Package 'catchr'

September 23, 2021

Type Package

Title Taking the Pain Out of Catching and Handling Conditions

Version 0.2.31

Maintainer Zachary Burchill <zach.burchill.code@gmail.com>

Description R has a unique way of dealing with warnings, errors, messages, and other conditions, but it can often be troublesome to users coming from different programming backgrounds. The purpose of this package is to provide flexible and useful tools for handling R conditions with less hassle. In order to lower the barrier of entry, keep code clean and readable, and reduce the amount of typing required, `catchr` uses a very simple domain-specific language that simplifies things on the front-end.
This package aims to maintain a continuous learning curve that lets new users jump straight in to condition-handling, while simultaneously offering depth and complexity for more advanced users.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.1.1

Imports rlang (>= 0.3.1), purrr (>= 0.2.0),

Depends R (>= 3.3.0)

Suggests testthat, beepr, crayon, covr, knitr, rmarkdown, spelling

VignetteBuilder knitr

Language en-US

NeedsCompilation no

Author Zachary Burchill [aut, cre, cph]

Repository CRAN

Date/Publication 2021-09-23 16:40:05 UTC

R topics documented:

`args_and_kwargs` *Separate dots into Python-esque* `*args` *and* `**kwargs`

Description

This function will return a named list with two sublists, 'args' and 'kwargs', which contain the unnamed and named arguments as quosures, respectively.

This is useful for when you want these two types of arguments to behave differently, e.g., as they do in `make_plans()`. The quosures will also have the attribute `'arg_pos'`, which will indicate their position in the original order in which they were supplied.

Usage

```
args_and_kwargs(..., .already_quosure = FALSE)
```

Arguments

`...`	Any mix of named and unnamed arguments
`.already_quosure`	if the arguments are already all quosures (in which case it will just sort them by named vs. unnamed arguments)

Value

A named list of lists, with $args being a list of quosures of the unnamed arguments and $kwargs being a list of quosures of the named arguments.

Note

This function started out in the `zplyr` package.

Examples

```
x <- args_and_kwargs(unnamed_1, named_1="ba", "unnamed_2", named_2 = letters)
print(x$args)
print(x$kwargs)

## Not run:
# Or see the `share_scales` from the `zplyr` package
share_scales <- function(...) {
  akw <- args_and_kwargs(...)
  # Unnamed arguments are ggplot scales
  geom_func_list <- purrr::map(akw$args, rlang::eval_tidy)
  # Named arguments are to be passed into those scales
  geoms <- purrr::map(geom_func_list, ~.(!!!akw$kwargs))
  return(geoms)
}

## End(Not run)
```

`beep_with` *Play short sounds*

Description

If you have the beepr package installed, catchr can use it to play sounds when certain conditions are being handled with `beep_with()`, similar to how `beep` works. But unlike `beep` and most catchr functions or special reserved terms, `beep_with()` is meant to be used as a user-defined function in a plan. It is particularly useful for when you're working with `futures` and busy doing something else while code is running in the background, or when you're working in a different window and want something to grab your attention.

`beep_with` can be used at the "top" level of a plan, since it returns a *function* (which is required custom input for a catchr plan) that will play the beeping sound you've specified.

Usage

```
beep_with(beepr_sound)
```

Arguments

beepr_sound	A character string or number specifying the sound to be played. See the sound argument in `beepr::beep()` documentation.

See Also

the beep special term, which will play the default beep; `user_exit()` and `exit_with()` for parallel functions for the exit special term, and `user_display()` and `display_with()` for parallel functions for the display special term.

Examples

```
warning_in_middle <- function() {
  message("It's time!")
  Sys.sleep(1)
  invisible("done")
}

if (requireNamespace("beepr", quietly = TRUE) == TRUE) {
  catch_expr(warning_in_middle(),
             message = c(beep_with(2), display, muffle))
  # Or you can just use the default sound with "beep":
  # catch_expr(warning_in_middle(), message = c(beep, display, muffle))
}
```

---

catchr-DSL *The language of catchr*

Description

catchr implements a small but helpful "domain-specific language" (DSL) to make building condition-handling functions simpler to read and type. Essentially, catchr reserves special 'terms' that mean something different than they do in the rest of R. When given as part of the input for a catchr plan, these terms will be substituted for special catchr functions used to handle conditions.

