A flexible library for shelling out in Clojure

122 commits 3 branches 3 releases 9 contributors

Branch: master Create new file Find file Clone or download

Raynes Whitespace cleanup Latest commit 3acbfe5 on 26 Oct 2016

src/me/raynes	Add :clear-env to the high level interface + tests	9 months ago
test	Whitespace cleanup	9 months ago
.gitignore	update gitignore	5 years ago
.travis.yml	Fix test task name in .travis.	5 years ago
README.md	Fix the name in the documentation, the correct key is :process	11 months ago
project.clj	remove org.flatland/useful dependency	10 months ago

conch

Conch is actually two libraries. The first, `me.raynes.conch.low-level` is a simple low-level interface to the Java process APIs. The second and more interesting library is an interface to low-level inspired by the Python sh library.

The general idea is to be able to call programs just like you would call Clojure functions. See Usage for examples.

Installation

In Leiningen:

```
[me.raynes/conch "0.8.0"]          @clojars.org
```

Usage

First of all, let's `require` a few things.

```
user> (require '[me.raynes.conch :refer [programs with-programs let-programs] :as sh])
nil
```

Let's call a program. What should we call? Let's call `echo`, because I obviously like to hear myself talk.

```
user> (programs echo)
#'user/echo
user> (echo "Hi!")
"Hi!\n"
```

Cool! `programs` is a simple macro that takes a number of symbols and creates functions that calls programs on the PATH with those names. `echo` is now just a normal function defined with `defn` just like any other.

`with-programs` and `let-programs` are lexical and specialized versions of `programs`. `with-programs` is exactly the same as `programs`, only it defines functions lexically:

```
user> (with-programs [ls] (ls))
"#README.md#\nREADME.md\nclasses\ndocs\nfoo.py\nlib\nlol\npom.xml\npom.xml.asc\nproject.clj\nsrc\ntarget\nt
user> ls
CompilerException java.lang.RuntimeException: Unable to resolve symbol: ls in
this context, compiling:(NO_SOURCE_PATH:1)
```

`let-programs` is similar, but is useful when you want to specify a path to a program that is not on the PATH:

```
user> (let-programs [echo "/bin/echo"] (echo "hi!"))
"hi!\n"
```

Bad example since `echo` *is* on the path, but if it wasn't there already it still would have worked. I promise.

Input

You can pass input to a program easily:

```
user> (programs cat)
#'user/cat
user> (cat {:in "hi"})
"hi"
user> (cat {:in ["hi" "there"]})
"hi\nthere\n"
user> (cat {:in (java.io.StringReader. "hi")})
"hi"
```

`:in` is handled by a protocol and can thus be extended to support other data.

Output

So, going back to our `ls` example. Of course, `ls` gives us a bunch of lines. In a lot of cases, we're going to want to process lines individually. We can do that by telling conch that we want a lazy seq of lines instead of a monolithic string:

```
user> (ls {:seq true})
("#README.md#" "README.md" "classes" "docs" "foo.py" "lib" "lol" "pom.xml"
"pom.xml.asc" "project.clj" "src" "target" "test" "testfile")
```

We can also redirect output to other places.

```
user> (let [writer (java.io.StringWriter.)] (echo "foo" {:out writer}) (str writer))
"foo\n"
user> (echo "foo" {:out (java.io.File. "conch")})
nil
user> (slurp "conch")
"foo\n"
```

And if that wasn't cool enough for you, `:out` is handled by a protocol and thus can be extended.

Need Moar INNNNPUUUUUUT

Need the exit code and stuff? Sure:

```
user> (echo "foo" {:verbose true})
{:proc {:out ("foo\n"), :in #<ProcessPipeOutputStream
java.lang.UNIXProcess$ProcessPipeOutputStream@19c12ee7>, :err (), :process
#<UNIXProcess java.lang.UNIXProcess@2bfabe2a>}, :exit-code
#<core$future_call$reify__6110@5adacdf4: 0>, :stdout "foo\n", :stderr ""}
```

Timeouts

```
user> (sleep "5")
... yawn ...
```

Tired of waiting for that pesky process to exit? Make it go away!

```
user> (sleep "5" {:timeout 2000})
... two seconds later ...
ExceptionInfo Program returned non-zero exit code :timeout  clojure.core/ex-info (core.clj:4227)
```

Much better.

Exceptions

Conch can handle exit codes pretty well. You can make it do pretty much whatever you want in failure scenarios.

