Package: futile.logger (via r-universe)

May 29, 2026

Type Package

Title A Logging Utility for R

Version 1.4.8

Date 2025-12-22

Maintainer Brian Lee Yung Rowe <r@zatonovo.com>

Depends R (>= 3.0.0)

Imports utils, lambda.r (>= 1.1.0), futile.options

Suggests testit, jsonlite, httr, crayon, rsyslog, glue

Description Provides a simple yet powerful logging utility. Based loosely on log4j, futile.logger takes advantage of R idioms to make logging a convenient and easy to use replacement for cat and print statements.

License LGPL-3

LazyLoad yes

NeedsCompilation no

ByteCompile yes

Collate 'options.R' 'appender.R' 'constants.R' 'layout.R' 'logger.R' 'scat.R' 'util.R' 'futile.logger-package.R'

RoxygenNote 7.1.2

URL https://github.com/zatonovo/futile.logger

Repository https://zatonovo.r-universe.dev

Date/Publication 2025-12-22 15:58:38 UTC

RemoteUrl https://github.com/zatonovo/futile.logger

RemoteRef HEAD

RemoteSha 9b8e53dca470ce3408b418ce835a10fe394a62dc

Contents

futile.logger-package . . . 2
flog.appender . . . 4
flog.carp . . . 6
flog.layout . . . 7
flog.logger . . . 9
flog.remove . . . 12
flog.threshold . . . 12
ftry . . . 13
logger.options . . . 14
prepare_arg . . . 15
scat . . . 15

Index **16**

`futile.logger-package` *A Logging Utility for R*

Description

This package implements a logging system inspired by log4j. The basic idea of layouts, appenders, and loggers is faithful to log4j, while the implementation and idiom is all R. This means that support for hierarchical loggers, custom appenders, custom layouts is coupled with a simple and intuitive functional syntax.

Details

Package:	futile.logger
Type:	Package
Version:	1.4.8
Date:	2025-12-22
License:	LGPL-3
LazyLoad:	yes

The latest version of futile.logger introduces zero-configuration semantics out of the box. This means that you can use the default configuration as is. It is also easy to interactively change the configuration of the ROOT logger, as well as create new loggers. Since loggers form a hierarchy based on their name, the ROOT logger is the starting point of the hierarchy and always exists. By default the ROOT logger is defined with a simple layout, printing to the console, with an INFO threshold. This means that writing to any logger with a threshold of INFO or higher will write to the console.

All of the logging functions take a format string so it is easy to add arbitrary values to log messages.

> flog.info("This song is just %s words %s", 7, "long")

Thresholds range from most verbose to least verbose: TRACE, DEBUG, INFO, WARN, ERROR, FATAL. You can easily change the threshold of the ROOT logger by calling > flog.threshold(TRACE) which changes will print all log messages from every package. To suppress most logging by default but turn on all debugging for a logger 'my.logger', you would execute

> flog.threshold(ERROR)
> flog.threshold(TRACE, name='my.logger')

Any arbitrary logger can be defined simply by specifying it in any futile.logger write operation (futile.threshold, futile.appender, futile.layout). If the logger hasn't been defined, then it will be defined dynamically. Any unspecified options will be copied from the parent logger.

When writing log messages, futile.logger will search the hierarchy based on the logger name. In our example, if 'my.logger' hasn't been defined then futile.logger will look for a logger named 'my' and finally the ROOT logger.

Functions calling futile.logger from a package are automatically assigned a logger that has the name of the package. Suppose we have log messages in a package called 'my.package'. Then any function that calls futile.logger from within the package will automatically be assigned a default logger of 'my.package' instead of ROOT. This means that it is easy to change the log setting of any package that uses futile.logger for logging by just updating the logger for the given package. For instance suppose you want to output log message for my.package to a file instead.

> flog.appender(appender.file('my.package.log'), name='my.package')

Now all log statements in the package my.package will be written to a file instead of the console. All other log messages will continue to be written to the console.

Appenders do the actual work of writing log messages to a writeable target, whether that is a console, a file, a URL, database, etc. When creating an appender, the implementation-specific options are passed to the appender at instantiation. The package defines two appender generator functions:

appender.file Write to a file

appender.console Write to the console

Each of these functions returns the actual appender function, so be sure to actually call the function!