These special terms can be inputted as strings (e.g., `warning = list('collect','muffle')`) or as unquoted terms (e.g., `warning = c(collect,muffle)`); catchr internally converts the unquoted terms to strings regardless, but being able to input them unquoted saves keystrokes and can highlight their special meanings for code readability.

Special reserved terms

The following are the special terms and what they do. Note that there are also some special condition names, but those are different from the following.

- `tomessage`, `towarning`, `toerror`: these terms will become functions that will convert captured conditions into a message, warning, or error, respectively, and raise them. The original classes of the condition will be lost.
- `beep`: if the beepr package is installed, this will play a sound via `beepr::beep`.

- `display`: the purpose of this term is to immediately display information about the captured condition on the output terminal without raising additional conditions (as would be done with `tomessage`). Currently, it attempts to display this information with bold, turquoise-blue text if the crayon package is installed. In future versions of catchr, this default styling (and other display options) may be able to be changed by the user.
- `muffle`: this term will be substituted for a function that 'muffles' (i.e., 'suppresses', 'catches', 'hides'—whatever you want to call it) the captured condition, preventing it from being raised to higher levels or subsequent plans. Anything in a plan *after* `muffle` will be ignored, so put it last.
  The function `muffle` is built on, `first_muffle_restart()`, searches for the first available restart with `"muffle"` in its name (the two typical ones are `"muffleMessage"` and `"muffleWarning"`) and calls `invokeRestart` with it. If the captured condition is an error, which can't be muffled, it will exit the evaluation and give `NULL` for the returned value of the evaluated expression.
- `exit`: when encountered, this will exit the evaluation of the expression immediately and by default muffle the captured condition (use `raise` in the plan if to ensure this doesn't happen). Any instructions after `exit` in the input will be ignored, so put it last.
- `collect`: this term will store the captured conditions and append them to the output of the evaluated expression. See the collecting conditions help topic for a full explanation.
- `raise`: this term will raise the captured condition "as is". The only *real* use for this term is when you want to use `exit` to stop the evaluation, but to still raise the condition past that as well (in which case, put `raise` in the plan before `exit`). The behavior of this raising might be slightly unpredictable for very odd edge-cases (e.g., if a condition were both a warning *and* an error).

Masking

catchr will turn unquoted special terms into functions, but what happens if these unquoted terms are identical to variables previously declared?

If `muffle` is the name of a user-defined function, e.g., `muffle <-function(x) print("Wooo!")`, in normal R we would expect `warning = muffle` to make `function(x) print("Wooo!")` the warning handler.

However, catchr's DSL "masks" any symbol that matches one of its reserved terms, and when it evaluates these symbols, they are converted into strings. For the most part, catchr will warn you when this happens.

Importantly, catchr does *not* mask reserved terms when:

- the reserved names are being used as calls, e.g., `warning = collect(foo)`. In these cases, it will attempt to use a previously defined function `collect` on `foo`, and will attempt to use whatever that evaluates to. The reserved terms are all strings/unquoted bare symbols, so it is never a problem anyway.
- the input specifically references a namespace/package, such as `warning = dplyr::collect`. When the symbol of a special terms is preceded by `::` or `:::`, it will be seen as the function of that package, and not as the special term `collect`.
- the reserved terms are used inside a previously defined function. For example, if the user had defined `muffle <-function(x) print("not special")`, and `fn <-function(x) muffle`, using the argument `warning = fn()` would not use the special term of `muffle`.

`catchr_opts` *Pass in catchr-specific options*

Description

catchr offers a number of options for planning condition handling, and the `catchr_opts` function provides a way of passing those options to whatever function is handling the planning. If any argument is left unspecified it defaults to the global defaults (accessible via `catchr_default_opts()`, `base::options()`, or `base::getOption()`).

Usage

```
catchr_opts(
  default_plan = NULL,
  warn_about_terms = NULL,
  bare_if_possible = NULL,
  drop_empty_conds = NULL
)
```

Arguments

`default_plan`	The default plan for unnamed input arguments. See `get_default_plan()` for more details.
`warn_about_terms`	A logical; if `FALSE`, will not warn about masking special terms
`bare_if_possible`	A logical; if `TRUE`, and no conditions are collected, will return the result of the evaluated expression as-is, without encompassing named list.
`drop_empty_conds`	A logical; if `TRUE`, the sublists for conditions that used `collect` but didn't collect anything will be dropped from the list. Otherwise, they will appear as empty sublists.

