Package: ggtrace (via r-universe)

October 18, 2024

Type Package

Title Programmatically explore, debug, and manipulate ggplot internals

Version 0.7.3

Description Programmatically explore, debug, and manipulate ggplot internals. Package ggtrace offers a low-level interface that extends base R capabilities of trace, as well as a family of workflow functions that make interactions with ggplot internals more accessible.

URL https://yjunechoe.github.io/ggtrace, https://github.com/yjunechoe/ggtrace

BugReports https://github.com/yjunechoe/ggtrace/issues

Depends R (>= 3.3.0)

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Collate 'utils.R' 'helpers.R' 'get_method.R' 'tracedump.R' 'ggedit.R' 'gguntrace.R' 'ggtrace.R' 'one-offs.R' 'with_ggtrace.R' 'topic-tracing-context.R' 'workflows-utils.R' 'workflows-inspect.R' 'workflows-capture.R' 'workflows-highjack.R' 'aliases.R' 'last-errorcontext.R' 'sublayer-data.R' 'zzz.R'

Imports cli, rlang (>= 1.0.0)

Suggests ggplot2 (>= 3.4.0), dplyr, grid, ggforce, ggh4x, patchwork, R6, rmarkdown, knitr, pkgdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

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

RemoteUrl https://github.com/yjunechoe/ggtrace

RemoteRef HEAD

RemoteSha 5b73d39698b932137e4bc039026e05e4c261aeba

Contents

`get_method` *Get information about ggproto methods*

Description

Get information about ggproto methods

Usage

```
get_method(method, inherit = FALSE)

get_method_inheritance(obj, trim_overriden = TRUE)

ggbody(method, inherit = FALSE, as.list = TRUE)

ggformals(method, inherit = FALSE)
```

Arguments

`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`

`inherit`	Whether the method should be searched from its closest parent. Defaults to `FALSE`. If `TRUE`, returns the parent's method and the corresponding `ggbody()` code as a message.
`obj`	A ggproto object
`trim_overriden`	Whether `get_method_inheritance` should recursively hide methods defined by a parent.
`as.list`	Whether `ggbody()` should return the body of the method as a list. Defaults to `TRUE`.

Details

- `get_method()` returns the method.
- `get_method_inheritance()` lists available methods from self and parent ggprotos.
- `ggbody()` returns the body of the method.
- `ggformals()` returns the formals of the method.

Value

A list

Gotchas

- If a method is being traced via `ggtrace()` or `ggedit()`, `get_method()` will return the current *modified state* of the method. As of v0.3.5, calling `get_method()` on a method that has a trace on it will return a warning to emphasize this fact.
- When using `inherit = TRUE`, make sure that all ggproto objects from `class(ggproto)` are available (by loading the packages where they are defined, for example). Under the hood, `get_method()` loops through the parents to search for the method, so it needs to be able to evaluate each element of `class(ggproto)` as an object.

Note

`get_method()` calls `get("method", ggproto)` under the hood. The `get("method", ggproto)` syntax is the long form of `ggproto$method` which retrieves the actual function body. This is a subtle but important difference for inspecting ggproto methods.

- For example, this works: `debugonce(get("compute_group", StatCount))`
- But this fails to insert a break point: `debugonce(StatCount$compute_group)`

`get_method()` was designed so that you do not have to worry about this distinction.

