The REBOL Documentation Project

-- FR - Documentation REBOL - Manuels --

Manuels

Manuel de l'utilisateur - Chapitre 7 - Les Séries de blocs

Philippe Le Goff

Première publication : 14 septembre 2005, et mis en ligne le mercredi 14 septembre 2005

Résumé :

Ce document est la traduction française du Chapitre 7 du User Guide de REBOL/Core, qui concerne les séries sous forme de blocs.

Ce document est la traduction française du Chapitre 7 du User Guide de REBOL/Core, qui concerne les séries sous forme de blocs.
Traducteur : Philippe Le Goff

1 Historique de la traduction

Date	Version	Commentaires	Auteur	Email
22 mai 2005 7:42	1.0.0	Traduction initiale	Philippe Le Goff	lp—legoff—free—fr

2 Blocs de Blocs

Quand un bloc apparaît en tant que valeur au sein d'un autre bloc, il est compté comme un **seul** élément et cela, quelque soit le nombre de valeurs qu'il contient.

Par exemple :

```
values: [
    "new" [1 2]
    %file1.txt ["one" ["two" %file2.txt]]
]
probe values
new" [1 2] %file1.txt ["one" ["two" %file2.txt]]]
```

La longueur de *values* est quatre éléments. La seconde et la quatrième valeur sont comptés comme des éléments unitaires :

```
print length? values
4
```

Les blocs à l'intérieur d'autres blocs ne perdent pas leur caractéristique de bloc. Dans l'exemple ci-dessous, la fonction **second** est utilisée pour extraire la deuxième valeur du bloc *values*. Pour afficher le bloc, saisissez :

```
probe second values
[1 2]
```

Pour connaître la longueur de ce bloc, tapez :

```
print length? second values
2
```

Pour afficher le type de données (datatype) :

```
print type? second values
block!
```

De la même manière, les fonctions sur les séries peuvent être exploitées sur d'autres types de valeurs dans les blocs. Dans l'exemple suivant, **pick** est utilisé pour extraire *%file1.txt* du bloc *values*.

Pour récupérer la valeur du troisiéme élément, saisissez :

```
probe pick values 3
%file1.txt
```

Pour récupérer la longueur de la valeur :

```
print length? pick values 3
9
```

Pour voir le type de données associé à la valeur extraite :

```
print type? pick values 3
file
```

3 Paths, chemins pour les blocs imbriqués

La notation avec les paths est très pratique pour les blocs imbriqués.

La quatrième valeur de la série *values* est un bloc contenant d'autres blocs. L'exemple suivant utilise un path pour récupérer des informations dans ce bloc.

Pour voir les valeurs de ce bloc, tapez :

```
probe values/4
["one" ["two" %file2.txt]]
probe values/4/2
["two" %file2.txt]
```

Pour obtenir les longueurs :

```
print length? values/4
2
print length? values/4/2
2
```

Et pour voir le type de données, saisissez :

```
print type? values/4
block!
print type? values/4/2
block!
```

Les deux séries contenues dans cette quatrième valeur sont aisément accessibles. Pour voir ces valeurs, tapez :

```
probe values/4/2/1
two
probe values/4/2/2
%file2.txt
```

Pour obtenir les longueurs de ces valeurs :

```
print length? values/4/2/1
3
print length? values/4/2/2
9
```

et pour leurs datatypes (types de données) :

```
print type? values/4/2/1
string
print type? values/4/2/2
file
```

Pour modifier ces valeurs :

```
change (next values/4/2/1) "o"
probe values/4/2/1
too
change/part (next find values/4/2/2 ".") "r" 3
probe values/4/2/2
%file2.r
```

Les exemples précédents illustrent la capacité que possède REBOL à manipuler des valeurs imbriquées dans des blocs. Notez que, dans les derniers exemples, la fonction **change** est utilisée pour modifier une chaîne et un nom de fichier avec trois niveaux d'imbrication.