By default, conch throws `ExceptionInfo` exceptions for non-zero exit codes, as demonstrated here:

```
user> (ls "-2")
ExceptionInfo Program returned non-zero exit code 1  clojure.core/ex-info
(core.clj:4227)
```

This exception's data is the same result you'd get by passing the `:verbose` option:

```
user> (ex-data *e)
{:proc {:out (), :in #<ProcessPipeOutputStream java.lang.UNIXProcess$ProcessPipeOutputStream@79bb65ee>, :er
option -- 2\nusage: ls [-ABCFGHLOPRSTUWabcdefghiklmnopqrstuwx1] [file ...]\n"), :process #<UNIXProcess java
```

You can control this behavior in two ways. The first way is to set `me.raynes.conch/*throw*` to `false`:

```
user> (binding [sh/*throw* false] (ls "-2"))
""
```

You can also just override whatever `*throw*` is with the `:throw` argument to the functions themselves:

```
user> (ls "-2" {:throw false})
""
```

Piping

You can pipe the output of one program as the input to another about how you'd expect:

```
user> (programs grep ps)
#'user/ps
user> (grep "ssh" {:in (ps "-e" {:seq true})})
" 4554 ??         0:00.77 /usr/bin/ssh-agent -l\n"
```

These functions also look for a lazy seq arg, so you can get rid of the explicit `:in` part.

```
user> (programs grep ps)
#'user/ps
user> (grep "ssh" (ps "-e" {:seq true}))
" 4554 ??         0:00.77 /usr/bin/ssh-agent -l\n"
```

Buffering

Conch gets rid of some ugly edge-cases by **always reading process output immediately when it becomes available**. It buffers this data into a queue that you consume however you want. This is how returning lazy seqs work. Keep in mind that if you don't consume data, it is being held in memory.

You can change how conch buffers data using the `:buffer` key.

```
user> (ls {:seq true :buffer 5})
("#READ" "ME.md" "#\nREA" "DME.m" "d\ncla" "sses\n" "conch" "\ndocs" "\nfoo." "py\nli" "b\nlol" "\npom." "
user> (ls {:seq true :buffer :none})
(\# \R \E \A \D \M \E \. \m \d \# \newline \R \E \A \D \M \E \. \m \d \newline
\c \l \a \s \s \e \s \newline \c \o \n \c \h \newline \d \o \c \s \newline \f \o
```

```
\o \. \p \y \newline \l \i \b \newline \l \o \l \newline \p \o \m \. \x \m \l
\newline \p \o \m \. \x \m \l \. \a \s \c \newline \p \r \o \j \e \c \t \. \c \l
\j \newline \s \r \c \newline \t \a \r \g \e \t \newline \t \e \s \t \newline \t
\e \s \t \f \i \l \e \newline)
```

Another nice thing gained by the way conch consumes data is that it is able to kill a process after a timeout and keep whatever data it has already consumed.

PTY stuff

PTY stuff seems like it'd be a lot of work and would involve non-Clojure stuff. If you need a PTY for output, I suggest wrapping your programs in 'unbuffer' from the expect package. It usually does the trick for unbuffering program output by making it think a terminal is talking to it.

Hanging

You might run into an issue where your program finishes after using conch but does not exit. Conch uses futures under the hood which spin off threads that stick around for a minute or so after everything else is done. This is an unfortunate side effect, but futures are necessary to conch's functionality so I'm not sure there is much I can do about it.

You can work around this by adding a `(System/exit 0)` call to the end of your program.

Low Level Usage

The low-level API is available in a separate package:

```
(use '[me.raynes.conch.low-level :as sh])
```

It is pretty simple. You spin off a process with `proc`.

```
user=> (def p (sh/proc "cat"))
#'user/p
user=> p
{:out #<BufferedInputStream java.io.BufferedInputStream@5809fdee>, :in #<BufferedOutputStream java.io.Buffe
```

When you create a process with `proc`, you get back a map containing the keys `:out`, `:err`, `:in`, and `:process`.

- `:out` is the process's stdout.
- `:err` is the process's stderr.
- `:in` is the process's stdin.
- `:process` is the process object itself.

Conch is more flexible than `clojure.java.shell` because you have direct access to all of the streams and the process object itself.

So, now we have a cat process. This is a unix tool. If you run `cat` with no arguments, it echos whatever you type in. This makes it perfect for testing input and output.

