A Package for “safe mode” R Sessions

Paul Murrell

Abstract

This document describes the R package **safemode**, which provides a function to monitor the activity that takes place on the console in an R session and issue warnings when expressions are evaluated in an inappropriate order.

The document describes both the use of the command and also provides a literate version of the function itself.

This package has as its basis the R code from Ross Ihaka’s “A Function for R Session Scripting.”

1 The `safemode()` Function

When `safemode()` is invoked, a sub-interpreter is run to process the user’s commands in “safe mode.” When this sub-interpreter is running, the the R command prompt is changed to `safe>` and the continuation prompt to `safe+`. The sub-interpreter is exited by typing the command `q()`.

```
> safemode()

safe> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10

safe> max(rnorm(100))
[1] 2.592984

safe> q()
```

While in “safe mode,” expressions are checked to make sure that the no symbols are “stale.” A symbol is stale if it was assigned a value less recently than one or more of its dependents. For example, in the following, the symbol `y` becomes dependent on `x`, so if `x` is modified, `y` becomes stale.

```
> safemode()

safe> x <- 1
safe> y <- x + 1
safe> x <- 2
safe> y
[1] 2
Warning message:
In withCallingHandlers(warning(staleWarnMsg(tracked[staleDeps])),  ... :
  Symbol 'y' is stale!
```

This is essentially all there is to know about using `safemode()`, other than to note that a `safemode()` command cannot be run from a `safemode()` sub-interpreter.

2 Implementation of the `safemode()` Function

The code for the `safemode` function is implemented as a *closure.* The support functions it uses are encapsulated in a private environment, visible only to that function. The mechanism used is as follows.

2a ⟨*safemode.R* 2a⟩≡

```
  ⟨comments-and-copyright 16⟩
  ⟨initialisation 2b⟩
  safemode <- local({
        ⟨warning state variables 13a⟩
        ⟨support functions 10b⟩
        ⟨read-eval-print loop 4⟩
        ⟨main function 2c⟩
  })
```

This code is written to file `safemode.R`.

2.1 Initialisation

The **safemode** package records a database of time stamps for symbols and a database of dependencies for symbols.

2b ⟨*initialisation* 2b⟩≡ (2a)

```
  timeDB <- new.env()
  depDB <- new.env()
```

2.2 The main function

The main function, `safemode()`, takes two arguments: whether to print out debugging information (`FALSE` by default) and a file to read input from (`NULL` by default, which means take input from the command line). This function calls the main workhorse function that provides a read-eval-print-loop.

2c ⟨*main function* 2c⟩≡ (2a)

```
  function(debug=FALSE, infile=NULL) {
        ⟨call the read-eval-print-loop 2d⟩
        ⟨shut down 3⟩
  }
```

The first argument passed to the `repl()` function is the environment that the `safemode()` function was called from. This will typically be the R global enviroment. The second argument is a debugging flag. The third argument is a file to read R code from (or `NULL`).

2d ⟨*call the read-eval-print-loop* 2d⟩≡ (2c)

```
  repl(sys.parent(), debug, infile)
```

Uses `repl` 4.

On exit, the main function erases the time stamp and dependency databases and returns an invisible (`NULL`) value.

3 ⟨*shut down* 3⟩≡ (2c)

```
rm(list=ls(timeDB), envir=timeDB)
rm(list=ls(depDB), envir=depDB)
invisible()
```

2.3 The read-eval-print loop

The `repl()` function takes over the role of the topmost level of functionality in R. It reads the lines of text that the user types, parses them and evaluates the results. It also has to handle exceptional conditions such as errors, warnings and user interrupts.

The important strategy employed in this function is used to accumulate the lines the user types until a complete expression has been read. Reading the lines is easy; it is done with `readline()`. Checking for a complete expression is trickier because parsing an incomplete expression trips an error. These must be caught using the `tryCatch()` mechanism and this type of error discriminated from other syntax errors.

