io.aviso/pretty Documentation

Release 1.4.4

Howard M. Lewis Ship

Jun 20, 2023

Contents

1 Sometimes, neatness counts **3**

2 License **5**

2.1 ANSI Formatting 5
2.2 Formatted Exceptions 6
2.3 REPL 9
2.4 Leiningen Plugin 10
2.5 Binary Output 11
2.6 Columnar Output 12

CHAPTER 1

Sometimes, neatness counts

If you are trying to puzzle out a stack trace, pick a critical line of text out of a long stream of console output, or compare two streams of binary data, a little bit of formatting can go a long way.

That's what the **io.aviso/pretty** library is for. It adds support for pretty output where it counts:

- Readable output for exceptions
- ANSI font and background color support
- Hex dump of binary data
- Hex dump of binary deltas
- Formatting data into columns

Here's an example of pretty at work:

```
user=> (pst)
                          clojure.core/eval   core.clj: 2852
                                          ...
                             user/eval2007  REPL Input
                        user/make-exception   user.clj:   31
                            user/update-row   user.clj:   23
user/make-jdbc-update-worker/reify/do-work   user.clj:   18
                           user/jdbc-update   user.clj:    7
     java.sql.SQLException: Database failure
                            SELECT FOO, BAR, BAZ
                            FROM GNIP
                            failed with ABC123
     SQLState: "ABC"
    errorCode: 123
java.lang.RuntimeException: Failure updating row
java.lang.RuntimeException: Request handling exception
nil
user=>
```

CHAPTER 2

License

Pretty is released under the terms of the Apache Software License 2.0.

2.1 ANSI Formatting

The `io.aviso.ansi` namespace defines a number of functions and constants for producing ANSI escape codes.

Starting with 1.4, the `compose` function is the best way to construct text with ANSI escape codes:

```
user=> (println (compose [:yellow.black-bg "WARNING: " [:bold.white "Shields at " [:bright-red "3%"]]]))
WARNING: Shields at 3%
nil
```

`compose` uses a Hiccup inspired data structure to identify how different vector blocks of text should be formatted.

2.1.1 Constants and Functions

`compose` is built on top of a large number of underlying functions and constants.

ANSI supports eight named colors, each with a bright variant. For each of the supported colors (black, red, green, yellow, blue, magenta, cyan, and white) there will be four functions and four constants:

- [bright-] *color* - function to set foreground text color
- [bright-] *color*-bg - function to set background color
- [bright-] *color*-font - constant that enables the text color
- [bright-] *color*-bg-font - constant that enables the color as background

For example, for the color green there will be `green`, `green-bg`, `bright-green`, and `bright-green-bg` functions, and constants `green-font`, `green-bg-font`, `bright-green-font`, and `bright-green-bg-font`.

The functions are passed a string and wrap the string with ANSI codes to enable the specific font attributes, with a reset of all attributes after the string.

Note that the exact color interpretation of the ANSI codes varies significantly between platforms and applications, and is frequently configurable, often using themes. You may need to adjust your application's settings to get an optimum display.

In addition to color, text can be: * `bold`, `faint`, or `plain` * `italic` or `roman` * `inverse` (which inverts the foreground and background colors) or `normal` * `underlined` or `not-underlined`

For each of these, there is a function and a `-font` constant.

Finally, `reset-font` is a constant that reverts all font characteristics back to defaults.

2.2 Formatted Exceptions

Pretty's main focus is on formatting of exceptions for readability, addressing one of Clojure's core weaknesses.

2.2.1 Rationale

Exceptions in Clojure are extremely painful for many reasons:

- They are often nested (wrapped and rethrown)
- Stack traces are output for every exception, which clogs output without providing useful detail
- Stack traces are often truncated, requiring the user to manually re-assemble the stack trace from several pieces
- Many stack frames represent implementation details of Clojure that are not relevant

This is addressed by the `io.aviso.exception/write-exception` function; it take an exception and writes it to the console, `*out*`.

