Package ‘ipc’
January 26, 2023

Type Package
Title Tools for Message Passing Between Processes
Version 0.1.4
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
Suggests testthat, knitr, rmarkdown, future, promises, redux
VignetteBuilder knitr
RoxygenNote 7.2.2
NeedsCompilation no
Repository CRAN
Date/Publication 2023-01-26 06:50:02 UTC

R topics documented:

ipc-package ......................................................... 2
AsyncInterruptor ................................................... 2
AsyncProgress ....................................................... 4
Consumer ............................................................. 7
defaultSource ....................................................... 9
Producer ............................................................. 10
Queue ................................................................. 11
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.

Author(s)

Ian Fellows <ian@fellstat.com>

AsyncInterruptor

An interruptor useful for stopping child processes.

Description

An interruptor useful for stopping child processes.

Details

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

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.
AsyncInterruptor

Methods

Public methods:

• AsyncInterruptor$new()
• AsyncInterruptor$interrupt()
• AsyncInterruptor$execInterrupts()
• AsyncInterruptor$getInterrupts()
• AsyncInterruptor$destroy()
• AsyncInterruptor$clone()

Method new(): Create the object

Usage:
AsyncInterruptor$new(queue = shinyQueue())

Arguments:
queue  The underlying queue object to use for interruption

Method interrupt(): signal an error

Usage:
AsyncInterruptor$interrupt(msg = "Signaled Interrupt")

Arguments:
msg  The error message

Method execInterrupts(): Execute any interruptions that have been signaled

Usage:
AsyncInterruptor$execInterrupts()

Method getInterrupts(): Get any interruptions that have been signaled without throwing them as errors

Usage:
AsyncInterruptor$getInterrupts()

Method destroy(): Cleans up object after use

Usage:
AsyncInterruptor$destroy()

Method clone(): The objects of this class are cloneable with this method.

Usage:
AsyncInterruptor$clone(deep = FALSE)

Arguments:
deep  Whether to make a deep clone.
Examples

```r
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 processes

Description

A progress bar object where inc and set are usable within other processes

Details

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.

Methods

Public methods:

- `AsyncProgress$new()`
- `AsyncProgress$setMax()`
- `AsyncProgress$setMin()`
- `AsyncProgress$sequentialClose()`
- `AsyncProgress$set()`
- `AsyncProgress$inc()`
- `AsyncProgress$close()`
- `AsyncProgress$clone()`

Method `new()`: Creates a new progress panel and displays it.
**Usage:**

AsyncProgress$new(
    ..., queue = shinyQueue(),
    millis = 250,
    value = NULL,
    message = NULL,
    detail = NULL
)

**Arguments:**

... Additional parameters to be passed to Shiny::Progress

queue  A Queue object for message passing

millis  How often in milliseconds should updates to the progress bar be checked for.

value  A numeric value at which to set the progress bar, relative to min and max.

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.

**Method getMax():** Returns the maximum

**Usage:**

AsyncProgress$getMax()

**Method getMin():** Returns the minimum

**Usage:**

AsyncProgress$getMin()

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

**Usage:**

AsyncProgress$sequentialClose()

**Method set():** Updates the progress panel. When called the first time, the progress panel is displayed.

**Usage:**

AsyncProgress$set(value = NULL, message = NULL, detail = NULL)

**Arguments:**

value  A numeric value at which to set

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.
**Method** inc(): Like set, this updates the progress panel. The difference is that inc increases the progress bar by amount, instead of setting it to a specific value.

*Usage:*
```r
AsyncProgress$inc(amount = 0.1, message = NULL, detail = NULL)
```

*Arguments:*
- `amount`: the size of the increment.
- `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`.

**Method** close(): Fires a close signal and may be used from any process.

*Usage:*
```r
AsyncProgress$close()
```

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*
```r
AsyncProgress$clone(deep = FALSE)
```

*Arguments:*
- `deep`: Whether to make a deep clone.

**Examples**

```r
## Only run examples in interactive R sessions
if (interactive()) {
  library(shiny)
  library(future)
  plan(multisession)
  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 #return something other than the future so the UI is not blocked
    })