There is also the problem of user interrupts. These can occur at any point in the read-eval-print process. To protect against such interrupts the whole read-eval-print process is embedded in a loop whose sole task is to catch and process interrupts.

The general structure of the `repl()` function is shown by the following function. Initial values are defined for the command prompt and the current expression, we determine whether we are running in batch mode (or interactively), and then the interrupt catching loop is run.

After exiting "safe mode", if we are in bacth mode, there is some shut down to perform.

4 ⟨*read-eval-print loop* 4⟩≡ (2a)

```
repl <- function(env, debug, infile) {
    prompt <- "safe> "
    cmd <- character()
    if (is.null(infile)) {
        batch <- FALSE
    } else {
        batch <- TRUE
        ⟨init batch mode 15a⟩
    }
    repeat {
        ⟨interrupt catching 5a⟩
    }
    if (batch) {
        ⟨batch shut down 15f⟩
    }
}
```

Defines:
  `cmd`, used in chunks 5–7 and 15.
  `prompt`, used in chunks 5 and 6b.
  `repl`, used in chunk 2d.

The code inside the `repeat` loop, in the function above, runs the *repl* and catches any interrupts that occur with a `tryCatch()` statement. The statement catches just interrupts and gives a fresh prompt.

5a ⟨*interrupt catching* 5a⟩≡ (4)

```
ans <- tryCatch(
    repeat {
        ⟨parse and evaluate expressions 5b⟩
    }, interrupt = function(x) x)
if (inherits(ans, "interrupt")) {
    cat("\nInterrupt!\n")
    prompt <- "script> "
    cmd <- character()
} else {
    stop("Interrupt catcher caught non-interrupt")
}
```

Uses `cmd` 4 and `prompt` 4.

Expressions are read and processed in a loop. A pass through the loop reads a single line of input with `readline()` and adds it to the `cmd` buffer (unless we are in batch mode). Each line of input is also added to the command line history with the `timestamp()` function.

We handle (whole-line) comments as a special case, immediately discarding them (or echoing them in batch mode).

Each time a line is added, an attempt is made to parse the contents of `cmd` and obtain a valid expression for evaluation. The parse is wrapped in a `tryCatch()` to trap any parsing errors that occur. The result of this attempted parse determines what happens next.

5b ⟨*parse and evaluate expressions* 5b⟩≡ (5a)

```
repeat {
    if (batch) {
        ⟨batch read 15b⟩
    } else {
        cmd <- c(cmd, readline(prompt))
        timestamp(cmd, prefix="", suffix="", quiet=TRUE)
    }
    # Handle EOF in batch mode
    if (!length(cmd)) {
        return()
    }
    if (grepl("^#", cmd)) {
        if (batch) {
            ⟨batch comment 15c⟩
        }
        cmd <- character()
        break
    }
    ans <- tryCatch(parse(text = cmd), error = function(e) e)
    ⟨handle the results of the parse 6a⟩
}
```

Uses `cmd` 4 and `prompt` 4.

The result returned by the `tryCatch()` is either a valid expression that can be evaluated or an error condition. We branch depending on the type of result obtained.

6a ⟨*handle the results of the parse* 6a⟩≡ (5b)

```
⟨handle the expression 6b⟩
```

There are two possible types of error to deal with. Errors can be caused by an incomplete parse or by some other type of syntax error. If the expression is incomplete, we change the prompt to indicate continuation and return to the top of the loop to fetch another line of input. If there was some other type of error, we deal with the error then we reset the command prompt and the state of the input buffer.

If there was no error, we have a valid expression. We then choose between a number of special cases (such as quitting "safe mode") and the general case of evaluating the expression typed by the user. When that is complete, we reset the command prompt and the state of the command buffer before continuing on to read the next expression.

