Události interpreta

Table of Contents

1. Koncept

Jako podporu pro psaní 'debuggers, profilers, analyzers' a dalších užitečných nástrojů, potřebných pro efektivní programování, nabízí interpret generické 'low-level event-oriented' API, jímž lze všechny tyto nástroje sestavit. Je to podobný přístup jako u `parse/trace` a `lexer/trace` instrumentace.

2. Instrumentace interpreta

Aby byl umožněn přístup k interním stavům interpreta, generuje tento interpret události v klíčových bodech interního procesu vyhodnocení kódu. Tyto události mohou být odchyceny použitím uživatelem zadané 'callback' funkce. Události jsou emitovány pouze tehdy, je-li v interpretu povolen *tracing* režim s použitím upřesnění (refinement) `/trace` u funkce `do`.

```
do/trace <code> <callback>

<code>     : blok vyhodnocovaného kódu ve sledovacím režimu (tracing mode)
```

```
<callback> : funkce odezvy (callback), spuštěné při každé události interpreta
```

Příklad

```
logger: function [
    event  [word!]                  ;-- název události
    code   [any-block! none!]       ;-- aktuálně vyhodnocovaný blok
    offset [integer!]               ;-- offset ve vyhodnocovaném bloku
    value  [any-type!]              ;-- aktuálně pojednávaná hodnota
    ref    [any-type!]              ;-- reference aktuálního volání (slovo nebo cesta)
    frame  [pair!]                  ;-- 'start/top' pozice paměť. zásobníku (stack
frame)
][
    print [
        pad uppercase form event 8
        mold/part/flat either any-function? :value [:ref][:value] 20
    ]
]

do/trace [print 1 + 2] :logger
```

vytvoří výstup:

```
INIT    none                    ;-- initializing tracing mode
ENTER   none                    ;-- entering block to evaluate
FETCH   print                   ;-- fetching and evaluating `print` value
OPEN    print                   ;-- results in opening a new call stack frame
FETCH   +                       ;-- fetching and evaluating `+` infix operator
OPEN    +                       ;-- results in opening a new call stack frame
FETCH   1                       ;-- fetching left operand `1`
PUSH    1                       ;-- pushing integer! value `1` on stack
FETCH   2                       ;-- fetching and evaluating right operand
PUSH    2                       ;-- pushing integer! value `2`
CALL    +                       ;-- calling `+` operator
RETURN  3                       ;-- returning the resulting value
CALL    print                   ;-- calling `print`
3                               ;-- outputting 3
RETURN  unset                   ;-- returning the resulting value
EXIT    none                    ;-- exiting evaluated block
END     none                    ;-- ending tracing mode
```

3. Události

Je-li aktivní 'tracing mode', spustí interpret níže popsané události (events). Tyto lze uspořádat do následujících kategorií:

- Globální události: `INIT`, `END`

- Vyhodnocení entity block/paren/path kódu: ENTER, EXIT
- Volání libovolné funkce: OPEN, CALL, RETURN
- Vyhodnocení bloku těla funkce: PROLOG, EPILOG
- Vyhodnocení výrazu: FETCH, PUSH , SET, EXPR, ERROR
- Ošetření výjimek: THROW, CATCH

Podrobný popis:

Event	Code	Offset	Value	Ref	Popis
INIT	none	-1	n/a (none)	n/a (none)	je-li iniciován 'tracing mode' (volání do/trace).
END	none	-1	n/a (none)	n/a (none)	je-li 'tracing mode' ukončen (do/trace call exiting).
ENTER [1]	any-block!	-1	n/a (none)	n/a (none)	má-li být vyhodnocen blok.
EXIT [1]	any-block!	-1	last any-type! evaluated value	n/a (none)	bylo-li dosaženo chvostu (tail) aktuálně vyhodnocovaného bloku nebo cesty.
OPEN	any-block!	integer!	any-function!	word!, path!	je-li iniciováno volání nové funkce (any-function!) a je otevřen nový zásobník (stack frame).
CALL	any-block!	integer!	any-function! to call	word!, path!, any-function!	je volána funkce pro všechny argumenty, převzaté ze zásobníku.
RETURN	any-block!	integer!	returned any-type! value	word!, path!	bylo-li vráceno volání funkce a stack frame byl uzavřen.
EXPR	any-block!	integer!	expression any-type! result	n/a (none)	byl-li 'top-level' výraz vyhodnocen.
PROLOG	any-block!	-1	called function! value	word!, path!	při vstupování do těla funkce.
EPILOG	any-block!	-1	called function! value	word!, path!	při opouštění těla funkce.
FETCH	any-block!	integer!	fetched any-type! value	n/a (none)	hodnota je přečtena ze vstupu vyhodnocovaného bloku.