Layouts are responsible for formatting messages. This operation usually consists of adding the log level, a timestamp, plus some pretty-printing to make the log messages easy on the eyes. The package supplies several layouts:

layout.simple Writes messages with a default format

layout.simple.parallel Writes messages with a default format with PID

layout.json Generates messages in a JSON format

layout.format Define your own format

layout.tracearg Print a variable name along with its value

Author(s)

Brian Lee Yung Rowe <r@zatonovo.com>

See Also

`flog.logger`, `flog.threshold`, `flog.layout`, `flog.appender`

Examples

```
flog.debug("This %s print", "won't")
flog.warn("This %s print", "will")

flog.info("This inherits from the ROOT logger", name='logger.a')
flog.threshold(DEBUG, name='logger.a')
flog.debug("logger.a has now been set to DEBUG", name='logger.a')
flog.debug("But the ROOT logger is still at INFO (so this won't print)")

## Not run:
flog.appender(appender.file("other.log"), name='logger.b')
flog.info("This writes to a %s", "file", name='logger.b')

## End(Not run)
```

`flog.appender` *Manage appenders for loggers*

Description

Provides functions for adding and removing appenders.

Arguments

...	Used internally by lambda.r

Usage

```
# Get the appender for the given logger
flog.appender(name) %::% character : Function
flog.appender(name='ROOT')

# Set the appender for the given logger
flog.appender(fn, name='ROOT')

# Print log messages to the console
appender.console()

# Write log messages to a file
appender.file(file)

# Write log messages to a dynamically-named file
appender.file2(format)

# Write log messages to console and a file
appender.tee(file)

# Write log messages to a Graylog2 HTTP GELF endpoint
appender.graylog(server, port)

# Write log message to syslog. Arguments are passed on to open_syslog.
appender.syslog(identifier, ...)
```

```
# Special meta appender that prints only when the internal counter mod n = 0
appender.modulo(n, appender=appender.console())
```

Details

Appenders do the actual work of writing log messages to some target. To use an appender in a logger, you must register it to a given logger. Use `flog.appender` to both access and set appenders.

The ROOT logger by default uses `appender.console`.

`appender.console` is a function that writes to the console. No additional arguments are necessary when registering the appender via flog.appender.

`appender.file` writes to a file, so you must pass an additional file argument to the function. To change the file name, just call `flog.appender(appender.file(file))` again with a new file name.

`appender.file2` is similar, but the filename is dynamically determined at runtime. It may include most of the same tokens as `layout.format` (all except `"~m"`, the message itself). This allows, for instance, having separate logfiles for each log level.

To use your own appender create a function that takes a single argument, which represents the log message. You need to pass a function reference to `flog.appender`.

`appender.tee` writes to both the console and file.

`appender.graylog` writes to a Graylog2 HTTP GELF endpoint.

`appender.syslog` writes to the POSIX system logger.

`appender.modulo` is a meta appender. It calls `appender` every `n` times.

Value

When getting the appender, `flog.appender` returns the appender function. When setting an appender, `flog.appender` has no return value.

Author(s)

Brian Lee Yung Rowe

See Also

`flog.logger` `flog.layout`

Examples

```
## Not run:
flog.appender(appender.console(), name='my.logger')

# Set an appender to the logger named 'my.package'. Any log operations from
# this package will now use this appender.
flog.appender(appender.file('my.package.out'), 'my.package')

# Set an appender to a file named using the message level and calling function.
# Also tee the messages to the console.
```

```
flog.appender(appender.file2('~l-~f.log', console = TRUE))

## End(Not run)
```

`flog.carp` *Always return the log message*

Description

Indicate whether the logger will always return the log message despite the threshold.

Arguments

`carp`	logical Whether to carp output or not
`name`	character The name of the logger

Details

This is a special option to allow the return value of the flog.* logging functions to return the generated log message even if the log level does not exceed the threshold. Note that this minorly impacts performance when enabled. This functionality is separate from the appender, which is still bound to the value of the logger threshold.

Usage

# Indicate whether the given logger should carp
flog.carp(name=ROOT)

# Set whether the given logger should carp
flog.carp(carp, name=ROOT)

Author(s)

Brian Lee Yung Rowe

Examples

```
flog.carp(TRUE)
x <- flog.debug("Returns this message but won't print")
flog.carp(FALSE)
y <- flog.debug("Returns nothing and prints nothing")
```

`flog.layout` *Manage layouts within the 'futile.logger' sub-system*

Description

Provides functions for managing layouts. Typically 'flog.layout' is only used when manually creating a logging configuration.