Catchr options

catchr's options are specified below. The names of the global default option are preceded by `"catchr."` so they don't collide with other packages' options (i.e., `drop_empty_conds` can be accessed via `getOption("catchr.drop_empty_conds")`:

- `default_plan`: The default plan that will be used for unnamed arguments (i.e., conditions specified without plans) to `make_plans()` or the like. See `get_default_plan()` for more details. The original package default is `c("collect","muffle")`.
- `warn_about_terms`: If one of its reserved terms would mask a previously defined variable name when catchr is compiling plans, you can specify whether or not a warning will be generated. The original package default is `TRUE`, which will warn the user of these occurrences.

- `bare_if_possible`: When no plans are set to collect conditions, you have the option of returning the value of the evaluated expression by itself, *without* being the $value element of a list. If `bare_if_possible` is `TRUE` and no plans collect conditions, it will return the value without the wrapping list. If one is using catchr extensively, it might be wise to set this option to `FALSE` so catchr's returned values are always consistent. The original package default is `TRUE`.
- `drop_empty_conds`: If conditions have plans that would collect them but none are raised in the evaluation of an expression, you have the option of dropping their sublists. For example, conditions that aren't warnings, messages, or errors are very rare. If you wanted to return the "misc" condition sublist only when such conditions were raised, you could do this by setting the value to `TRUE`. The original package default is `FALSE`.

See Also

The default catchr options, `set_default_plan()`, `get_default_plan()`

`catch_expr` *Catch conditions*

Description

These are function that actually evaluate expression and "catch" the conditions. `catch_expr()` evaluates an expression, catching and handling the conditions it raises according to whatever catchr plans are specified. `make_catch_fn()` is a function factory that returns a function that behaves like `catch_expr()` with the plans already specified.

Plans can be passed in as output from `make_plans()` or as input that follows the same format as the input to `make_plans()`.

Usage

```
catch_expr(expr, ..., .opts = NULL)

make_catch_fn(..., .opts = NULL)
```

Arguments

`expr`	the expression to be evaluated
`...`	a catchr plan as made by `make_plans()` or input for plans that follows the same format as input to `make_plans()`
`.opts`	The options to be used for the plans (generally passed in using `catchr_opts()`). If the input plans were already made by `make_plans()`, setting this will override whatever options were specified earlier.

Value

For `catch_expr()`: The value of the evaluated expression if there isn't an error and if the plans don't force an exit. If `getOption("catchr.bare_if_possible")` is `FALSE` (or if any conditions have been collect), it will return a named list, with the "value" element containing the value of the evaluated expression and sublists containing any collected conditions.

For `make_catch_fn()` A function that catches conditions for expressions the same way `catch_expr()` would, but with the plans already specified.

Examples

```
warner <- function() {
  warning("Suppress this!")
  "done!"
}

compiled_warning_plans <- make_plans(warning = muffle)
warning_catcher <- make_catch_fn(warning = muffle)
warning_catcher2 <- make_catch_fn(compiled_warning_plans)

# `results` 1-4 are equivalent
results1 <- catch_expr(warner(), warning = muffle)
results2 <- warning_catcher(warner())
results3 <- catch_expr(warner(), compiled_warning_plans)
results4 <- warning_catcher2(warner())
```

`collecting-conditions` *Collect conditions, without halting processes*

Description

One of the most useful aspects of catchr is its ability to catch and 'collect' the conditions (e.g., warnings, errors, messages, etc.) raised by an expression without halting/redoing the evaluation of that expression. This can be particularly useful in a number of scenarios:

- If you are trying to catch the warning messages from code that takes a long time to run, where having to restart the whole process from square one would be too costly.
- If you want to collect warnings, messages, and errors from code that is running remotely, where these conditions would not be returned with the rest of the results, such as with the **future** package.
- If you are running lots of code in parallel and want to log all of the conditions within R, such as in a large-scale power simulation, or with packages such `purrr`.

