Emacs Deferred Documentation

Release 0.3.2

SAKURAI Masashi

February 06, 2016

Contents

Contents:

CHAPTER 1

Installation

1.1 Installing from Melpa

If you have already used Melpa to install some other package then all you have to do is:

```
M-x package-install RET deferred RET
```

1.2 Installing from Marmalade

WRITE HERE

1.3 Installing from Git

If you want to contribute to deferred and concurrent you should run it directly from the Git reposiotry.

First get the repository:

```
$ git clone git://github.com/kiwanami/emacs-deferred.git
```

Then add this to your init.el:

```
(add-to-list 'load-path "/path/to/emacs-deferred")
(require 'deferred)
(require 'concurrent)
```

To build deferred and concurrent manually:

```
WRITE HERE
```

deferred.el manual

2.1 Sample

You can find following sample codes in `deferred-sample.el`. Executing `eval-last-sexp` (C-x C-e), you can try those codes.

2.1.1 Basic usage

This is a basic deferred chain. This code puts some outputs into message buffer, and then require a number from minibuffer.

Chain:

```
(deferred:$
  (deferred:next
    (lambda () (message "deferred start")))
  (deferred:nextc it
    (lambda ()
      (message "chain 1")
      1))
  (deferred:nextc it
    (lambda (x)
      (message "chain 2 : %s" x)))
  (deferred:nextc it
    (lambda ()
      (read-minibuffer "Input a number: ")))
  (deferred:nextc it
    (lambda (x)
      (message "Got the number : %i" x)))
  (deferred:error it
    (lambda (err)
      (message "Wrong input : %s" err))))
```

- This s-exp returns immediately.
- Asynchronous tasks start subsequently.
- The macro *deferred:$* chains deferred objects.
- The anaphoric variable **:el:var:‘it‘** holds a deferred object in the previous line.
- The next deferred task receives the value that is returned by the previous deferred one.

- Inputting a wrong value, such as alphabets, this s-exp raises an error. The error is caught by the errorback function defined by *deferred:error*.

2.1.2 Timer

After evaluating this s-exp and waiting for 1 second, a message is shown in the minibuffer.

Timer:

```
(deferred:$
  (deferred:wait 1000) ; 1000msec
  (deferred:nextc it
    (lambda (x)
      (message "Timer sample! : %s msec" x))))
```

- The next deferred task subsequent to deferred:wait receives the actual elapse time in millisecond.

2.1.3 Commands and Sub-process

This s-exp inserts the result that is performed by the command 'ls -la'. (This s-exp may not run in windows. Try 'dir' command.)

Command process:

```
(deferred:$
  (deferred:process "ls" "-la")
  (deferred:nextc it
    (lambda (x) (insert x))))
```

- This s-exp hardly blocks Emacs because of asynchronous mechanisms.

2.1.4 HTTP GET : Text

This s-exp inserts a text from http://www.gnu.org asynchronously. (You can clear the result with undo command.)

HTTP GET:

```
(require 'url)

(deferred:$
  (deferred:url-retrieve "http://www.gnu.org")
  (deferred:nextc it
    (lambda (buf)
      (insert  (with-current-buffer buf (buffer-string)))
      (kill-buffer buf))))
```

2.1.5 HTTP Get : Image

This s-exp inserts an image from google asynchronously.

Get an image:

```
(deferred:$
  (deferred:url-retrieve "http://www.google.co.jp/intl/en_com/images/srpr/logo1w.png")
  (deferred:nextc it
```

```
    (lambda (buf)
      (insert-image
       (create-image
        (let ((data (with-current-buffer buf (buffer-string))))
          (substring data (+ (string-match "\n\n" data) 2)))
        'png t))
      (kill-buffer buf))))
```

2.1.6 Parallel

This s-exp retrieves two images from google concurrently and wait for the both results. Then, the file sizes of the images are inserted the current buffer.

Parallel deferred:

```
(deferred:$
  (deferred:parallel
    (lambda ()
      (deferred:url-retrieve "http://www.google.co.jp/intl/en_com/images/srpr/logo1w.png"))
    (lambda ()
      (deferred:url-retrieve "http://www.google.co.jp/images/srpr/nav_logo14.png")))
  (deferred:nextc it
    (lambda (buffers)
      (loop for i in buffers
            do
            (insert
             (format
              "size: %s\n"
              (with-current-buffer i (length (buffer-string)))))
            (kill-buffer i)))))
```

- The function 'deferred:parallel' runs asynchronous tasks concurrently.
- The function wait for all results, regardless normal or abnormal. Then, the subsequent tasks are executed.
- The next task receives a list of the results.
  - The order of the results is corresponding to one of the argument.
  - Giving an alist of tasks as the argument, the results alist is returned.

