backtrace

Warren Wilkinson

December 28, 2012

Contents

Overview

It would be nice if your webserver gave you a backtrace when it went "kaput". This library captures backtraces so you can log or email them.

It works by creating a 'backtrace' restart that prints to a stream. If there is a problem, you can log the backtrace.

```
(defun main-or-thread-main (args)
  (with-printable-backtrace (*logging*)
     (do-something-important)))
```

Features

- Shows files and file positions (when it can)
- Show local variables (use :variables t argument)
- Customizable depth (use :depth N argument)

Limitations

- **It only works on SBCL.**
- Invoking the restart is left to the end-user. A simple example:

```
(handler-bind ((error-you-want-backtraces-for
                #'(lambda (c)
                    (invoke-restart 'backtrace:backtrace)
                    (error c))))
  (with-printable-backtrace (*logging-stream* :depth 6 :variables t)
    (do-something-important)))
```

  You might choose something more sophisticated, like logging for some users but sending developers to the debugger.

- The restart is always 'backtrace:backtrace. For now, you can't give them unique names.

Installation

Manual Installation

In summary: Untar the .tar package and then symlink the .asd files into a place where ASDF can find them.

1. Untar the files where you want them to be. On windows download the .zip and unzip it instead, it's the same files.
2. ASDF could be looking anywhere – it depends on your setup. Run this in your lisp repl to get a clue as to where ASDF is seeking libraries[1]:

   ```
   (mapcan #'funcall asdf:*default-source-registries*)
   ```

3. Symlink the .asd files to the source directory. If you use windows, these instructions on symlink alternatives apply to you.

[1] you might need to (require 'asdf) before running this example

Once the files are in place, the package can be loaded with ASDF by:

```
(asdf:operate 'asdf:load-op :backtrace)
```

If you have problems, see the support section. If you don't have problems you may want to run the tests anyway, because you can.

Running the Tests

Once the system is loaded, it can be tested with asdf.

```
(asdf:operate 'asdf:test-op :backtrace)
```

This should display something like the following. There should be **zero failures**, if you have failures see the support section of this document.

```
RUNNING BACKTRACE TESTS...
BACKTRACE TEST RESULTS:
     Tests: 9
   Success: 9
  Failures: 0
```

Getting Support

You can email Warren Wilkinson, or look at the github repository.

Implementation

The macro "with-printable-backtrace" creates

1. A restart case that will print the backtrace
2. A handler to grab the current backtrace.

We need both parts; if the stack unwinds to invoke the restart then the backtrace is lost. The handler grabs it before that can happen.

```
(defmacro with-printable-backtrace ((&optional stream &key (depth 5) variables) &body body)
  (let ((backtrace (gensym)))
    `(let ((,backtrace nil))
       (restart-case
           (handler-bind
               ((error (lambda (condition)
                         (declare (ignore condition))
                         (setf ,backtrace
                               (nthcdr 3
                                 (loop for i from 0 upto (+ 2 ,depth)
                                       for frame = (sb-di:top-frame) then (sb-di:frame-down frame)
                                       collect frame)))
                         nil)))
             ,@body)
         (backtrace ()
           :report (lambda (r) (format r "Print ~a-level backtrace to ~a with~:[out~;~] variables"
```

```
                          ,depth ',stream ,variables))
          (print-backtrace ,stream ,variables ,backtrace)
          nil)))))
```

There magic numbers (nthcdr 3 ...) and (+ 2 depth) are to skip backtrace items related to fetching the backtrace. The handler returns nil to *decline* the error, meaning other handlers are free to attempt fixing it.