L'affichage du bloc *values* produit le résultat suivant :

```
probe values
["new" [1 2] %file1.txt ["one" ["too" %file2.r]]]
```

4 Tableaux

Les blocs sont utilisés pour créer des tableaux. Un exemple de tableau bi-dimensionnel statique est :

```
arr: [
    [1   2   3  ]
    [a   b   c  ]
    [$10 $20 $30]
]
```

Vous pouvez obtenir les valeurs d'un tableau avec les fonctions d'extraction relatives aux séries :

```
probe first arr
[1 2 3]
probe pick arr 3
[$10.00 $20.00 $30.00]
probe first first arr
1
```

Vous pouvez aussi utiliser des paths pour obtenir les valeurs d'un tableau :

```
probe arr/1
[1 2 3]
probe arr/3
[$10.00 $20.00 $30.00]
probe arr/3/2
$20.00
```

Les paths peuvent encore être utilisés pour changer les valeurs dans un tableau :

```
arr/1/2: 20

probe arr/1
[1 20 3]
arr/3/2: arr/3/1 + arr/3/3

probe arr/3/2
$40.00
```

4.1 Création de tableaux

La fonction **array** crée dynamiquement un tableau.

Cette fonction prend en argument soit un nombre entier, soit un bloc de nombres entiers, et elle retourne en résultat un bloc : le tableau. Par défaut, les cellules d'un tableau sont initialisées à **none**.

Pour intialiser les cellules d'un tableau avec d'autres valeurs, utilisez le raffinement **/initial**, qui est expliqué dans la section suivante.

Lorsqu'un tableau est fourni avec **un seul nombre entier**, c'est un tableau à **une dimension**, de la taille du nombre, qui est retourné.

```
arr: array 5
probe arr
[none none none none none]
```

Quand un bloc de plusieurs nombres entiers est passé en argument, le tableau est à plusieurs dimensions. Chaque nombre entier donne respectivement la taille de la dimension correspondante.

Voici un exemple d'un tableau possédant six cellules, sur deux lignes et trois colonnes :

```
arr: array [2 3]
probe arr
[[none none none] [none none none]]
```

Il est possible de faire un tableau à trois dimensions en rajoutant un autre nombre entier au bloc en argument :

```
arr: array [2 3 2]
foreach lst arr [probe lst]
[[none none] [none none] [none none]]
[[none none] [none none] [none none]]
```

Le bloc d'entiers qui est passé à la fonction **array** peut être très grand selon ce que la capacité de la mémoire de votre système.

4.2 Valeurs initiales

Pour initialiser les cellules d'un tableau à une valeur autre que **none**, utilisez le raffinement **/initial**.

Voici quelques exemples :

```
arr: array/initial 5 0
probe arr
[0 0 0 0 0]
arr: array/initial [2 3] 0
probe arr
[[0 0 0] [0 0 0]]
arr: array/initial 3 "a"
probe arr
["a" "a" "a"]
arr: array/initial [3 2] 'word
probe arr
[[word word] [word word] [word word]]
arr: array/initial [3 2 1] 11:11
probe arr
```

```
[[[11:11] [11:11]] [[11:11] [11:11]] [[11:11] [11:11]]]
```

5 Composition de blocs

La fonction **compose** est pratique pour créer des blocs avec des valeurs dynamiques. Elle peut être utilisée pour créer aussi bien des données et du code.

La fonction **compose** attend un bloc en argument et renvoie en résultat un bloc composé de chacune des valeurs du bloc en argument.

Les valeurs entre parenthèses sont évaluées en priorité, avant que le bloc ne soit retourné. Par exemple :

```
probe compose [1 2 (3 + 4)]
[1 2 7]
probe compose ["The time is" (now/time)]
["The time is" 10:32:45]
```

Si des parenthèses encadrent un bloc, alors chaque valeur de ce bloc sera utilisée :

```
probe compose [a b ([c d])]
[a b c d]
```

Pour éviter cela dans le résultat, vous devez inclure ce bloc dans un autre bloc :

(**Ndt** : c'est-à-dire protéger le bloc par un autre bloc).

```
probe compose [a b ([[c d]])]
[a b [c d]]
```

Un bloc sans éléments est sans effet :

```
probe compose [a b ([]) c d]
[a b c d]
```

Lorsque **compose** s'applique sur un bloc comprenant des sous-blocs, les sous-blocs ne sont pas évalués, même s'ils contiennent des parenthèses :

```
probe compose [a b [c (d e)]]
[a b [c (d e)]]
```

Si vous souhaitez que les sous-blocs soient évalués, utilisez le raffinement **/deep**. Le raffinement **/deep** entraine l'évaluation de toutes les valeurs entre parenthèses, indépendamment de la position où elles se trouvent :

```
probe compose/deep [a b [c (d e)]]
[a b [c d e]]
```