core, spawn(fun() -> loop([]) end)).

loop(Modules) ->
    receive
        {core, load, {Module, Args}} ->
            io:format("Chargement de ~s ...~n", [Module]),
            Pid = spawn(fun() -> Module:init(Args) end),
            loop([{Module, Pid}|Modules]);

        {core, reload, {Module, NewArgs}} ->
            io:format("Fermetume de ~s ...~n", [Module]),
            {Module, Pid} = lists:keysearch(Module, 1, Modules),
            Pid ! quit,
            io:format("Chargement de ~s ...~n", [Pid]),
            NewPid = spawn(fun() -> Module:init(NewArgs) end),
            NewList = lists:keyreplace(Module, 1, Modules, {Module,
NewPid}),
            loop(NewList);

        {core, unload, {Module}} ->
            io:format("Fermeture de ~s ...~n", [Module]),
            {Module, Pid} = lists:keysearch(Module, 1, Modules),
            Pid ! quit,
            NewList = lists:keydelete(Module, 1, Modules),
            loop(NewList);

        {core, quit} ->
            %% ici, plutot lists:foreach, car on utilise pas le
```

```
    resultat
                lists:foreach(fun({_, Pid}) -> Pid ! quit end, Modules),
                io:format("Modules fermés.~n");

            {_From, show, String} ->
                io:format("~s~n", [String]),
                loop(Modules);

            {_From, Else, _Args} ->
                 io:format("Unknow request: ~p.~n", [Else]),
                 loop(Modules)
        end.
```

Celui-ci, bien qu'un peu plus complexe que le précédent reste **très** basique. Néanmoins, il illustre bien le principe détaillé plus haut, à savoir : pouvoir charger et recharger des modules à volonté sans avoir à relancer tout le programme.

Vous aurez pu remarquer que les 4 premiers messages, ou du moins les 4 premiers messages *recherchés*, sont ceux mettant en jeu une intervention humaine. Parmi ceux-ci, les 2 premiers ont pour but de charger ou recharger des modules, alors que le deux suivants sont destinés à fermer 1 ou la totalité des modules chargés.

Les deux messages suivants servent à la réception des messages des modules, le premier doit tout simplement afficher à l'écran le message envoyé alors que le second... eh bien il doit gérer tous les autres types de requête, mais comme on ne sait pas les traiter, ou plutôt qu'on ne veut pas les traiter dans le cadre de ce tutoriel (la première requête étant là à titre d'exemple), on affiche la requête puis on relance la boucle.

Les Modules

La structure

Si vous avez regardé le code précédent, vous avez pu remarquer une ligne ressemblant à :

Code : Erlang

```
Pid = spawn(fun() -> Module:init(Args) end)
```

Vous connaissez tous (du moins, je l'espère) la fonction spawn, inutile donc de s'y attarder. Ce qui peut par contre être intéressant, c'est la fonction utilisée pour initialiser le processus : Module:init ; pour ceux qui auraient la mémoire courte : Module est la variable contenant le nom du module à charger, on lance donc le process avec la fonction init du module (comme prévu), en lui passant la liste d'arguments fournis par l'utilisateur.

Vos modules devront donc **obligatoirement** contenir une fonction : init/1. Après il faudra bien entendu une fonction principale récursant jusqu'à ce qu'on ferme le module, cette fonction devra servir à recevoir les messages du core.
Ensuite, libre à vous de créer les fonctions que vous voulez, c'est votre programme après tout. 

La communication

Là aussi, un protocole strict devrait être respecté, du moins, ça permet d'éviter les problèmes. Ce qu'il faut, c'est que tous les messages envoyés par ou aux modules passent par le core. Pour faciliter cela, vous avez pu voir dans le code du core qu'à la création du process nous avons utilisé la fonction : register/2. Pour ceux qui ne connaîtraient pas encore cette fonction, elle associe un atome à un pid. Le premier argument est l'atome, le second un Pid ; ici, vous pouvez voir une fonction, *spawn* mais que renvoie cette fonction ? Eh oui : un Pid ! Une fois ceci fait, lorsque vous écrirez cet atome, cela fera référence au Pid qui lui est associé ainsi : core ! X enverra le contenu de X au core.

Ici le core est très basique, il n'envoie pas de message à ses modules à l'exception de quit ; pour que vos modules lui envoient des messages, il faudra donc qu'il fasse les choses par eux-mêmes, sans attendre les instructions du core. En réalité, il est très rare que cela se passe ainsi, le core envoie la plupart du temps des messages à ses modules, il arrive néanmoins que les modules doivent (comme ici) entreprendre des actions sans que le core le leur dise, la méthode la plus simple pour cela est d'utiliser un timer.

Voilà, vous connaissez désormais le hot code swapping et savez le mettre rapidement et simplement en application avec Erlang. Si jamais certains d'entre vous décident de l'utiliser dans le cadre de leurs programmes, il faudra qu'ils fassent un corps bien plus élaboré, capable de communiquer à ses modules, et établir une structure permettant la communication des modules.

Notes

Ce tuto a été écrit, non pas en zCode, mais en mdown, un petit langage de mise en forme agréable à utiliser (et qui peut produire du zCode), conçu par rz0 ; vous pourrez trouver une comparaison entre la syntaxe mdown et le zCode sur cette page.

Merci à Cygal, bluestorm, lastsseldon et igwana pour leur relecture et leurs conseils.

Un tutoriel signé PHM