6b ⟨*handle the expression* 6b⟩≡ (6a)

```
if (inherits(ans, "error")) {
    if (incompleteParse((ans))) {
        prompt <- "safe+ "
    } else {
        handleParseError(ans)
        prompt <- "safe> "
        cmd <- character()
    }
} else {
    ⟨handle special expression cases 7⟩
    ⟨handle the general expression case 8⟩
    prompt <- "safe> "
    cmd <- character()
}
```

Uses `cmd` 4, `handleParseError` 12a, `incompleteParse` 11c, and `prompt` 4.

If the expression was empty (the user idly typed the enter key) we simply go back to fetch another expression. If the user typed `q()` then we exit from the repl and return to the top-level function. If for some reason the user tried to invoke `safemode()` we issue an error. (This probably needs further thought.)

7 ⟨*handle special expression cases* 7⟩≡ (6b)

```
special <- TRUE
if (length(ans) == 0) {
    if (batch) {
        ⟨batch blank line 15d⟩
    }
    cmd <- character()
    break
} else if (isQuitCall(ans)) {
    return()
} else if (grepl("^safemode\\(",
               deparse(ans[[1]], nlines = 1))) {
    cat("Error: You can't call safemode() while in \"safe mode\"\n")
    break
} else {
    special <- FALSE
}
```

Uses `cmd` 4 and `isQuitCall` 15g.

If none of these special cases hold, we are in the general situation. We evaluate the expression that the user typed and print the answer. Note that it is possible for parsing to produce several calls in the expression returned from the parse. (Such calls are separated by semicolons.) To handle the general case, we loop over the elements of the expression evaluating and printing each one in turn.

After evaluation, a check is made of whether any new warnings have been issued. If there were, the warnings are transferred to the global variable `last.warning`. There, they can be accessed with calls to the function `warnings()`. Finally, a call is made to `displayWarnings()` to display the warning messages in the correct way.

8 ⟨*handle the general expression case* 8⟩≡ (6b)

```
if (!special) {
    renewwarnings <<- TRUE
    newwarnings <<- FALSE
    if (batch) {
        ⟨batch expression 15e⟩
    }
    for(e in ans) {
        ⟨evaluate expression in safe mode 9a⟩
    }
    if (newwarnings) {
        warnings = warningCalls
        names(warnings) = warningMessages
        assign("last.warning",
               warnings[1:nwarnings],
               "package:base")
        displayWarnings(nwarnings)
    }
}
```

Uses `displayWarnings` 14a, `newwarnings` 13a, `nwarnings` 13a, `renewwarnings` 13a, `warningCalls` 13a, and `warningMessages` 13a.

2.4 Evaluating expressions in “safe mode”

For each expression, `e`, we determine which symbols need checking, check for any stale symbols, then evaluate the expression.

Before we evaluate the expression, we deparse it so we have a text version of the code.

Evaluation is carried out inside a `tryCatchWithWarnings()` call. This means that any warnings that occur are recorded (in the variables `warningCalls` and `warningMessages`). Evaluation also occurs in the parent environment of the `safemode()` call, `env` (which will typically be the global environment).

If there were no errors, we record new time stamps and dependencies for any symbols assigned in the expression.

9a ⟨*evaluate expression in safe mode* 9a⟩≡ (8)

```
⟨determine tracked symbols in expression 9b⟩
⟨check for stale symbols in expression 9c⟩
code <- deparse(e)
e <- tryCatchWithWarnings(withVisible(eval(e,
                                           envir = env)))
if (inherits(e, "error")) {
    handleError(e)
} else {
    handleValue(e)
    ⟨record time stamps and dependencies 10a⟩
}
```

Uses `handleError` 12b, `handleValue` 12c, and `tryCatchWithWarnings` 13b.

To determine which symbols need to be checked, we use `findGlobals()` from the **codetools** package. This involves setting up a dummy function (with no arguments) because `findGlobals()` only works on closures. We also can only check symbols for which we already have a time stamp.