Examples

```
library(ggplot2)

# Uninformative
StatCount$compute_group
formals(StatCount$compute_group)
```

```
body(StatCount$compute_group)

# Errors
# get(StatCount$compute_group)

# Informative
get_method(StatCount$compute_group)
ggformals(StatCount$compute_group) # formals(get_method(StatCount$compute_group))
ggbody(StatCount$compute_group)    # body(get_method(StatCount$compute_group))

# Works for ggproto in extension packages

ggbody(ggforce::StatDelaunaySegment$compute_group)

library(ggforce)
ggbody(StatBezier$compute_panel)

# `inherit = TRUE` will return the method from the closest parent

## ERRORS:
## get_method(StatBoxplot$compute_panel)
## ggbody(StatBoxplot$compute_panel)
## ggformals(StatBoxplot$compute_panel)
ggbody(StatBoxplot$compute_panel, inherit = TRUE)
ggbody(Stat$compute_panel)

# Navigating complex inheritance
class(GeomArcBar)
invisible(ggbody(GeomArcBar$default_aes, inherit = TRUE)) # self
invisible(ggbody(GeomArcBar$draw_panel, inherit = TRUE))  # parent
invisible(ggbody(GeomArcBar$draw_key, inherit = TRUE))    # grandparent
invisible(ggbody(GeomArcBar$draw_group, inherit = TRUE))  # top-level

# Getting information about method inheritance all at once
# - default `trim_overriden = TRUE` hides redundant methods defined in parent
get_method_inheritance(GeomArcBar, trim_overriden = TRUE)

# Works for custom ggproto
# - Example from {ggplot2} "Extending ggplot2" vignette
StatDensityCommon <- ggproto("StatDensityCommon", Stat,
  required_aes = "x",

  setup_params = function(data, params) {
    if (!is.null(params$bandwidth))
      return(params)

    xs <- split(data$x, data$group)
    bws <- vapply(xs, bw.nrd0, numeric(1))
    bw <- mean(bws)
    message("Picking bandwidth of ", signif(bw, 3))

    params$bandwidth <- bw
    params
  },

  compute_group = function(data, scales, bandwidth = 1) {
    d <- density(data$x, bw = bandwidth)
    data.frame(x = d$x, y = d$y)
  }
)

as.list(body(get("compute_group", StatDensityCommon)))

ggbody(StatDensityCommon$compute_group)

# As of v.0.4.0, ggbody works for functions as well
ggbody(sample)
ggtrace(sample, 1)
invisible(ggbody(sample))
is_traced(sample)
gguntrace(sample)
```

---

ggdebug *Debug a ggproto method*

Description

Debug a ggproto method

Usage

```
ggdebug(method, ...)

ggdebugonce(method, ...)

ggundebug(method, ...)
```

Arguments

`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`...`	Ignored. Designed for the ease of calling this function by modifying the call to an earlier `{ggtrace}` function in interactive contexts.

ggedit — *Interactively edit a masking copy of the source code*

Description

Interactively edit a masking copy of the source code

Usage

```
ggedit(method, remove_trace = FALSE, ...)
```

Arguments

`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`remove_trace`	Whether to edit from a clean slate. Defaults to `FALSE`.
`...`	Unused, for extensibility.

Details

Like `base::trace()`, the edit is in effect until `gguntrace()` is called. Changes with `ggedit()` are cumulative, so `ggedit()` will inform you via a warning if you're making an edit on top of an existing edit. Call `gguntrace()` on the object first if you'd like to edit the method's original unaltered source code.

Only works in interactive contexts.

Gotchas

- Calling `ggtrace()` on an method that that has changes from `ggedit()` will remove the changes from `ggedit()`. It *is* possible to combine both features, but disabled in the package to keep the API consistent. It is against the philosophy of `{ggtrace}` to mix programmatic and interactive workflows.

See Also

`gguntrace()`, `is_traced()`

Examples

```
## Not run:

jitter_plot <- ggplot(diamonds[1:1000,], aes(cut, depth)) +
  geom_point(position = position_jitter(width = 0.2, seed = 2021))

# Interactively modify the method's source code in text editor
ggedit(PositionJitter$compute_layer)

# Check the edited code
ggbody(PositionJitter$compute_layer)

# Execute method with edit
jitter_plot

# Untrace
gguntrace(PositionJitter$compute_layer)

# Edit is removed in the next call
jitter_plot


## End(Not run)
```

`ggtrace` *Insert traces for delayed evaluation*

Description

Insert traces for delayed evaluation

Usage

```
ggtrace(
  method,
  trace_steps,
  trace_exprs,
  once = TRUE,
  use_names = TRUE,
  ...,
  print_output = TRUE,
  verbose = FALSE
)
```

Arguments

`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms:

