Bi-Directional Bindings

Janko Thyson

Thursday, October 16, 2014

Scenario 1: use references "as is" (no functional transformation/processing)

Scenario explanation

- Type/Direction:

`A` references `B` and `B` references `A` –> bidirectional binding type

- Binding/Relationship:

`A` uses value of `B` "as is" and `B` uses value of `A` "as is". This results in a **steady state**.

Example

A cool feature of this binding type is that you are free to alter the values of *both* objects and still keep everything "in sync"

```
setReactiveS3(id = "x_1", function() "object-ref: {id: x_2}")
setReactiveS3(id = "x_2", function() "object-ref: {id: x_1}")
```

Note that the call to `setReactiveS3()` merely initializes objects with bidirectional bindings to the value `numeric(0)`:

```
x_1
```

```
## NULL
```

```
x_2
```

```
## NULL
```

You must actually assign a value to either one of them via `<-` **after** establishing the binding:

```
## Set actual initial value to either one of the objects //
(x_1 <- 100)
```

```
## [1] 100
```

```
x_2
```

```
## [1] 100
```

```
x_1
```

```
## [1] 100
```

```
## Changing the other one of the two objects //
(x_2 <- 1000)
```

```
## [1] 1000
```

```
x_1
```

```
## [1] 1000
```

Clean up

```
## [1] TRUE
```

```
## [1] TRUE
```

Scenario 2: use functionally transformed/process value (steady state)

Scenario explanation

- Type/Direction:

`A` references `B` and `B` references `A` –> bidirectional binding type

- Binding/Relationship:

`A` uses transformed value of `B` and `B` uses transformed value of `A`.

The binding functions used result in a **steady state**.

Example

As the binding functions are "inversions"" of each other, we still get to a steady state.

```
setReactiveS3(id = "x_1", function() {
  "object-ref: {id: x_2}"
  x_2 * 2
})

setReactiveS3(id = "x_2", function() {
  "object-ref: {id: x_1}"
  x_1 / 2
})
```

Note that due to the structure of the binding functions, the visible object values are initialized to `numeric()` instead of `NULL` now.

```
x_1
```

```
## numeric(0)
```

```
x_2
```

```
## numeric(0)
```

Here, we always reach a steady state, i.e. a state in which cached values can be used instead of the need to executed the binding functions.

```
## Set actual initial value to either one of the objects //
(x_1 <- 100)
```

```
## [1] 100
```

```
x_2
```

```
## [1] 50
```

```
x_1
```

```
## [1] 100
```

```
## Changing the other one of the two objects //
(x_2 <- 1000)
```

```
## [1] 1000
```

```
x_1
```

```
## [1] 2000
```

```
x_2
```

```
## [1] 1000
```

Clean up

```
## [1] TRUE
```

```
## [1] TRUE
```

Scenario 3: use functionally transformed/process values (no steady state)

Scenario explanation

- Type/Direction:

`A` references `B` and `B` references `A` –> bidirectional binding type

- Binding/Relationship:

`A` uses transformed value of `B` and `B` uses transformed value of `A`.

The binding functions used result in a **non-steady state**.

Example

As the binding functions are **not** "inversions"" of each other, we never reach/stay at a steady state. Cached values are/can never be used as by the definition of the binding functions the two objects are constantly updating each other.

```
setReactiveS3(id = "x_1", function() {
  "object-ref: {id: x_2}"
  x_2 * 2
})

setReactiveS3(id = "x_2", function() {
  "object-ref: {id: x_1}"
  x_1 * 10
})
```

Here, we have "non-steady-state" behavior, i.e. we never reach a state were cached values can be used. We always need to execute the binding functions as each request of a visible object value results in changes.

This is best verified when using `verbose = TRUE` and comparing it to the other scenarios (not done at this point).

```
x_1
```

```
## numeric(0)
```

```
x_2
```

```
## numeric(0)
```

```
## Set actual initial value to either one of the objects //
(x_1 <- 1)
```

```
## [1] 1
```

```
x_2
```

```
## [1] 10
```

```
## --> `x_1` * 10

x_1
```

```
## [1] 20
```

```
## --> x_2 * 2

x_2
```

```
## [1] 200
```

```
## --> `x_1` * 10

## Changing the other one of the two objects //
(x_2 <- 1)
```

```
## [1] 1
```

```
x_1
```

```
## [1] 2
```

```
x_2
```

```
## [1] 20
```

```
x_1
```

```
## [1] 40
```

Clean up

```
## [1] TRUE
```

```
## [1] TRUE
```