Chapter 8: Conditions

Ryan Heslin

May 30, 2022

Signaling Conditions

You can produce errors, warnings, and messages, in descending order of severity.

```
f <- function() g()
g <- function() h()
h <- function() stop("This is an error!")

f()
```

```
Error in h(): This is an error!
```

```
#> Error in h(): This is an error!
```

`rlang::abort` takes a single message argument rather than taking them via `...`.

```
library(rlang)
h <- function() abort("This is an error!")
f()
```

```
Error in ‘h()‘:
! This is an error!
```

```
#> Error: This is an error!
```

1.

```r
file.remove2 <- function(path) {
  if (!file.exists(path)) stop(path, " does not exist")
  file.remove(path)
}
try(file.remove2("not_a_file.txt"))
```

```
Error in file.remove2("not_a_file.txt") : not_a_file.txt does not exist
```

2.

It adds a trailing newline, which `cat` does not do by default.

Handling Conditions

Warnings should be reserved for deprecation notices and errors that can *probably* be recovered from.

`message` is best reserved for side-effects functions; `cat` should be used where printing is the main task of the function.

Messages are (by default) saved and printed when control returns to the top level; warnings are printed immediately. `try` can be used to wrap errors, but better `tryCatch`.

```r
f2 <- function(x) {
  try(log(x))
  10
}
f2("a")
```

```
Error in log(x) : non-numeric argument to mathematical function
```

```
[1] 10
```

```
#> Error in log(x) : non-numeric argument to mathematical function
#> [1] 10
```

Unlike `tryCatch`, `withCallingHandlers` returns control to the context where the condition was triggered, not a handler function. `tryCatch` is better for errors for that reason. It also has a `finally` argument that does the obvious thing. It is better suited for warnings and messages.

```
withCallingHandlers(
  warning = function(cnd) {
    # code to run when warning is signalled
  },
  message = function(cnd) {
    # code to run when message is signalled
  },
  code_to_run_while_handlers_are_active
)
```

Conditions are implemented with condition objects, which contain call and message objects.

An illustration of the differences.

```
tryCatch(
  message = function(cnd) cat("Caught a message!\n"),
  {
    message("Someone there?")
    message("Why, yes!")
  }
)
```

```
Caught a message!
```

```
#> Caught a message!
```

```
withCallingHandlers(
  message = function(cnd) cat("Caught a message!\n"),
  {
    message("Someone there?")
    message("Why, yes!")
  }
)
```

```
Caught a message!
Caught a message!
```

```
#> Caught a message!
#> Someone there?
#> Caught a message!
#> Why, yes!
```

`rlang` has a function to “muffle” conditions and prevent them from propagating upward.

```
# Muffles the default handler which prints the messages
withCallingHandlers(
  message = function(cnd) {
    cat("Level 2\n")
    cnd_muffle(cnd)
  },
  withCallingHandlers(
    message = function(cnd) cat("Level 1\n"),
    message("Hello")
  )
)
```

```
Level 1
Level 2
```

```
#> Level 1
#> Level 2

# Muffles level 2 handler and the default handler
withCallingHandlers(
  message = function(cnd) cat("Level 2\n"),
  withCallingHandlers(
    message = function(cnd) {
      cat("Level 1\n")
      cnd_muffle(cnd)
    },
    message("Hello")
  )
)
```

```
Level 1
```

```
#> Level 1
```

1.

`abort` has the stack trace and cleaner printing. They can also contain a class, parent classes, and arbitrary metadata via `...`.

```
catch_cnd(stop("An error"))
```

```
<simpleError in force(expr): An error>
```

```
catch_cnd(abort("An error"))
```

```
<error/rlang_error>
Error:
! An error
Backtrace:
```

2.

Error, `NULL`, warning, message

```
show_condition <- function(code) {
  tryCatch(
    error = function(cnd) "error",
    warning = function(cnd) "warning",
    message = function(cnd) "message",
    {
      code
      NULL
    }
  )
}

show_condition(stop("!"))
```

```
[1] "error"
```

```
show_condition(10)
```

```
NULL
```

```
show_condition(warning("?!"))
```

```
[1] "warning"
```

```
show_condition({
  10
  message("?")
  warning("?!")
})
```

```
[1] "message"
```

3.

First control bubbles up to the outer handler, then the inner handler, then the inner handler triggers the outer handler again, then finally the innermost call to `message` evaluates.

```
withCallingHandlers(
  message = function(cnd) message("b"),
  withCallingHandlers(
    message = function(cnd) message("a"),
    message("c")
  )
)
#> b
#> a
#> b
#> c
```

4.