	• `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`trace_steps`	A sorted numeric vector of positions in the method's body to trace. Negative indices reference steps from the last, where `-1` references the last step in the body. Special value `"all"` traces all steps of the method body.
`trace_exprs`	A list of expressions to evaluate at each position specified in `trace_steps`. If a single expression is provided, it is recycled to match the length of `trace_steps`. To simply run a step and return its output, you can use the `~step` keyword. If the step is an assign expression, the value of the assigned variable is returned. If `trace_exprs` is not provided, `ggtrace()` is called with `~step` by default.
`once`	Whether to `untrace()` the method on exit. If `FALSE`, creates a persistent trace which is active until `gguntrace()` is called on the method. Defaults to `TRUE`.
`use_names`	Whether the trace dump should use the names from `trace_exprs`. If `trace_exprs` is not specified, whether to use the method steps as names. Defaults to `TRUE`.
`...`	Unused, for extensibility.
`print_output`	Whether to `print()` the output of each expression to the console. Defaults to `TRUE`.
`verbose`	Whether logs should be printed when trace is triggered. Encompasses `print_output`, meaning that `verbose = FALSE` also triggers the effect of `print_output = FALSE` by consequence. Defaults to `FALSE`.

Details

`ggtrace()` is a wrapper around `base::trace()` which is called on the ggproto method. It calls `base::untrace()` on itself on exit by default, to make its effect ephemeral like `base::debugonce()`. A major feature is the ability to pass multiple positions and expressions to `trace_steps` and `trace_exprs` to inspect, capture, and modify the run time environment of ggproto methods. It is recommended to consult the output of `ggbody()` when deciding which expressions to evaluate at which steps.

The output of the expressions passed to `trace_exprs` is printed while tracing takes place. The list of outputs from `ggtrace()` ("trace dumps") can be returned for further inspection with `last_ggtrace()` or `global_ggtrace()`.

Workflows

Broadly, there are four flavors of working with the `{ggtrace}` package, listed in the order of increasing complexity:

- **Inspect**: The canonical use of `ggtrace()` to make queries, where expressions are passed in and their evaluated output are returned, potentially for further inspection.
- **Capture**: The strategy of returning the method's runtime environment for more complex explorations outside of the debugging context. A method's environment contextualizes the `self` object in addition to making all inherited params and local variables available.

  A reference to the method's runtime environment can be returned with `environment()`, as in `trace_exprs = quote(environment())`. Note that environments are mutable, meaning that environment() returned from the first and last steps will reference the same environment. To get a snapshot of the environment at a particular step, you can return a deep copy with rlang::env_clone(environment()).

- **Inject**: The strategy of modifying the behavior of a method as it runs by passing in expressions that make assignments.

  For example, trace_steps = c(1, 10) with trace_exprs = rlang::exprs(a <- 5, a) will first assign a new variable a at step 1, and return its value 5 at step 10. This can also be used to modify important variables like quote(data$x <- data$x * 10). If you would like to inject an object from the global environment, you can make use of the !! (bang-bang) operator from {rlang}, like so: rlang::expr(data <- !!modified_data).

  Note that the execution environment is created anew each time the method is ran, so modifying the environment from its previous execution will not affect future calls to the method.

  If you would like to capture the modified plot output and assign it to a variable, you can do so with ggplotGrob(). You can then render the modified plot with print().

- **Edit**: It is also possible to make any arbitrary modifications to the method's source code, which stays in effect until the method is untraced. While this is also handled with base::trace(), this workflow is fundamentally interactive. Therefore, it has been refactored as its own function ggedit(). See ?ggedit for more details.

Gotchas

- If you wrap a ggplot in invisible() to silence ggtrace(), the plot will not build, which also means that the tracing is not triggered. This is because the print/plot method of ggplot is what triggers the evaluation of the plot code. It is recommended to allow ggtrace() to print information, but if you'd really like to silence it, you can do so by wrapping the plot in a function that forces its evaluation first, like ggplotGrob, as in invisible(ggplotGrob(<plot>)).
- If for any reason ggtrace(once = TRUE) fails to untrace itself on exit, you may accidentally trigger the trace again. To check if a method is being traced, call is_traced(). You can also always call gguntrace() since unlike base::untrace(), it will not error if a trace doesn't exist on the method. Instead, gguntrace() will do nothing in that case and simply inform you that there is no trace to remove.
- Because base::trace() wraps the method body in a special environment, it is not possible to inspect the method/function which called it, even with something like rlang::caller_env(). You will traverse through a few wrapping environments created by base::trace() which eventually ends up looping around.

Messages

Various information is sent to the console whenever a trace is triggered. You can control what gets displayed with print_output and verbose, which are both TRUE by default. print_output simply calls print() on the evaluated expressions, and turning this off may be desirable if expressions in trace_exprs evaluates to a long dataframe or vector. verbose controls all information printed to the console including those by print(), and setting verbose = FALSE will mean that only message()s will be displayed. Lastly, you can suppress message() from ggtrace() with options(ggtrace.suppressMessages = TRUE), though suppressing messages is generally not recommended.