Arguments

...	Used internally by lambda.r

Usage

```
# Get the layout function for the given logger
flog.layout(name) %::% character : Function
flog.layout(name='ROOT')

# Set the layout function for the given logger
flog.layout(fn, name='ROOT')

# Decorate log messages with a standard format
layout.simple(level, msg, ...)

# Decorate log messages with a standard format colored by log level
layout.colored(level, msg, ...)

# Decorate log messages with a standard format using glue instead of sprintf
layout.glue(level, msg, ...)

# Decorate log messages with a standard format and a pid
layout.simple.parallel(level, msg, ...)

# Generate log messages as JSON
layout.json(level, msg, ...)

# Decorate log messages using a custom format
layout.format(format, datetime.fmt="

# Show the value of a single variable layout.tracearg(level, msg, ...)

# Generate log messages in a Graylog2 HTTP GELF accetable format layout.graylog(common.fields)
```

Details

Layouts are responsible for formatting messages so they are human-readable. Similar to an appender, a layout is assigned to a logger by calling `flog.layout`. The `flog.layout` function is used internally to get the registered layout function. It is kept visible so user-level introspection is possible.

`layout.simple` is a pre-defined layout function that prints messages in the following format: LEVEL [timestamp] message

This is the default layout for the ROOT logger.

`layout.format` allows you to specify the format string to use in printing a message. The following tokens are available.

~l Log level

~t Timestamp

~n Namespace

~f The calling function

~m The message

~p The process PID

~i Logger name

`layout.json` converts the message and any additional objects provided to a JSON structure. E.g.:

flog.info("Hello, world", cat='asdf')

yields something like

{"level":"INFO","timestamp":"2015-03-06 19:16:02 EST","message":"Hello, world","func":"(shell)","cat":["asdf"]}

`layout.tracearg` is a special layout that takes a variable and prints its name and contents.

`layout.graylog` is a special layout for use with the appender.graylog to generate json acceptable to a Graylog2 HTTP GELF endpoint. Standard fields to be included with every message can be included by setting the common.fields to a list of properties. E.g.:

flog.layout(layout.graylog(common.fields = list(host_ip = "10.10.11.23", env = "production")))

Author(s)

Brian Lee Yung Rowe

See Also

`flog.logger` `flog.appender`

Examples

```
# Set the layout for 'my.package'
flog.layout(layout.simple, name='my.package')

# Update the ROOT logger to use a custom layout
layout <- layout.format('[~l] [~t] [~n.~f] ~m')
flog.layout(layout)

# Create a custom logger to trace variables
flog.layout(layout.tracearg, name='tracer')
x <- 5
flog.info(x, name='tracer')
```

`flog.logger` *Manage loggers*

Description

Provides functions for writing log messages and managing loggers. Typically only the flog.[trace|debug|info|warn|error|fatal] functions need to be used in conjunction with flog.threshold to interactively change the log level.

Arguments

`msg`	The message to log
`name`	The logger name to use
`capture`	Capture print output of variables instead of interpolate
`logger`	The logger to use. If `NULL` (the default), it is looked up based on `name`. Provide `logger` explicitely if the speed of the evaluation of log level is of concern (e.g., a `flog.trace` call in your function which has to be run many times).
`...`	Optional arguments to populate the format string
`expr`	An expression to evaluate
`finally`	An optional expression to evaluate at the end

Usage

# Conditionally print a log statement at TRACE log level
flog.trace(msg, ..., name=flog.namespace(), logger=NULL, capture=FALSE)

# Conditionally print a log statement at DEBUG log level
flog.debug(msg, ..., name=flog.namespace(), logger=NULL, capture=FALSE)

# Conditionally print a log statement at INFO log level
flog.info(msg, ..., name=flog.namespace(), logger=NULL, capture=FALSE)

# Conditionally print a log statement at WARN log level
flog.warn(msg, ..., name=flog.namespace(), logger=NULL, capture=FALSE)

# Conditionally print a log statement at ERROR log level
flog.error(msg, ..., name=flog.namespace(), logger=NULL, capture=FALSE)

# Print a log statement at FATAL log level
flog.fatal(msg, ..., name=flog.namespace(), logger=NULL, capture=FALSE)

Execute an expression and capture any warnings or errors ftry(expr, error=stop, silent=FALSE, finally=NULL, details='')

Additional Usage

These functions generally do not need to be called by an end user.

# Get the ROOT logger
flog.logger()

```
# Get the logger with the specified name
flog.logger(name)

# Set options for the given logger
flog.logger(name, threshold=NULL, appender=NULL, layout=NULL, carp=NULL)
```

Details

These functions represent the high level interface to futile.logger.