2.1.7 Deferred Combination : try-catch-finally

This s-exp executes following tasks:

- Getting an image by wget command,
- Resizing the image by convert command in ImageMagick,
- Insert the re-sized image into the current buffer. You can construct the control structure of deferred tasks, like try-catch-finally in Java.

Get an image by wget and resize by ImageMagick:

```
(deferred:$

  ;; try
  (deferred:$
    (deferred:process "wget" "-O" "a.jpg" "http://www.gnu.org/software/emacs/tour/images/splash.png")
```

```
    (deferred:nextc it
      (lambda () (deferred:process "convert" "a.jpg" "-resize" "100x100" "jpg:b.jpg")))
    (deferred:nextc it
      (lambda ()
        (clear-image-cache)
        (insert-image (create-image (expand-file-name "b.jpg") 'jpeg nil)))))

  ;; catch
  (deferred:error it ;
    (lambda (err)
      (insert "Can not get a image! : " err)))

  ;; finally
  (deferred:nextc it
    (lambda ()
      (deferred:parallel
        (lambda () (delete-file "a.jpg"))
        (lambda () (delete-file "b.jpg")))))
  (deferred:nextc it
    (lambda (x) (message ">> %s" x))))
```

- In this case, the deferred tasks are statically connected.

Here is an another sample code for try-catch-finally blocks. This is simpler than above code because of the 'deferred:try' macro. (Note: They bring the same results practically, but are not perfectly identical. The 'finally' task may not be called because of asynchrony.)

Try-catch-finally:

```
(deferred:$
  (deferred:try
    (deferred:$
      (deferred:process "wget" "-O" "a.jpg" "http://www.gnu.org/software/emacs/tour/images/splash.png
      (deferred:nextc it
        (lambda () (deferred:process "convert" "a.jpg" "-resize" "100x100" "jpg:b.jpg")))
      (deferred:nextc it
        (lambda ()
          (clear-image-cache)
          (insert-image (create-image (expand-file-name "b.jpg") 'jpeg nil)))))
    :catch
    (lambda (err) (insert "Can not get a image! : " err))
    :finally
    (lambda ()
      (delete-file "a.jpg")
      (delete-file "b.jpg")))
  (deferred:nextc it
    (lambda (x) (message ">> %s" x))))
```

2.1.8 Timeout

Although a long time command is executed (3 second sleeping), the task is canceled by timeout for 1 second.

The function 'deferred:earlier' also runs asynchronous tasks concurrently, however, the next deferred task receives the first result. The other results and tasks will be canceled.

Timeout Process:

```
(deferred:$
  (deferred:earlier
    (deferred:process "sh" "-c" "sleep 3 | echo 'hello!'")
    (deferred:$
      (deferred:wait 1000) ; timeout msec
      (deferred:nextc it (lambda () "canceled!"))))
  (deferred:nextc it
    (lambda (x) (insert x))))
```

- Changing longer timeout for 'deferred:wait', the next task receives a result of the command.
- When a task finishes abnormally, the task is ignored.
  - When all tasks finishes abnormally, the next task receives nil.
- The functions 'deferred:parallel' and 'deferred:earlier' may be corresponding to 'and' and 'or', respectively.

Here is an another sample code for timeout, employing 'deferred:timeout' macro.

Timeout macro:

```
(deferred:$
  (deferred:timeout
    1000 "canceled!"
    (deferred:process "sh" "-c" "sleep 3 | echo 'hello!'"))
  (deferred:nextc it
    (lambda (x) (insert x))))
```

2.1.9 Loop and Animation

This s-exp plays an animation at the cursor position for few seconds. Then, you can move cursor freely, because the animation does not block Emacs.

Returning a deferred object in the deferred tasks, the returned task is executed before the next deferred one that is statically connected on the source code. (In this case, the interrupt task is dynamically connected.)

Employing a recursive structure of deferred tasks, you can construct a deferred loop. It may seem the multi-thread in Emacs Lisp.