Event	Code	Offset	Value	Ref	Popis
PUSH	any-block!	integer!	pushed any-type! value	n/a (none)	hodnota byla vložena do zásobníku (stack frame).
SET	any-block!	integer!	any-type!	set-word!, set-path!	entitě set-word nebo set-path je zadána hodnota.
ERROR	none	-1	error! value	n/a (none)	vyskytne-li se chyba a má být vyřazena ze zásobníku.
THROW	none	-1	thrown any-type! value	n/a (none)	při vyřazení hodnoty s použitím nativní funkce throw.
CATCH	none	-1	thrown any-type! value	n/a (none)	při odchycení hodnoty s použitím nativní funkce catch.

[1] Vězte že události enter a exit jsou generovány při jakémkoli vyhodnocení cesty (stejně jako u jiných hodnot typu block.

Události jsou doprovázeny extra informacemi:

- code: pokud se vyskytuje, poskytuje vstupní aktuálně interpretované řady typu block! či paren!.
- offset: odlišná hodnota od -1 indikuje ofset vstupí řady v okamžiku události.
- value: je-li přítomná, jde o aktuálně zpracovávanou hodnotu.
- ref: je-li přítomno, odkazuje na slovo či cestu, z nichž vyhodnocení vyprodukovalo aktuální 'event/value'.

4. Ovladač událostí

Zde je prototyp ovladače událostí, vhodného pro zadání jako argumentu funkce do/trace:

```
func [
    event  [word!]
    code   [any-block! none!]
    offset [integer!]
    value  [any-type!]
    ref    [any-type!]
    frame  [pair!]
][
    [events]                    ;-- optional restricted event names list
    ...body...
]
```

Argument	Popis
event	Jméno události.

Argument	Popis
`code`	Blok aktuálně vyhodnocovaného kódu.
`offset`	Offset v aktuálně vyhodnocovaném bloku.
`value`	Hodnota, aktuálně zpracovávaná v události.
`ref`	Reference volání (word or path) spojeného s událostí.
`frame`	Dvojice indexů v interním zásobníku Redu, označující začátek a konec 'call frame'. [1]

[1] Vězte, že `frame index range` je pro interní zásobník Redu, nikoliv pro zásobník debuggeru, který je řízen debuggerem samotným.

Blok těla (funkce?) může začínat volitelným filtrovacím blokem k označení událostí, které budou spuštěny. To umožňuje redukovat počet odezev (callback calls), což má za následek mnohem lepší procesní výkon.

5. Řízené sledovaní

5.1. Funkce `trace`

Syntax

```
    trace <mode>

    <mode>: new event generation mode (logic!)
```

Popis

Funkce `trace` umožňuje zapnout/vypnout generování události během sledovaného vyhodnocení (uvnitř vyhodnocovaného kódu příkazem `do/trace`). Použití `trace` tímto způsobem mimo sledované vyhodnocení nemá žádný účinek. Funkci `trace` lze použít i jiným způsobem, jak je popsáno v části "Předdefinované nástroje".

