cl-hooks README

Jan Moringen

07 November 2010

Contents

1 Introduction 1

2 Hooks 1
- 2.1 Definition . . . 1
- 2.2 Variable Hooks . . . 2
- 2.3 Internal Object Hooks . . . 2
- 2.4 External Object Hooks . . . 3
- 2.5 Hook Combination . . . 3

3 Tracking State 4

4 Restarts 4
- 4.1 Hook Restarts . . . 4
- 4.2 Handler Restarts . . . 4

5 Convenience Marcos 4

1 Introduction

A hook, in the present context, is a certain kind of extension point in a program that allows interleaving the execution of arbitrary code with the execution of a the program without introducing any coupling between the two. Hooks are used extensively in the extensible editor Emacs.

In the Common LISP Object System (CLOS), a similar kind of extensibility is possible using the flexible multi-method dispatch mechanism. It may even seem that the concept of hooks does not provide any benefits over the possibilites of CLOS. However, there are some differences:

- There can be only one method for each combination of specializers and qualifiers. As a result this kind of extension point cannot be used by multiple extensions independently.
- Removing code previously attached via a :before, :after or :around method can be cumbersome.
- There could be other or even multiple extension points besides :before and :after in a single method.
- Attaching codes to individual objects using eql specializers can be cumbersome.
- Introspection of code attached a particular extension point is cumbersome since this requires enumerating and inspecting the methods of a generic function.

This library tries to complement some of these weaknesses of method-based extension-points via the concept of hooks.

2 Hooks

2.1 Definition

A hook is an extension point consisting of the following pieces:

- A name (a symbol or some form)
- A list of handlers
- ftype?

- A result combination
- A documentation string

There are several kinds of hooks defined in this library, but new kinds of hooks can easily be defined by adding methods to the generic functions that form the hook protocol:

- `hook-handlers`
- `(setf hook-handlers)`,
- `hook-combination`
- `(setf hook-combination)`
- `documentation`
- `(setf documentation)`

The following sections briefly discuss the kinds of hooks that are currently defined in the library.

2.2 Variable Hooks

The most straightforward approach to implementing a hook is to use a variable. The variable is used as followed

Symbol Name name of the hook

Symbol Value list of handlers currently attached to the hook

Symbol Documentation if no dedicated hook documentation is installed using `(setf (hook-documentation ...) ...)`, the documentation of the symbol as a variable is used

Consider the following example

```
(defvar *my-hook* nil
  "My␣hook␣is␣only␣run␣for␣educational␣purposes.")

(hooks:add-to-hook '*my-hook*
                   (lambda (x)
                     (format t "my-hook␣called␣with␣argument␣~S~%" x)))

(hooks:run-hook '*my-hook* 1)
```

```
NIL
```

```
(documentation '*my-hook* 'hooks::hook)
```

```
My hook is only run for educational purposes.
```

2.3 Internal Object Hooks

Hooks can also live in other places like object slots:

```
(defclass my-class ()
  ((my-hook :initarg  :my-hook
            :type     list
            :initform nil
            :documentation
            "This␣hook␣bla␣bla")))

(defvar *my-object* (make-instance 'my-class))

(hooks:object-hook *my-object* 'my-hook)
```

```
#<OBJECT-HOOK MY-HOOK PROGN (0) {C11AEF9}>
```

Operation on an intern object hook work in the usual way:

```
(hooks:add-to-hook (hooks:object-hook *my-object* 'my-hook)
                   (lambda (x)
                     (format t "my-hook␣called␣with␣argument␣~S~%" x)))

(hooks:object-hook *my-object* 'my-hook)
```

```
#<OBJECT-HOOK MY-HOOK PROGN (1) {C11AEF9}>
```

```
(format t "bla~%")
(hooks:run-hook (hooks:object-hook *my-object* 'my-hook) 1)
```

```
NIL
```

