Package 'mirai'

May 11, 2023

Type Package

Title Minimalist Async Evaluation Framework for R

Version 0.8.7

Description Lightweight parallel code execution and distributed computing. Designed for simplicity, a 'mirai' evaluates an R expression asynchronously, on local or network resources, resolving automatically upon completion. Features efficient task scheduling, scalability beyond R connection limits, and transports faster than TCP/IP for inter-process communications, courtesy of 'nanonext' and 'NNG' (Nanomsg Next Gen).

License GPL (>= 3)

BugReports https://github.com/shikokuchuo/mirai/issues

URL https://shikokuchuo.net/mirai/, https://github.com/shikokuchuo/mirai/

Encoding UTF-8

Depends R (>= 2.12)

Imports nanonext (>= 0.8.3)

RoxygenNote 7.2.3

NeedsCompilation no

Author Charlie Gao [aut, cre] (<https://orcid.org/0000-0002-0750-061X>), Hibiki AI Limited [cph]

Maintainer Charlie Gao <charlie.gao@shikokuchuo.net>

Repository CRAN

Date/Publication 2023-05-11 07:30:02 UTC

R topics documented:

`mirai-package` *mirai: Minimalist Async Evaluation Framework for R*

Description

Lightweight parallel code execution and distributed computing. Designed for simplicity, a 'mirai' evaluates an R expression asynchronously, on local or network resources, resolving automatically upon completion. Features efficient task scheduling, scalability beyond R connection limits, and transports faster than TCP/IP for inter-process communications, courtesy of 'nanonext' and 'NNG' (Nanomsg Next Gen).

Notes

For local mirai processes, the default transport for inter-process communications is platform-dependent: abstract Unix domain sockets on Linux, Unix domain sockets on MacOS, Solaris and other POSIX platforms, and named pipes on Windows.

This may be overriden if required by specifying 'url' in the `daemons` interface, and starting server and/or dispatcher processes manually using `server` and `dispatcher` respectively, on the local machine.

Links

mirai website: https://shikokuchuo.net/mirai/
mirai on CRAN: https://cran.r-project.org/package=mirai

nanonext website: https://shikokuchuo.net/nanonext/
nanonext on CRAN: https://cran.r-project.org/package=nanonext

NNG website: https://nng.nanomsg.org/

Author(s)

Charlie Gao <charlie.gao@shikokuchuo.net> (ORCID)

`call_mirai` *mirai (Call Value)*

Description

Call the value of a mirai, waiting for the the asynchronous operation to resolve if it is still in progress.

Usage

```
call_mirai(aio)
```

Arguments

`aio`	a 'mirai' object.

Details

This function will wait for the async operation to complete if still in progress (blocking).

A blocking call can be sent a user interrupt with e.g. ctrl+c. If the ongoing execution in the mirai is interruptible, it will resolve into an object of class 'miraiInterrupt' and 'errorValue'. `is_mirai_interrupt` may be used to handle such cases.

If an error occurs in evaluation, the error message is returned as a character string of class 'miraiError' and 'errorValue'. `is_mirai_error` may be used to test for this.

`is_error_value` tests for all error conditions including mirai errors, interrupts, and timeouts.

The mirai updates itself in place, so to access the value of a mirai `x` directly, use `call_mirai(x)$data`.

Value

The passed mirai (invisibly). The retrieved value is stored at `$data`.

Alternatively

The value of a mirai may be accessed at any time at `$data`, and if yet to resolve, an 'unresolved' logical NA will be returned instead.

Using `unresolved` on a mirai returns TRUE only if a mirai has yet to resolve and FALSE otherwise. This is suitable for use in control flow statements such as `while` or `if`.

