Package 'tryCatchLog'

March 26, 2019

Title Advanced 'tryCatch()' and 'try()' Functions

Version 1.1.4

Description Advanced tryCatch() and try() functions for better error handling
(logging, stack trace with source code references and support for post-
mortem analysis via dump files).

Imports utils

Depends R (>= 3.1.0)

License GPL-3 | file LICENSE

URL https://github.com/aryoda/tryCatchLog

BugReports https://github.com/aryoda/tryCatchLog/issues

Encoding UTF-8

LazyData true

RoxygenNote 6.1.0

Suggests futile.logger, testthat, knitr, rmarkdown

VignetteBuilder knitr

NeedsCompilation no

Author Juergen Altfeld [aut, cre, cph],
Charles Epaillard [ctb]

Maintainer Juergen Altfeld <jaltfeld@altfeld-im.de>

Repository CRAN

Date/Publication 2019-03-25 23:10:03 UTC

R topics documented:

---

append.to.last.tryCatchLog.result

> *Appends a new log entry to the stored logging output of the last call to*
> tryCatchLog *or* tryLog

Description

You can get the last logging output by calling [last.tryCatchLog.result](#last.tryCatchLog.result).

Usage

```
append.to.last.tryCatchLog.result(new.log.entry)
```

Arguments

new.log.entry   the new log entry (a data.frame created with link{build.log.entry})

Details

THIS FUNCTION IS USED ONLY PACKAGE INTERNALLY!

Value

the complete logging result of the last call to tryCatchLog or tryLog as data.frame

Note

THIS IS A PACKAGE INTERNAL FUNCTION AND THEREFORE NOT EXPORTED.

See Also

[last.tryCatchLog.result](#last.tryCatchLog.result), [reset.last.tryCatchLog.result](#reset.last.tryCatchLog.result),

build.log.entry	*Creates a log entry row containing all relevant logging information in columns*

Description

The serverity level should correspond to the condition class.

Usage

```
build.log.entry(timestamp, severity, msg.text, call.stack, dump.file.name,
  omit.call.stack.items = 0)
```

Arguments

timestamp	logging timestamp as POSIXct (normally by calling Sys.time)
severity	severity level of the log entry ((ERROR, WARN, INFO etc.)
msg.text	Logging message (e. g. error message)
call.stack	a call stack created by sys.calls
dump.file.name	name of the created dump file (leave empty if the tryCatchLog argument write.error.dump.file is FALSE
omit.call.stack.items	
	the number of stack trace items to ignore (= last x calls) in the passed call.stack since they are caused by using tryCatchLog

Value

An object of class tryCatchLog.log.entry and data.frame with one and the following columns:

1. timestamp - creation date and time of the logging entry
2. severity - the serverity level of the log entry (ERROR, WARN, INFO etc.)
3. msg.text - the message text of the log entry
4. compact.stack.trace - the short stack trace containing only entries with source code references down to line of code that has thrown the condition
5. full.stack.trace - the full stack trace with all calls down to the line of code that has thrown the condition (including calls to R internal functions and other functions even when the source code in not available).
6. dump.file.name - name of the created dump file (if any)

Note

THIS IS A PACKAGE INTERNAL FUNCTION AND THEREFORE NOT EXPORTED.

See Also

last.tryCatchLog.result build.log.output

---

build.log.output   *Creates a single string suited as logging output*

Description

To view the formatted output print the logging output in a console use [cat](#) (instead of printing the output with [print](#) which shows the newline escape codes).

Usage

```
build.log.output(log.results, include.full.call.stack = TRUE,
  include.severity = TRUE, include.timestamp = FALSE,
  use.platform.newline = FALSE)
```

Arguments

log.results   A data.frame and member of the class tryCatchLog.log.entry with log en-
                     try rows as returned by [last.tryCatchLog.result](#) containing the logging in-
                     formation to be prepared for the logging output.

include.full.call.stack

                     Flag of type [logical](#): Shall the full call stack be included in the log output?
                     Since the full call stack may be very long it can be omitted by passing FALSE.

include.severity

                     logical switch if the severity level (e. g. ERROR) shall be included in the
                     output

include.timestamp

                     logical switch if the timestamp of the catched condition shall be included in
                     the output

use.platform.newline

                     logical: If TRUE the line breaks ("newline") will be inserted according to the
                     current operationg system (Windows: CR+LF, else: CR). If FALSE R's usual \n
                     esacpe character will be inserted and it is left to the client to convert this later
                     into the operation-system-specific characters. This argument is rarely required
                     (except e. g. if you want to write the return value into a database table column).

