

Logging using ParallelLogger

Martijn J. Schuemie

2020-03-28

Contents

| | | |
|----------|---|----------|
| 1 | Introduction | 1 |
| 1.1 | Terminology | 1 |
| 2 | Creating a console logger | 2 |
| 2.1 | Shorthand | 2 |
| 3 | Creating a file logger | 3 |
| 3.1 | Shorthand | 3 |
| 4 | Creating an e-mail logger | 3 |
| 4.1 | Shorthand | 4 |
| 5 | Warnings and fatal errors | 4 |
| 6 | Logging when parallel processing | 4 |
| 7 | Shiny log viewer | 5 |

1 Introduction

This vignette describes how you can use the `ParallelLogger` package to perform logging. Logging is the activity of recording events that occur during an analysis in a log. The log can be used for example for debugging, profiling (understanding performance bottlenecks), and audits.

1.1 Terminology

- **Logger:** An object that can receive **events**, and writes them to a log. A logger has a **name**, a prespecified **event level** (only events at or above that level are logged), and one or more **appenders**.
- **Event:** Consists of a message and an event level.
- **Event level:** Each event has an associated level. These levels (in ranked order) are
 - **TRACE:** Events to mark the analysis has passed through some code.
 - **DEBUG:** Events to help understand the state of the code (e.g. whether a variable has a value).
 - **INFO:** Events typically displayed to the user to inform of the progress.
 - **WARN:** Events that indicate something probably requires attention.
 - **ERROR:** Events indicating something went wrong.
 - **FATAL:** Events indicating something went wrong, causing the analysis to terminate.
- **Appender:** An object that writes to a destination, for example the console or a file. An appender uses a **layout** to format its messages. There currently are three types of appenders:
 - **Console appender:** Writes to the console, created using the `createConsoleAppender` function.
 - **File appender:** Writes to a file, created using the `createFileAppender` function.
 - **E-mail appender:** Sends an e-mail, created using the `createEmailAppender` function.

- **Layout:** Objects specifying the format in which the log will be created. The following layouts are available:
 - `layoutSimple`: Only outputs the message.
 - `layoutTimestamp`: Adds the current time and date to the message.
 - `layoutStackTrace`: Adds the time and date, and full stack trace to the message.
 - `layoutParallel`: Includes the thread identifier, name of the package and function raising the event, the current time and date, the message level, and the message itself.
 - `layoutEmail`: This layout adds the thread ID and stack trace to the message.

2 Creating a console logger

The code below demonstrates how one would create a logger that writes all events at level `INFO` or greater to the console using a layout with time stamp:

```
logger <- createLogger(name = "SIMPLE",
                      threshold = "INFO",
                      appenders = list(createConsoleAppender(layout = layoutTimestamp)))

registerLogger(logger)

logTrace("This event is below the threshold (INFO)")

logInfo("Hello world")
```

```
#> Hello world
#> 2020-03-28 04:33:28 Hello world
```

Note that the message is displayed twice. This is because there is a default logger that uses the simple layout and threshold = “INFO”, and writes to console. We can remove this logger before registering our logger to avoid duplication:

```
clearLoggers()

logger <- createLogger(name = "SIMPLE",
                      threshold = "INFO",
                      appenders = list(createConsoleAppender(layout = layoutTimestamp)))

registerLogger(logger)

logInfo("Hello world")
```

```
#> 2020-03-28 04:33:28 Hello world
```

2.1 Shorthand

A shorthand for creating a simple console logger is offered by the `addDefaultConsoleLogger` function. The code

```
addDefaultConsoleLogger()
```

is equivalent to

```
registerLogger(createLogger(name = "SIMPLE",
                          threshold = "INFO",
                          appenders = list(createConsoleAppender(layout = layoutSimple))))
```