Using the `collect` term lets you do this. When the plan for a condition uses `collect`, the captured condition will be added to a list of other conditions of that same type. When the expression is done being evaluated, catchr will return a named list, where $value is the output of the expression, and the other named elements are sublists with all their collected conditions. The exact behavior of this process is determined by options in `catchr_opts()`.

See Also

dispense_collected() to raise the collected conditions and return the bare result

Examples

```
one_of_each <- function(with_error) {
  rlang::inform("This is a message")
  rlang::warn("This is a warning")
  if (with_error)
    stop("This is an error", call.=FALSE)
  "return value!"
}

collecting_plans <- make_plans(message, warning, error,
                               .opts = catchr_opts(default_plan = c(collect, muffle),
                                                   drop_empty_conds = FALSE))

# When the evaluation completes, the "value" element is the value the expression returns
no_error <- catch_expr(one_of_each(FALSE), collecting_plans)
no_error$value

# If it doesn't return, the value is generally NULL
with_error <- catch_expr(one_of_each(TRUE), collecting_plans)
with_error$value

# If the option `drop_empty_conds` == TRUE, then
#   sublists without collected condition will be dropped
catch_expr(one_of_each(FALSE), collecting_plans,
           .opts = catchr_opts(drop_empty_conds=TRUE))

# If the option `bare_if_possible` == TRUE, then even
#   functions that don't use `collect` will return the value
#   of the expression as a "value" sublist
catch_expr("DONE", fake_cond = muffle, .opts = catchr_opts(bare_if_possible=FALSE))
```

---

default-catchr-options

Default catchr-specific options

Description

catchr's options for planning condition handling are passed into catchr functions with catchr_opts(), but when an option isn't specified in the call, catchr_opts() uses whatever the default for that option is. You can get and set these global defaults with catchr_default_opts() and do a "factory reset" on them to restore the original package values with restore_catchr_defaults().

Usage

```
catchr_default_opts(...)

restore_catchr_defaults(...)
```

Arguments

...	Default options to get or set. See the Arguments section for details.

Arguments

For `catchr_default_opts()`, unnamed arguments (unquoted terms / strings of the option names) will have their current default values returned, similar to `getOption()`. Named arguments (whose names are option names) will have their default values *set* to whatever their value is. If no arguments are specified, it will return all the current default values.

`restore_catchr_defaults()` only accepts unnamed arguments (unquoted terms / strings of the option names). The options specified will have their default values set to the *original* default package values. Leaving the arguments empty will result in *all* the option defaults being reset to their original values.

See Also

`catchr_opts()` for what the options mean; `get_default_plan()` and `set_default_plan()`, which are equivalent to `catchr_default_opts(default_plan)` and `catchr_default_opts(default_plan = ...)`, respectively.

`dispense_collected` *Return the value after raising all collected conditions*

Description

This function takes in the collected conditions list that is the output of `catch_expr()` or a function from `make_catch_fn()`, raises all the conditions that were collected, and then returns the value the original evaluated expression had returned. This might be useful in situations in which one had collected conditions from a remote evaluation of an expression, and wishes to raise the conditions locally.

The way the errors are treated can be changed as well: they can either be raised as-is, displayed on screen, or raised as warnings.

Usage

```
dispense_collected(l, treat_errs = c("raise", "display", "warn"))
```

Arguments

l	The results of `catch_expr()` or a function from `make_catch_fn()`
treat_errs	One of three strings governing how errors are treated: `"raise"` which will simply raise errors as they are, `"display"` which will just print the error messages on-screen, and `"warn"` which will raise the errors as warnings.

---

extract_display_string

Make a string to display from a condition

Description

Turn a condition into a string comprised of its message, name, and call, in a variety of configurations.

Usage

```
extract_display_string(cond, cond_name = NA, include_call = T)
```

Arguments

cond	A condition to display or turn into a string
cond_name	Either the name of the condition you want to display, `NA` if you want the condition name to be assigned by default (the first class of the condition), or `NULL` if you don't want the condition type displayed at all.
include_call	A logical; if `FALSE` the call won't be included in the string even if present in the condition.

Value

A string

---

first_muffle_restart *Find the first 'mufflable' restart*

Description

This function attempts to return the first available restart with the string "muffle" in its name. If the condition is an error, it will attempt to find the first restart named "return_error" (used internally in catchr to return a `NULL` value). If the condition is an "interrupt", it will attempt to find the first restart named "resume". If no such restarts can be found, it returns `NULL`.