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

# using call_mirai()
df1 <- data.frame(a = 1, b = 2)
df2 <- data.frame(a = 3, b = 1)
m <- mirai(as.matrix(rbind(df1, df2)), .args = list(df1, df2), .timeout = 1000)
```

```
call_mirai(m)$data

# using unresolved()
m <- mirai({
  res <- rnorm(n)
  res / rev(res)
  },
  n = 1e6)
while (unresolved(m)) {
  cat("unresolved\n")
  Sys.sleep(0.1)
}
str(m$data)

}
```

---

daemons *daemons (Persistent Server Processes)*

Description

Set 'daemons' or background persistent server processes receiving `mirai` requests. These are, by default, automatically created on the local machine. Alternatively, a client URL may be specified to receive connections from remote servers started with `server` for distributing tasks across the network. Daemons may take advantage of the dispatcher, which ensures that tasks are assigned to servers efficiently on a FIFO basis, or else the low-level approach of distributing tasks to servers in an even fashion.

Usage

```
daemons(n, url = NULL, dispatcher = TRUE, ..., .compute = "default")
```

Arguments

`n`	integer number of daemons (server processes) to set.
`url`	[default NULL] if specified (for connecting to remote servers), the client URL as a character vector, including a port accepting incoming connections (and optionally for websockets a path) e.g. 'tcp://192.168.0.2:5555' or 'ws://192.168.0.2:5555/path'.
`dispatcher`	[default TRUE] logical value whether to use dispatcher. Dispatcher is a background process that connects to servers on behalf of the client and ensures FIFO scheduling, queueing tasks if necessary (see Dispatcher section below).
`...`	additional arguments passed through to `dispatcher` if using dispatcher and/or `server` if launching local daemons.
`.compute`	[default 'default'] character compute profile to use for creating the daemons (each compute profile has its own set of daemons for connecting to different resources).

Details

For viewing the currrent status, specify `daemons()` with no arguments.

Use `daemons(0)` to reset daemon connections:

- A reset is required before revising settings for the same compute profile, otherwise changes are not registered.
- All connected daemons and/or dispatchers exit automatically.
- {mirai} reverts to the default behaviour of creating a new background process for each request.

When specifying a client URL, all daemons dialing into the client are detected automatically and resources may be added or removed at any time.

If the client session ends, for whatever reason, all connected dispatcher and daemon processes automatically exit as soon as their connections are dropped. If a daemon is processing a task, it will exit as soon as the task is complete.

Value

Setting daemons: integer number of daemons set, or the character client URL.

Viewing current status: a named list comprising:

- **connections** - number of active connections at the client. Always 1L when using dispatcher as there is only a single connection to the dispatcher, which then in turn connects to the servers.
- **daemons** - if using dispatcher: a status matrix (see Status Matrix section below), or else an integer 'errorValue' if communication with the dispatcher was unsuccessful. If not using dispatcher: the number of daemons set, or else the client URL.

Dispatcher

By default `dispatcher = TRUE`. This launches a background process running `dispatcher`. A dispatcher connects to servers on behalf of the client and queues tasks until a server is able to begin immediate execution of that task, ensuring FIFO scheduling. Dispatcher uses synchronisation primitives from `nanonext`, waiting rather than polling for tasks, which is efficient both in terms of consuming no resources while waiting, and also being fully synchronised with events (having no latency).

By specifying `dispatcher = FALSE`, servers connect to the client directly rather than through a dispatcher. The client sends tasks to connected servers immediately in an evenly-distributed fashion. However, optimal scheduling is not guaranteed as the duration of tasks cannot be known *a priori*, such that tasks can be queued at a server behind a long-running task while other servers remain idle. Nevertheless, this provides a resource-light approach suited to working with similar-length tasks, or where concurrent tasks typically do not exceed available daemons.

Local Daemons

Daemons provide a potentially more efficient solution for asynchronous operations as new processes no longer need to be created on an *ad hoc* basis.

Supply the argument 'n' to set the number of daemons. New background `server` processes are automatically created on the local machine connecting back to the client process, either directly or via a dispatcher.

Distributed Computing

Specifying 'url' allows tasks to be distributed across the network.

The client URL should be in the form of a character string such as: 'tcp://192.168.0.2:5555' at which server processes started using `server` should connect to.

Alternatively, to listen to port 5555 on all interfaces on the local host, specify either 'tcp://:5555', 'tcp://*:5555' or 'tcp://0.0.0.0:5555'.

