Package 'ipc'

June 23, 2019

Type Package

Title Tools for Message Passing Between Processes

Version 0.1.3

Author Ian E. Fellows

Maintainer Ian E. Fellows <ian@fellstat.com>

Description Provides tools for passing messages between R processes.
Shiny Examples are provided showing how to perform useful tasks such as:
updating reactive values from within a future, progress bars for long running
async tasks, and interrupting async tasks based on user input.

URL https://github.com/fellstat/ipc

BugReports https://github.com/fellstat/ipc/issues

Imports R6, shiny, txtq

License MIT + file LICENCE

Encoding UTF-8

LazyData true

Suggests testthat, knitr, rmarkdown, future, promises, redux

VignetteBuilder knitr

RoxygenNote 6.1.1

NeedsCompilation no

Repository CRAN

Date/Publication 2019-06-23 06:00:03 UTC

R topics documented:

---

ipc-package	*Tools for performing async communication between workers in shiny*

Description

Tools for performing async communication between workers in shiny

Author(s)

Ian Fellows <ian@fellstat.com>

AsyncInterruptor	*An interruptor useful for stopping child processes.*

Description

This class is a simple wrapper around a Queue object making adding interrupt checking to future code easy to implement and read.

Arguments

queue	a shiny queue
msg	An error message string.

Details

Methods

`initialize(queue=shinyQueue())` Creates a new interruptor.

`interrupt(msg="Signaled Interrupt")` Signals an interrupt

`execInterrupts()` Executes anything pushed to the queue, including interrupts.

`getInterrupts()` Gets the result of the queue's executing, not throwing the interrupts.

Examples

```
library(future)
strategy <- "future::multisession"
plan(strategy)
inter <- AsyncInterruptor$new()
fut <- future({
  for(i in 1:100){
    Sys.sleep(.01)
    inter$execInterrupts()
  }
})
inter$interrupt("Error: Stop Future")
try(value(fut))
inter$destroy()

# Clean up multisession cluster
plan(sequential)
```

AsyncProgress   *A progress bar object where inc and set are usable within other pro-*
                                 cesses

Description

An async compatible wrapper around Shiny's progress bar. It should be instatiated from the main process, but may be closed, set and incremented from any process.

Arguments

session	The Shiny session object, as provided by `shinyServer` to the server function.
min	The value that represents the starting point of the progress bar. Must be less tham `max`.
max	The value that represents the end of the progress bar. Must be greater than `min`.
message	A single-element character vector; the message to be displayed to the user, or `NULL` to hide the current message (if any).
detail	A single-element character vector; the detail message to be displayed to the user, or `NULL` to hide the current detail message (if any). The detail message will be shown with a de-emphasized appearance relative to `message`.
value	A numeric value at which to set the progress bar, relative to `min` and `max`.
queue	A Queue object for message passing
millis	How often in milliseconds should updates to the progress bar be checked for.

Details

Methods

`initialize(..., queue=shinyQueue(), millis=250, value=NULL, message=NULL, detail=NULL)`
Creates a new progress panel and displays it.

`set(value = NULL, message = NULL, detail = NULL)` Updates the progress panel. When called
the first time, the progress panel is displayed.

`inc(amount = 0.1, message = NULL, detail = NULL)` Like set, this updates the progress panel.
The difference is that inc increases the progress bar by amount, instead of setting it to a spe-
cific value.

`sequentialClose()` Removes the progress panel and destroys the queue. Must be called from
main process.

`close()` Fires a close signal and may be used from any process.

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
library(shiny)
library(future)
plan(multiprocess)
ui <- fluidPage(
  actionButton("run","Run"),
  tableOutput("dataset")
)

server <- function(input, output, session) {

  dat <- reactiveVal()
  observeEvent(input$run, {
    progress <- AsyncProgress$new(session, min=1, max=15)
    future({
      for (i in 1:15) {
        progress$set(value = i)
        Sys.sleep(0.5)
      }
      progress$close()
      cars
    }) %...>% dat
    NULL
  })

  output$dataset <- renderTable({
    req(dat())
  })
}

shinyApp(ui, server)
}
```

Consumer	*A Class for reading and executing tasks from a source*

Description

Methods

`initialize(source, env=parent.frame(2))` Creates a Consumer object linked to the `source`.

`setSource(source)` Sets the Source for this consumer.

`getSource(source)` Gets the Source of this consumer.

`consume(throwErrors=TRUE, env=parent.frame())` Executes all (unprocessed) signals fired to source from a Producer. if `throwErrors` is TRUE, the first error encountered is thrown after executing all signals. Signals are executed in the env environment. If env is NULL, the enviroment set at initialization is used.

`start(millis=250, throwErrors=TRUE, env=parent.frame())` Starts executing `consume` every `millis` milliseconds. `throwErrors` and env are passed down to `consume`

`stop()` Stops the periodic execution of `consume`.

`clearHandlers()` Removes all handlers

`removeHandler(signal, index)` Removes handler from 'signal' with position index

`addHandler(func, signal)` Adds a handler for 'signal'. func should take three parameters: 1. the signal, 2. the message object, and 3. the evaluation environment.

`initHandlers()` Adds the two default executeors.

`finalize()` runs stop on object distruction

Arguments

`source`	a source, e.g. TextFileSource.
`millis`	milliseconds.
`env`	An enviroment specifying where to execute signals.
`signal`	A string.
`index`	A position.

---

defaultSource	*Get/set the class used to sink/read from the file system*

Description

Get/set the class used to sink/read from the file system

Usage