Příklad

```
do/trace [
    print "before"
    trace off
    print "between"
    trace on
    print "after"
] :logger
```

vytvoří výstup

```
INIT     none
ENTER    none
FETCH    print
OPEN     print
FETCH    "before"
PUSH     "before"
CALL     print
before
RETURN   unset
FETCH    trace
OPEN     trace
FETCH    off
PUSH     false
CALL     trace                ;-- calling `trace off`
between                       ;-- only `print` output but no related events
RETURN   true                 ;-- next event is the return of `trace on`
FETCH    print
OPEN     print
FETCH    "after"
PUSH     "after"
CALL     print
after
RETURN   unset
EXIT     unset
END      none
```

5.2. Atributy funkce

Každá funkce, volaná během sledovaného vyhodnocení, může být nastavena na to aby negenerovala žádnou událost nebo být nucena generovat události, i když je generování události vypnuto. Toho lze dosíci s použitím následujících atributů funkce.

5.2.1. trace

Syntax

```
    func [[trace]...][...]
```

Popis

Při použití tohoto atributu je funkce nucena generovat události při sledovaném vyhodnocení. To se také vztahuje na vnořená volání (pokud explicitně nevypínají generovaní událostí). Pamatujte si, že je stále možné vypnout/zapnout generování událostí funkcí `trace`.

Příklad

```
foo: func [[trace]][1 + 2]
```

```
do/trace [
    trace off
    print "before"
    foo
    print "after"
    trace on
] :logger
```

vytvoří výstup

```
INIT     none
ENTER    none
FETCH    trace
OPEN     trace
FETCH    off
PUSH     false
CALL     trace
before                          ;-- no related events for `print "before"`
PROLOG   foo                    ;-- events enabled from beginning of `foo` body
evaluation
ENTER    none
FETCH    +
OPEN     +
FETCH    1
PUSH     1
FETCH    2
PUSH     2
CALL     +
RETURN   3
EXIT     3
EPILOG   foo                    ;-- event generation stopped again when `foo` exits
after                           ;-- no related events for `print "after"`
RETURN   true                   ;-- next event is the return of `trace on`
EXIT     true
END      none
```

5.2.2. no-trace

Syntax

```
    func [[no-trace]...][...]
```

Popis

Při použití tohoto atributu bude generování událostí během sledovaného vyhodnocování blokováno. To se také vztahuje na vnořená volání (pokud explicitně nevypínají generovaní událostí). Pamatujte si, že je stále možné vypnout/zapnout generování událostí funkcí `trace`.

Příklad

```
foo: func [[no-trace]][print 1 + 2]
do/trace [print "before" foo print "after"] :logger
```

vytvoří výstup

```
INIT     none
ENTER    none
FETCH    print
OPEN     print
FETCH    "before"
PUSH     "before"
CALL     print
before
RETURN   unset
FETCH    foo
OPEN     foo
CALL     foo                    ;-- last event before entering `foo`
3                               ;-- no event generated from inside `foo`
RETURN   unset                  ;-- next event is the return from `foo`
FETCH    print
OPEN     print
FETCH    "after"
PUSH     "after"
CALL     print
after
RETURN   unset
EXIT     unset
END      none
```

6. Předdefinované nástroje

V knihovně pro Red runtime je vždy k disposici několik ovladačů, pomáhajících uživatelům lépe analyzovat a ladit programy Redu.

6.1. debug

Syntax

```
    debug <code>
    debug/later <code>

    <code> : code to evaluate through the debugger (any-type!)
```

Popis

Spouští interaktivní ladící (debugging) seanci, umožňující vyhodnotit argument `kódu` kontrolovaným způsobem. Ladící konzola je prezentována promptem `debug>_`, čekajícím na příkazy uživatele (viz seznam níže).

Upřesnění (refinement) `/later` zajišťuje nepřerušené vyhodnocování, dokud se nenarazí na hodnotu `@stop`, která aktivuje ladící konzolu. V normálním vyhodnocování lze pokračovat zadáním hodnoty `@go` (spolu s ladícím příkazem `continue`). Hodnota `@stop` působí vlastně jako přerušení. Použití hodnot `@stop` a `@go` umožňuje selektivní vyhodnocování krok za krokem pouze ve zvolených částech kódu.

