Programmation réactive

Table of Contents

1. Concept

Dans la version 0.6.0, Red a introduit le support de la "programmation réactive" pour aider à réduire la taille et la complexité des programmes. Le modèle réactif de Red repose sur les flux de données et les objets événements pour construire un graphe dirigé et propager les changements dans les objets. Il utilise un modèle "push". Plus spécifiquement, Red implémente le modèle de programmation réactive orientée objet, dans lequel seuls les champs des objets peuvent être la source du changement.

L'API réactive et son usage sont simples et pratiques, même si la description est abstraite. Voici quelques diagrammes pour aider à visualiser les relations réactives.

[react-simple] | *react-simple.png*

Les graphes A et B montrent des relations simples entre un ou plusieurs réacteurs (objets qui agissent comme une source réactive).

[react-graph] | *react-graphs.png*

Les graphes C, D et E montrent des réactions en chaîne, où certaines cibles sont elles-mêmes des réacteurs, ce qui instaure une chaîne de relations qui peut avoir n'importe-quelle forme.

Les réactions sont exécutées de manière asynchrone, lorsque la valeur d'un champ source est changée. La relation de réaction est maintenue jusqu'à ce que la réaction soit explicitement détruite en utilisant `react/unlink` ou `clear-reactions`.

Seuls l'objet source dans une expression réactive doit être un réacteur. La cible peut être un simple objet. Si la cible est également un réacteur, les réactions sont chaînées et un graphe de relations est implicitement construit.

NOTE

- Le support réactif de Red pourrait être étendu à l'avenir pour supporter un modèle "pull".
- Ceci n'est pas un framework de programmation réactive fonctionnelle, bien que les flux d'événements pourraient être supportés dans l'avenir.
- Le moteur d'interface graphique de Red/View repose sur les objets *face!* (figure) afin d'opérer sur des objets graphiques. Les figures sont des réacteurs, et elles peuvent être utilisées pour mettre en place des relations réactives entre les figures et des objets non-réacteurs.

1.1. Glossaire

Expression	Définition
programmation réactive	Un paradigme de programmation, sous-ensemble de la programmation par flux de données, basé sur des événement qui provoquent ("push") des changements.
réaction	Un bloc de code qui contient une ou plusieurs expressions réactives.
expression réactive	Une expression qui fait référence à au moins une source réactive.
relation réactive	Une relation entre deux ou plusieurs objets implémentée en utilisant des expressions réactives.
source réactive	Une valeur de type `path!` faisant référence à un champ dans un objet réactif.
formule réactive	Une réaction qui renvoie le dernier résultat d'une expression à l'évaluation.
objet réactif	Un objet dont les champs peuvent être utilisés comme sources réactives.
réacteur	Alias pour "objet réactif".

1.2. Relations statiques

La forme la plus simple de réaction est une "relation statique" créée entre des objets *nommés*. Elle est *statique* parce qu'elle lie statiquement les objets. Elle s'applique uniquement à ses réacteurs sources, et ne peut pas être réutilisée pour d'autres objets.

Exemple 1

```
view [
    s: slider return
    b: base react [b/color/1: to integer! 255 * s/data]
]
```

Cet exemple établit une relation réactive entre un curseur nommé `s` et une figure de base nommée `b`. Lorsque le curseur est déplacé, la composante rouge de la couleur de fond de la figure de base est changée à mesure. L'expression réactive ne peut pas être réutilisée pour un autre jeu de figures. Ceci est la forme la plus simple de comportement réactif pour les objets graphiques dans Red/View.

Exemple 2

```
vec: reactor [x: 0 y: 10]
box: object [length: is [square-root (vec/x ** 2) + (vec/y ** 2)]]
```

Une autre forme de relation statique peut être définie en utilisant l'opérateur `is`, dans lequel le résultat de l'évaluation de la réaction est placé dans un mot (dans n'importe quel contexte).

Cet exemple n'est pas lié au système d'interface graphique. Il calcule la longueur d'un vecteur défini par `vec/x` et `vec/y` en utilisant une expression réactive. Là encore l'objet source est spécifié statiquement par son nom (`vec`) dans l'expression réactive.