9b ⟨*determine tracked symbols in expression* 9b⟩≡ (9a)

```
dummy <- function() {}
body(dummy) <- e
vars <- findGlobals(dummy)
tracked <- vars[vars %in% ls(timeDB)]
⟨debug globals 14b⟩
```

If there are any symbols to check, and any of those symbols are stale, we issue a warning.

9c ⟨*check for stale symbols in expression* 9c⟩≡ (9a)

```
if (length(tracked) > 0) {
    staleDeps <- sapply(tracked, stale)
    if (any(staleDeps)) {
        withCallingHandlers(warning(staleWarnMsg(tracked[staleDeps])),
                            warning = warningHandler)
    }
}
```

To determine whether the expression involved an assignment, we use `getInputs()` from the **CodeDepends** package. If that function determines that there are “output” or “update” symobls in the expression, then we have an assignment, so we record a new time stamp (and update the dependencies) for the symbol that was assigned a new value. The `get_nanotime()` function from the **microbenchmark** package is used to get more accurate timings.

10a ⟨*record time stamps and dependencies* 10a⟩≡ (9a)

```
# test for whether expression was an assignment
sc <- readScript("", txt=code)
info <- scriptInfo(sc)
inputs <- info[[1]]@inputs
outputs <- info[[1]]@outputs
updates <- info[[1]]@updates
⟨debug inputs and outputs 14c⟩
assignment <- FALSE
symbol <- character()
if (length(outputs) > 0) {
    symbol <- c(symbol, outputs)
    assignment <- TRUE
}
if (length(updates) > 0) {
    symbol <- c(symbol, updates)
    assignment <- TRUE
}
if (assignment) {
    for (i in symbol) {
        assign(i, get_nanotime(), envir=timeDB)
        assign(symbol, tracked, envir=depDB)
    }
    ⟨debug time and dependency databases 14d⟩
}
```

2.5 Stale symbol support functions

The functions `age()` and `deps()` provide convenient access to the time stamp and dependencies databases.

10b ⟨*support functions* 10b⟩≡ (2a) 11a ▷

```
age <- function(x) {
    get(x, timeDB, inherits=FALSE)
}
deps <- function(x) {
    get(x, depDB, inherits=FALSE)
}
```

The `stale()` function finds all dependencies for a symbol and checks that the symbol is older than all of its dependents, and that all of its dependents are not stale.

11a ⟨*support functions* 10b⟩+≡ (2a) ◁10b 11b▷

```
stale <- function(x) {
    dependents <- deps(x)
    length(dependents) &&
        (any(age(x) < sapply(dependents, age)) ||
         any(sapply(dependents, stale)))
}
```

The `staleWarnMsg()` function generates text for a warning message.

11b ⟨*support functions* 10b⟩+≡ (2a) ◁11a 11c▷

```
staleWarnMsg <- function(deps) {
    N <- length(deps)
    if (N == 1) {
        paste0("Symbol ’", deps, "’ is stale!")
    } else if (N == 2) {
        paste0("Symbols ’",
               paste(deps, collapse="’ and ’"),
               "’ are stale!")
    } else {
        paste0("Symbols ’",
               paste(paste(deps[-N], collapse="’, ’"),
                     deps[N], sep="’, and ’"),
               "’ are stale!")
    }
}
```

2.6 Parsing support functions

An incomplete parse is detected when the result of the parse is an error that contains the string `"unexpected end of input"`.

11c ⟨*support functions* 10b⟩+≡ (2a) ◁11b 12a▷

```
incompleteParse <- function(e) {
    (inherits(e, "error") &&
     grepl("unexpected end of input", e$message))
}
```

Defines:
`incompleteParse`, used in chunk 6b.

The most complicated support function is the one that handles the printing of error messages from parsing. Because the parse is taking place using a character vector as input, the error messages produced look rather different from those produced when the parser gets its input from the console. This function transforms the error messages into that form.

