nREPL Looper

ClojureDC 2015

Brock Wilcox
awwaiid@thelackthereof.org
"Live Coding"
Meta-Ex
Repl-Electric
user=> (use 'overtone.live)
user=> (use 'overtone.inst.piano)
user=> (piano 62)
user=> 1
Tedious!
(kick)
(hat)
(apply-at (+ (now) 100) hat [])
(apply-at (+ (now) 200) hat [])
(apply-at (+ (now) 300) hat [])
apply_by / apply_at
Temporal Recursion
;; (From overtone wiki)

;; setup a sound for our metronome to use
(def kick (sample (freesound-path 2086)))

;; setup a tempo for our metronome to use
(def one-twenty-bpm (metronome 120))

;; this function will play our sound at the given temp
(defn looper [nome sound]
  (let [beat (nome)]
    (at (nome beat) (sound))
    (apply-by (nome (inc beat)) looper nome sound [])))

;; turn on the metronome
(looper one-twenty-bpm kick)
(Side note: had to run patched master Overtone to get freesound samples working in the last month or so)
(defn drumloop []
  (kick)
  (hat)
  (apply-at (+ (now) 100) hat [])
  (apply-at (+ (now) 200) hat [])
  (apply-at (+ (now) 300) hat [])
  (apply-at (+ (now) 400) drumloop []))
Emacs. Vim. Edit Live. Fun!
How can I keep living on the console?
Key-driven

jline.console ConsoleReader
`char->note`

"used to convert char a to midi note 54 (F#4) etc...

the characters below should resemble of a keyboard, with the key left of z
and the two right of m specified with keycode instead of a clojure char

please note that the midi note numbers always increases +2, and that there's
an odd (5,7) offset per row, according to the Wicki-Hayden note layout:

+fourth  +oct  +fifth
  ^     ^     _
  |     |     /|
  |_ ****** /
  *      *
  *        *
  *   C    *  -> +2
  *        *
  *      *
  *****

```
{1 64 2 66 3 68 4 70 5 72 6 74 7 76 8 78 9 80 0 82
 q 59 w 61 e 63 r 65 t 67 y 69 u 71 i 73 o 75 p 77
 a 54 s 56 d 58 f 60 g 62 h 64 j 66 k 68 l 70
 60 47 z 49 x 51 c 53 v 55 b 57 n 59 m 61 , 63 . 65}
```

jline to get keys

(lein trampoline repl)
Bang on keyboard

(wish I had an n-key-rollover gamer board)
Moar REPL
Declare a list of offsets and events
(def loop-1
  (atom
    {:length 4000
     :events [{:offset 0     :cmd "(piano 60)"
              {:offset 1000 :cmd "(piano 60)"
              {:offset 2000 :cmd "(piano 63)"
              {:offset 3000 :cmd "(piano 65)"}]}}))
Playback
; Play a single event

(defn play-event [start-time length event]
  (let [offset (mod (event :offset) length)
        cmd (event :cmd)]
    (apply-at (+ start-time offset) eval [cmd])))
; Play a bunch of events

(defn play-events-at [events start-time length]
  (doseq [event events]
    (play-event start-time length event)))
; Play a whole loop, temporally recursively

(defn play-loop [event-loop]

  (let [events (@event-loop :events)
        length (@event-loop :length)
        start-time (now)]

    (swap! event-loop assoc :start-time start-time)

    ; Queue up all the events
    (play-events-at events start-time length)

    ; Recurse! In the future!
    (apply-at (+ start-time length) play-loop [event-loop]))
Recording Live Events
Reply

Intercept reply-execute
(ns reply.eval-modes.standalone)

(def execute
  (looper.core/wrap-repl-execute execute))

(ns looper.core)

(defn wrap-repl-execute [repl-execute]
  "Modify reply's execute to record the events with timing"
  (fn [options form]
    (let [loop (options :loop)
           start-time (@loop :start-time)]
      (add-event-now loop form)
      (repl-execute options form))))
ick. Better way?
nREPL Middleware
nREPL

client → server
Cool way to allow clients, like emacs, to connect to a server and run stuff. Let's the server be stateful!
Handler, Middleware, Transport
Handlers:

Input message -> do stuff -> write to transport
Middleware:

Wrap a handler to add some functionality
Transport:

Send messages back and forth

Especially, how to send responses back to the client
; Example message
{"op" "eval",
 "code" "(+ 1 2 3)",
 "transport" ...}

; Example responses
{":ns "user",
 :value "6",
 :session "8ff94bb1-06a3-489d-9a3a-7ba950681600",
 :id "b850b366-99b2-4516-8f50-952be5424eb6"}

{":status #{:done},
 :session "708f7673-9077-4d88-8173-92b6a4c750c0",
 :id "47d42421-8619-43f8-a0a5-8b87cc6c0df4"}
client → middleware → middleware → ...
Middleware wrapping:

Given a handler, return a new handler.
(require 'clojure.tools.nrepl.transport :as t)
(use 'clojure.tools.nrepl.misc :only (response-for))

(defn wrap-current-time
  [handler]
  (fn [{:keys [op transport] :as msg}]
    (if (== "time?" op)
      (t/send transport
        (response-for msg
          :status :done
          :time (System/currentTimeMillis)))
      (handler msg))))
(defn wrap-looper
  "As nREPL middleware, look for commands to intercept or stuff to record"
)

[handler]

(fn [{:keys [op code] :as msg}]
  (if (and (= op "eval") (not= code ""))
    (let [input (read-string code)]
      (if (and (seq? input) (= 'looper (first input)))
        (looper-msg handler msg)
        (looper-eval-handler handler msg)))
    (handler msg))))
Demo
(Sometimes doesn't work?)
More than one client!
Demo
The Future

- Visualization!
- Midi repl client
- Live sample recording
- Write a song I guess?
THE END
Halp!

getting the current namespace from the eval-er
; Right now this is done via a shared var, waiting for the other thread to set ; it. I hope there is a better way, but I don't know it :)  

(def handler-ns (atom false))

(defn get-ns
  
  "Get the namespace from the eval handler."
  [handler session]
  (reset! handler-ns false)

  (let [msg {:transport dummy-transport
            :op "eval"
            :session session
            :code "(reset! nrepl-looper.middleware/handler-ns *ns*)"}]

    (logr "Going to call the handler with:" msg)
    (handler msg)

    ; This is probably a horrible way to wait for the value!
    (while (not @handler-ns))
    @handler-ns))