3 Creating a file logger

Probably more useful is a file logger. In the code below, we instantiate a logger that writes to file, using a threshold of TRACE (so including all events), and using the layout for parallel processing.

```
logFileName <- "log.txt"

logger <- createLogger(name = "PARALLEL",
                      threshold = "TRACE",
                      appenders = list(createFileAppender(layout = layoutParallel,
                                                         fileName = logFileName)))

registerLogger(logger)

logTrace("Executed this line")

logDebug("There are ", length(getLoggers()), " loggers")

logInfo("Hello world")
```

```
#> 2020-03-28 04:33:29 Hello world
```

We can read the log file:

```
writeLines(readChar(logFileName, file.info(logFileName)$size))
```

```
#> 2020-03-28 04:33:29 [Main thread] TRACE evaluate timing_fn Executed this line
#> 2020-03-28 04:33:29 [Main thread] DEBUG evaluate timing_fn There are 2 loggers
#> 2020-03-28 04:33:29 [Main thread] INFO evaluate timing_fn Hello world
```

And clean it up when we're done:

```
unlink(logFileName)
```

3.1 Shorthand

A shorthand for creating the file logger detailed here is offered by the `addDefaultFileLogger` function. The code

```
addDefaultFileLogger(logFileName)
```

is equivalent to

```
registerLogger(createLogger(name = "DEFAULT",
                          threshold = "TRACE",
                          appenders = list(createFileAppender(layout = layoutParallel,
                                                             fileName = logFileName))))
```

4 Creating an e-mail logger

We can also add a logger that sends an e-mail whenever an event is logged above the specified threshold. For example, for a process running on a remote machine it might be useful to receive e-mails of fatal events:

```
mailSettings <- list(from = "someone@gmail.com",
                    to = c("someone_else@gmail.com"),
                    smtp = list(host.name = "smtp.gmail.com",
                               port = 465,
                               user.name = "someone@gmail.com",
                               passwd = "super_secret!"))
```

```

        ssl = TRUE),
    authenticate = TRUE,
    send = TRUE)

logger <- createLogger(name = "EMAIL",
    threshold = "FATAL",
    appenders = list(createEmailAppender(layout = layoutEmail,
        mailSettings = mailSettings)))

registerLogger(logger)

logFatal("No more data to process")

```

Note that the `mailSettings` object will be passed on to the `send.mail` function in the `mailR` package, so for more details see `?mailR::send.mail`

4.1 Shorthand

A shorthand for creating the e-mail logger detailed here is offered by the `addDefaultEmailLogger` function. The code

```
addDefaultEmailLogger(mailSettings)
```

is equivalent to

```

registerLogger(createLogger(name = "DEFAULT",
    threshold = "FATAL",
    appenders = list(createEmailAppender(layout = layoutEmail,
        mailSettings = mailSettings))))

```

5 Warnings and fatal errors

All R warnings and errors are automatically logged, and therefore do not require explicit logging. For example:

```

clearLoggers()
addDefaultFileLogger(logFileName)

warning("Danger!")

# This throws a warning:
as.numeric('a')

# This throws an error:
a <- b

writeLines(readChar(logFileName, file.info(logFileName)$size))

```

```

#> 2020-03-28 04:33:29 [Main thread]   WARN    evaluate    timing_fn    Danger!
#> 2020-03-28 04:33:29 [Main thread]   WARN    evaluate    timing_fn    Warning: NAs introduced by coercion
#> 2020-03-28 04:33:29 [Main thread]   FATAL    evaluate    timing_fn    Error: object a not found

```

6 Logging when parallel processing

The logging functions are designed to work with the parallel processing functions included in this package. The `layoutParallel` records thread identifiers, making it possible to later untangle the various threads.

