literate clojure

a literate programming tool to write clojure codes in org mode.

Jingtao Xu

August 15, 2019

Contents

1 Introduction **1**

2 how to do it? **1**

3 Implementation **2**
3.1 Preparation 2
3.2 stream read operations 3
3.3 reader macros 4
3.4 handle org syntax 4
3.5 handle end of source code block 5
3.6 install new dispatcher functions 5
3.7 install new dispatcher functions to tools.reader . . . 6
3.8 tangle org file to clojure file 6

4 References **7**

1 Introduction

This is a clojure library to show a way how to use literate programming in clojure.

It extends the clojure read syntax so clojure can load org file as source file directly.

literate programming examples show the reason why use org mode,

By using clojure package literate-clojure , emacs org mode and elisp library polymode, literate programming can be easy in one org file containing both documentation and source codes, and this org file works well with cider.

2 how to do it?

In org mode, the comment line start with character # (see org manual), and the clojure codes exists between #+begin_src clojure and #+end_src (see org manual).

```
#+BEGIN_SRC clojure :load no
(ns literate-clojure.clojure-example (:use [clojure.pprint]))
(defn test []
  (pprint "This is a test function."))
#+END_SRC

#+BEGIN_SRC clojurescript :load no
(ns literate-clojure.clojurescript-example (:use [cljs.pprint :refer [pprint]]
(defn test []
  (pprint "This is a test function."))
#+END_SRC
```

So to let clojure can rean an org file directly, all lines out of surrounding by #+begin_src and #+end_src should mean nothing, and even codes surrounding by them should mean nothing if the header arguments in a code block request such behavior.

Here is a trick, a new clojure dispatch syntax "# "(sharpsign whitespace) will be defined to make clojure reader enter into org mode syntax, then ignore all lines after that until it meet #+begin_src.

When #+begign_src lisp occurs, org header arguments for this code block give us a chance to switch back to normal clojure reader or not.

And if it switch back to normal clojure reader, the end line #+END_SRC should mean the end of current code block,so a new clojure dispatch syntax for "#+"(sharp plus)will have an additional meaning to determine if it is #+END_SRC, if it is, then clojure reader will switch back to org mode syntax, if it is not, clojure reader will continue to read subsequent stream as like the original clojure reader.

This workflow restricts the org file starting with a comment character and a space character("# "), but it should not be a problem but indeed a convenient way for us to specify some local variables, for example I often put them in the first line of an org file:

```
# -*- encoding:utf-8 Mode: POLY-ORG;  -*- ---
```

Which make emacs open file with utf-8 encoding and poly-org-mode.

3 Implementation

3.1 Preparation

3.1.1 namespace

Let's create a new namespace for this library.

```
(ns literate-clojure.core
  (:require
    [clojure.pprint :refer [cl-format]]
    [clojure.string :refer [starts-with? lower-case trim split]]
    [clojure.tools.reader.reader-types :as reader-types]
    [clojure.tools.reader])
  (:import (clojure.lang LispReader LispReader$WrappingReader)))
```

3.1.2 debug function

A boolean variable to toggle debug on/off

```
(defonce ^:dynamic debug-p nil)
```

a debug function to print out some log messages.

```
(defn debug [& args]
  (when debug-p
    (apply println "literate-clojure: " args)))
```

3.2 stream read operations

The reader class used by clojure and tools.reader is different.

The reader class used by clojure to parse source forms is PushbackReader, and the reader class used by tools.reader is different reader$_{types}$.

We use a dynamic variable to distinguish them

```
(def ^:dynamic tools-reader-p nil)
```

Let's implement some common read facilities. The first one is `read one character` by the reader, we will simulate the behavior of PushbackReader.

```
(defn read-char [reader]
  (if tools-reader-p
    (let [c (reader-types/read-char reader)]
      (if c
        (int c)
        -1))
    (.read reader)))
```

And now let's implement a `read line` operation.