Value

A ready to use logging output with stack trace (as character)

Note

Supports also a single row created by the package internal function [build.log.entry](#) as log.results
argument.

See Also

[last.tryCatchLog.result](#) [build.log.entry](#)

---

determine.platform.NewLine

Determines the operating system specific new line character(s)

Description

CR + LF on Windows, else only LF...

Usage

```
determine.platform.NewLine()
```

Details

This function is pendant to Microsoft's .Net "Environment.NewLine".

Value

the new line character(s) for the current operating system

Note

THIS IS A PACKAGE INTERNAL FUNCTION AND THEREFORE NOT EXPORTED.

References

https://stackoverflow.com/questions/47478498/build-string-with-os-specific-newline-characters-crlf-

---

get.pretty.call.stack *Pretty formatted call stack enriched with the source file names and row numbers*

Description

Enriches the current call stack with the source file names and row numbers to track the location of thrown conditions and generates a prettily formatted list of strings

Usage

```
get.pretty.call.stack(call.stack, omit.last.items = 0, compact = FALSE)
```

Arguments

call.stack   Call stack object created by sys.calls

omit.last.items

          Number of call stack items to drop from the end of the full stack trace

compact   TRUE will return only call stack items that have a source code reference (FALSE all)

Details

How to read the call stack:

1. Call stack items consist of:
   `<call stack item number> [<file name>#<row number>:] <expression executed by this code line>`
2. The last call stack items with a file name and row number points to the source code line causing the error.
3. Ignore all call stack items that do not start with a file name and row number (R internal calls only)

You should only call this function from within [`withCallingHandlers`](withCallingHandlers), NOT from within [`tryCatch`](tryCatch) since tryCatch unwinds the call stack to the tryCatch position and the source of the condition cannot be identified anymore.

Value

The call stack ([`sys.calls`](sys.calls)) without the last number of function calls (given by "omit.last.items") to remove irrelevant calls caused e. g. by exception handler ([`withCallingHandlers`](withCallingHandlers)) or restarts (of warnings).

See Also

[`tryCatchLog`](tryCatchLog), [`tryLog`](tryLog), [`limitedLabelsCompact`](limitedLabelsCompact)

---

get.pretty.option.value

gets the current value of an option as key/value string

Description

The data type is also indicated if an option is set (since a wrong data type may cause problems). If an option is not set "(not set)" is shown as value.

Usage

```
get.pretty.option.value(option.name)
```

Arguments

option.name   Name of the option (as character)

Details

THIS IS AN INTERNAL PRIVATE FUNCTION OF THE PACKAGE.

Value

The option as key/value string in one line

See Also

get.pretty.tryCatchLog.options

Examples

```
## Not run:
tryCatchLog:::get.pretty.option.value("warn")
# [1] "Option warn = 0 (double)"
## End(Not run)
```

---

get.pretty.tryCatchLog.options

Gets the current option values of all options supported by the 'tryCatchLog' package

---

Description

This is a convenience function whose result can be used e. g. to log the current settings.

Usage

```
get.pretty.tryCatchLog.options()
```

Details

If an option is not set the string "(not set)" is shown as value.

The data type is also indicated if an option is set (since a wrong data type may cause problems).

Value

The current option settings as string (one per line as key/value pair), e. g.

```
Option tryCatchLog.write.error.dump.file = FALSE (logical)
Option tryCatchLog.write.error.folder = . (character)
Option tryCatchLog.silent.warnings = FALSE (logical)
Option tryCatchLog.silent.messages = (not set)
```

Examples

```
cat(get.pretty.tryCatchLog.options())  # "cat" does apply new line escape characters
```

---

is.duplicated.log.entry

Check if a new log entry would be a duplicate of on an already existing log entry

---

Description

The `log.entry` is checked against the existing log entries from `last.tryCatchLog.result` using the following columns:

1. msg.text
2. full.stack.trace

Usage

```
is.duplicated.log.entry(log.entry)
```

Arguments

log.entry   A `data.frame` with the new log entry (exactly one row)

Value

TRUE if the `log.entry` is a duplicate, else FALSE

Note

Required function to fix issue # 18 (<https://github.com/aryoda/tryCatchLog/issues/18>)

See Also

`last.tryCatchLog.result`, `build.log.entry`

---

is.package.available   *Checks if a package is installed and can be loaded*

Description

Use this function to check for optional package dependencies within this package.

