リアクティブプログラミング

目次

1. コンセプト

バージョン

0.6.0から、Redは「リアクティブプログラミング」のサポートを開始しました。上手く活用することで、Redのプログラムのサイズと複雑さを抑えることが可能です。Redのリアクティブモデルはデータフローとオブジェクトのイベントを直接的に関連付け、オブジェクトの変更を伝播させます。これは「プッシュ型」のモデルです。より具体的に言うと、Redは object-oriented reactive programming モデルを実装しており、オブジェクトのフィールドの変更によってのみ起こされます。

リアクティブAPIの説明は抽象的になりがちですが、しかし実際には非常にシンプルで実用的です。リアクティブなリレーションを表現する、以下の図を見てください。

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

グラフAとBは１つ以上のシンプルなリアクター（リアクティブな動作のソースとなるオブジェクト）の関係性を示しています。

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

グラフC・D・Eは連鎖的なリアクションを示しています。ここではリアクションのターゲットはそれ自身がリアクターになっていて、鎖状の関連性を持っています。このように、どのような形状の関係性も構築することが可能です。

リアクションはソースとなるフィールドの値が変更された時、非同期に実行されます。リアクションのリレーションは `react/unlink` または `clear-reaction` で明示的に破棄されるまで維持されます。

リアクターとなる必要があるのは、リアクティブエクスプレッション内のソースオブジェクトのみです。ターゲットとなる側は通常のオブジェクトのままとなります。ターゲットがリアクターとなる場合、連鎖的にリアクションとなり、暗黙的にリアクションのリレーショングラフが構築されます。

NOTE:

- 将来的には「プル型」のリアクティブプログラミングもサポートされるように拡張される可能性があります。

- 将来的にはイベントストリームがサポートされる可能性がありますが、Redのリアクティブプログラミングは FRP フレームワークではありません。
- Red/View GUIエンジンはグラフィックオブジェクトを処理する上で *face!*（フェイス）オブジェクトに依拠しています。フェイスはリアクターで、他のフェイスや、リアクターではないオブジェクトとの間のリアクティブリレーションを設定するために使用することができます。

1.1. 用語集

用語集	意味
リアクティブプログラミング	データフロープログラミングのサブセットであり、イベントが変更を「プッシュ」することに基づくプログラミングパラダイム。
リアクション	リアクティブエクスプレッションを含むコードブロック
リアクティブエクスプレッション	1つ以上のリアクティブソースを参照する式
リアクティブリレーション	リアクティブエクスプレッションを使って、オブジェクトの間で結ばれるリレーション
リアクティブソース	リアクティブオブジェクトのフィールド尾を参照する path! の値
リアクティブフォーミュラ	式によって評価された最後の値を返すリアクション
リアクティブオブジェクト	フィールドがリアクティブソースとして使われているオブジェクト
リアクター	「リアクティブオブジェクト」と同じ意味

1.2. スタティックリレーション

最もシンプルな形式のリアクションは 名前付けされたオブジェクトの間で作られる「スタティックリレーション」です。「スタティック」というのはオブジェクトに静的にリンクしているということです。それはソースのリアクターへのみ適用され、他のオブジェクトに対して再利用することはできません。

例1

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

この例では「s」という名前のスライダー（slider）と「b」という名前のベースフェイス（base）の間でリアクティブリレーションを設定しています。スライダーが動かされると、追随してベースフェイスの背景色の赤要素が変更されます。このリアクティブエクスプレッションは他のフェイスに対して再利用することはできません。これがRed/Viewにおけるグラフィックオブジェクトのリアクティブな挙動としては、最もシンプルなものです。

例2

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

スタティックリレーションを定義するもうひとつの方法は「is」演算子を使うことです。is演算子を使うと、wordにリアクティブに評価された値がセットされます。

この例はGUIシステムとは無関係です。上記のコードはリアクティブエクスプレッションにより、「vec/x」と「vec/y」という定義に基づいてvectorの長さを計算します。繰り返しになりますが、このリアクティブエクスプレッションでは、ソースオブジェクトは「vec」という名前によりスタティックに指定されています。

例３

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

上記の「total」というwordは「x + y」という式を値としてセットされています。「x」か「y」が変更されるたびに、「total」は更新されます。このケースではリアクティブソースを指定するためにパスが必要ないことに注目してください。これは、「is」がリアクターのボディの中で直接使われており、コンテキストが同一であるためです。

例４

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

これは例３のバリエーションで、グローバルのwordもリアクティブリレーションのターゲットにできることを示しています（グローバルのwordをリアクティブソースにすることはできません）。この形式はExcelなどのスプレッドシートにおける計算式のモデルと最も近いタイプと言えます。

NOTE:グローバルコンテキストのサイズの関係上、それをリアクティブにする（上記の「total」がそうですが）と重大なパフォーマンスのオーバーヘッドが発生することがあります。この問題は将来のバージョンでは解消される可能性があります。

1.3. ダイナミックリレーション

スタティックリレーションは簡単に指定できますが、たくさんのリアクターに同じリアクションを設定する必要がある場合や、リアクターが匿名（補足：すべてのオブジェクトはデフォルトでは匿名です。）の場合に、うまく対応することができません。そのようなケースでは、リアクションは 関数 と「react/link」で指定する必要があります。

例

```
;-- 赤いボールをマウスドラッグで上下に動かし、他のボールがどのように動くか確認してください。

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
```