Usage

```
first_muffle_restart(cond)
```

Arguments

cond	A condition

Value

A restart or NULL if none can be found.

---

give_newline *Make a string end with a newline character*

Description

give_newline will append a line return ('\n') to the end of a string if it doesn't already end with one. There is also the option to remove any trailing whitespace before doing so.

Usage

```
give_newline(s, trim = FALSE)
```

Arguments

s	A string.
trim	Indicates whether to remove trailing whitespace before adding newline.

---

has_handler_args *Make sure a function can be a handler*

Description

This makes sure that a given function doesn't *require* more than one argument to be passed into it, and takes in at least one argument (which is what a handler needs).

Usage

```
has_handler_args(fn)
```

Arguments

fn	A function that is a candidate for being a handler

is_catchr_plan *Check if list is a catchr plan*

Description

Currently, this function just checks whether a list was made via `make_plans()`. It does so simply by looking at the class of the list.

In the future, catchr plans may become more complicated and this will become a more useful part of the API.

Usage

```
is_catchr_plan(x)
```

Arguments

`x`	An object to test

make_plans *Making catchr plans*

Description

Customizing how conditions are handled in catchr is done by giving catchr 'plans' for when it encounters particular conditions. These plans are essentially just lists of functions that are called in order, and that take in the particular condition as an argument.

However, since catchr evaluates things slightly differently than base R, the user input to make these plans has to first be passed into `make_plans` (or, for setting the default plan, `set_default_plan()`). `make_plans` also lets users specify options for how they want these plans to be evaluated with the `.opts` argument (see `catchr_opts()` for more details).

See the 'Input' section below and the examples for how to use `make_plans`.

Usage

```
make_plans(..., .opts = catchr_opts())
```

Arguments

`...`	Named and unnamed arguments for making plans. See 'Input' for more detail.
`.opts`	The options to be used for the plan. Generally passed in using `catchr_opts()`.

Input

User input to make_plans is very similar to how one makes handlers for base::withCallingHandlers(), base::tryCatch() and rlang's rlang::with_handlers(), albeit with some important differences.

Like the functions above, the name of each argument determines which type of condition it will be the plan for. Hence, warnings = fn will apply the fn function to the warnings raised in evaluating expr.

However, *unnamed* arguments are *also* accepted: the value of any unnamed arguments will be treated as the type of a condition, which will then have the default plan assigned to it, as specified either in .opts = catchr_opts(...) or via getOption("catchr.default_plan"). Unnamed arguments must be either strings or unquoted expressions which will then be converted to strings. Currently, unnamed arguments are *never* evaluated, so cannot be calls that evaluate to strings.

However, this may change in future versions of catchr.

Passing input in programmatically

make_plans supports quasiquotation, so if for some reason one wishes to pass input into make_plans via a different function, programmatically, etc., one may do so by splicing in quosures. See below for examples.

Examples

```
# ### INPUT EXAMPLES ############################

# Named arguments -------------------------------------

#   * single functions:
p <- make_plans(warning = str, message = function(x) print(x))

#   * single unquoted expressions and strings
#     (must match catchr's special reserved terms, e.g., 'muffle', 'exit', etc.):
p <- make_plans(message = muffle, condition = "collect")

#   * lists or vectors of any combinatin of the above:
p <- make_plans(error = list(collect, "exit"),
                message = c(cat, "muffle"))

#   * anything that evaluates to the above:
fn <- function() { list(cat, "muffle") }
p <- make_plans(message = fn() )

# Unnamed arguments ----------------------

#   * single strings:
p <- make_plans("warning","condition")

#   * unquoted expressions:
p <- make_plans(warning,condition)

#   * Currently, does NOT accept anything that evaluates to strings:
```

```
#       (However, this may change in the future)
## Not run:
string_fn <- function() { "condition" }
make_plans(string_fn()) # will currently raise error

## End(Not run)

# Mixes of both --------------------------
p <- make_plans("warning", message = c(towarning, muffle),
                condition = print)

# ### Quasiquotation and splicing in the arguments ###############

q <- rlang::quo(function(cond) {print(cond)})
name <- "warning"

print_plan <- make_plans(!!name := !!q)

# 'message' will be assigned the default plan
qs <- rlang::quos(warning = muffle, error = exit, message)
random_plan <- make_plans(!!!qs)
```

`print.catchr_compiled_plans` *View and print 'compiled' catchr plans*

Description

'Compiled' catchr plans returned by `make_plans()` look very ugly "naked". These functions make plans understandable at a single glance.