Conch defines a few utility functions for streaming output and feeding input. Since we want to make sure that our input is going to the right place, let's set up a way to see the output of our process in realtime:

```
user=> (future (sh/stream-to-out p :out))
#<core$future_call$reify__5684@77b5c22f: :pending>
```

The `stream-to-out` function takes a process and either `:out` or `:err` and streams that to `System/out`. In this case, it has the effect of printing everything we pipe into our cat process, since our cat process just outputs whatever we input.

```
user=> (sh/feed-from-string p "foo\n")
nil
foo
```

The `feed-from-string` function just feeds a string to the process. It automatically flushes (which is why this prints immediately) but you can stop it from doing that by passing `:flush false`.

I think our cat process has lived long enough. Let's kill it and get its exit code. We can use the `exit-code` function to get the exit code. However, since `exit-code` stops the thread and waits for the process to terminate, we should run it in a future until we actually destroy the process.

```
user=> (def exit (future (sh/exit-code p)))
#'user/exit
```

Now let's kill. R.I.P process.

```
user=> (sh/destroy p)
nil
```

And the exit code, which we should be able to obtain now that the process has been terminated:

```
user=> @exit
0
```

Awesome! Let's go back to `proc` and see what else we can do with it. We can pass multiple strings to `proc`. The first string will be considered the executable and the rest of them the arguments to that executable.

```
user=> (sh/proc "ls" "-l")
{:out #<BufferedInputStream java.io.BufferedInputStream@7fb6634c>, :in #<BufferedOutputStream java.io.Buffe
```

low-level

```
(require '[me.raynes.conch.low-level :as sh])
```

Here is an easy way to get the output of a one-off process like this as a string:

```
user=> (sh/stream-to-string (sh/proc "ls" "-l") :out)
"total 16\n-rw-r--r--  1 anthony  staff  2545 Jan 24 16:37 README.md\ndrwxr-xr-x  2 anthony  staff    68 Ja
```

Let's print that for readability:

```
user=> (print (sh/stream-to-string (sh/proc "ls" "-l") :out))
total 16
-rw-r--r--  1 anthony  staff  2545 Jan 24 16:37 README.md
drwxr-xr-x  2 anthony  staff    68 Jan 19 19:23 classes
drwxr-xr-x  3 anthony  staff   102 Jan 19 19:23 lib
-rw-r--r--  1 anthony  staff   120 Jan 20 14:45 project.clj
drwxr-xr-x  3 anthony  staff   102 Jan 20 14:45 src
drwxr-xr-x  3 anthony  staff   102 Jan 19 16:36 test
nil
```

So, that's the `ls` of the current directory. I ran this REPL in the conch project directory. We can, of course, pass a directory to `ls` to get it to list the files in that directory, but that isn't any fun. We can pass a directory to `proc` itself and it'll run in the context of that directory.

```
user=> (print (sh/stream-to-string (sh/proc "ls" "-l" :dir "lib/") :out))
total 6624
-rw-r--r--  1 anthony  staff  3390414 Jan 19 19:23 clojure-1.3.0.jar
nil
```

You can also pass a `java.io.File` or anything that can be passed to `clojure.java.io/file`.

We can also set environment variables:

```
user=> (print (sh/stream-to-string (sh/proc "env" :env {"FOO" "bar"}) :out))
FOO=bar
nil
```

The map passed to `:env` completely replaces any other environment variables that were in place.

Finally, there a couple of low-level functions for streaming from and feeding to a process. They are `stream-to` and `feed-from`. These functions are what the utility functions are built off of, and you can probably use them to stream to and feed from your own special places.

You might want to fire off a program that listens for input until EOF. In these cases, you can feed it data for as long as you want and just tell it when you are done. Let's use `pygmentize` as an example:

```
user=> (def proc (sh/proc "pygmentize" "-fhtml" "-lclojure"))
#'user/proc
user=> (sh/feed-from-string proc "(+ 3 3)")
nil
user=> (sh/done proc)
nil
user=> (sh/stream-to-string proc :out)
"<div class=\"highlight\"><pre><span class=\"p\">(</span><span class=\"nb\">+ </span><span class=\"mi\">3</
```

When we call `done`, it closes the process's output stream which is like sending EOF. The process processes its input and then puts it on its input stream where we read it with `stream-to-string`.

Other options

All of conch's streaming and feeding functions (including the lower level ones) pass all of their keyword options to `clojure.java.io/copy`. It can take an `:encoding` and `:buffer-size` option. Guess what they do.

Key names

You might notice that the map that `proc` returns is mapped like so:

- `:in` -> output stream
- `:out` -> input stream

I did this because swapping them feels counterintuitive. The output stream is what you put `:in` to and the input stream is what you pull `:out` from.

License