See Also

gguntrace(), is_traced(), last_ggtrace(), global_ggtrace()

Examples

```
# One example of an Inspect workflow ----

library(ggplot2)

jitter_plot <- ggplot(diamonds[1:1000,], aes(cut, depth)) +
  geom_point(position = position_jitter(width = 0.2, seed = 2021))

jitter_plot

ggbody(PositionJitter$compute_layer)

## Step 1 ====
## Inspect what `data` look like at the start of the function
ggtrace(PositionJitter$compute_layer, trace_steps = 1, trace_exprs = quote(head(data)))

jitter_plot

## Step 2 ====
## What does `data` look like at the end of the method? Unfortunately, `trace()` only lets us enter
## at the beginning of a step, so we can't inspect what happens after the last step is evaluated. To
## address this, `ggtrace()` offers a `~step` keyword which gets substituted for the current line.
## We also set `print_output = FALSE` to disable printing of the output
ggtrace(
  PositionJitter$compute_layer,
  trace_steps = 14,
  trace_exprs = quote(~step), # This is the default if `trace_exprs` is not provided
  print_output = FALSE
)

# We wrap the plot in `ggplotGrob()` and `invisible()` to force
# its evaluation while suppressing its rendering
invisible(ggplotGrob(jitter_plot))

# The output of the evaluated expressions can be inspected with `last_ggtrace()`
head(last_ggtrace()[[1]])

## Step 3 ====
## If we want both to be returned at the same time for an easier comparison, we can pass in a
## (named) list of expressions.
ggtrace(
  PositionJitter$compute_layer,
  trace_steps = c(1, 14),
  trace_exprs = rlang::exprs(
    before_jitter = data,
    after_jitter = ~step
  ),
  verbose = FALSE
)

invisible(ggplotGrob(jitter_plot))

## Step 4 ====
## The output of the evaluated expressions can be inspected with `last_ggtrace()`
jitter_tracedump <- last_ggtrace()

lapply(jitter_tracedump, head, 3)

jitter_distances <- jitter_tracedump[["before_jitter"]]$x -
  jitter_tracedump[["after_jitter"]]$x

range(jitter_distances)
jitter_plot$layers[[1]]$position$width
```

`ggtrace_capture_env` *Capture a snapshot of a method's execution environment*

Description

Capture a snapshot of a method's execution environment

Usage

```
ggtrace_capture_env(x, method, cond = 1L, at = -1L, ...)

capture_env(x, method, cond = 1L, at = -1L, ...)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`cond`	When the method environment should be captured. Defaults to `1L`.
`at`	Which step of the method body the environment should be captured. See `ggbody()` for a list of expressions/steps in the method body.
`...`	Unused.

Value

An environment

Tracing context

When quoted expressions are passed to the cond or value argument of workflow functions they are evaluated in a special environment which we call the "tracing context".

The tracing context is "data-masked" (see `rlang::eval_tidy()`), and exposes an internal variable called `._counter_` which increments every time a function/method has been called by the ggplot object supplied to the x argument of workflow functions. For example, cond = `quote(._counter_ == 1L)` is evaluated as `TRUE` when the method is called for the first time. The cond argument also supports numeric shorthands like cond = `1L` which evaluates to `quote(._counter_ == 1L)`, and this is the default value of cond for all workflow functions that only return one value (e.g., `ggtrace_capture_fn()`). It is recommended to consult the output of `ggtrace_inspect_n()` and `ggtrace_inspect_which()` to construct expressions that condition on `._counter_`.

For highjack functions like `ggtrace_highjack_return()`, the value about to be returned by the function/method can be accessed with `returnValue()` in the `value` argument. By default, `value` is set to `quote(returnValue())` which simply evaluates to the return value, but directly computing on `returnValue()` to derive a different return value for the function/method is also possible.

Examples

```
library(ggplot2)

# Example from https://ggplot2.tidyverse.org/reference/aes_eval.html
after_scale_plot <- ggplot(mpg, aes(class, hwy)) +
  geom_boxplot(aes(colour = class, fill = after_scale(alpha(colour, 0.4))))
after_scale_plot