This is best explained by example; here's a SQLException wrapped inside two RuntimeExceptions, and printed normally:

```
user=> (throw (make-ex-info))
Execution error (SQLException) at user/jdbc-update (REPL:1).
Database failure
SELECT FOO, BAR, BAZ
FROM GNIP
failed with ABC123
user=> (pst)
SQLException Database failure
SELECT FOO, BAR, BAZ
FROM GNIP
failed with ABC123
      user/jdbc-update (NO_SOURCE_FILE:1)
      user/jdbc-update (NO_SOURCE_FILE:1)
      user/make-jdbc-update-worker/reify--169 (NO_SOURCE_FILE:5)
      user/update-row (NO_SOURCE_FILE:4)
      user/update-row (NO_SOURCE_FILE:1)
      user/make-exception (NO_SOURCE_FILE:5)
      user/make-exception (NO_SOURCE_FILE:1)
      user/make-ex-info (NO_SOURCE_FILE:5)
      user/make-ex-info (NO_SOURCE_FILE:1)
      user/eval175 (NO_SOURCE_FILE:1)
      user/eval175 (NO_SOURCE_FILE:1)
      clojure.lang.Compiler.eval (Compiler.java:7176)
nil
```

(continues on next page)

(continued from previous page)

```
user=> *clojure-version*
{:major 1, :minor 10, :incremental 0, :qualifier nil}
user=>
```

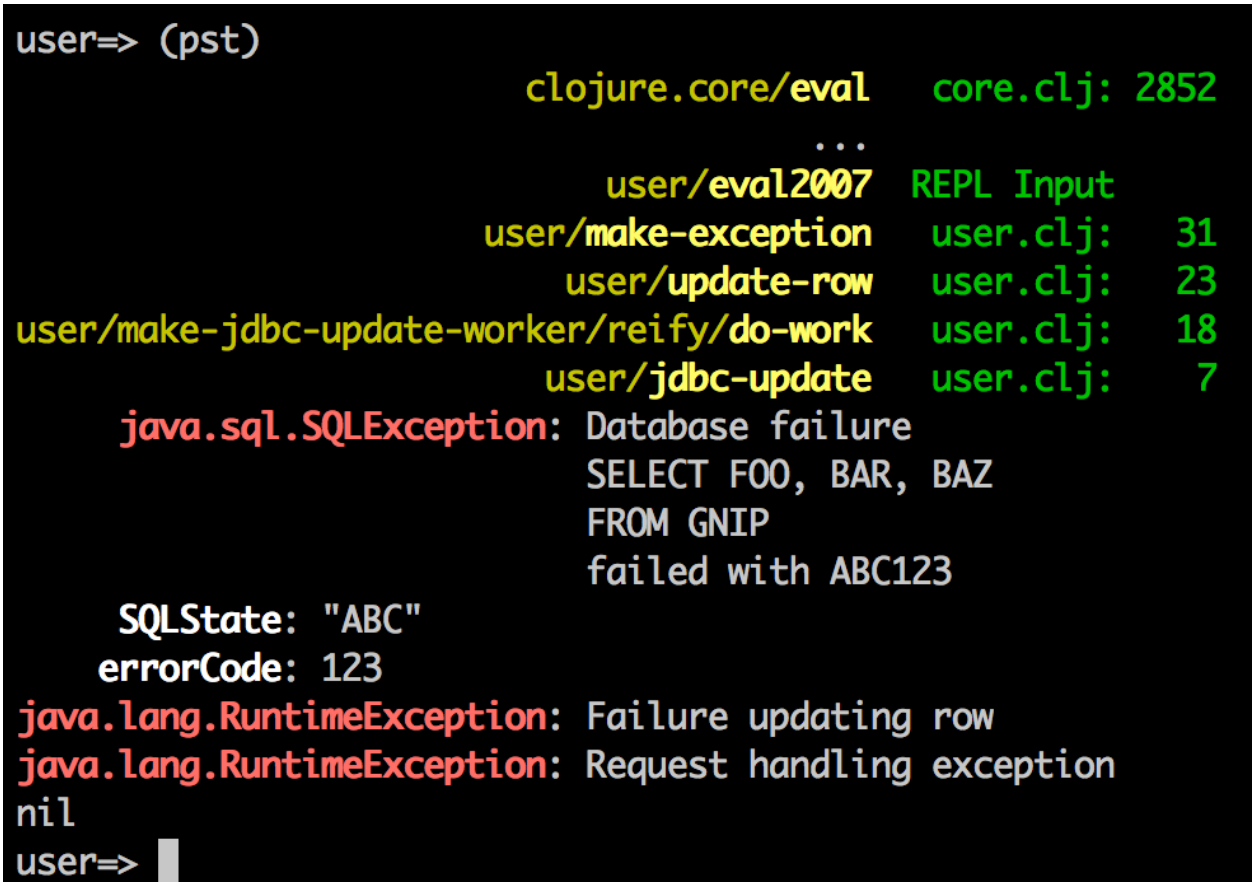
This is greatly improved in Clojure 1.10 over prior Clojure releases, but still quite minimal.