Specifying the wildcard value zero for the port number e.g. 'tcp://:0' or 'ws://:0' will automatically assign a free ephemeral port. Use `daemons()` to inspect the actual assigned port at any time.

With Dispatcher

When using dispatcher, it is recommended to use a websocket URL rather than TCP, as this requires only one port to connect to all servers: a websocket URL supports a path after the port number, which can be made unique for each server.

Specifying a single client URL such as 'ws://192.168.0.2:5555' with `n = 6` will automatically append a sequence to the path, listening to the URLs 'ws://192.168.0.2:5555/1' through 'ws://192.168.0.2:5555/6'.

Alternatively, specify a vector of URLs to listen to arbitrary port numbers / paths. In this case it is optional to supply 'n' as this can be inferred by the length of vector supplied.

Individual `server` instances should then be started on the remote resource, which dial in to each of these client URLs. At most one server should be dialled into each URL at any given time.

The dispatcher automatically adjusts to the number of servers actually connected. Hence it is possible to dynamically scale up or down the number of servers as required, subject to the maximum number initially specified.

Alternatively, supplying a single TCP URL will listen on a block of URLs with ports starting from the supplied port number and incrementing by one for 'n' specified e.g. the client URL 'tcp://192.168.0.2:5555' with `n = 6` listens to the contiguous block of ports 5555 through 5560.

Without Dispatcher

A TCP URL may be used in this case as the client listens at only one address, utilising a single port.

The network topology is such that server daemons (started with `server`) or indeed dispatchers (started with `dispatcher`) dial into the same client URL.

'n' is not required in this case, and disregarded if supplied, as network resources may be added or removed at any time. The client automatically distributes tasks to all connected servers and dispatchers.

Compute Profiles

By default, the 'default' compute profile is used. Providing a character value for '.compute' creates a new compute profile with the name specified. Each compute profile retains its own daemons settings, and may be operated independently of each other. Some usage examples follow:

local / remote daemons may be set via a client URL and creating a new compute profile by specifying '.compute' as 'remote'. Subsequent mirai calls may then be sent for local computation by not specifying its '.compute' argument, or for remote computation to connected daemons by specifying its '.compute' argument as 'remote'.

cpu / gpu some tasks may require access to different classes of server, such as those with GPUs. In this case, `daemons()` may be called twice to set up client URLs for CPU-only and GPU servers

to dial into, specifying the '.compute' argument as 'cpu' and 'gpu' respectively. By supplying the '.compute' argument to subsequent mirai calls, tasks may be sent to either 'cpu' or 'gpu' servers as appropriate.

Note: further actions such as viewing the status of daemons or resetting via `daemons(0)` should be carried out with the desired '.compute' argument specified.

Status Matrix

When using dispatcher, calling `daemons()` returns a matrix with the following columns:

'online' shows as 1 when there is an active connection, or else 0 if a server has yet to connect or has disconnected.

'instance' increments by 1 every time there is a new connection at a URL. When this happens, the 'assigned' and 'complete' statistics reset to zero. This counter is designed to track new server instances connecting after previous ones have ended (due to time-outs etc.). 'instance' itself resets to zero if the URL is regenerated by `saisei`.

'assigned' shows the cumulative number of tasks assigned to the server instance by the dispatcher.

'complete' shows the cumulative number of tasks completed by the server instance.

The URLs are stored as row names to the matrix.

Timeouts

Specifying the `.timeout` argument in `mirai` will ensure that the 'mirai' always resolves.

However, the task may not have completed and still be ongoing in the daemon process. In such situations, dispatcher ensures that queued tasks are not assigned to the busy process, however overall performance may still be degraded if they remain in use. If a process hangs and cannot be restarted manually, `saisei` specifying `force = TRUE` may be used to regenerate any particular URL for a new `server` to connect to.

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

# Create 2 local daemons (using dispatcher)
daemons(2)
# View status
daemons()
# Reset to zero
daemons(0)

# Create 2 local daemons (not using dispatcher)
daemons(2, dispatcher = FALSE)
# View status
daemons()
# Reset to zero
daemons(0)

# 2 remote daemons via dispatcher (using zero wildcard)
```

```
daemons(2, url = "ws://:0")
# View status
daemons()
# Reset to zero
daemons(0)

# Set client URL for remote servers to dial into (using zero wildcard)
daemons(url = "tcp://:0", dispatcher = FALSE)
# View status
daemons()
# Reset to zero
daemons(0)

}
```