Usage

```
## S3 method for class 'catchr_compiled_plans'
print(
  x,
  ...,
  show_opts = FALSE,
  total_len = getOption("width"),
  show_full = FALSE
)

## S3 method for class 'catchr_compiled_plans'
summary(object, ...)
```

Arguments

x	The "compiled" plans, i.e., from make_plans()
...	Currently unused.
show_opts	A logical; if TRUE, prints the catchr options set for the plans.
total_len	An integer; sets the total number of characters each line can span before being cut off with "..."
show_full	A logical; if TRUE, will print out the full length of each line.
object	The "compiled" plans, i.e., from make_plans()

---

reserved-conditions *Special condition names*

Description

In addition to having reserved terms for use in making condition-handling plans, catchr also places special meaning on two types of conditions, misc and last_stop. The misc is very useful, last_stop is something most users should probably avoid.

The misc condition

The names of the named arguments passed to make_plans correspond to the type of conditions each plan is designed for—specifically, if any of a condition's classes match a plan's name, it will be caught by that plan. By default, all conditions have a class of "condition".

There is nothing special about a condition with a class of "misc" in base R, although there are no normal base R functions that would automatically raise such a condition. However, in catchr, using the name misc for a plan means that this plan will be applied to any condition that does *not* already have a plan specified for it. Consider the following example:

```
plans <-make_plans(warning = collect,message = collect,error = exit,misc = collect)
```

These plans will collect every non-error condition into three sublists, one for warnings, one for messages, and one for everything else—"misc". If one used condition = collect instead of misc, warnings and messages would be collected twice: once in each of their respective sublists, and another time in "condition", since each type also has that class. misc will *not* catch warnings or messages in the scenario above.

Since ~99\

The last_stop condition

This condition name is reserved for exit, user_exit() and exit_with(). There is basically zero chance any code other than catchr will ever raise a condition of "last_stop", so this shouldn't be a problem, but until catchr becomes more mature, do not use this name for any condition or plan.

set_default_plan *Get/set the input for the default catchr plan*

Description

These functions allow the user to set and retrieve the input that will be assigned to any conditions passed to `make_plans()` without plans (i.e., as unnamed arguments). Using the same inputting style as `make_plans()`, the argument `new_plan` will essentially be treated the same way a single named argument would be in `make_plans()`, without actually having a name/specific condition.

Usage

```
set_default_plan(new_plan)

get_default_plan()
```

Arguments

`new_plan`	The input (in the style of named arguments to `make_plans()`) that will become the input of default plan.

Value

`set_default_plan()` will invisibly return a "cleaned up" version of the input (i.e., evaluated, and with the unquoted terms replaced with strings), which is what will also be returned by `get_default_plan()` until a new default is set.

See Also

default catchr options

user_display *Display conditions in output terminal*

Description

These functions make a catchr plan immediately print the condition to the output terminal, similar to how `display` works. But unlike `display` and most catchr functions or special reserved terms, these functions are meant to be used in user-defined functions of a plan.

`user_display()` immediately displays a condition n the output terminal, and if crayon is installed, will style the text with whatever `crayon` styling is supplied (either as a `crayon` function or a character vector for `crayon::combine_styles()`). This function should be used *within* a custom function, i.e., `function(x) {user_display(x,"pink")}`, and not called by itself, since when it is called, it doesn't evaluate to a function, string or unquoted term, which are required input types to a catchr plan.

`display_with` can be used at the "top" level of a plan, since it returns a *function* that calls `user_display()`. Thus `user_display("pink")` is equivalent to the example above.

Usage

```
user_display(cond, crayon_style, ...)

display_with(crayon_style, ...)
```

Arguments

cond	A condition one wishes to display
crayon_style	If **crayon** is installed, this can be either a `crayon style` (e.g., `crayon::green()`, `blue$bold`, etc.) or a character vector for `crayon::combine_styles()`. These styles will be applied to the output.
...	Parameters to pass into `extract_display_string()`; namely `cond_name` (which controls how the condition is introduced) and `include_call`, which determines whether the call is included.