この例では、リアクションの対象はボールfaceが２つずつ順番に適用されるfollow関数になっています。これはリレーションの連鎖を生成し、全てのボールをリンクさせます。リアクティブになっている対象はパラメータのため、スタティックリレーションと異なり、違うオブジェクトに使用することができます。

2. API

2.1. react

構文

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

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

react/later <code>

<code>    : 少なくとも1つのリアクティブソースを含むコードブロック
<func>    : 少なくとも1つのリアクティブソースを含む関数
<objects> : リアクティブ関数に対しての引数として使われるオブジェクトのリスト
<source>  : 「'all」というword、オブジェクト（object!）、オブジェクトのリスト（block!）のいずれか

Returns   : リアクションへの参照のための <code> または <func>
```

説明

「react」は1つ以上のリアクティブソースに対し。新しいリアクティブリレーションをセットします。リレーションをセットする元となるのはコードのブロック（この場合「スタティックリレーション」となります）か、関数（この場合「ダイナミックリレーション」となり、「/link」リファインメントを使用する必要があります）です。いずれの場合もコードはスタチェックに解析され、リアクターのフィールドを参照するリアクティブソース（path!の形式で表現されます）が決定されます。

デフォルトでは新しく生成されたリアクションは「react」関数の結果が返される前に、生成されると同時に一度実行されます。
この動作が望ましくない場合、「/later」オプションを使用することもできます。

リアクションは任意のRedのコード、リアクティブソースとリアクティブエクスプレッションを好きなだけ含めることができます。どのような組み合わせが適切か、自由に決めることが可能です。

「/link」オプションはリアクションとなる関数と、リアクションの中で評価に使われるオブジェクトのリストを取ります。この方法の場合、異なるオブジェクトのセットでリアクションコードを使いまわせるダイナミックリアクションが可能となります。（基本形となる「react」は、スタティックに 名前付けされた オブジェクトに対してのみ機能します。）

リアクションは「/unlink」リファインメントと以下のいずれかの「<source>」引数により、削除されます。

- 「'all」というword。この場合リアクションによって作成された全てのリアクティブリレーションが削除されます。
- オブジェクト。この場合、そのオブジェクトがリアクティブソースとなっているリレーションのみ削除されます。
- オブジェクトのリスト。この場合、渡されたオブジェクトがリアクティブソースとなっているリレーションのみ削除されます。

「/unlink」はリアクションブロックか関数を引数として取ります。そのため そのリアクション によって作られたリアクションのみ削除されます。

2.2. is

構文

```
<word>: is <code>

<word> : リアクションの結果をセットするword(set-word!型)
<code> : 1つ以上のリアクティブソースを含むコードブロック(block!型)
```

説明

「is」はリアクティブフォーミュラを生成し、式の結果がwordに割り当てられるようになります。「<code>」ブロックは次のどちらかへの参照を含むコードブロックです。1つ目は、リアクターのボディブロックの中で使用されている場合の、ラッピングオブジェクトのフィールドへの参照です。2つ目は、外部のリアクターのフィールドへの参照です。

NOTE:この演算子はExcelの計算式のモデルに似たリアクティブフォーミュラを生成します。

例

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

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

2.3. react?

構文

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

<obj>   : チェックするオブジェクト(object!)
<field> : チェックするオブジェクトのフィールド(word!)

Returns : リアクション(block!またはfunction!)か none! の値
```

説明

「react?」はオブジェクトのフィールドがリアクティブソースになっているかチェックします。もしそうであれば、そのフィールドがソースになっているリアクションのうち最初に見つかったものが返されます。それ以外の場合、「none」が返されます。「/target」リファインメントはフィールドがソースではなくターゲットになっているかをチェックし、そのフィールドをターゲットにしている最初のリアクションか、該当するものがなければ「none」を返します。

2.4. clear-reactions

構文

```
clear-reactions
```

説明

全ての定義済みのリアクションを削除します。

2.5. dump-reactions

構文

```
dump-reactions
```

説明

登録済みのリアクションのリストを出力します。この機能はデバッグを目的としています。

3. リアクティブオブジェクト

通常のRedのオブジェクトはリアクティブな挙動は行いません。オブジェクトがリアクティブソースになるためには、以下のリアクターのプロトタイプのいずれかを使用して生成される必要があります。

3.1. reactor!

構文

```
make reactor! <body>

<body> : オブジェクトのボディブロック(block!)

戻り値: リアクティブオブジェクト
```

説明

ボディブロックを元に新しいリアクティブオブジェクトを生成します。生成されたオブジェクトでは、フィールドに新しい値をセットすると、そのフィールドに定義されたリアクションが起動されます。

NOTE:ボディブロックは「is」式を含むのが通常です。

3.2. deep-reactor!

構文

```
make deep-reactor! <body>

<body> : オブジェクトのボディブロック(block!)

戻り値: リアクティブオブジェクト
```

説明

ボディブロックを元にリアクティブオブジェクトを生成します。生成されたオブジェクトでは、フィールドに新しい値をセットするか、フィールドが参照するseriesを変更するとそのフィールドに定義され

たリアクションが起動されます。seriesはネストしているものも対象になります。

NOTE:ボディブロックは「is」式を含むのが通常です。

例

これはネストしたseriesへの変更がリアクションを起動することを示す例です。

NOTE:今のところ、リアクションが循環しないようにするのは実装者が意識する必要があります。たとえば、もし「deep-reactor!」がリアクターフォーミュラ（「is」など）内のseriesの値を変更すると、永久にリアクションの循環が起きてしまいます。

```
r: make 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
```