On a good day, the exception messages will include all the details you need to resolve the problem, which is strangely at odds with Clojure's `ex-info` function; `ex-info` encourages you to put useful information into the `ex-data` of the exception, yet Clojure doesn't print out this data. `write-exceptions` by contrast, does output the `ex-data`.

Meanwhile, you will have to mentally scan and parse the above text explosion, to parse out file names and line numbers.

It's one more bit of cognitive load you just don't need in your day.

Instead, here's the equivalent, using a *hooked* version of Clojure's `clojure.repl/pst`, modified to use `write-exception`.



As you can see, this lets you focus in on the exact cause and location of your problem. . `write-exception` flips around the traditional order, providing a chronologically sequential view:

- The stack trace leading to the root exception comes first, and is ordered outermost frame to innermost frame.
- The exception stack comes after the stack trace, and is ordered root exception (innermost) to outermost, reflecting how the stack has unwound, and the root exception was wrapped in new exceptions and rethrown.

The stack trace is carefully formatted for readability, with the left-most column identifying Clojure functions or Java class and method, and the right columns presenting the file name and line number.

The stack frames themselves are filtered to remove details that are not relevant. This filtering is via an optional function, so you can define filters that make sense for your code. For example, the default filter omits frames in the clojure.lang package (they are reduced to ellipses), and truncates the stack trace when when it reaches `clojure.main/repl/read-eval-print`.

Repeating stack frames are also identified and reduced to a single line (that identifies the number of frames). This allows your infinite loop that terminates with a StackOverflowException to be reported in just a few lines, not thousands.

The inverted (from Java norms) ordering has several benefits:

- Chronological order is maintained, whereas a Java stack trace is in reverse chronological order.
- The most relevant details are at (or near) the *bottom* not the *top*; this means less "scrolling back to see what happened".

The related function, `format-exception`, produces the same output, but returns it as a string.

For both `format-exception` and `write-exception`, output of the stack trace is optional, or can be limited to a certain number of stack frames.

Frames can also be highlighted by customizing `io.aviso.exception/*app-frame-names*`. This adds extra visual clarity to identify frames that belong in your Clojure code vs. library code.

Before:

```
user=> (alter-var-root #'io.aviso.exception/*app-frame-names* (constantly []))
[]
user=> (my-app.handler-test/test-failure)

ERROR in () (:)
expected: nil
  actual:
                                        clojure.core/eval            core.clj: 3105
                                                        ...
                                            user/eval2522           REPL Input
                        my-app.handler-test/test-failure  demo_appframes.clj:   62
                             my-app.handler/make-ex-info  demo_appframes.clj:   49
                           my-app.handler/make-exception  demo_appframes.clj:   40
                                     my-app.db/update-row  demo_appframes.clj:   30
provided.db-worker/make-jdbc-update-worker/reify/do-work  demo_appframes.clj:   23
                                  provided.db/jdbc-update  demo_appframes.clj:    8
     java.sql.SQLException: Database failure
     SQLState: "ABC"
    errorCode: 123
java.lang.RuntimeException: Failure updating row
java.lang.RuntimeException: Request handling exception
clojure.lang.ExceptionInfo: Exception in make-ex-info.
    function: make-exception
nil
```