Exemple 3

```
a: reactor [x: 1 y: 2 total: is [x + y]]
```

Le mot `total` ci-dessus prend la valeur de l'expression `x + y`. Chaque fois que l'une des valeurs `x` ou `y` change, `total` est immédiatement mis à jour. Remarquez que les chemins ne sont pas requis dans ce cas pour spécifier les sources réactives, parce-que `is` est utilisé directement à l'intérieur du corps du reacteur et connaît donc son contexte.

Exemple 4

```
a: reactor [x: 1 y: 2]
total: is [a/x + a/y]
```

Cette variation de l'Exemple 3 mointre qu'un mot global peut être la cible d'une relation réactive (quoiqu'il ne puisse pas en être la source). Cette forme est la plus proche du modèle de formule dans un tableur (eg. Excel).

NOTE	de par la taille du contexte global, le rendre réactif (comme ci-dessus avec `total`) peut être coûteux en termes de performances, quoi que cela pourrait être surmonté dans le futur.

1.3. Relations dynamiques

Les relations statiques sont faciles à spécifier, mais elles ne passent pas facilement à l'échelle si vous avez besoin de donner la même réaction à un certain nombre de réacteurs, ou si les réacteurs sont anonymes (rappel: les objets sont anonymes par défaut). Dans de tels cas, la réaction devrait être spécifiée avec une **function** et `react/link`.

Exemple

```
;-- Faire glisser la balle rouge de haut en bas avec la souris. Observer comment
réagissent les autres balles.

win: layout [
    size 400x500
    across
    style ball: base 30x30 transparent draw [fill-pen blue circle 15x15 14]
    ball ball ball ball ball ball ball b: ball loose
    do [b/draw/2: red]
]

follow: func [left right][left/offset/y: to integer! right/offset/y * 108%]

faces: win/pane
while [not tail? next faces][
    react/link :follow [faces/1 faces/2]
    faces: next faces
]
view win
```

Dans cet exemple, la réaction est une fonction (`follow`) qui est appliquée aux figures de balles par paires. Cela crée une chaîne de relations qui relient toutes les balles entre elles. Les termes de la réaction sont des paramètres, qui peuvent ainsi être utilisés pour différents objets (contrairement aux relations statiques).

2. API

2.1. react

Syntaxe

```
react <code>
react/unlink <code> <source>

react/link <func> <objects>
react/unlink <func> <source>

react/later <code>

<code>    : bloc de code qui contient au moins une source réactive (block!).
<func>    : fonction qui contient au moins une source réactive (function!).
<objects> : liste d'objets utilisés comme arguments à une fonction réactive (block! of
object! values).
<source>  : le mot 'all, ou un objet, ou une liste d'objets (word! object! block!).

Renvoie   : <code> ou <func> pour les références futures à la réaction.
```

Description

`react` définit une nouvelle relation réactive, qui contient au moins une source réactive, à partir d'un bloc de code (définissant une "relation statique") ou d'une fonction (définissant une "relation dynamique" et nécessitant le raffinement `/link`). Dans les deux cas, le code est analysé statiquement pour déterminer les sources réactives (sous la forme de valeurs de type `path!`) qui font référence aux champs du réacteur.

Par défaut, la réaction nouvellement formée **est appelée une fois à sa création** avant que la fonction `react` ne se termine. Cela peut être indésirable dans certains cas, et peut donc être évité avec l'option `/later`.

Une réaction contient n'importe quel code Red, une ou plusieurs sources réactives, et une ou plusieurs expressions réactives. C'est à l'utilisateur de déterminer le jeu de relations qui correspondent le mieux à ses besoins.

L'option `/link` prend une fonction comme réaction et une liste d'objets arguments à utiliser dans l'évaluation de la réaction. Cette forme alternative permet des réactions dynamiques, où le code de la réaction peut être réutilisé avec différents types d'objets (le `react` de base ne peut fonctionner qu'avec des objets *nommés* statiquement).

On supprime une réaction en utilisant le raffinement `unlink` avec l'un des éléments suivants comme argument `<source>`:

- Le mot `'all`, supprimera toutes les relations réactives créées par la réaction.
- Une valeur de type objet, supprimera seulement les relations dont cet objet est la source réactive.
- Une liste d'objets, supprimera seulement les relations dont ces objets sont les sources réactives.