Seznam ladících příkazů:

Příkaz	**Zkratka**	**Popis**
`help`	`?`	Vytisknout seznam ladicích příkazů.
`next`	`n` or ENTER key	Další krok vyhodnocení.
`continue`	`c`	Opustit ladící konzolu ale pokračovat ve vyhodnocování.
`quit`	`q`	Opustit ladící konzolu a ukončit vyhodnocení.
`stack`	`s`	Zobrazit aktuální lokální invokace a zásobník výrazů.
`parents`	`p`	Zobrazit rodičovský zásobník invokací (call stack).
`:word`	n/a	Vrací hodnotu entity `word`. Je-li to objekt typu `function!`, vrací lokální kontext.
`:a/b/c`	n/a	Vrací hodnotu cesty `a/b/c`.
`watch <word1> <word2>⋯`	`w ⋯`	Sleduje jedno či více slov.
`-watch <word1> <word2>⋯`	`-w ⋯`	Ukončí sledování jednoho či více slov.
`+stack`	`+s`	Vrací zásobník výrazů pro každou novou událost.
`-stack`	`-s`	Nevrací zásobník výrazů pro každou novou událost.
`+locals`	`+l`	Vrací lokální kontext pro každý vstup v zásobníku invokací (call stack).
`-locals`	`-l`	Nevrací lokální kontext pro každý vstup v zásobníku invokací.
`+indent`	`+i`	Odsadí výstup zásobníku výrazů (expression stack).
`-indent`	`-i`	Neodsadí výstup zásobníku výrazů.

Je-li nastaven interaktivní režim, vrací ladící konzola sadu kontextuálních informací pro každý krok. Zde je krátký popis:

Typický výstup:

```
debug> n                  ;-- poslední ladící příkaz (`next`)
-----> EVAL n             ;-- následující krok: vyzvednutí a vyhodnocení `n`
Input: n < 1              ;-- aktuálně vyhodnocovaný kód
Stack: print              ;-- dno aktuálního (lokálního) zásobníku invokací (call
stack)
Stack: fibo               ;-- invokace funkcí i jejich argumenty jsou zobrazeny v
zásobníku (stack = štos)
Stack: 4
Stack: either
Stack: <                  ;-- stack top
```

Tento zásobník je označen jako "zásobník výrazů" pro odlišení od "rodičovského zásobníku", který se vztahuje na call stack před voláním debuggeru. Je-li zásobník výrazů prázdný, zobrazí se označení `-empty stack-` pod řádkem `Input:`.

6.2. `profile`

Syntax

```
    profile <code>
    profile/by <code> <category>

    <code>     : code to profile (any-type!)
    <category> : sort by a specific category: 'name, 'count, 'time (word!)
```

Popis

Profiluje zadaný kód, počítaje invokace funkcí a délky měření. Po vyhodnocení kódu se tiskne zpráva. Implicitní třídění je podle počtu invokací. Alternativní třídění lze zadat upřesněním `/by`. Příkaz `profile` přijímá stejné argumenty jako příkaz `do`.

Poznámky:

- Časování není aktuálně na platformě Windows příliš přesné pro trvání kratší než 20 ms. *This will be improved in the future with better timers and functions prolog/epilog more accurate exclusions.*
- Trvání vložených funkcí je aktuálně přičteno k trvání rodičovských funkcí. *Proper function timing (excluding nested calls) will be added in the future.*
- Invokace funkcí s upřesněním jsou hodnoceny odděleně jako specifické instance funkcí *(same refinements in different order will be counted separately too currently).*

Volby

Příkaz `profile` lze implicitně použít pro jakýkoliv typ funkce (typeset (`any-function!`). Argumenty lze také omezit na podskupinu přímou modifikací volby `system/tools/options/profile/types` (_

setting it to a different typeset_).