Before reading a line, let's define a function to determine whether a character is line terminator.

```
(defn- line-terminator? [c]
  (or (= c (int \return)) (= c (int \newline))))
```

Now let's implement the operation `read line`.

```
(defn- literate-read-line [reader]
  (let [c (read-char reader)]
    (cond (= c -1) nil
          (line-terminator? c) ""
          :else (with-out-str
                  (do (cl-format *out* "~c" (char c))
                      (loop [c (read-char reader)]
                        (when (and (not= c -1)
                                   (not (line-terminator? c)))
                          (cl-format *out* "~c" (char c))
                          (recur (read-char reader)))))))))
```

3.3 reader macros

Unlike Common Lisp, Clojure doesn't support user-defined reader macros.

Based on clojure's LispReader, it is easy to define a dispatch reader macro (i.e. one starting with # and some specified second character):

```
(defn- dispatch-reader-macro [ch fun]
  (let [dm (.get (doto (.getDeclaredField clojure.lang.LispReader "dispatchMacros")
                   (.setAccessible true))
                 nil)]
    (when (nil? (aget dm (int ch)))
      (debug (cl-format nil "install dispatch reader macro for character '~a'" ch))
      (aset dm (int ch) fun))))
```

But it only works in clojure instead of clojurescript, because clojurescript use tools.reader.

3.4 handle org syntax

There are a lot of different lisp codes occur in one org file, some for function implementation, some for demo, so a new org code block header argument `load` to decide to read them or not should define,and it has three meanings:

- yes
  It means that current code block should load normally, it is the default mode when the header argument `load` is not provided.
- no
  It means that current code block should ignore by lisp reader.

The parameter `arguments` is a string vector contains all head block arguments.

```
(defn- load? [arguments]
  (debug (cl-format nil "header arguments is: ~s" arguments))
  (loop [left-arguments arguments]
    (cond (nil? left-arguments) true
          (= (first left-arguments) ":load")
          (case (second left-arguments)
            nil true
            "" true
            "yes" true
            "no" nil)
          :else (recur (next left-arguments)))))
```

Let's implement a function to read header arguments after #+BEGIN_SRC clojure or #+BEGIN_SRC clojurescript .

```
(def id-of-begin-src "#+begin_src")
(def literate-begin-src-ids (for [lang '("clojure" "clojurescript")]
                              (format "%s %s" id-of-begin-src lang)))
(defn- read-org-code-block-header-arguments [line]
  (let [trimmed-line (trim line)]
    ;; remove two head tokens.
    (rest (rest (split (lower-case trimmed-line) #"\s+")))))
```

Let's define a new dispatch function for "# "(sharp space) to enter into org syntax, until it meet #+begin_src clojure. The reader is returned so LispReader will continue to read rest forms with clojure syntax.

```
(defn- dispatch-sharp-space [reader quote opts pending-forms]
  (debug "enter into org syntax.")
  (loop [line (literate-read-line reader)]
    (cond (nil? line) (debug "reach end of stream in org syntax.")
          (some #(starts-with? (format "%s " (lower-case (trim line))) (format "%s " %)
              ↪ ) literate-begin-src-ids)
          (do (debug "reach begin of code block.")
              (if (load? (read-org-code-block-header-arguments line))
                  (debug "enter into clojure syntax.")
                (recur (literate-read-line reader))))
          :else (do
                  (debug (cl-format nil "ignore line: ~a" line))
                  (recur (literate-read-line reader)))))
  reader)
(defn- tools-reader-dispatch-sharp-space [reader quote opts pending-forms]
  (binding [tools-reader-p true]
    (dispatch-sharp-space reader quote opts pending-forms)))
```