After:

```
user=> (alter-var-root #'io.aviso.exception/*app-frame-names* (constantly [#"my-app.*"]))
[#"my-app.*"]
user=> (my-app.handler-test/test-failure)

ERROR in () (:)
expected: nil
  actual:
                                        clojure.core/eval            core.clj: 3105
                                                      ...
                                            user/eval2526           REPL Input
                        my-app.handler-test/test-failure  demo_appframes.clj:   62
                             my-app.handler/make-ex-info  demo_appframes.clj:   49
                           my-app.handler/make-exception  demo_appframes.clj:   40
                                    my-app.db/update-row  demo_appframes.clj:   30
provided.db-worker/make-jdbc-update-worker/reify/do-work  demo_appframes.clj:   23
                                  provided.db/jdbc-update  demo_appframes.clj:    8
     java.sql.SQLException: Database failure
     SQLState: "ABC"
    errorCode: 123
java.lang.RuntimeException: Failure updating row
java.lang.RuntimeException: Request handling exception
clojure.lang.ExceptionInfo: Exception in make-ex-info.
    function: make-exception
nil
```

Notice with custom app-frame-names, the matched frame names are also in bold font. This is customized by re-binding or altering `*app-frame-names*`, which is a list of string or patterns to match on the frame's name.

```
;; marks any frame that begins with demo
(alter-var-root #'io.aviso.exception/*app-frame-names* (constantly [#"my-app.*"]))
```

2.2.2 io.aviso.repl

This namespace includes a function, `install-pretty-exceptions`, which hooks into all the common ways that exceptions are output in Clojure and redirects them to use write-exception.

When exceptions occur, they are printed out without a stack trace or properties. The `clojure.repl/pst` function is overridden to fully print the exception (*with* properties and stack trace).

In addition, `clojure.stacktrace/print-stack-trace` and `clojure.stacktrace/print-cause-trace` are overwritten; these are used by `clojure.test`. Both do the same thing: print out the full exception (again, with properties and stack trace).

You may not need to invoke this directly, as pretty can also act as a *Leiningen Plugin*.

2.2.3 io.aviso.logging

This namespace includes functions to change `clojure.tools.logging` to use Pretty to output exceptions, and to add a default Thread.UncaughtExceptionHandler that uses `clojure.tools.logging`.

2.3 REPL

Pretty includes some useful REPL utilities in the `io.aviso.repl` namespace.

Primarily, the `install-pretty-exceptions` function overrides several internal Clojure functions to enable *formatted exceptions*. This function is normally invoked for you when Pretty is *used as a Leiningen plugin*.

The remaining functions exist to help you make things pretty that *don't* originate in the REPL. You will often see output in logs: EDN data perhaps, or often, raw exceptions.

If you have a REPL running, you can use the following functions to get a better view of that data:

2.3.1 copy

The `copy` function will return the current contents of the system clipboard as a string. This requires that AWT is running. On OS X, you will see a window for your application start when you first invoke this function

The `pretty-print` and `format-exception` functions can be invoked with no arguments, in which case the call to `copy` happens automatically.

Consult the API documentation for more details.

2.3.2 pretty-print

This will pretty-print the contents of the clipboard; the clipboard text is parsed as EDN.

2.3.3 format-exception

This will parse a normal Java stack trace and format it for readability.

2.3.4 paste

This will copy a string back on to the clipboard, so it can be pasted into another window.

```
(use 'io.aviso.repl)

(-> (copy) format-exception paste)
```

2.3.5 -main

Sets up pretty exceptions before passing arguments to *clojure.main/main*; this is used when starting a REPL to ensure that pretty exception are enabled before starting to interact.

2.4 Leiningen Plugin

pretty can act as a plugin to Leiningen.

To enable pretty exception reporting automatically, add pretty to *both* the :plugins and the :dependencies lists of your `project.clj`.

```
(defproject ...
 :plugins [[io.aviso/pretty "1.3"]]
 :middleware [io.aviso.lein-pretty/inject]
 :dependencies [...
```

(continues on next page)

(continued from previous page)

```
                [io.aviso/pretty "1.3"]]
 ...)
```

Adjust the version number for the current version, "1.4.4".

Tip: Often, you only add `io.aviso/pretty` to your :dev profile dependencies.

This adds middleware to enable pretty exception reporting when running a REPL, tests, or anything else that starts code in the project.