See Also

the display special term, which essentially uses a version of `user_display`; `user_exit()` and `exit_with()` for parallel functions for the exit special term, and `beep_with()` for a parallel function for the beep special term.

Examples

```
make_warnings <- function() {
  warning("This warning has a call")
  warning("This warning does not", call. = FALSE)
  invisible("done")
}

# The crayon stylings won't work if `crayon` isn't installed.
catch_expr(make_warnings(), warning = c(display_with("pink"), muffle))
catch_expr(make_warnings(),
           warning = c(display_with(c("pink","bold"), include_call = FALSE), muffle))
catch_expr(make_warnings(), warning = c(display_with("inverse", cond_name=NULL), muffle))
# If you don't want to use crayon styles, just use `NULL`
catch_expr(make_warnings(), warning = c(display_with(NULL, cond_name="Warning"), muffle))

# You can get a lot of weird crayon styles
if (requireNamespace("crayon", quietly = TRUE) == TRUE) {
  freaky_colors <- crayon::strikethrough$yellow$bgBlue$bold$blurred
  catch_expr(make_warnings(),
             warning = c(function(x) user_display(x, freaky_colors), muffle))
}
```

user_exit *Force an exit*

Description

These functions force a catchr plan to immediately exit the evaluation of an expression (and the rest of the plan), similar to how `exit` works. But unlike `exit` and most catchr functions or special reserved terms, these functions are meant to be used in the user-defined functions of a plan.

`user_exit()` forces the code to exit, and after exiting, evaluate whatever expression was supplied. This function should be used *within* a custom function, i.e., `function(x) {user_exit(print("DONE!"))}`.

`exit_with()` can be used at the "top" level of a plan, since it returns a *function* that calls `user_exit()`. Thus `exit_with(print("DONE!"))` is equivalent to the example above. Additionally, if `as_fn` is set to `TRUE`, it will attempt to coerce `expr` into a function via `rlang`'s `rlang::as_function()`. If `expr` can be converted, `exit_with()` will return a function that takes in a condition, modifies it via `expr`, and then supplies this to `user_exit`. E.g., `exit_with(~.$message)` is equivalent to `function(cond) {user_exit(cond$message)}`

Usage

```
user_exit(expr = NULL)

exit_with(expr, as_fn = FALSE)
```

Arguments

`expr`	An optional expression which if specified, will be evaluated after `user_exit` exits the evaluation.
`as_fn`	A logical; if `TRUE`, catchr will try to convert `expr` into a function via `rlang::as_function()` which will be applied to the condition. It will fall back to normal behavior if this coercion raises an error.

See Also

the exit special term, which essentially becomes `exit_with(NULL)`; `user_display()` and `display_with()` for parallel functions for the display special term, and `beep_with()` for a parallel function for the beep special term..

Examples

```
yay <- catch_expr({warning("oops"); "done!"},
                  warning = exit_with("YAY"))

# This won't work, since `user_exit("YAY")` doesn't evaluate to a function/string
## Not run:
yay <- catch_expr({warning("oops"); "done!"},
                  warning = user_exit("YAY"))
```

```
## End(Not run)

check <- function(cond) {
  if (inherits(cond, "simpleWarning"))
    user_exit(rlang::warn(paste0("Check it: ", cond$message)))
  else
    invokeRestart(first_muffle_restart(cond))
  NULL
}

result <- catch_expr(
  { rlang::warn("This will be muffled")
    warning("This won't be muffled") },
  warning = check)
# Notice that `result` takes whatever the last (invisibly)
#   returned value is. Here, that's the message from the warning
result

# If you don't want to accidentally assign what is returned by `user_exit`,
#   either add `NULL` to the end of the expresion:
result2 <- catch_expr(
  { rlang::warn("This will be muffled")
    warning("This won't be muffled")},
  warning = function(x) { user_exit({ warning("This won't be assigned"); NULL})})
result2

# Or you can just do the assignment _within_ the expression being evaluated:
result3 <- NULL
catch_expr({result3 <- {
    rlang::warn("This will be muffled")
    warning("This won't be muffled")}},
  warning = check)
result3
```

Index