3.5 handle end of source code block

Let's define a new dispatch function for "#+" (sharp plus) to return back org syntax, until it meet #+begin_src clojure.

```
(defn- dispatch-sharp-plus [reader quote opts pending-forms]
  (let [line (literate-read-line reader)]
    (cond (nil? line) (debug "reach end of stream in org syntax.")
          (starts-with? (lower-case (trim line)) "end_src")
          (do (debug "reach begin of code block.")
```

```
              (debug "switch back from clojure syntax to org syntax.")
              (dispatch-sharp-space reader quote opts pending-forms))
          :else (throw (Exception. (cl-format nil "invalid syntax in line :~a" line))))
              ↪ ))
(defn- tools-reader-dispatch-sharp-plus [reader quote opts pending-forms]
  (binding [tools-reader-p true]
    (dispatch-sharp-plus reader quote opts pending-forms)))
```

3.6 install new dispatcher functions

```
(defn install-org-dispatcher []
  (dispatch-reader-macro \+ dispatch-sharp-plus)
  (dispatch-reader-macro \space dispatch-sharp-space))
(println "install literate syntax to clojure reader.")
(install-org-dispatcher)
```

3.7 install new dispatcher functions to tools.reader

Sadly tools.reader use a private function to return dispatch functions(see function dispatch-macros). So we have to advice this function to add new dispatch reader macro.

```
(defn tools.reader.additional-dispatch-macros [orig-fn]
  #(or (orig-fn %)
       (case %
         \+ tools-reader-dispatch-sharp-plus
         \space tools-reader-dispatch-sharp-space
         nil)))
(println "install literate syntax to tools.reader.")
(alter-var-root (var clojure.tools.reader/dispatch-macros) #'tools.reader.
    ↪ additional-dispatch-macros)
```

3.8 tangle org file to clojure file

To build clojure file from an org file, we implement a function `tangle-file`.

The basic method is simple here, we use function `dispatch-sharp-space` to ignore all lines should be ignored, then export all code lines until we reach `#+end_src`, this process is repeated to end of org file.

This mechanism is good enough because it will not damage any codes in org code blocks.

```
(def exception-id-of-end-of-stream "end-of-litereate-stream")
(defn tangle-file [org-file]
  (with-open [reader (clojure.lang.LineNumberingPushbackReader. (clojure.java.io/reader
      ↪  org-file))]
    (with-open [writer (clojure.java.io/writer (str (.substring org-file 0 (.
        ↪ lastIndexOf org-file "."))
                                                    ".clj"))]
      (.write writer (cl-format nil ";;; This file is automatically generated from file
          ↪  '~a'.
```

```
;;; It is not designed to be readable by a human.
;;; It is generated to load by clojure directly without depending on 'literate-clojure
    ↪ '.
;;; Please read file '~a' to find out the usage and implementation detail of this
    ↪ source file.~%~%"
                                org-file org-file))
      (try
        (while true
          ;; ignore all lines of org syntax.
          (dispatch-sharp-space reader \space nil nil)
          ;; start to read clojure codes.
          (loop [line (literate-read-line reader)]
            (cond (nil? line) (do (debug "reach end of stream in org syntax.")
                                  (throw (Exception. exception-id-of-end-of-stream)))
                  (starts-with? (lower-case (trim line)) "#+end_src")
                  (debug "reach end of code block.")
                  :else (do
                          (debug (cl-format nil "tangle line: ~a" line))
                          (.write writer line)
                          (.write writer "\n")
                          (recur (literate-read-line reader)))))
          (.write writer "\n")
          (.flush writer))
        (catch Exception e
          (if (not= exception-id-of-end-of-stream (.getMessage e))
            ;; we don't know about this exception, throw it again.
            (throw e)))))))
```

So if we want to release ./core.clj, the following codes should execute:

```
(tangle-file "src/literate_clojure/core.org")
```

4 References

- Literate. Programming. by Donald E. Knuth
- Literate Programming a site of literate programming
- Literate Programming in the Large a talk video from Timothy Daly,one of the original authors of Axiom.
- A collection of literate programming examples using Emacs Org mode
- literate programming in org babel
- a reader macro library for clojure: https://github.com/klutometis/reader-macros
- org babel example: https://github.com/lambdatronic/org-babel-example
- clojure reader macros: https://cdaddr.com/programming/clojure-reader-mac
- literate lisp: https://github.com/jingtaozf/literate-lisp