12a ⟨*support functions* 10b⟩+≡ (2a) ◁11c 12b▷

```
handleParseError <- function(e) {
    msg = strsplit(conditionMessage(e), "\n")[[1]]
    errortxt = msg[1]
    msg = gsub("[0-9]+: ", "", msg[-c(1, length(msg))])
    msg = msg[length(msg) - 1:0]
    if (length(msg) == 1)
        msg = paste(" in: \"", msg, "\"\n", sep = "")
    else
        msg = paste(" in:\n\"",
                    paste(msg, collapse = "\n"),
                    "\"\n", sep = "")
    cat("Error",
        gsub("\n.*", "",
             gsub("<text>:[0-9]+:[0-9]+", "",
                  errortxt)),
        msg, sep = "")
}
```

Defines:
`handleParseError`, used in chunk 6b.

2.7 Input-output support

The error messages produced during evaluation are easy to process. We simply cat them to the output.

12b ⟨*support functions* 10b⟩+≡ (2a) ◁12a 12c▷

```
handleError <- function(e) {
    cat("Error in", deparse(conditionCall(e)),
        ":", conditionMessage(e), "\n")
}
```

Defines:
`handleError`, used in chunk 9a.

Printing the values that result from evaluating expressions has one wrinkle to it. We have to check the visibility of the result and only print “visible” results.

12c ⟨*support functions* 10b⟩+≡ (2a) ◁12b 13b▷

```
handleValue <- function(e) {
    if (e$visible) {
        print(e$value)
    }
}
```

Defines:
`handleValue`, used in chunk 9a.

2.8 Warning support

A number of top-level closure variables are used to manage the warning messages produced by evaluation of expressions. The following variables manage the accumulation of error messages.

`warningCalls`	holds the calls that produced warnings
`warningMessages`	holds the warning messages
`nwarnings`	the number or warnings accumulated
`renewwarnings`	purge the warning list on next warning?
`newwarnings`	has the evaluation produced new warnings

The variables are initialised as follows.

13a ⟨*warning state variables* 13a⟩≡ (2a)

```
warningCalls <- vector("list", 50)
warningMessages <- character(50)
nwarnings <- 0
renewwarnings <- TRUE
newwarnings <- FALSE
```

Defines:
`newwarnings`, used in chunks 8 and 13b.
`nwarnings`, used in chunks 8, 13b, and 14a.
`renewwarnings`, used in chunks 8 and 13b.
`warningCalls`, used in chunks 8 and 13b.
`warningMessages`, used in chunks 8 and 13b.

Warnings are trapped by the following two functions. The effect is to simply add warnings to the accumulated list of warnings and then call the built-in `muffleWarning()` restart.

13b ⟨*support functions* 10b⟩+≡ (2a) ◁12c 14a▷

```
warningHandler <- function(w) {
    newwarnings <<- TRUE
    if (renewwarnings) {
        renewwarnings <<- FALSE
        nwarnings <<- 0
    }
    n <- nwarnings + 1
    if (n <= 50) {
        warningCalls[[n]] <<- conditionCall(w)
        warningMessages[n] <<- conditionMessage(w)
        nwarnings <<- n
    }
    invokeRestart("muffleWarning")
}
tryCatchWithWarnings <- function(expr) {
    withCallingHandlers(tryCatch(expr,
                                 error = function(e) e),
                        warning = warningHandler)
}
```

Defines:
`tryCatchWithWarnings`, used in chunk 9a.
Uses `newwarnings` 13a, `nwarnings` 13a, `renewwarnings` 13a, `warningCalls` 13a, and `warningMessages` 13a.

The `displayWarnings()` function is used to display warnings at the end of an evaluation. If there are 10 or fewer messages they are displayed. If there are more than 10 messages, the user is told to inspect them with `warnings()`. Only the first 50 messages are stored.

