The REBOL Documentation Project

-- FR - Documentation REBOL - Divers --

Divers

Introduction du concept de "Module courant"

Philippe Le Goff

Première publication : 8 juin 2006, et mis en ligne le jeudi 8 juin 2006

Résumé :

La traduction de l'article 29 du blog de CArl sur REBOL 3.0, à propos des modules.

Introduction du concept de "Module courant"

Carl Sassenrath, CTO REBOL Technologies 24-May-2006 23:53 GMT Article #0029

Permettez-moi de dire ceci en vérité : mon document sur les Modules n'est pas très bon. Désolé, mais c'est juste pas très clair. Il date un peu, et dès que j'aurais écrit un meilleur document, je promets d'effacer l'ancien.

Bon, pour clarifier tout cela : **les Modules fournissent une manière d'étendre l'environnement REBOL**. C'est leur propos. Bien sûr, cette déclaration ne signifie pas que tout avec les modules est simple. Ce n'est pas le cas. Mais, comprendre et utiliser les modules doit rester simple. N'importe qui devrait être capable de les utiliser. (Autrement, je ne le ferais pas, n'est-ce pas ? Vous me connaissez.)

Donc, ceci implique quelques petites choses :

1. Vous devrez être capable de créer un module aussi simplement qu'en chargeant un script qui ressemblerait à :

```
REBOL [title : "My module" type : module] ...
```

et, alternativement, vous pourrez utiliser cette méthode n'importe quand dans votre code :

```
my-mod : make module ! [ [title : "My module"] a : 10 func1 : func [...] [...] ]
```

Note : Cela signifie que vous pourrez créer plusieurs modules au sein d'un unique script, si vous avez besoin de le faire.

2. Vous devriez être capables d'accéder facilement aux valeurs exportées par un module. Comment "facilement" ? Et bien automatiquement ....

par exemple, si votre module est :

```
REBOL [title : "My Math" type : module export : [buy sell]] ... buy : func [value price] [...] sell : func [value price] [...]
```

et bien, par la suite, vous pourrez exécuter un script qui fera ceci :

```
buy "google" $200 sell "microsoft" $10
```

Il devrait fonctionner comme vous l'espérez. Ce doit être juste facile.

3. Il y a de nombreuses options en plus, qui représentent le reste. Oui, ces options sont utiles, mais la plupart du temps, vous n'y prendrez pas garde.

Maintenant, voyons ce qui est un peu plus compliqué. Laissez moi vous donner un bon exemple :

Disons que nous avons un bout de code qui exécute un script externe utilisant la fonction do.

A quel module sont liés les mots de ce script ? **Ce n'est pas une question triviale**. Je la mentionne pour vous faire réfléchir. En profondeur, j'espère.

Pourquoi cela est-il important ? Voici pourquoi : si vous utilisez une fonction d'un module REBOL (appelons le Mod-A) qui ressemble à :

```
do-it : func [file] [   ...   do file   ...  ]
```

est-ce que les variables du fichier nouvellement évalué sont liées au module Mod-A ou sont-elles liées au module qui a appelé la fonction " do-it " en premier lieu ?

Avec les années, j'ai constaté que lorsque ces questions surviennent, il y a une sorte d'action implicite qui s'est produit. C'est une hypothèse en cours. Dans ce cas-ci, à quel module les nouveaux mots sont-ils attachés ? Nous avons deux choix :

* Le fichier où la fonction "do" a été évaluée

* Le module original où la fonction " do-it " a été évaluée.

Dans les langages statiques, peu puissants, comme C et d'autres, ceci serait un choix facile. Mais, dans l'évaluation dynamique de REBOL, il n'est pas aussi simple de décider. Les deux choix sont valides.

Comme avec REBOL ou comme dans d'autres domaines, le meilleur choix doit venir d'une sorte de principe de bon sens.

Examinons cela :

Il vaut mieux éviter une auto-expansion accidentelle des modules. Certains résultats peuvent être problématiques, car arbitrairement, de nouvelles fonctions peuvent être ajoutées à celles existant dans les modules standard du système. Ceci n'est pas une bonne chose.

Partant de là, selon cette règle, la fonction do précédente devrait binder les mots nouveaux du module actif courant. Bon, voici une action implicite : le concept du module courant.

Lorsque votre script va être évalué, REBOL l'exécutera dans son propre et unique module utilisateur ("user module"). Et lorsque des scripts supplémentaires seront chargés et évalués, ils seront par défaut liés au même module utilisateur, à moins d'être explicitement dirigés vers ailleurs.

A présent, que se passe-t-il si vous voulez dans un bout de code faire référence explicitement à un module ? Que se passe-t-il si vous faites un "load" ou un "do" et que vous voulez affecter un module spécifique.

C'est le propos de la fonction **with**.

Exactement comme vous pouvez dire :

```
with object [... expressions ...]
```

avec ici "object" égal à n'importe quel objet valide, et que le bloc est évalué au sein du contexte de

l'objet, vous pouvez écrire aussi :

```
with module [... expressions ...]
```

avec "module" égal à n'importe quel module existant. Cette ligne spécifie que les expressions sont liées au contexte du module fourni.

Vous pouvez toujours écrire :

```
with module [do %script.r]
```

pour forcer le module à prendre en compte les bindings du nouveau script.

En plus, pour rendre cette action plus facile, vous pouvez aussi utilisez n'importe quel mot d'un contexte pour représenter le contexte lui-même. De cette manière, vous n'avez pas à vous préoccuper de la manière d'obtenir la référence du contexte. Par example :

```
with 'word [do %script.r]
```

où le mot "word" est relié au contexte cible.

Je pense que ceux d'entres vous qui sont des experts en REBOL verront que c'est bien plus qu'un autre nom pour " bind ". Une variable cachée a été définie pour indiquer le module, l'environnement courant pour le bloc. Et, il y a quelques implications plus profondes qui sont la conséquence de tout ceci (comme que ce qui se produit quand une erreur survient, et rejaillit en dehors du bloc fourni à **with**.)

Bien, vous y êtes. De la simplicité avec juste un pincée d'épice pour donner le meilleur de la saveur. Maintenant, si seulement je pouvais faire cuire tout ce qui est bon.

S'il vous plaît, merci de poster vos commentaires. Je suis ouvert à toutes les idées. Merci.

(Traduction : Philippe Le Goff)