Usage

```
is.package.available(package.name)
```

Arguments

package.name   Name of the package (as string)

Details

This is a package-internal function!

See section 'Good practice' in '?.onAttach'.

Value

TRUE if the packages is installed, otherwise FALSE

http://r-pkgs.had.co.nz/description.html

Examples

```
tryCatchLog:::is.package.available("tryCatchLog")  # must be TRUE :-)
```

---

is.windows	*Determines if R is running on a Windows operating system*

Description

Throws a warning if an indication for Windows OS were found but the Windows OS cannot be recognized for sure (via a second different check).

Usage

```
is.windows()
```

Value

TRUE of running on a Windows OS else FALSE

Examples

```
is.windows()
```

`last.tryCatchLog.result`

Gets the logging result of the last call to `tryCatchLog` *or* `tryLog`

Description

The last logging result after calling `tryCatchLog` or `tryLog` can be retrieved by calling this function.

Usage

```
last.tryCatchLog.result()
```

Details

The typical use case is to get and store the log output not only in a log file but also in another place that is not supported by the logging framework, e. g. in a data base table of your application or displaying it in a GUI (user interface).

Another use case is to review the last log output on the console during debugging.

Value

the logging result of the last call to tryCatchLog or tryLog as data.frame comprised of one row per logged condition with these columns:

1. timestamp - creation date and time of the logging entry
2. severity - the serverity level of the log entry (ERROR, WARN, INFO etc.)
3. msg.text - the message text of the log entry
4. compact.stack.trace - the short stack trace containing only entries with source code references down to line of code that has thrown the condition
5. full.stack.trace - the full stack trace with all calls down to the line of code that has thrown the condition (including calls to R internal functions and other functions even when the source code in not available).
6. dump.file.name - name of the created dump file (if any)

If no condition is logged at all an empty data.table is returned.

See Also

tryCatchLog, tryLog

Examples

```
last.tryCatchLog.result()
```

limitedLabelsCompact   *Convert a call stack into a list of printable strings*

Description

Converts a call stack into a list of printable strings ("labels") with a limited length per call. If source code references are available they are also added in the form \<file name\>#\<line number\>: executed R expression (call)

Usage

```
limitedLabelsCompact(value, compact = FALSE,
  maxwidth = getOption("width") - 5L)
```

Arguments

value	a list of calls ("call.stack") generated by `sys.calls`
compact	if TRUE only calls that contain a source code reference (attribute "srcref") are returned (plus always the first call); if FALSE all calls will be returned.
maxwidth	Maximum number of characters per call in the return value (longer strings will be cutted). Must be between 40 and 1000

Details

R does track source code references only if you set the option "keep.source" to TRUE via `options(keep.source = TRUE)`. Without this option this function cannot enrich source code references. If you use Rscript to start a non-interactive R script as batch job you have to set this option since it is FALSE by default. You can add this option to your .Rprofile file or use a startup R script that sets this option and sources your actual R script then.

This function is based on the undocumented `limitedLabels` function of the base package. The source code can be viewed by entering `limitedLabels` in the R console. The attributes required to add source file names and line numbers to the calls (srcref and srcfile) and how they are created internally are explained in this article: [https://journal.r-project.org/archive/2010-2/RJournal_](https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Murdoch.pdf) [2010-2_Murdoch.pdf](https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Murdoch.pdf)

Value

A list of strings (one for each call)

See Also

`sys.calls`, `tryCatchLog`, `get.pretty.call.stack`

Examples

```
limitedLabelsCompact(sys.calls(), TRUE)
```

---

log2console   *Prints a time-stamped log message to the console incl. the severity*
                                level

Description

This is a package-internal function.

Usage

```
log2console(severity.level, msg)
```

Arguments

severity.level   String containing the severity level (ERROR, WARN or INFO) of the message

msg   The message to be printed (as character).

Value

The log message as it was printed to the console. NA is printed as empty string.

Examples

```
tryCatchLog:::log2console("WARN", "this is my last warning")
```

---

platform.NewLine   *Gets the operating system specific new line character(s)*

Description

CR + LF on Windows, else only LF...

Usage

```
platform.NewLine()
```

Details

The newline character(s) are determined once at package loading time.