Příklady

```
profile [print 1 + 2 + 3 * 5]

30
#1   +               | 2          | 0:00:00
#2   *               | 1          | 0:00:00
#3   print           | 1          | 0:00:00.001
```

Soubory a adresy URL lze jako argument zadat přímo:

```
profile https://raw.githubusercontent.com/red/red/master/tests/demo.red

        RedRed              d
        d     d             e
        e     e             R
        R     R   edR    dR d
        d     d  d   R  R  Re
        edRedR   e   d  d   R
        R   e    RedR   e   d
        d    e   d      R   e
        e    R   e   d  d  dR
        R     R   edR    dR d
#1   if              | 420        | 0:00:00
#2   <=              | 391        | 0:00:00
#3   prin            | 241        | 0:00:00.240773
#4   +               | 220        | 0:00:00
#5   either          | 210        | 0:00:00
#6   all             | 210        | 0:00:00.0028192
#7   >               | 210        | 0:00:00.0020021
#8   =               | 210        | 0:00:00.0010021
#9   tail?           | 37         | 0:00:00
#10  unless          | 37         | 0:00:00
#11  skip            | 37         | 0:00:00
#12  repeat          | 10         | 0:00:00.212984
#13  next            | 10         | 0:00:00
#14  foreach         | 1          | 0:00:00.251109
```

Volby pro výběr zpracovávané sady funkcí lze modifikovat předem:

```
system/tools/options/profile/types: make typeset! [op!]
profile https://raw.githubusercontent.com/red/red/master/tests/demo.red

        RedRed              d
        d     d             e
        e     e             R
```

```
        R     R   edR    dR d
        d     d  d   R  R  Re
        edRedR   e   d  d   R
        R   e    RedR   e   d
        d    e   d      R   e
        e    R   e   d  d  dR
        R     R   edR    dR d
#1   <=               | 391        | 0:00:00.0000038
#2   +                | 220        | 0:00:00
#3   >                | 210        | 0:00:00
#4   =                | 210        | 0:00:00.0010005
```

6.3. trace

Syntax

```
    trace <code>
    trace <mode>
    trace/raw <code>

    <code> : code to trace (any-type!)
    <mode> : turn tracing on/off (logic!)
```

Popis

Generuje prostý sled kroků při vyhodnocení argumentu. Zobrazí se pouze následující podmnožina všech možných událostí interpreta: `open push call prolog epilog set return error catch throw`. *In order to display a lower level trace with all the events, a* `/raw` *refinement is provided. The output then just dumps the following information for each event: event name, offset, reference, value, frame range (basically the event handler arguments, except for the* `code` *argument).*

Je-li jako argument funkce `trace` zadána hodnota typu `logic!`, dojde pouze k přepnutí sledování (tracing) na `on/off`, což umožňuje těsnější kontrolu zevnitř vyhodnocení sledovaného kódu.

Volby

Výstupní trace je implicitně u vnořeních invokací odsazen. Je možné provést "flat trace" nastavením volby `system/tools/options/trace/indent?` na `false`.

Examples

```
trace [a: 1 + 2]
```

má výstup

```
-> PUSH a:
-> OPEN +
```

```
->  PUSH 1
->  PUSH 2
->  CALL op!  (+)
-> RETURN 3  (+)
-> SET 3  (a)
== 3
```

S použitím upřesnění `/raw`:

```
trace/raw [a: 1 + 2]
```

will output

```
INIT -1 none none 36x38
ENTER 0 none none 38x38
FETCH 0 none a: 38x38
PUSH 0 none a: 38x39
FETCH 1 none + 38x39
OPEN 1 none + 38x39
FETCH 1 none 1 39x39
PUSH 1 none 1 39x40
FETCH 3 none 2 39x40
PUSH 3 none 2 39x41
CALL 4 + make op! [["Returns the sum of 39x41
RETURN 4 + 3 39x41
SET 4 a: 3 38x40
EXPR 4 none 3 38x39
EXIT 4 none 3 38x39
END -1 none none 36x39
== 3
```

6.4. `tracing?`

Syntax

```
    <state>: tracing?

    <state> : vrací aktuální režim generace událostí (logic!)
```

Popis

Hlásí stav aktuální tvorby událostí interpreta (`true` or `false`).

Příklad

```
foo: func [[no-trace]][probe tracing? print 1 + 2]
```

```
no-log: func [e c o v r f][]
do/trace [probe tracing? foo probe tracing?] :no-log
```

má výstup

```
true
false
3
true
```

7. Implementační poznámky

7.1. Obecně

minimální kód

7.2. Profiler improvements

7.3. Stack access