`catch_cnd` creates a named list of handlers that are all `identity`. Then the expression is evaluated in a call to `tryCatch` with the dummy handlers spliced in. This catches whatever condition is generated.

5.

```
show_condition2 <- function(code) {
  handler <- function(cnd) rlang::class(catch_cnd(cnd))[[2]]
  tryCatch(
    error = handler,
    warning = handler,
    message = handler,
    {
      code
      NULL
    }
  )
}
show_condition(stop("!"))
```

```
[1] "error"
```

```
show_condition(10)
```

```
NULL
```

```r
show_condition(warning("?!"))
```

```
[1] "warning"
```

```r
show_condition({
  10
  message("?")
  warning("?!")
})
```

```
[1] "message"
```

Custom Conditions

Base conditions don't allow complex metadata, which is annoying.

> Name, message, then metadata

```r
abort(
  "error_not_found",
  message = "Path `blah.csv` not found",
  path = "blah.csv"
)
```

```
Error:
! Path `blah.csv` not found
```

```r
#> Error: Path `blah.csv` not found

abort_bad_argument <- function(arg, must, not = NULL) {
  msg <- glue::glue("`{arg}` must {must}")
  if (!is.null(not)) {
    not <- typeof(not)
    msg <- glue::glue("{msg}; not {not}.")
  }

  abort("error_bad_argument",
    message = msg,
    arg = arg,
    must = must,
    not = not
  )
}
```

Handlers can be specified for each class. The first matching handler is chosen, not the most specific; be careful!

```
tryCatch(
  error = function(cnd) "other error",
  error_bad_argument = function(cnd) "bad_argument",
  my_log("a")
)
#> [1] "other error"
```

1.

```
check_loaded <- function(pkg) {
  if (!requireNamespace(pkg, quietly = FALSE)) {
    abort(
      message = paste("Package", sQuote(pkg), "is required but not available"),
      class = "error_missing_package",
      package = pkg,
      call = parent.frame()
    )
  }
}
check_loaded("foo")
```

```
Error:
! Package 'foo' is required but not available
```

2.

Make all my error messages belong to a new subclass that takes the error from a parent class and prints the added message: "The error handling interface is unstable and should not be relied on by unit tests or package code."

Applications

1.

I admit I took the trick of the invisible return from the solutions manual.

```
suppressConditions <- function(expr, handler = stop()) {
  tryCatch(withCallingHandlers(expr,
    message = function(c) tryInvokeRestart("muffleMessage"),
    warning = function(w) {
      tryInvokeRestart("muffleWarning")
    }
  ),
  error = function(e) invisible(e)
  )
}

suppressConditions(5 + "a", return("Failure"))
suppressConditions(message("Hi"))
suppressConditions(5 + 8)
```

```
[1] 13
```

```
suppressConditions(warning("Bad news!"))
```

2.

Using `withCallingHandlers` ensures the traceback starts from the code that triggered the error, not the error handler.

3.

```
catch_cnds <- function(expr) {
  conds <- list()
  add_cond <- function(cnd) {
    conds <<- append(conds, list(cnd))
    conditionMessage(cnd)
    # cnd_muffle(cnd)
  }

  tryCatch(
    error = function(cnd) {
      conds <<- append(conds, list(cnd))
    },
    withCallingHandlers(
      message = add_cond,
      warning = add_cond,
      expr
    )
  )
  conds
}

catch_cnds({
  inform("a")
  warn("b")
  abort("C")
})
```

```
[[1]]
<message/rlang_message>
Message:
a

[[2]]
<warning/rlang_warning>
Warning:
b

[[3]]
<error/rlang_error>
Error:
! C
Backtrace:
```

```
#> [[1]]
#> <message: a
#> >
#>
#> [[2]]
#> <warning: b>
#>
#> [[3]]
#> <error/rlang_error>
#> C
#> Backtrace:
#>  1. global::catch_cnds(...)
#>  6. base::withCallingHandlers(...)
```

4.

You'll take down every last bottle, and you'll like it!

```
bottles_of_beer <- function(i = 99) {
  message(
    "There are ", i, " bottles of beer on the wall, ",
    i, " bottles of beer."
  )
  while (i > 0) {
    tryCatch(
      Sys.sleep(1),
      interrupt = function(err) {
        i <<- i - 1
        if (i > 0) {
          message(
            "Take one down, pass it around, ", i,
            " bottle", if (i > 1) "s", " of beer on the wall."
          )
        }
      }
    )
  }
  message(
    "No more bottles of beer on the wall, ",
    "no more bottles of beer."
  )
}
bottles_of_beer()
```