`dispatcher` *mirai Dispatcher*

Description

Implements a dispatcher for tasks from a client to multiple servers for processing, using a FIFO scheduling rule, queuing tasks as required.

Usage

```
dispatcher(
  client,
  url = NULL,
  n = NULL,
  asyncdial = FALSE,
  token = FALSE,
  lock = FALSE,
  ...,
  monitor = NULL
)
```

Arguments

`client`	the client URL to dial as a character string (where tasks are sent from), including the port to connect to and (optionally) a path for websocket URLs e.g. 'tcp://192.168.0.2:5555' or 'ws://192.168.0.2:5555/path'.
`url`	(optional) the URL or range of URLs the dispatcher should listen at as a character vector, including the port to connect to and (optionally) a path for websocket URLs e.g. 'tcp://192.168.0.2:5555' or 'ws://192.168.0.2:5555/path'. Tasks are sent to servers dialled into these URLs. If not supplied, 'n' URLs accessible from the same computer will be assigned automatically.

n	(optional) if specified, the integer number of servers to listen for. Otherwise 'n' will be inferred from the number of URLs supplied as '...'. Where a single URL is supplied and 'n' > 1, 'n' unique URLs will be automatically assigned for servers to dial into.
asyncdial	[default FALSE] whether to perform dials asynchronously. The default FALSE will error if a connection is not immediately possible (e.g. daemons has yet to be called on the client, or the specified port is not open etc.). Specifying TRUE continues retrying (indefinitely) if not immediately successful, which is more resilient but can mask potential connection issues.
token	[default FALSE] if TRUE, appends a unique 40-character token to each URL path the dispatcher listens at (not applicable for TCP URLs which do not accept a path).
lock	[default FALSE] if TRUE, sockets lock once a connection has been accepted, preventing further connection attempts. This provides safety against more than one server trying to connect to a unique URL.
...	additional arguments passed through to server if launching local daemons i.e. 'url' is not specified.
monitor	(for package internal use only) do not set this parameter.

Details

The network topology is such that a dispatcher acts as a gateway between clients and servers, ensuring that tasks received from clients are dispatched on a FIFO basis to servers for processing. Tasks are queued at the dispatcher to ensure tasks are only sent to servers that can begin immediate execution of the task.

Value

Invisible NULL.

---

is_mirai *Is mirai*

Description

Is the object a 'mirai'.

Usage

```
is_mirai(x)
```

Arguments

x	an object.

Value

Logical TRUE if 'x' is of class 'mirai', FALSE otherwise.