For object internal hooks, the documentation of the backing slot is used as the hook's documentation:

```
(documentation (hooks:object-hook *my-object* 'my-hook) 'hooks::hook)
```

```
This hook bla bla
```

2.4 External Object Hooks

Or outside of objects:

```
(defparameter *external-hook* (hooks:external-hook *my-object* 'my-external-hook))

*external-hook*
```

```
#<EXTERNAL-HOOK PROGN (0) {AE0AB49}>
```

We stored the hook object in a variable since we are going to use it in some other examples.

```
(hooks:add-to-hook *external-hook*
                   (lambda (x)
                     (format t "my-external-hook called with argument ~S~%" x)))

(hooks:run-hook *external-hook* 1)
```

```
NIL
```

2.5 Hook Combination

Hook combination refers to the different possible way of constructing the resulting value of running a hook. While bearing a strong resemblance to method combination in CLOS namewise, hook combination is a much more restricted and less powerful concept.

The default hook combination is `progn`:

```
(hooks:hook-combination (hooks:external-hook *my-object* 'my-external-hook))
```

```
PROGN
```

`progn` hook combination means the final result is the return value of the handler run last: TODO

Let's set up the hook to test some other combinations

```
(hooks:clear-hook *external-hook*)
(hooks:add-to-hook *external-hook* #'(lambda (x) (mod x 5)))
(hooks:add-to-hook *external-hook* #'(lambda (x) (- x)))
```

```
(#<FUNCTION (LAMBDA #) {AEF407D}> #<FUNCTION (LAMBDA #) {AECCDD5}>)
NIL
```

Combination using `list`

```
(setf (hooks:hook-combination *external-hook*) #'list)

(list
 (hooks:run-hook *external-hook* -3)
 (hooks:run-hook *external-hook* 1)
 (hooks:run-hook *external-hook* 7))
```

```
3   2
-1  1
-7  2
```

Combination using `max`

```
(setf (hooks:hook-combination *external-hook*) #'max)

(list
 (hooks:run-hook *external-hook* -3)
 (hooks:run-hook *external-hook* 1)
 (hooks:run-hook *external-hook* 7))
```

3 1 2

Note:

Some functions can be used for hook combination, but will not work as expected in all cases. `max` is one such examples. Running a hook with `max` hook combination that does not have any handlers will result in an error because `max` cannot be called without any arguments (which is the result of calling zero handlers).

3 Tracking State

```
(defmethod hooks:on-become-active :after ((hook t))
  (format t "hook ~S is now active~%" hook))

(defmethod hooks:on-become-inactive :after ((hook t))
  (format t "hook ~S is now inactive~%" hook))

(setf *my-object* (make-instance 'my-class))

(hooks:add-to-hook (hooks:object-hook *my-object* 'my-hook) (lambda (x)))

(setf (hooks:hook-handlers (hooks:object-hook *my-object* 'my-hook)) nil)
```

NIL

4 Restarts

This library uses restart to recover from errors during the execution of hooks or their handlers. This section briefly discusses the restarts that are installed at the hook and handler levels.

4.1 Hook Restarts

`retry` When this restart is invoked, the hook is ran again.

`use-value` When this restart is invoked, the hook is not ran and a replacement value is read interactively and returned in place of the result of running the hook.

4.2 Handler Restarts

`retry` When this restart is invoked, the handler is executed again.

`use-value` When this restart is invoked, the handler is not executed and a replacement value is read interactively and returned in place of the result of executing the handler.

`skip` When this restart is invoked, the handler is skipped without producing any return value. If there are other handlers, the hook may still produce a return value.

5 Convenience Marcos

```
(hooks:with-handlers
    (((hooks:external-hook *my-object* 'my-hook)
      (lambda (x)))

     ((hooks:external-hook *my-object* 'my-other-hook)
      (lambda (y z))))
  (hooks:run-hook (hooks:external-hook *my-object* 'my-hook)))
```