Value

the new line character(s) for the current operating system

Examples

```
platform.NewLine()
```

reset.last.tryCatchLog.result

> *Resets the stored logging output of the last call to* tryCatchLog *or* tryLog *to an empty list*

Description

You can get the last logging output by calling last.tryCatchLog.result.

Usage

```
reset.last.tryCatchLog.result()
```

Value

invisible: TRUE

Note

THIS IS A PACKAGE INTERNAL FUNCTION AND THEREFORE NOT EXPORTED.

See Also

last.tryCatchLog.result, append.to.last.tryCatchLog.result,

set.logging.functions *Sets the logging functions that shall be used by* tryCatchLog *for the different severity levels*

Description

The logging functions must have at least one parameter: The logging message (as character) which must be the first argument.

Usage

```
set.logging.functions(error.log.func = function(msg)
  tryCatchLog:::log2console("ERROR", msg), warn.log.func = function(msg)
  tryCatchLog:::log2console("WARN", msg), info.log.func = function(msg)
  tryCatchLog:::log2console("INFO", msg))
```

Arguments

error.log.func	The logging function for errors
warn.log.func	The logging function for warning
info.log.func	The error function for messages

Details

The default logging functions are internal functions without any dependencies to other logging packages. They use the same logging output format as **futile.logger** version 1.4.3.

If you want to disable any logging output you should use a decent logging framework which allows to set the logging threshold (e. g. futile.logger's `flog.threshold`).

The package-internal default logging functions are only a minimal implementation and are not meant to replace a decent logging framework.

Value

Nothing

See Also

[tryCatchLog](#)

Examples

```
# To disable any logging you could use "empty" functions
set.logging.functions( error.log.func = function(msg) invisible(),
                       warn.log.func  = function(msg) invisible(),
                       info.log.func  = function(msg) invisible())
```

---

tryCatchLog   *Try an expression with condition logging and error handling*

Description

This function evaluates an expression passed in the `expr` parameter and executes the error handler function passed as parameter `error` in case of an error condition.

Usage

```
tryCatchLog(expr, ..., finally = NULL,
  write.error.dump.file = getOption("tryCatchLog.write.error.dump.file",
  FALSE),
  write.error.dump.folder = getOption("tryCatchLog.write.error.dump.folder",
  "."), silent.warnings = getOption("tryCatchLog.silent.warnings",
  FALSE), silent.messages = getOption("tryCatchLog.silent.messages",
  FALSE))
```

Arguments

`expr`	expression to be evaluated
`...`	condition handler functions (as in [tryCatch](#)). Usual condition names are `error`, `warning`, `message` and `interrupt`. All condition handlers are passed to [tryCatch](#) as is (no filtering, wrapping or changing of semantics).
`finally`	expression to be evaluated at the end
`write.error.dump.file`	TRUE: Saves a dump of the workspace and the call stack named dump_<YYYYMMDD>_at_<HHMMSS.sss>_PII This dump file name pattern shall ensure unique file names in parallel processing scenarios.
`write.error.dump.folder`	path: Saves the dump of the workspace in a specific folder instead of the working directory
`silent.warnings`	TRUE: Warnings are logged, but not propagated to the caller. FALSE: Warnings are logged and treated according to the global setting in [getOption](#)("warn"). See also [warning](#).
`silent.messages`	TRUE: Messages are logged, but not propagated to the caller. FALSE: Messages are logged and propagated to the caller.

Details

The `finally` expression is then always evaluated at the end.

Conditions are logged with the function call stack (including file names and line numbers).

This function shall overcome some drawbacks of the standard [tryCatch](#) function.
For more details see <https://github.com/aryoda/tryCatchLog>.

If the package **futile.logger** is installed it will be used for writing logging output, otherwise an internal basic logging output function is used.

Before you call tryCatchLog for the first time you should initialize the logging framework you are using (e. g.**futile.logger** to control the log output (log to console or file etc.):

```
library(futile.logger)
flog.appender(appender.file("my_app.log"))
flog.threshold(INFO)    # TRACE, DEBUG, INFO, WARN, ERROR, FATAL
```

If you are using the **futile.logger** package tryCatchLog calls these log functions for the different R conditions to log them:

1. error -> [flog.error](#)

2. warning -> [flog.warn](#)

3. message -> [flog.info](#)

'tryCatchLog' does only log the above conditions, other (user-defined) conditions are currently not not logged but can be catched of course by passing additional handler functions via the `...` **argument.**

The log contains the call stack with the file names and line numbers (if available).