Loop and animation:

```
(lexical-let ((count 0) (anm "-/|\\-")
              (end 50) (pos (point))
              (wait-time 50))
  (deferred:$
    (deferred:next
      (lambda (x) (message "Animation started.")))

    (deferred:nextc it
      (deferred:lambda (x)
        (save-excursion
          (when (< 0 count)
            (goto-char pos) (delete-char 1))
          (insert (char-to-string
                   (aref anm (% count (length anm))))))
        (if (> end (incf count)) ; return nil to stop this loop
            (deferred:nextc (deferred:wait wait-time) self)))) ; return the deferred

    (deferred:nextc it
      (lambda (x)
```

```
        (save-excursion
          (goto-char pos) (delete-char 1))
        (message "Animation finished.")))))
```

- 'deferred:lambda' is an anaphoric macro in which 'self' refers itself. It is convenient to construct a recursive structure.

2.1.10 Wrapping asynchronous function

Let's say you have an asynchronous function which takes a callback. For example, dbus.el, xml-rpc.el and web-socket.el has such kind of asynchronous APIs. To use such libraries with deferred.el, you can make an unregistered deferred object using deferred:new and then start the deferred callback queue using deferred:callback-post in the callback given to the asynchronous function. If the asynchronous function supports "errorback", you can use deferred:errorback-post to pass the error information to the following callback queue.

In the following example, run-at-time is used as an example for the asynchronous function. Deferred.el already has deferred:wait for this purpose so that you don't need the following code if you want to use run-at-time.

```
(deferred:$
  (deferred:next
    (lambda ()
      (message "1")
      1))
  (deferred:nextc it
    (lambda (x)
      (lexical-let ((d (deferred:new #'identity)))
        (run-at-time 0 nil (lambda (x)
                             ;; Start the following callback queue now.
                             (deferred:callback-post d x))
                     x)
        ;; Return the unregistered (not yet started) callback
        ;; queue, so that the following queue will wait until it
        ;; is started.
        d)))
  ;; You can connect deferred callback queues
  (deferred:nextc it
    (lambda (x)
      (message "%s" (1+ x)))))
```

CHAPTER 3

concurrent.el manual

WRITE HERE

CHAPTER 4

deferred.el API

4.1 Basic functions

Function `deferred:next` (**&optional** callback, arg)

```
    Create a deferred object and schedule executing.

    :Arguments:
        CALLBACK
            a function with zero or one argument
    :Return: a deferred object

    This function is a short cut of following code::

     (deferred:callback-post (deferred:new callback)).
```

Function `deferred:nextc` (d, callback)

```
    Create a deferred object with OK callback and connect it to
    the given deferred object.

    :Arguments:
        D
            a deferred object
        CALLBACK
            a function with zero or one argument
    :Return: a deferred object

    Return a deferred object that wrap the given callback
    function.  Then, connect the created deferred object with the
    given deferred object.
```

Function `deferred:error` (d, callback)

```
    Create a deferred object with ERRORBACK and connect it to the
    given deferred object D.

    :Arguments:
        D
            a deferred object
        ERRORBACK
            a function with zero or one argument
    :Return: a deferred object
```

```
Return a deferred object that wraps the given function as an
ERRORBACK.  Then, connect the created deferred object with the
given deferred object.  The given ERRORBACK function catches the
error occurred in the previous task.

If this function does not throw an error, the subsequent callback
functions are executed.
```

Function **deferred:cancel** (d)

```
Cancel all callbacks and deferred chain in the deferred object.
```

Function **deferred:watch** (d, callback)

```
Create a deferred object with watch task and connect it to the given deferred object.
The watch task CALLBACK can not affect deferred chains with
return values. This function is used in following purposes,
simulation of try-finally block in asynchronous tasks, progress
monitoring of tasks.
```

Function **deferred:wait** (msec)

```
Return a deferred object scheduled at MSEC millisecond later.
```

Function **deferred:$** (**&rest** elements)

```
Anaphoric function chain macro for deferred chains.
```

4.2 Utility functions

Function **deferred:loop** (times-or-list, func)

```
Return a iteration deferred object.
```

Function **deferred:parallel** (**&rest** args)

```
Return a deferred object that calls given deferred objects or
functions in parallel and wait for all callbacks. The following
deferred task will be called with an array of the return
values. ARGS can be a list or an alist of deferred objects or
functions.
```

Function **deferred:earlier** (**&rest** args)

```
Return a deferred object that calls given deferred objects or
functions in parallel and wait for the first callback. The
following deferred task will be called with the first return
value. ARGS can be a list or an alist of deferred objects or
functions.
```

4.3 Wrapper functions

Function **deferred:call** (f **&rest** , args)

```
Call the given function asynchronously.
```

Function **deferred:apply** (f **&optional** , args)

```
Call the given function asynchronously.
```