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

df <- data.frame()
m <- mirai(as.matrix(df), .args = list(df))
is_mirai(m)
is_mirai(df)

}
```

`is_mirai_error` *Error Validators*

Description

Validator functions for error value types created by {mirai}.

Usage

```
is_mirai_error(x)

is_mirai_interrupt(x)

is_error_value(x)
```

Arguments

x	an object.

Details

Is the object a 'miraiError'. When execution in a mirai process fails, the error message is returned as a character string of class 'miraiError' and 'errorValue'.

Is the object a 'miraiInterrupt'. When a mirai is sent a user interrupt, e.g. by ctrl+c during an ongoing `call_mirai`, the mirai will resolve to an empty character string classed as 'miraiInterrupt' and 'errorValue'.

Is the object an 'errorValue', such as a mirai timeout, a 'miraiError' or a 'miraiInterrupt'. This is a catch-all condition that includes all returned error values, such as timeouts, as well as the error types above.

Value

Logical value TRUE or FALSE.

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

m <- mirai(stop())
call_mirai(m)
is_mirai_error(m$data)
is_mirai_interrupt(m$data)
is_error_value(m$data)

m2 <- mirai(Sys.sleep(1L), .timeout = 100)
call_mirai(m2)
is_mirai_error(m2$data)
is_mirai_interrupt(m2$data)
is_error_value(m2$data)

}
```

`launch_server` *Launch mirai Server*

Description

Utility function which calls `server` in a background `Rscript` process. May be used to re-launch local daemons that have timed out.

Usage

```
launch_server(url, ...)
```

Arguments

`url`	the client URL for the server to dial into as a character string, including the port to connect to and (optionally) a path for websocket URLs e.g. tcp://192.168.0.2:5555' or 'ws://192.168.0.2:5555/path'.
`...`	(optional) additional arguments passed to `server`.

Details

Consider specifying the argument 'asyncdial' [default FALSE] whether to perform dials asynchronously. The default FALSE will error if a connection is not immediately possible (e.g. `daemons` has yet to be called on the client, or the specified port is not open etc.). Specifying TRUE continues retrying (indefinitely) if not immediately successful, which is more resilient but can mask potential connection issues.

Value

Invisibly, integer system exit code (zero upon success).

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

launch_server("abstract://mirai", asyncdial = FALSE, idletime = 60000L)

}
```

`mirai` *mirai (Evaluate Async)*

Description

Evaluate an expression asynchronously in a new background R process or persistent daemon (local or remote). This function will return immediately with a 'mirai', which will resolve to the evaluated result once complete.

Usage

```
mirai(.expr, ..., .args = list(), .timeout = NULL, .compute = "default")
```

Arguments

`.expr`	an expression to evaluate asynchronously (of arbitrary length, wrapped in {} if necessary), or a language object passed by name.
`...`	(optional) named arguments (name = value pairs) specifying objects referenced in '.expr'. Used in addition to and taking precedence over, any arguments specified via '.args'.
`.args`	(optional) either (i) a list of objects to be passed by name, i.e. also found in the current scope with the same name, or else (ii) a list of name = value pairs, as in '...'.
`.timeout`	[default NULL] for no timeout, or an integer value in milliseconds. A mirai will resolve to an 'errorValue' 5 (timed out) if evaluation exceeds this limit.
`.compute`	[default 'default'] character value for the compute profile to use when sending the mirai.

Details

This function will return a 'mirai' object immediately.

The value of a mirai may be accessed at any time at `$data`, and if yet to resolve, an 'unresolved' logical NA will be returned instead.

`unresolved` may be used on a mirai, returning TRUE if a 'mirai' has yet to resolve and FALSE otherwise. This is suitable for use in control flow statements such as `while` or `if`.

Alternatively, to call (and wait for) the result, use `call_mirai` on the returned mirai. This will block until the result is returned (although interruptible with e.g. ctrl+c).

The expression '.expr' will be evaluated in a separate R process in a clean environment, which is not the global environment, consisting only of the named objects passed as '...' and/or the list supplied to '.args'.

If an error occurs in evaluation, the error message is returned as a character string of class 'miraiError' and 'errorValue'. `is_mirai_error` may be used to test for this.

`is_error_value` tests for all error conditions including 'mirai' errors, interrupts, and timeouts.

Specify '.compute' to send the mirai using a specific compute profile (if previously created by `daemons`), otherwise leave as 'default'.

Value

A 'mirai' object.

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

# specifying objects via '...'
n <- 3
m <- mirai(x + y + 2, x = 2, y = n)
m
m$data
Sys.sleep(0.2)
m$data

# passing existing objects by name via '.args'
df1 <- data.frame(a = 1, b = 2)
df2 <- data.frame(a = 3, b = 1)
m <- mirai(as.matrix(rbind(df1, df2)), .args = list(df1, df2), .timeout = 1000)
call_mirai(m)$data

# using unresolved()
m <- mirai(
  {
    res <- rnorm(n)
    res / rev(res)
  },
  n = 1e6
)
```

```
while (unresolved(m)) {
  cat("unresolved\n")
  Sys.sleep(0.1)
}
str(m$data)

# evaluating scripts using source(local = TRUE) in '.expr'
n <- 10L
file <- tempfile()
cat("r <- rnorm(n)", file = file)
m <- mirai({source(file, local = TRUE); r}, .args = list(file, n))
call_mirai(m)[["data"]]
unlink(file)

# specifying global variables using list2env(envir = .GlobalEnv) in '.expr'
n <- 10L
file <- tempfile()
cat("r <- rnorm(n)", file = file)
globals <- list(file = file, n = n)
m <- mirai(
  {
    list2env(globals, envir = .GlobalEnv)
    source(file)
    r
  },
  globals = globals
)
call_mirai(m)[["data"]]
unlink(file)

# passing a language object to '.expr' and a named list to '.args'
expr <- quote(a + b + 2)
args <- list(a = 2, b = 3)
m <- mirai(.expr = expr, .args = args)
call_mirai(m)$data

}
```