Another option is to add the following to your `~/.lein/profiles.clj`:

```
:pretty {
  :plugins [[io.aviso/pretty "1.3"]]
  :dependencies [[io.aviso/pretty "1.3"]]
  :middleware [io.aviso.lein-pretty/inject]
}
```

This creates an opt-in profile that adds and enables pretty exception reporting.

You can then enable pretty in any project, even one that does not normally have pretty as a dependency, as follows:

```
lein with-profiles +pretty run
```

or:

```
lein with-profiles +pretty do clean, test, install
```

You may also want to add the following to your `~/.bash_profile`:

```
alias pretty="lein with-profile +pretty"
```

At which point, you can use the command `pretty` instead of `lein`.

2.5 Binary Output

The `io.aviso.binary` namespace provides support output of binary data.

Binary data is represented using the protocol BinaryData; this protocol is extended on byte arrays, on String, and on nil. BinaryData is simply a randomly accessible collection of bytes, with a known length.

```
(write-binary "Choose immutability and see where it takes you.")
```

```
0000: 43 68 6F 6F 73 65 20 69 6D 6D 75 74 61 62 69 6C 69 74 79 20 61 6E 64 20 73 65␣
↪65 20 77 68 65 72
0020: 65 20 69 74 20 74 61 6B 65 73 20 79 6F 75 2E
```

```
(write-binary *out* "Choose immutability and see where it takes you." {:ascii true})
0000: 43 68 6F 6F 73 65 20 69 6D 6D 75 74 61 62 69 6C |Choose immutabil|
0010: 69 74 79 20 61 6E 64 20 73 65 65 20 77 68 65 72 |ity and see wher|
0020: 65 20 69 74 20 74 61 6B 65 73 20 79 6F 75 2E    |e it takes you. |
=> nil
(write-binary *out* "Choose immutability and see where it takes you." {:ascii true :line-bytes 20})
0000: 43 68 6F 6F 73 65 20 69 6D 6D 75 74 61 62 69 6C 69 74 79 20 |Choose immutability |
0014: 61 6E 64 20 73 65 65 20 77 68 65 72 65 20 69 74 20 74 61 6B |and see where it tak|
0028: 65 73 20 79 6F 75 2E                                       |es you.             |
=> nil
```

Alternately, `format-binary` will return the formatted binary output string.

You can also compare two binary data values with `write-binary-delta`:

```
(write-binary-delta "Can you spot the difference?" "Can you spot the d1fference?")
0000: 43 61 6E 20 79 6F 75 20 73 70 6F 74 20 74 68 65 |  43 61 6E 20 79 6F 75 20 73 70 6F 74 20 74 68 65
0010: 20 64 69 66 66 65 72 65 6E 63 65 3F             |  20 64 31 66 66 65 72 65 6E 63 65 3F
=> nil
```

If the two data are of different lengths, the shorter one is padded with `--` to make up the difference.

As with `write-binary`, there's a `format-binary-delta`.

2.6 Columnar Output

The `io.aviso.columns` namespace is what's used by the exceptions namespace to format the exceptions, properties, and stack traces.

The `format-columns` function is provided with a number of column definitions, each of which describes the width and justification of a column. Some column definitions are just a string to be written for that column, such as a column separator. `format-columns` returns a function that accepts the column values and writes the formatted column data to *out*.

`write-rows` takes the function provided by `format-columns`, plus a set of functions to extract column values, plus a seq of rows. In most cases, the rows are maps, and the extraction functions are keywords (isn't Clojure magical that way?).

Here's an example, based on code in the exception namespace:

```
(defn ^:private write-stack-trace
  [exception]
  (let [elements (->> exception expand-stack-trace (map preformat-stack-frame))
        formatter (c/format-columns [:right (c/max-value-length elements :formatted-
↪name)]
                                    "  " (:source *fonts*)
                                    [:right (c/max-value-length elements :file)]
                                    2
                                    [:right (->> elements (map :line) (map str) c/max-
↪length)]
                                    (:reset *fonts*))]
    (c/write-rows formatter
                  [:formatted-name
                   :file
                   #(if (:line %) ": ")
                   :line]
                  elements)))
```