`/unlink` prend un bloc ou une fonction de réaction en argument, de sorte que seules les relations créées par **cette** réaction sont supprimées.

2.2. is

Syntaxe

```
<word>: is [<body>]
<word>: is [[<default>] <body>]

<word>    : mot qui doit prendre pour valeur le résultat de la réaction (set-word!).
<body>    : bloc de code Red quelconque contenant au moins une source réactive.
<default> : bloc de code Red quelconque qui renvoie une valeur utilisée initialement
par défaut pour <word>.
```

Description

`is` crée un formule réactive dont le résultat sera assigné à un mot. Le bloc `<code>` peut contenir des références à la fois aux champs de l'objet encapsulant, s'il est utilisé dans le bloc de corps d'un réacteur, et à des champs de réacteurs externes. Pour spécifier une valeur par défaut, un bloc renvoyant cette valeur par défaut peut être renvoyé comme premier élément du bloc de la formule réactive. Ceci est particulièrement utile quand on utilise des références vers l'avant à des sources réactives qui ne sont pas définies (`unset`) au moment de l'évaluation de la formule.

NOTE	Cet opérateur crée des formules réactives qui imitent étroitement le modèle des formules d'Excel.

Exemples

```
a: reactor [x: 1 y: 2 total: is [x + y]]

a/total
== 3
a/x: 100
a/total
== 102
```

```
reactor [a: 1 b: is [[none] c] c: is [a < 4]]
== make object! [
    a: 1
    b: true
    c: true
]
```

2.3. react?

Syntaxe

```
react? <obj> <field>
react?/target <obj> <field>

<obj>   : object to check (object!).
<field> : object's field to check (word!).

Renvoie : une reaction (block! function!) ou une valeur none! .
```

Description

`react?` vérifie si un champ d'un objet est une source réactive. Si c'est le cas, la première réaction trouvée où ce champ d'objet est présent comme source sera renvoyée, sinon `none` est renvoyé. Le raffinement `/target` vérifie si le champ est une cible au lieu d'une source, et renverra la première réaction trouvée ciblant ce champ ou `none` s'il n'y a pas de correspondance.

2.4. clear-reactions

Syntaxe

```
clear-reactions
```

Description

Supprime inconditionnellement toutes les réactions définies.

2.5. dump-reactions

Syntaxe

```
dump-reactions
```

Description

Produit une liste des réactions enregistrées à des fins de débogage.

3. Objets réactifs

Les objets ordinaires en Red ne montrent pas de comportements réactifs. Afin qu'un objet soit une source réactive, il doit être construit à partir de l'un des prototypes de réacteur suivants.

3.1. reactor!

Syntaxe

```
make reactor! <body>

<body> : bloc corps de l'objet  (block!).

Renvoie : un objet réactif.
```

Description

Construit un nouvel objet réactif à partir du bloc corps. Dans l'objet renvoyé, donner une nouvelle valeur à un champ déclenchera les réactions définies pour ce champ. La fonction `reactor` est un raccourci pratique pour ce type de constructeur.

NOTE	Le corps peut contenir des expressions `is`.

3.2. deep-reactor!

Syntaxe

```
make deep-reactor! <body>

<body> : bloc corps de l'objet  (block!).

Renvoie : un objet réactif.
```

Description

Construit un nouvel objet réactif à partir du bloc corps. Dans l'objet renvoyé, donner une nouvelle valeur à un champ ou changer une série à laquelle le champ fait référence, y compris des séries imbriquées, déclenchera les réaction définies pour ce champ. La fonction `deep-reactor` est un raccourci pratique pour ce type de constructeur.

NOTE	Le corps peut contenir des expressions `is`.

Exemple

Ceci montre comment le changement d'une série, même imbriquée, déclenche une réaction.

NOTE	C'est actuellement à l'utilisateur de prévenir les bouclages. Par exemple, si un `deep-reactor!` change les valeurs d'une série dans une formule de réacteur (e.g. `is`), cela peut créer des cycles de réaction sans fin.

```
r: deep-reactor [
    x: [1 2 3]
    y: [[a b] [c d]]
    total: is [append copy x copy y]
]
append r/y/2 'e
print mold r/total
```