`saisei` *Saisei - Regenerate Token*

Description

When using daemons with a local dispatcher service, regenerates the token for the URL a dispatcher socket listens at.

Usage

```
saisei(i = 1L, force = FALSE, .compute = "default")
```

Arguments

i	[default 1L] integer ith URL to replace.
force	[default FALSE] logical value whether to replace the listener even when there is an existing connection.
.compute	[default 'default'] character compute profile to use (each compute profile has its own set of daemons for connecting to different resources).

Details

As the specified listener is closed and replaced immediately, this function will only be successful if there are no existing connections at the socket (i.e. 'online' status shows 0), unless the argument 'force' is specified as TRUE.

Value

The regenerated character URL upon success, or else NULL.

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

daemons(1L)
Sys.sleep(1L)
daemons()
saisei(i = 1L, force = TRUE)
daemons()

daemons(0)

}
```

---

server *mirai Server (Async Executor Daemon)*

Description

Implements a persistent executor/server for the remote process. Awaits data, evaluates an expression in an environment containing the supplied data, and returns the result to the caller/client.

Usage

```
server(
  url,
  asyncdial = FALSE,
  maxtasks = Inf,
```

```
  idletime = Inf,
  walltime = Inf,
  timerstart = 0L,
  exitlinger = 1000L,
  ...,
  cleanup = 7L
)
```

Arguments

url	the client or dispatcher URL to dial into as a character string, including the port to connect to and (optionally) a path for websocket URLs e.g. 'tcp://192.168.0.2:5555' or 'ws://192.168.0.2:5555/path'.
asyncdial	[default FALSE] whether to perform dials asynchronously. The default FALSE will error if a connection is not immediately possible (e.g. daemons has yet to be called on the client, or the specified port is not open etc.). Specifying TRUE continues retrying (indefinitely) if not immediately successful, which is more resilient but can mask potential connection issues.
maxtasks	[default Inf] the maximum number of tasks to execute (task limit) before exiting.
idletime	[default Inf] maximum idle time, since completion of the last task (in milliseconds) before exiting.
walltime	[default Inf] soft walltime, or the minimum amount of real time taken (in milliseconds) before exiting.
timerstart	[default 0L] number of completed tasks after which to start the timer for 'idletime' and 'walltime'. 0L implies timers are started upon launch.
exitlinger	[default 1000L] time in milliseconds to linger before exiting due to a timer / task limit, to allow sockets to complete sends currently in progress. The default can be set wider if computations are expected to return very large objects (> GBs).
...	reserved but not currently used.
cleanup	[default 7L] Integer additive bitmask controlling whether to perform cleanup of the global environment (1L), reset loaded packages to an initial state (2L), reset options to an initial state (4L), and perform garbage collection (8L) after each evaluation. This option should not normally be modified. Do not set unless you are certain you require persistence across evaluations. Note: it may be an error to reset options but not loaded packages if packages set options on load.

Details

The network topology is such that server daemons dial into the client or dispatcher, which listens at the 'url' address. In this way, network resources may be added or removed dynamically and the client or dispatcher automatically distributes tasks to all available servers.

Value

Invisible NULL.

stop_mirai *mirai (Stop Evaluation)*

Description

Stop evaluation of a mirai that is in progress.

Usage