The primary use case for futile.logger is to write out log messages. There are log writers associated with all the predefined log levels: TRACE, DEBUG, INFO, WARN, ERROR, FATAL. Log messages will only be written if the log level is equal to or more urgent than the current threshold. By default the ROOT logger is set to INFO.

> flog.debug("This won't print")
> flog.info("But this %s", 'will')
> flog.warn("As will %s", 'this')

Typically, the built in log level constants are used in the call, which conform to the log4j levels (from least severe to most severe): TRACE, DEBUG, INFO, WARN, ERROR, FATAL. It is not a strict requirement to use these constants (any numeric value will work), though most users should find this level of granularity sufficient.

Loggers are hierarchical in the sense that any requested logger that is undefined will fall back to its most immediate defined parent logger. The absolute parent is ROOT, which is guaranteed to be defined for the system and cannot be deleted. This means that you can specify a new logger directly.

> flog.info("This will fall back to 'my', then 'ROOT'", name='my.logger')

You can also change the threshold or any other setting associated with a logger. This will create an explicit logger where any unspecified options are copied from the parent logger.

> flog.appender(appender.file("foo.log"), name='my')
> flog.threshold(ERROR, name='my.logger')
> flog.info("This won't print", name='my.logger')
> flog.error("This

If you have a function which gets called many times, it is a good strategy to pass the logger directly instead of its name.

Instead of this: > simulation_fun <- function(i)
> flog.trace("We are in loop > i
>

... you can do this:: > my_logger <- flog.logger("my.logger")
> simulation_fun2 <- function(i)
> flog.trace("We are in loop > i
>

> system.time(for (i in 1:1000) simulation_fun(i)) > system.time(for (i in 1:1000) simulation_fun2(i))

If you define a logger that you later want to remove, use flog.remove.

The option 'capture' allows you to print out more complicated data structures without a lot of ceremony. This variant doesn't accept format strings and instead appends the value to the next line of output. Consider

> m <- matrix(rnorm(12), nrow=3)
> flog.info("Matrix:",m, capture=TRUE)

which preserves the formatting, whereas using capture=FALSE will have a cluttered output due to recycling.

Author(s)

Brian Lee Yung Rowe

See Also

flog.threshold flog.remove flog.carp flog.appender flog.layout

Examples

```
flog.threshold(DEBUG)
flog.debug("This debug message will print")

flog.threshold(WARN)
flog.debug("This one won't")

m <- matrix(rnorm(12), nrow=3)
flog.info("Matrix:",m, capture=TRUE)

ftry(log(-1))

## Not run:
s <- c('FCX','AAPL','JPM','AMZN')
p <- TawnyPortfolio(s)

flog.threshold(TRACE,'tawny')
ws <- optimizePortfolio(p, RandomMatrixDenoiser())
z <- getIndexComposition()

flog.threshold(WARN,'tawny')
ws <- optimizePortfolio(p, RandomMatrixDenoiser())
z <- getIndexComposition()

## End(Not run)

## Not run:
flog.appender(appender.modulo(1000), name='counter')
lapply(1:10000, function(i) flog.info("value is %s",i, name='counter'))

## End(Not run)
```

flog.remove *Remove a logger*

Description

In the event that you no longer wish to have a logger registered, use this function to remove it. Then any references to this logger will inherit the next available logger in the hierarchy.

Arguments

name	The logger name to use

Usage

```
# Remove a logger
flog.remove(name)
```

Author(s)

Brian Lee Yung Rowe

Examples

```
flog.threshold(ERROR, name='my.logger')
flog.info("Won't print", name='my.logger')
flog.remove('my.logger')
flog.info("Will print", name='my.logger')
```

flog.threshold *Get and set the threshold for a logger*

Description

The threshold affects the visibility of a given logger. When a log statement is called, e.g. `flog.debug('foo')`, futile.logger compares the threshold of the logger with the level implied in the log command (in this case DEBUG). If the log level is at or higher in priority than the logger threshold, a message will print. Otherwise the command will silently return.