14a ⟨*support functions* 10b⟩+≡ (2a) ◁13b 15g▷

```
displayWarnings <- function(n) {
    if (n <= 10) {
        print(warnings())
    } else if (n < 50) {
        cat("There were",
            nwarnings,
            "warnings (use warnings() to see them)\n")
    } else {
        cat("There were 50 or more warnings",
            "(use warnings() to see the first 50)\n")
    }
}
```

Defines:
  `displayWarnings`, used in chunk 8.
Uses `nwarnings` 13a.

2.9 Debugging support

If the `debug` flag is set to `TRUE` a variety of debugging information is spewed out for each expression.

14b ⟨*debug globals* 14b⟩≡ (9b)

```
if (debug) {
    cat(paste("globals: ", paste(vars, collapse=", "), "\n"))
    cat(paste("tracked: ", paste(tracked, collapse=", "), "\n"))
}
```

14c ⟨*debug inputs and outputs* 14c⟩≡ (10a)

```
if (debug) {
    cat(paste("inputs: ", paste(inputs, collapse=", "), "\n"))
    cat(paste("outputs: ", paste(outputs, collapse=", "), "\n"))
    cat(paste("updates: ", paste(updates, collapse=", "), "\n"))
}
```

14d ⟨*debug time and dependency databases* 14d⟩≡ (10a)

```
if (debug) {
    cat("Time stamp database:\n")
    print(sapply(ls(timeDB), get, envir=timeDB))
    cat("Dependencies database:\n")
    print(sapply(ls(depDB), get, envir=depDB))
}
```

2.10 Batch mode

If the `infile` argument to `safemode()` is non-`NULL`, we open a file to read from (rather than reading from the command line).

15a ⟨*init batch mode* 15a⟩≡ (4)

```
con <- file(infile, "r")
```

In batch mode, we read from the connection rather than from the command line.

15b ⟨*batch read* 15b⟩≡ (5b)

```
cmd <- c(cmd, readLines(con, n=1))
```

Uses `cmd` 4.

In batch mode, comments are echoed to stdout.

15c ⟨*batch comment* 15c⟩≡ (5b)

```
cat(paste0("safe> ", cmd), "\n")
```

Uses `cmd` 4.

In batch mode, blank lines are echoed to stdout.

15d ⟨*batch blank line* 15d⟩≡ (7)

```
cat("\n")
```

In batch mode, we echo the expression text to stdout.

15e ⟨*batch expression* 15e⟩≡ (8)

```
cat(paste0(c("safe> ",
             rep("safe+ ",
                 max(0, length(cmd) - 1))),
           cmd), sep="\n")
```

Uses `cmd` 4.

In batch mode, we must close the input connection.

15f ⟨*batch shut down* 15f⟩≡ (4)

```
close(con)
```

2.11 Miscellany

The following function does a quick-and-dirty check of whether a user typed `q()` at the command prompt. It is rather easy to defeat this. For example, typing `(q())` will cause an immediate exit from R.

15g ⟨*support functions* 10b⟩+≡ (2a) ◁14a

```
isQuitCall <- function(e) {
    (!inherits(e, "error") &&
      length(e) == 1 &&
      deparse(e[[1]], nlines = 1) == "q()")
}
```

Defines:
`isQuitCall`, used in chunk 7.

2.12 Comments and copyright

16 ⟨*comments-and-copyright* 16⟩≡ (2a)

```
### Original code and documentation copyright Ross Ihaka, 2011
###
### Modifications copyright Paul Murrell, 2015
###
### Distributed under the terms of GPL3, but may also be
### redistributed under any later version of the GPL.
###
### DO NOT edit this file directly.
### This R code was generated from a literate document;
### all changes should be made to that literate document.
###
### Safe mode for R
###
### Synopsis:
###
### This function provides an environment that provides some
### protection from stupidity arising from laziness
###
###   safemode()
###   ...
###   q()
###
### Exit from safe mode using using q()
###
### This is best regarded as an exercise in getting familar
### with R's condition system and a demonstration of how
### to write an interpreted REPL and an exploration of
### the 'codetools' and 'CodeDepends' packages.
```

Chunk Index

Identifier Index