```
defaultSource(sourceClass)
```

Arguments

sourceClass   An R6 object

Producer	*A Class for sending signals to a source*

Description

Methods

`initialize(source)` Creates a Producer object linked to the `source`.

`setSource(source)` Sets the `Source` for this producer.

`getSource(source)` Gets the `Source` of this producer.

`fire(signal, obj=NA)` Sends a signal to the source with associates object `obj`.

`fireEval(expr, env)` Signals for execution of the expression `obj` with values from the environment (or list) `env` substituted in.

`fireDoCall(name, param)` Signals for execution of the function whose string value is `name` with the parameters in list `param`.

`fireDoCall(name, ...)` Signals for execution of the function whose string value is `name` with the parameters ....

Details

@param obj The object to associate with the signal. @param signal A string signal to send. @param env An environment or list for substitution @param param A list of function parameters. @param expr An expression to evaluate. @param name the name of the function @param ... parameters to be passed to function

---

queue   *Create a Queue object*

Description

Create a Queue object

Usage

```
queue(source = defaultSource()$new(), producer = Producer$new(source),
  consumer = Consumer$new(source))
```

Arguments

source	The source for reading and writing the queue
producer	The producer for the source
consumer	The consumer of the source

---

redisConfig   *Get/set redis configuration*

Description

Get/set redis configuration

Usage

```
redisConfig(config)
```

Arguments

config	a function generating id strings

---

redisIdGenerator   *Get/set the location for temporary files*

Description

Get/set the location for temporary files

Usage

```
redisIdGenerator(generator)
```

Arguments

generator	a function generating id strings

RedisSource   *Reads and writes the queue to a redis db*

Description

Reads and writes the queue to a redis db

Arguments

id	An identifier to use for the queue
config	A configuration list for redux::hiredis
n	The number of records to pop (-1 indicates all available).
msg	A string indicating the signal.
obj	The object to associate with the signal.

ShinyConsumer   *A Consumer class with common task handlers useful in Shiny apps*

Description

In addtion to 'eval' and 'function' signals, ShinyConsumer object process 'interrupt' and 'notify' signals for throwing errors and displaying Shiny notifictions.

---

shinyExample   *Run Example Shiny Apps*

Description

Run Example Shiny Apps

Usage

```
shinyExample(application = c("progress", "changeReactive", "cancel"))
```

Arguments

application	The example to run

Details

'progress' is an example application with a long running analysis that is cancelable and has a progress bar. 'changeReaction' is the old faithful example, but with the histogram colors changing over time. 'cancel' is an example with a cancelable long running process.

ShinyProducer *A Producer with methods specific for Shiny*

Description

A Producer object with additional methods for firing interrupts, shiny notifications, and reactive value assignments.

Details

Methods

`fireInterrupt(msg="Interrupt")` Sends an error with message `msg`.

`fireNotify(msg="Interrupt")` Sends a signal to create a shiy Notifiction with message `msg`.

`fireAssignReactive(name, value)` Signals for assignment for reactive `name` to `value`.

> @param msg A string @param name The name of the reactive value. @param value The value to assign the reactive to.

---

shinyQueue *Create a Queue object*

Description

Create a Queue object

Usage

```
shinyQueue(source = defaultSource()$new(),
  producer = ShinyProducer$new(source),
  consumer = ShinyConsumer$new(source),
  session = shiny::getDefaultReactiveDomain())
```

Arguments

source	The source for reading and writing the queue
producer	The producer for the source
consumer	The consumer of the source
session	A Shiny session

Details

Creates a Queue object for use with shiny, backed by ShinyTextSource, ShiyProducer and Shiny-Consumer objects by default. The object will be cleaned up and destroyed on session end.

---

stopMulticoreFuture *Stops a future run in a multicore plan*

Description

Stops a future run in a multicore plan

Usage

```
stopMulticoreFuture(x)
```

Arguments

x   The MulticoreFuture

Details

This function sends terminate and kill signals to the process running the future, and will only work for futures run on a multicore plan. This approach is not recommended for cases where you can listen for interrupts within the future (with `AsyncInterruptor`). However, for cases where long running code is in an external library for which you don't have control, this can be the only way to terminate the execution.

---

tempFileGenerator *Get/set the location for temporary files*

Description

Get/set the location for temporary files

Usage

```
tempFileGenerator(tempfile)
```

Arguments

tempfile   a function generating working file path (e.g. tempfile())

TextFileSource *Reads and writes the queue to a text file*

Description

A wrapper around `txtq`. This object saves signals and associated objects to and queue, and retrieves them for processing.

Arguments

filePath	The path to the file
n	The number of records to pop (-1 indicates all available).
msg	A string indicating the signal.
obj	The object to associate with the signal.

Index