Function **deferred:process** (command **&rest** , args)

```
A deferred wrapper of `start-process'. Return a deferred
object. The process name and buffer name of the argument of the
`start-process' are generated by this function automatically.
The next deferred object receives stdout string from the command
process.
```

Function `deferred:process-shell` (command **&rest** , args)

```
A deferred wrapper of `start-process-shell-command'. Return a deferred
object. The process name and buffer name of the argument of the
`start-process-shell-command' are generated by this function automatically.
The next deferred object receives stdout string from the command
process.
```

Function `deferred:process-buffer` (command **&rest** , args)

```
A deferred wrapper of `start-process'. Return a deferred
object. The process name and buffer name of the argument of the
`start-process' are generated by this function automatically.
The next deferred object receives stdout buffer from the command
process.
```

Function `deferred:process-shell-buffer` (command **&rest** , args)

```
A deferred wrapper of `start-process-shell-command'. Return a deferred
object. The process name and buffer name of the argument of the
`start-process-shell-command' are generated by this function automatically.
The next deferred object receives stdout buffer from the command
process.
```

Function `deferred:wait-idle` (msec)

```
Return a deferred object which will run when Emacs has been
idle for MSEC millisecond.
```

Function `deferred:url-retrieve` ()

Function `deferred:url-get` ()

Function `deferred:url-post` ()

4.4 Primitive functions

Function `deferred:new` (**&optional** callback)

```
Create a deferred object.
```

Function `deferred:succeed` (**&optional** arg)

```
Create a synchronous deferred object.
```

Function `deferred:fail` (**&optional** arg)

```
Create a synchronous deferred object.
```

Function `deferred:callback` (d **&optional** , arg)

```
Start deferred chain with a callback message.
```

Function `deferred:callback-post` (d **&optional** , arg)

```
Add the deferred object to the execution queue.
```

Function `deferred:errorback` (d **&optional** , arg)

```
Start deferred chain with an errorback message.
```

Function `deferred:errorback-post` (d **&optional** , arg)

```
Add the deferred object to the execution queue.
```

4.5 Utility Macros

Function `deferred:try` (d **&key** , catch, finally)

```
Try-catch-finally macro. This macro simulates the
try-catch-finally block asynchronously. CATCH and FINALLY can be
nil. Because of asynchrony, this macro does not ensure that the
task FINALLY should be called.
```

Function `deferred:timeout` (timeout-msec, timeout-form, d)

```
Time out macro on a deferred task D.  If the deferred task D
does not complete within TIMEOUT-MSEC, this macro cancels the
deferred task and return the TIMEOUT-FORM.
```

Function `deferred:processc` (d, command **&rest** , args)

```
Process chain of `0api:el.function.deferred:processdeferred0-
api:el.function.deferred:process`deferred:process'.
```

Function `deferred:process-bufferc` (d, command **&rest** , args)

```
Process chain of `0api:el.function.deferred:process0bufferdeferred0-
api:el.function.deferred:process0buffer`deferred:process-buffer'.
```

Function `deferred:process-shellc` (d, command **&rest** , args)

```
Process chain of `0api:el.function.deferred:processdeferred0-
api:el.function.deferred:process`deferred:process'.
```

Function `deferred:process-shell-bufferc` (d, command **&rest** , args)

```
Process chain of `0api:el.function.deferred:process0bufferdeferred0-
api:el.function.deferred:process0buffer`deferred:process-buffer'.
```

CHAPTER 5

concurrent.el API

5.1 Pseudo-thread

Function **cc:thread** (wait-time-msec **&rest** , body)

```
Return a thread object.
```

5.2 Generator

Function **cc:generator** (callback **&rest** , body)

```
Create a generator object. If BODY has `yield' symbols, it
means calling callback function CALLBACK.
```

5.3 Semaphore

Function **cc:semaphore-create** (permits-num)

```
Return a semaphore object with PERMITS-NUM permissions.
```

Function **cc:semaphore-acquire** (semaphore)

```
Acquire an execution permission and return deferred object to chain.
If this semaphore object has permissions, the subsequent deferred
task is executed immediately.  If this semaphore object has no
permissions, the subsequent deferred task is blocked. After the
permission is returned, the task is executed.
```

Function **cc:semaphore-release** (semaphore)

```
Release an execution permission. The programmer is responsible to return the permissio
```

Function **cc:semaphore-with** (semaphore, body-func **&optional** , error-func)

```
Execute the task BODY-FUNC asynchronously with the semaphore block.
```