Below is a simple example:

```
unlink(logFileName) # Clean up log file from the previous example
clearLoggers() # Clean up the loggers from the previous example

addDefaultFileLogger(logFileName)

cluster <- makeCluster(3)

fun <- function(x) {
  ParallelLogger::logInfo("The value of x is ", x)
  # Do something
  if (x == 6)
    ParallelLogger::logDebug("X equals 6")
  return(NULL)
}

dummy <- clusterApply(cluster, 1:10, fun, progressBar = FALSE)

stopCluster(cluster)

writeLines(readChar(logFileName, file.info(logFileName)$size))
```

```
#> 2020-03-28 04:33:29 [Main thread] TRACE evaluate timing_fn Initiating cluster with 3 threads
#> 2020-03-28 04:33:31 [Thread 1] TRACE Thread 1 initiated
#> 2020-03-28 04:33:31 [Thread 2] TRACE Thread 2 initiated
#> 2020-03-28 04:33:32 [Thread 3] TRACE Thread 3 initiated
#> 2020-03-28 04:33:32 [Thread 1] INFO The value of x is 1
#> 2020-03-28 04:33:32 [Thread 3] INFO The value of x is 3
#> 2020-03-28 04:33:32 [Thread 1] INFO The value of x is 4
#> 2020-03-28 04:33:32 [Thread 3] INFO The value of x is 5
#> 2020-03-28 04:33:32 [Thread 2] INFO The value of x is 6
#> 2020-03-28 04:33:32 [Thread 1] INFO The value of x is 7
#> 2020-03-28 04:33:32 [Thread 3] INFO The value of x is 8
#> 2020-03-28 04:33:32 [Thread 1] INFO The value of x is 9
#> 2020-03-28 04:33:32 [Thread 2] INFO The value of x is 10
#> 2020-03-28 04:33:32 [Main thread] TRACE evaluate timing_fn Stopping cluster
#> 2020-03-28 04:33:32 [Thread 3] TRACE Thread 3 terminated
#> 2020-03-28 04:33:32 [Thread 2] TRACE Thread 2 terminated
```

7 Shiny log viewer

A Shiny app for viewing a log file created using the `layoutParallel` is included in the package. To explore the log created in the prior example, run

```
launchLogViewer(logFileName)
```

to launch the viewer shown in Figure 1.

Log File Viewer - C:\Users\mschuemi\Git\OhdsiRTools\vignettes\log.txt

Level

TRACE ▾

Thread

All ▾

Package

All ▾

Search:

| Timestamp | Thread | Level | Package | Function | Message |
|---------------------|---------------|-------|---------|----------|-----------------------------------|
| 2018-04-23 15:06:04 | [Main thread] | TRACE | | | Initiating cluster with 3 threads |
| 2018-04-23 15:06:10 | [Thread 1] | TRACE | | | Thread 1 initiated |
| 2018-04-23 15:06:10 | [Thread 2] | TRACE | | | Thread 2 initiated |
| 2018-04-23 15:06:10 | [Thread 3] | TRACE | | | Thread 3 initiated |
| 2018-04-23 15:06:10 | [Thread 3] | INFO | | | The value of x is 3 |
| 2018-04-23 15:06:10 | [Thread 2] | INFO | | | The value of x is 2 |
| 2018-04-23 15:06:10 | [Thread 1] | INFO | | | The value of x is 1 |
| 2018-04-23 15:06:10 | [Thread 3] | INFO | | | The value of x is 4 |
| 2018-04-23 15:06:10 | [Thread 2] | INFO | | | The value of x is 5 |
| 2018-04-23 15:06:10 | [Thread 1] | INFO | | | The value of x is 6 |
| 2018-04-23 15:06:10 | [Thread 3] | INFO | | | The value of x is 7 |
| 2018-04-23 15:06:10 | [Thread 2] | INFO | | | The value of x is 8 |
| 2018-04-23 15:06:10 | [Thread 2] | INFO | | | The value of x is 10 |
| 2018-04-23 15:06:10 | [Thread 3] | INFO | | | The value of x is 9 |
| 2018-04-23 15:06:10 | [Thread 1] | DEBUG | | | X equals 6 |
| 2018-04-23 15:06:11 | [Main thread] | TRACE | | | Stopping cluster |
| 2018-04-23 15:06:11 | [Thread 1] | TRACE | | | Thread 1 terminated |
| 2018-04-23 15:06:11 | [Thread 2] | TRACE | | | Thread 2 terminated |
| 2018-04-23 15:06:11 | [Thread 3] | TRACE | | | Thread 3 terminated |

Showing 1 to 19 of 19 entries

Figure 1: Shiny log viewer app