```
stop_mirai(aio)
```

Arguments

aio	a 'mirai' object.

Details

Stops the asynchronous operation associated with the mirai by aborting, and then waits for it to complete or to be completely aborted. The mirai is then deallocated and attempting to access the value at `$data` will result in an error.

Value

Invisible NULL.

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

s <- mirai(Sys.sleep(n), n = 5)
stop_mirai(s)

}
```

unresolved *Query if a mirai is Unresolved*

Description

Query whether a mirai or mirai value remains unresolved. Unlike `call_mirai`, this function does not wait for completion.

Usage

```
unresolved(aio)
```

Arguments

aio	a 'mirai' object or 'mirai' value stored at $data.

Details

Suitable for use in control flow statements such as while or if.

Note: querying resolution may cause a previously unresolved 'mirai' to resolve.

Value

Logical TRUE if 'aio' is an unresolved mirai or mirai value, or FALSE otherwise.

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

m <- mirai(Sys.sleep(0.1))
unresolved(m)
Sys.sleep(0.3)
unresolved(m)

}
```

%>>% *Deferred Evaluation Pipe*

Description

Pipe a possibly unresolved value forward into a function. The piped expression should be wrapped in .().

Usage

```
x %>>% f

.(expr)
```

Arguments

x	a 'mirai' or mirai value at $data that is possibly an 'unresolvedValue'.
f	a function that accepts 'x' as its first argument.
expr	a piped expression.

Details

An 'unresolvedExpr' encapsulates the eventual evaluation result. Query its `$data` element for resolution. Once resolved, the object changes into a 'resolvedExpr' and the evaluated result will be available at `$data`.

Supports stringing together a series of piped expressions (as per the below example).

Wrap a piped expression in `.()` to ensure that the return value is always an 'unresolvedExpr' or 'resolvedExpr' as the case may be, otherwise if 'x' is already resolved, the evaluated result would be returned directly.

`unresolved` may be used on an expression or its `$data` element to test for resolution.

Value

The evaluated result, or if the mirai value of x is an 'unresolvedValue', an 'unresolvedExpr'.

It is advisable to wrap `resolve()` around a piped expression to ensure stability of return types, as this is guaranteed to return either an 'unresolvedExpr' or 'resolvedExpr'.

Usage

Usage is similar to R's native `|>` pipe.

`x %>>% f` is equivalent to `f(x)`

`x %>>% f()` is equivalent to `f(x)`

`x %>>% f(y)` is equivalent to `f(x, y)`

Please note that other usage is not supported and it is not a drop-in replacement for magrittr's `%>%` pipe.

Examples

```
if (interactive()) {
# Only run examples in interactive R sessions

m <- mirai({Sys.sleep(0.5); 1})
b <- .(m %>>% c(2, 3) %>>% as.character)
unresolved(b)
b
b$data

call_mirai(m)
unresolved(b)
b
b$data

}
```

Index