All other functions support this macro, they interface low level SBCL routines. Many of them are derived from watching how Swank does it.

```
(defun call-signature (frame)
  (with-output-to-string (a)
    (sb-debug::print-frame-call frame a)))

(defun print-variable (stream arg colonp atsignp)
  (declare (ignore colonp atsignp))
  (format stream "~a = ~a" (first arg) (second arg)))

(defun print-backtrace (stream variables frames &aux (i 0))
  (dolist (frame frames)
    (format stream "~%~d. ~a~%   - SOURCE: ~s~{~%    (with) ~/backtrace:print-variable/~}"
            i
            (call-signature frame)
            (source-location frame)
            (and variables (frame-variables frame)))
    (incf i)))
```

Fetching the frame variables is tricky since they are complex objects.

```
(flet ((frame-vars (frame) ;; adapted from swank
         (ignore-errors (sb-di::debug-fun-debug-vars (sb-di:frame-debug-fun frame))))
       (debug-var-value (var frame location) ;; adapted from swank
         (ecase (sb-di:debug-var-validity var location)
           (:valid (sb-di:debug-var-value var frame))
           ((:invalid :unknown) ':<not-available>))))
  (defun frame-variables (frame) ;; adapted from swark
    (let ((loc (sb-di:frame-code-location frame))
          (vars (frame-vars frame)))
      (when vars
        (loop for v across vars collect
             (list (sb-di:debug-var-symbol v) (debug-var-value v frame loc)))))))
```

Computing the line number is very difficult. The compiler doesn't keep track of them. Instead, it counts the top-level forms it sees, and

we can get that. The function file-line reads that many top-level-forms, and then counts the newlines in that space.

```
(defun file-line (file top-level-form-number)
  (with-open-file (s file)
    (dotimes (i top-level-form-number (sb-impl::flush-whitespace s))
      (read s))
    (let* ((position (file-position s))
           (upto-start (make-string position)))
      (file-position s 0)
      (read-sequence upto-start s)
      (1+ (count #\Newline upto-start)))))
```

Producing a human-readable source-location is hard because so much can go wrong. This function attempts to do so, with a focus on reliability. It doesn't try hard, but it works.

```
(defun source-location (frame)
  (let* ((loc (sb-di:frame-code-location frame))
         (dsource (sb-di:code-location-debug-source loc)))
    (aif (sb-di:debug-source-namestring dsource)
         (let ((truename (ignore-errors (truename it))))
           (if truename
               (concatenate 'string (namestring truename)
                            "@"
                            (or (ignore-errors
                                  (princ-to-string
                                   (file-line truename
                                              (sb-di::code-location-toplevel-form-offset
                                               (sb-debug::maybe-block-start-location loc)))))
                                "?"))
               it))
         (or (ignore-errors (sb-debug::code-location-source-form loc 100))
             "REPL, unknown location"))))
```

Tests

Mosts tests are simple regexes on the output of running (alpha 4 5). These test functions are in their own file, test-functions.lisp, so they have known line numbers.

```
(in-package :backtrace.test)
(defvar *fail-p* t)
(defun tertiary () ;; Line 4
  (declare (optimize debug))
```

```
  (if *fail-p* (error "In last") 4))

(defun beta (a) ;; Line 8
  (declare (optimize debug))
  (* (tertiary) a))

(defun alpha (a b) ;; Line 12
  (declare (optimize debug))
    (dotimes (i 4)
      (incf b (beta a))))
```

test	vars	depth	regex	notes
variables-off	nil	6	!with	When variables are disabled, we shouldn't see any.
variables-on	t	6	with	When enabled, we should see some.
tertiary-first	nil	1	!BETA	Tertiary should be first.
beta-second	nil	2	!ALPHA	Beta should be second
alpha-third	nil	3	ALPHA	Alpha should be second
tertiary-fp	nil	1	test-functions.lisp@4	Correct file location for tertiary
beta-fp	nil	2	test-functions.lisp@8	Correct file location for beta
alpha-fp	nil	3	test-functions.lisp@12	Correct file location for alpha

The only other test ensures that our restart is present.

```
(defun ensure-restart-exists-test ()
  "Test that the backtrace restart is created."
  (block nil
    (handler-bind ((error (lambda (c) (declare (ignore c)) (return (find-restart 'backtrace)))))
      (test-setup 5 nil))))
(pushnew 'ensure-restart-exists-test *all-tests*)
```

License

backtrace is distributed under the LGPL2 License.