R does track source code references of scripts only if you set the option keep.source to TRUE via options(keep.source = TRUE). Without this option this function cannot enrich source code references.

If you use Rscript to start a non-interactive R script as batch job you have to set this option since it is FALSE by default. You can add this option to your .Rprofile file or use a startup R script that sets this option and sources your actual R script then.

By default, most packages are built without source reference information. Setting the environment variable R_KEEP_PKG_SOURCE=yes before installing a source package will tell R to keep the source references. You can also use options(keep.source.pkgs = TRUE) before you install a package.

Setting the parameter tryCatchLog.write.error.dump.file to TRUE allows a post-mortem analysis of the program state that led to the error. The dump contains the workspace and in the variable "last.dump" the call stack (`sys.frames`). This feature is very helpful for non-interactive R scripts ("batches").

Setting the parameter tryCatchLog.write.error.dump.folder to a specific path allows to save the dump in a specific folder. If not set, the dump will be saved in the working directory.

To start a post-mortem analysis after an error open a new R session and enter: load("dump_20161016_164050.rda")   # re debugger(last.dump)

Note that the dump does **not** contain the loaded packages when the dump file was created and a dump loaded into memory does therefore **not** use exactly the same search path. This means:

1. the program state is not exactly reproducible if objects are stored within a package namespace
2. you cannot step through your source code in a reproducible way after loading the image if your source code calls functions of non-default packages

Value

the value of the expression passed in as parameter "expr"

Best practices

To **avoid that too many dump files filling your disk space** you should omit the write.error.dump.file parameter and instead set its default value using the option tryCatchLog.write.error.dump.file in your .Rprofile file instead (or in a startup R script that sources your actual script). In case of an error (that you can reproduce) you set the option to TRUE and re-run your script. Then you are able to examine the program state that led to the error by debugging the saved dump file.

To see the **source code references (source file names and line numbers)** in the stack traces you must set this option before executing your code:
options(keep.source = TRUE)

You can **execute your code as batch with** Rscript **using this shell script command**:
Rscript -e "options(keep.source = TRUE); source('my_main_function.R')"

References

http://adv-r.had.co.nz/beyond-exception-handling.html
https://stackoverflow.com/questions/39964040/r-catch-errors-and-continue-execution-after-logging-tl

See Also

tryLog, limitedLabels, get.pretty.call.stack, getOption, last.tryCatchLog.result, set.logging.functions

Examples

```
tryCatchLog(log(-1))   # logs a warning
```

---

tryLog   *Try an expression with condition logging and error recovery*

Description

tryLog is a wrapper function around tryCatchLog that traps any errors that occur during the evalu-
ation of an expression without stopping the execution of the script (similar to try). Errors, warnings
and messages are logged. In contrast to tryCatchLog it returns but does not stop in case of an error
and therefore does not have the error and finally parameters to pass in custom handler functions.

Usage

```
tryLog(expr,
  write.error.dump.file = getOption("tryCatchLog.write.error.dump.file",
  FALSE),
  write.error.dump.folder = getOption("tryCatchLog.write.error.dump.folder",
  "."), silent.warnings = getOption("tryCatchLog.silent.warnings",
  FALSE), silent.messages = getOption("tryCatchLog.silent.messages",
  FALSE))
```

Arguments

expr   expression to be evaluated

write.error.dump.file
                  TRUE: Saves a dump of the workspace and the call stack named dump_<YYYYMMDD>_at_<HHMMSS.sss>_PII
                  This dump file name pattern shall ensure unique file names in parallel processing
                  scenarios.

write.error.dump.folder
                  path: Saves the dump of the workspace in a specific folder instead of the work-
                  ing directory

silent.warnings
                  TRUE: Warnings are logged, but not propagated to the caller.
                  FALSE: Warnings are logged and treated according to the global setting in getOption("warn").
                  See also warning.

silent.messages

        TRUE: Messages are logged, but not propagated to the caller.
        FALSE: Messages are logged and propagated to the caller.

Details

tryLog is implemented using [tryCatchLog](). If you need need more flexibility for catching and handling errors use the latter. Error messages are never printed to the [stderr]() connection but logged only.

Value

The value of the expression (if expr is evaluated without an error.
In case of an error: An invisible object of the class "try-error" containing the error message and error condition as the "condition" attribute.

See Also

[tryCatchLog](), [last.tryCatchLog.result]()

Examples

```
tryLog(log(-1))   # logs a warning
tryLog(log("a"))  # logs an error
```

Index