Arguments

threshold	integer The new threshold for the given logger
name	character The name of the logger

Usage

```
# Get the threshold for the given logger
flog.threshold(name) %::% character : character
flog.threshold(name=ROOT)

# Set the threshold for the given logger
flog.threshold(threshold, name=ROOT)
```

Author(s)

Brian Lee Yung Rowe

Examples

```
flog.threshold(ERROR)
flog.info("Won't print")
flog.threshold(INFO)
flog.info("Will print")
```

---

ftry *Wrap a try block in futile.logger*

Description

This function integrates futile.logger with the error and warning system so problems can be caught both in the standard R warning system, while also being emitted via futile.logger.

Usage

```
ftry(expr, error = stop, finally = NULL, silent = FALSE, details = "")
```

Arguments

expr	The expression to evaluate in a try block
error	An error handler
finally	Pass-through to tryCatch finally
silent	Boolean - should errors be rethrown? The same as the silent option on 'try'. If a custom error handler is being used that takes control over this option. Note you should test the return value if you are dependent on it.
details	An extra string to print when there's a warning message

Author(s)

Brian Lee Yung Rowe

Examples

```
## Not run:
ftry(log("a")) # Logs the warning (but the warning still bubbles)

x <- 'a'
y <- 2 # Some ID associated with x value
ftry(log("a"), details=sprintf("y = %s",y))

ftry(log(-1)) # Logs the error and rethrows it

## End(Not run)
ftry(log(-1),silent=TRUE) # logs the error and silently continues
```

---

logger.options *Constants for 'futile.logger'*

Description

Log level constants and the logger options.

Usage

```
logger.options(..., simplify = FALSE, update = list())
```

Arguments

...	TODO
simplify	TODO
update	TODO

Details

The logging configuration is managed by 'logger.options', a function generated by OptionsManager within 'futile.options'.

Author(s)

Brian Lee Yung Rowe

See Also

futile.options

prepare_arg *Provide basic parsing for layout string*

Description

Return name of argument if arg is empty. Otherwise return the value.

Usage

```
prepare_arg(x)
```

Arguments

x	The argument to prepare

scat *Print formatted messages*

Description

A replacement for cat that has built-in sprintf formatting

Usage

```
scat(format, ..., use.newline = TRUE)
```

Arguments

format	A format string passed to sprintf
...	Arguments to pass to sprintf for dereferencing
use.newline	Whether to append a new line at the end

Details

Like cat but you can use format strings.

Value

A formatted string printed to the console

Author(s)

Brian Lee Yung Rowe

Examples

```
apply(array(2:5),1, function(x) scat('This has happened %s times', x) )
```

Index

∗ **attribute**
futile.logger-package, 2
∗ **data**
flog.appender, 4
flog.carp, 6
flog.layout, 7
flog.logger, 9
flog.remove, 12
flog.threshold, 12
ftry, 13
logger.options, 14
scat, 15
∗ **logic**
futile.logger-package, 2
∗ **package**
futile.logger-package, 2

appender.console *(*flog.appender*)*, 4
appender.file *(*flog.appender*)*, 4
appender.file2 *(*flog.appender*)*, 4
appender.graylog *(*flog.appender*)*, 4
appender.modulo *(*flog.appender*)*, 4
appender.syslog *(*flog.appender*)*, 4
appender.tee *(*flog.appender*)*, 4

DEBUG *(*logger.options*)*, 14

ERROR *(*logger.options*)*, 14

FATAL *(*logger.options*)*, 14
flog.appender, *3*, 4, *8*, *11*
flog.carp, 6, *11*
flog.debug *(*flog.logger*)*, 9
flog.error *(*flog.logger*)*, 9
flog.fatal *(*flog.logger*)*, 9
flog.info *(*flog.logger*)*, 9
flog.layout, *3*, *5*, 7, *11*
flog.logger, *3*, *5*, *8*, 9
flog.namespace *(*futile.logger-package*)*, 2
flog.remove, *11*, 12
flog.threshold, *3*, *11*, 12
flog.trace *(*flog.logger*)*, 9
flog.warn *(*flog.logger*)*, 9
ftry, 13
futile.logger *(*futile.logger-package*)*, 2
futile.logger-package, 2

INFO *(*logger.options*)*, 14

layout.colored *(*flog.layout*)*, 7
layout.format *(*flog.layout*)*, 7
layout.glue *(*flog.layout*)*, 7
layout.graylog *(*flog.layout*)*, 7
layout.json *(*flog.layout*)*, 7
layout.simple *(*flog.layout*)*, 7
layout.tracearg *(*flog.layout*)*, 7
logger.options, 14

open_syslog, *4*

prepare_arg, 15

scat, 15

TRACE *(*logger.options*)*, 14

WARN *(*logger.options*)*, 14