  }
}
```
Consumer

A Class for reading and executing tasks from a source

Description
A Class for reading and executing tasks from a source

Public fields

handlers  A list of handlers
stopped   Is currently stopped.
laterHandle  A callback handle.

Methods

Public methods:

• Consumer$new()
• Consumer$setSource()
• Consumer$getSource()
• Consumer$consume()
• Consumer$start()
• Consumer$stop()
• Consumer$addHandler()
• Consumer$clearHandlers()
• Consumer$removeHandler()
• Consumer$initHandlers()
• Consumer$finalize()
• Consumer$clone()

Method new(): Creates the object.

Usage:
Consumer$new(source)

Arguments:
source  A source, e.g. TextFileSource.

**Method setSource():** Sets the source.

*Usage:*
Consumer$setSource(source)

*Arguments:*
source  A source, e.g. TextFileSource.

**Method getSource():** Gets the source.

*Usage:*
Consumer$getSource()

**Method consume():** 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 environment set at initialization is used.

*Usage:*
Consumer$consume(throwErrors = TRUE, env = parent.frame())

*Arguments:*
throwErrors  Should errors be thrown or caught.
env  The execution environment.

**Method start():** Starts executing consume every millis milliseconds. throwErrors and env are passed down to consume

*Usage:*
Consumer$start(millis = 250, env = parent.frame())

*Arguments:*
millis  milliseconds.
env  The execution environment.

**Method stop():** Stops the periodic execution of consume.

*Usage:*
Consumer$stop()

**Method addHandler():** Adds a handler for 'signal'. func

*Usage:*
Consumer$addHandler(func, signal)

*Arguments:*
func  The function which takes three parameters: 1. the signal, 2. the message object, and 3. the evaluation environment.
signal  A string to bind the function to.

**Method clearHandlers():** Removes all handlers

*Usage:*
Consumer$clearHandlers()
Method `removeHandler()`: Removes a single handler.

Usage:
Consumer$removeHandler(signal, index)

Arguments:
signal  The signal of the handler.
index   The index of the handler to remove from the signal.

Method `initHandlers()`: Adds default handlers.

Usage:
Consumer$initHandlers()

Method `finalize()`: Cleans up object.

Usage:
Consumer$finalize()

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
Consumer$clone(deep = FALSE)

Arguments:
depth Whether to make a deep clone.

---

**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

A Class for sending signals to a source

Methods

Public methods:

- `Producer$new()`
- `Producer$setSource()`
- `Producer$getSource()`
- `Producer$fire()`
- `Producer$fireEval()`
- `Producer$fireDoCall()`
- `Producer$fireCall()`
- `Producer$clone()`

Method `new()`: Creates a Producer object linked to the source.

Usage:

```
Producer$new(source)
```

Arguments:

- `source` A source.

Method `setSource()`: Setter for source.

Usage:

```
Producer$setSource(source)
```

Arguments:

- `source` A source.

Method `getSource()`: Getter for source.

Usage:

```
Producer$getSource()
```

Method `fire()`: Sends a signal to the source with associates object `obj`.

Usage:

```
Producer$fire(signal, obj = NA)
```

Arguments:

- `signal` A string signal to send.
- `obj` The object to associate with the signal.
**Method** fireEval(): Signals for execution of the expression obj with values from the environment (or list) env substituted in.

*Usage:*
Producer$fireEval(expr, env)

*Arguments:*
expr An expression to evaluate.
env An environment or list for substitution

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

*Usage:*
Producer$fireDoCall(name, param)

*Arguments:*
name the name of the function
param A list of function parameters.

**Method** fireCall(): Signals for execution of the function whose string value is name with the parameters ....

*Usage:*
Producer$fireCall(name, ...)

*Arguments:*
name the name of the function
... The arguments to the function.

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*
Producer$clone(deep = FALSE)

*Arguments:*
deep Whether to make a deep clone.

---

**Queue**

*A Class containing a producer and consumer*

---

**Description**

Creates a Queue object for inter-process communication. Its members producer and consumer are the main entry points for sending and receiving messages respectively.

**Usage**

```r
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

Details

This function creates a queue object for communication between different R processes, including forks of the same process. By default, it uses txtq backage as its backend. Technically, the information is sent through temporary files, created in a new directory inside the session-specific temporary folder (see *tempfile*). This requires that the new directory is writeable, this is normally the case but if *Sys.umask* forbids writing, the communication fails with an error.

Public fields

- producer   A Producer object
- consumer   a Consumer object.

Methods

**Public methods:**

- `Queue$new()`
- `Queue$destroy()`
- `Queue$clone()`

**Method new():** Create a Queue object

*Usage:*

```
Queue$new(source, prod, cons)
```

*Arguments:*

- `source`   The source to use for communication.
- `prod`     A Producer object.
- `cons`     A Consumer object.

**Method destroy():** clean up object after use.

*Usage:*

```
Queue$destroy()
```

**Method clone():** The objects of this class are cloneable with this method.

*Usage:*

```
Queue$clone(deep = FALSE)
```

*Arguments:*

- `deep`   Whether to make a deep clone.
Examples

```r
## Not run:
library(parallel)
library(future)
library(promises)
plan(multisession)

q <- queue()

# communicate from main session to child
fut <- future({
  for(i in 1:1000){
    Sys.sleep(.1)
    q$consumer$consume()
  }
})

q$producer$fireEval(stop("Stop that child"))
cat(try(value(fut)))

# Communicate from child to main session
j <- 0
fut <- future({
  for(i in 1:10){
    Sys.sleep(.2)
    # set j in the main thread substituting i into the expression
    q$producer$fireEval(j <- i, env=list(i=i))
  }
})

while(j < 10){
  q$consumer$consume() # collect and execute assignments
  cat("j = ", j, "\n")
  Sys.sleep(.1)
}

fut <- future({
  for(i in 1:10){
    Sys.sleep(.2)
    # set j in the main thread substituting i into the expression
    q$producer$fireEval(print(i), env=list(i=i))
  }
})

q$consumer$start() # execute `consume` at regular intervals

# clean up
q$destroy()
```
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