# `after_scale()` is resolved by `Geom$use_defaults` (at Step 6)

before_applying <- ggtrace_capture_env(
  x = after_scale_plot,
  method = Geom$use_defaults,
  at = 1  # To be more specific, do `at = 6`
)
after_applying <- ggtrace_capture_env(
  x = after_scale_plot,
  method = Geom$use_defaults,
  at = -1  # To be more specific, do `at = 7`
)

colnames(before_applying$data)
colnames(after_applying$data)

library(dplyr)

before_applying$data %>%
  select(any_of(c("colour", "fill")))
after_applying$data %>%
  select(any_of(c("colour", "fill")))

identical(
  before_applying$data %>%
    select(any_of(c("colour", "fill"))) %>%
    mutate(fill = alpha(colour, 0.4)),       #< after_scale() logic here
  after_applying$data %>%
    select(any_of(c("colour", "fill")))
)


# Using the captured environment for further evaluation
ggbody(Geom$draw_panel)

by_group_drawing_code <- rlang::call_args(ggbody(Geom$draw_panel)[[3]])[[2]]
by_group_drawing_code

draw_panel_env <- ggtrace_capture_env(
  x = after_scale_plot,
  method = Geom$draw_panel
)
draw_panel_env

boxes <- eval(by_group_drawing_code, draw_panel_env)

library(grid)
grid.newpage()
grid.draw(editGrob(boxes[[1]], vp = viewport()))
```

ggtrace_capture_fn   *Capture a snapshot of a method as a pre-filled function*

Description

Returns a ggproto method as a function with arguments pre-filled to their values when it was first called

Usage

```
ggtrace_capture_fn(x, method, cond = 1L, ...)

capture_fn(x, method, cond = 1L, ...)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`cond`	When the method function should be captured. Defaults to `1L`.
`...`	Unused.

Value

A function

Tracing context

When quoted expressions are passed to the `cond` or `value` argument of workflow functions they are evaluated in a special environment which we call the "tracing context".

The tracing context is "data-masked" (see `rlang::eval_tidy()`), and exposes an internal variable called `._counter_` which increments every time a function/method has been called by the ggplot object supplied to the `x` argument of workflow functions. For example, `cond = quote(._counter_ == 1L)` is evaluated as `TRUE` when the method is called for the first time. The `cond` argument also supports numeric shorthands like `cond = 1L` which evaluates to `quote(._counter_ == 1L)`, and this is the default value of `cond` for all workflow functions that only return one value (e.g., `ggtrace_capture_fn()`). It is recommended to consult the output of `ggtrace_inspect_n()` and `ggtrace_inspect_which()` to construct expressions that condition on `._counter_`.

For highjack functions like `ggtrace_highjack_return()`, the value about to be returned by the function/method can be accessed with `returnValue()` in the `value` argument. By default, `value` is set to `quote(returnValue())` which simply evaluates to the return value, but directly computing on `returnValue()` to derive a different return value for the function/method is also possible.

Note

For functions and methods that take `...`, arguments passed to `...` are captured and promoted to function arguments. The captured values are available for inspection via `formals()`.

Examples

```
library(ggplot2)

set.seed(47)
df <- as.data.frame(matrix(sample(5, 1000, TRUE), ncol = 2))
table(df)

base <- ggplot(df, aes(x = V1, y = V2))

p1 <- base + stat_summary(orientation = "x")
p1

p1_compute_panel <- ggtrace_capture_fn(p1, method = StatSummary$compute_panel)

# `p1_compute_panel` is a copy of the ggproto method
body(p1_compute_panel)
ggbody(StatSummary$compute_panel, as.list = FALSE)

# Its arguments are pre-filled (captured at runtime)
sapply(formals(p1_compute_panel), class)

# Runs as it should
p1_compute_panel()
```

```
# You can inspect changes to its behavior outisde of ggplot
# For example, see what happens when aes is flipped via `orientation = "y"`
p1_compute_panel(flipped_aes = TRUE)

# We confirm this output to be true when `orientation = "y"` in `stat_summary()`
p2 <- base + stat_summary(orientation = "y")
p2_compute_panel <- ggtrace_capture_fn(p2, method = StatSummary$compute_panel)

identical(p1_compute_panel(flipped_aes = TRUE), p2_compute_panel())

# You can interactively explore with `debugonce(p2_compute_panel)`


# Note that the captured method looks slightly different if the method takes `...`
p3 <- base + stat_smooth() + geom_jitter()
p3

p3_compute_panel <- ggtrace_capture_fn(p3, method = Stat$compute_panel)

# For one, the body is different - it's a "wrapper" around the captured method
body(p3_compute_panel)

# The captured method is stored in the `"inner"` attribute
attr(p3_compute_panel, "inner")

# Captured argument defaults are again available for inspection via `formals()`
# Note that arguments passed to the `...` are promoted to function arguments
names(ggformals(Stat$compute_panel))
names(formals(p3_compute_panel))

# It works the same otherwise - plus you get the benefit of autocomplete
head(p3_compute_panel())
head(p3_compute_panel(level = .99)[, c("ymin", "ymax")])
head(p3_compute_panel(flipped_aes = TRUE))

# Interactively explore with `debugonce(attr(p3_compute_panel, "inner"))`
```