Function **cc:semaphore-release-all** (semaphore)

```
Release all permissions for resetting the semaphore object.
If the semaphore object has some blocked tasks, this function
return a list of the tasks and clear the list of the blocked
tasks in the semaphore object.
```

Function **cc:semaphore-interrupt-all** (semaphore)

```
Clear the list of the blocked tasks in the semaphore and return a deferred object to c
This function is used for the interruption cases.
```

5.4 Signal

Function **cc:signal-channel** (**&optional** name, parent-channel)

```
Create a channel.
NAME is a channel name for debug.
PARENT-CHANNEL is an upstream channel. The observers of this channel can receive the u
In the case of using the function `0api:el.function.cc:signal0-
sendconcurrent0api:el.function.cc:signal0send`cc:signal-send', the observers of the up
api:el.function.cc:signal0send0globalconcurrent0api:el.function.cc:signal0-
send0global`cc:signal-send-global' can send a signal to the upstream channels from the
```

Function **cc:signal-connect** (channel, event-sym **&optional** , callback)

```
Append an observer for EVENT-SYM of CHANNEL and return a deferred object.
If EVENT-SYM is `t', the observer receives all signals of the channel.
If CALLBACK function is given, the deferred object executes the
CALLBACK function asynchronously. One can connect subsequent
tasks to the returned deferred object.
```

Function **cc:signal-send** (channel, event-sym **&rest** , args)

```
Send a signal to CHANNEL. If ARGS values are given, observers can get the values by fo
```

Function **cc:signal-send-global** (channel, event-sym **&rest** , args)

```
Send a signal to the most upstream channel.
```

Function **cc:signal-disconnect** (channel, deferred)

```
Remove the observer object DEFERRED from CHANNEL and return
the removed deferred object.
```

Function **cc:signal-disconnect-all** (channel)

```
Remove all observers.
```

5.5 Dataflow

Function **cc:dataflow-environment** (**&optional** parent-env, test-func, channel)

```
Create a dataflow environment.
PARENT-ENV is the default environment. If this environment doesn't have the entry A an
TEST-FUNC is a test function that compares the entry keys. The default function is `eq
CHANNEL is a channel object that sends signals of variable events. Observers can recei

``get-first``
    the fist referrer is waiting for binding
``get-waiting``
    another referrer is waiting for binding
``set``
    a value is bound
``get``
    returned a bound value
``clear``
    cleared one entry
``clear-all``
    cleared all entries
```

Function **cc:dataflow-get** (df, key)

```
Return a deferred object that can refer the value which is indicated by KEY.
If DF has the entry that bound value, the subsequent deferred task is executed immedia
If not, the task is deferred till a value is bound.
```

Function `cc:dataflow-get-sync` (df, key)

```
Return the value which is indicated by KEY synchronously.
If the environment DF doesn't have an entry of KEY, this function returns nil.
```

Function `cc:dataflow-set` (df, key, value)

```
Bind the VALUE to KEY in the environment DF.
If DF already has the bound entry of KEY, this function throws an error signal.
VALUE can be nil as a value.
```

Function `cc:dataflow-clear` (df, key)

```
Clear the entry which is indicated by KEY.
This function does nothing for the waiting deferred objects.
```

Function `cc:dataflow-get-avalable-pairs` (df)

```
Return an available key-value alist in the environment DF and the parent ones.
```

Function `cc:dataflow-get-waiting-keys` (df)

```
Return a list of keys which have waiting deferred objects in the environment DF and th
```

Function `cc:dataflow-clear-all` (df)

```
Clear all entries in the environment DF.
This function does nothing for the waiting deferred objects.
```

Function `cc:dataflow-connect` (df, event-sym **&optional** , callback)

```
Append an observer for EVENT-SYM of the channel of DF and return a deferred object.
See the docstring of `0api:el.function.cc:dataflow0environmentconcurrent0-
api:el.function.cc:dataflow0environment`cc:dataflow-environment' for details.
```

CHAPTER 6

Resources for deferred.el/concurrent.el development

6.1 How to build document

Document of deferred.el and concurrent.el are build using Sphinx and sphinxcontrib-emacs. It is available via read the docs.

To build document, you need to install Sphinx and sphinxcontrib-emacs, for example, using pip install:

```
cd PATH/TO/emacs-deferred
pip install Sphinx
pip install -r doc/requirements.txt
```

Then, goto `doc/` directory and build document:

```
cd doc/
make html
```

You have html document in `doc/build/html/`.

If autodocument does not fetch updated docstrings from `*.el` files, use `make clean html` instead.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

C

D

E