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

**Methods**

**Public methods:**
- RedisSource\$new()
- RedisSource\$getRedisConnection()
- RedisSource\$pop()
- RedisSource\$push()
- RedisSource\$destroy()
- RedisSource\$finalize()
- RedisSource\$clone()

**Method new()**: Creates a redis source object.

*Usage:*
RedisSource\$new(id = redisIdGenerator(), config = redisConfig())

*Arguments:*
- id An identifier to use for the queue
- config A configuration list for redux::hiredis

**Method getRedisConnection()**: Returns the underlying redis connection.

*Usage:*
RedisSource\$getRedisConnection()

**Method pop()**: Removes n items from the source and returns them.

*Usage:*
RedisSource\$pop(n = -1)

*Arguments:*
- n The number of records to pop (-1 indicates all available).

**Method push()**: Adds an item to the source.

*Usage:*
RedisSource\$push(msg, obj)

*Arguments:*
- msg A string indicating the signal.
- obj The object to associate with the signal.
Method `destroy()`: Cleans up source after use.

Usage:
RedisSource$destroy()

Method `finalize()`: finalize

Usage:
RedisSource$finalize()

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
RedisSource$clone(deep = FALSE)

Arguments:
- deep  Whether to make a deep clone.

---

**ShinyConsumer**

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

Description

A Consumer class with common task handlers useful in Shiny apps

A Consumer class with common task handlers useful in Shiny apps

Details

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

Super class

`ipc::Consumer` -> ShinyConsumer

Methods

**Public methods:**

- `ShinyConsumer$initHandlers()
- `ShinyConsumer$clone()

Method `initHandlers()`: Adds default handlers

Usage:
ShinyConsumer$initHandlers()

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
ShinyConsumer$clone(deep = FALSE)

Arguments:
- deep  Whether to make a deep clone.
shinyExample

**Description**

Run Example Shiny Apps

**Usage**

```r
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. 'changeReactive' 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 with methods specific for Shiny

**Details**

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

**Super class**

```r
ipc::Producer -> ShinyProducer
```
Methods

Public methods:

- ShinyProducer$fireInterrupt()
- ShinyProducer$fireNotify()
- ShinyProducer$fireAssignReactive()
- ShinyProducer$clone()

Method `fireInterrupt()`: Sends an error with message `msg`.

Usage:
ShinyProducer$fireInterrupt(msg = "Interrupt")

Arguments:
msg A string

Method `fireNotify()`: Sends a signal to create a shiny Notification with message `msg`.

Usage:
ShinyProducer$fireNotify(msg = "Notification")

Arguments:
msg A string

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

Usage:
ShinyProducer$fireAssignReactive(name, value)

Arguments:
name The name of the reactive value.
value The value to assign the reactive to.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
ShinyProducer$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

shinyQueue | Create a Queue object

Description

Create a Queue object
Usage

```r
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

```r
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.

Note that multicore is not supported on Windows machines or within RStudio.
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

Reads and writes the queue to a text file

Reads and writes the queue to a text file

Details

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

Methods

Public methods:

- TextFileSource$new()
- TextFileSource$pop()
- TextFileSource$push()
- TextFileSource$destroy()
- TextFileSource$clone()

Method new(): Creates a TextFileSource

Usage:

TextFileSource$new(filePath = tempFileGenerator()())

Arguments:

filePath  The path to the file.

Method pop(): removes n items from the source and returns them
Usage:
TextFileSource$pop(n = -1)

Arguments:
 n The number of records to pop (-1 indicates all available).

Method push(): Adds an item to the source.
 Usage:
 TextFileSource$push(msg, obj)

Arguments:
 msg A string indicating the signal.
 obj The object to associate with the signal.

Method destroy(): Cleans up source after use.
 Usage:
 TextFileSource$destroy()

Method clone(): The objects of this class are cloneable with this method.
 Usage:
 TextFileSource$clone(deep = FALSE)

Arguments:
 deep Whether to make a deep clone.
Index

AsyncInterruptor, 2
AsyncProgress, 4

Consumer, 7
defaultSource, 9

ipc-package, 2
dpc::Consumer, 16
dpc::Producer, 17

Producer, 10
Queue, 11
queue (Queue), 11

redisConfig, 14
redisIdGenerator, 14
RedisSource, 15

ShinyConsumer, 16
shinyExample, 17
ShinyProducer, 17
shinyQueue, 18
stopMulticoreFuture, 19
Sys.umask, 12

tempfile, 12
tempFileGenerator, 20
TextFileSource, 20