`ggtrace_highjack_args` *Highjack a method's execution and modify its argument values*

Description

Highjack a method's execution and modify its argument values

Usage

```
ggtrace_highjack_args(x, method, cond = 1L, values, ..., draw = TRUE)

highjack_args(x, method, cond = 1L, values, ..., draw = TRUE)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`cond`	When the return value should be replaced. Defaults to `1L`.
`values`	A named list of variable-value pairings. When values are expressions, they are evaluated in the formals.
`...`	Unused.
`draw`	Whether to draw the modified graphical output from evaluating `x`. Defaults to `TRUE`.

Value

A gtable object with class <`ggtrace_highjacked`>

Tracing context

When quoted expressions are passed to the `cond` or `value` argument of workflow functions they are evaluated in a special environment which we call the "tracing context".

The tracing context is "data-masked" (see `rlang::eval_tidy()`), and exposes an internal variable called `._counter_` which increments every time a function/method has been called by the ggplot object supplied to the `x` argument of workflow functions. For example, `cond = quote(._counter_ == 1L)` is evaluated as `TRUE` when the method is called for the first time. The `cond` argument also supports numeric shorthands like `cond = 1L` which evaluates to `quote(._counter_ == 1L)`, and this is the default value of `cond` for all workflow functions that only return one value (e.g., `ggtrace_capture_fn()`). It is recommended to consult the output of `ggtrace_inspect_n()` and `ggtrace_inspect_which()` to construct expressions that condition on `._counter_`.

For highjack functions like `ggtrace_highjack_return()`, the value about to be returned by the function/method can be accessed with `returnValue()` in the `value` argument. By default, `value` is set to `quote(returnValue())` which simply evaluates to the return value, but directly computing on `returnValue()` to derive a different return value for the function/method is also possible.

Examples

```
set.seed(1116)
library(ggplot2)
library(dplyr)


p <- ggplot(mtcars, aes(mpg, hp, color = factor(cyl))) +
  geom_point() +
  geom_smooth(method = "lm")
p
```

```
# Fit predictions from loess regression just for second group
ggtrace_highjack_args(
  x = p,
  method = StatSmooth$compute_group,
  cond = quote(data$group[1] == 2),
  values = list(method = "loess")
)

# If value is an expression, it's evaluated in the Tracing Context
ggtrace_highjack_args(
  x = p,
  method = StatSmooth$compute_group,
  values = rlang::exprs(

    # Every time the method is called, call it with a bigger CI
    level = ._counter_ * 0.3,

    # Fit models to just a random sample of the data
    data = data %>%
      slice_sample(prop = .8)

  )
)
```

ggtrace_highjack_return

Highjack a method's execution and make it return a user-supplied value

Description

Highjack a method's execution and make it return a user-supplied value

Usage

```
ggtrace_highjack_return(
  x,
  method,
  cond = 1L,
  value = quote(returnValue()),
  ...,
  draw = TRUE
)

highjack_return(
  x,
  method,
  cond = 1L,
  value = quote(returnValue()),
  ...,
  draw = TRUE
)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`cond`	When the return value should be replaced. Defaults to `1L`.
`value`	What the method should return instead. Defaults to `quote(returnValue())`.
`...`	Unused.
`draw`	Whether to draw the modified graphical output from evaluating `x`. Defaults to `TRUE`.

Value

A gtable object with class <`ggtrace_highjacked`>

Tracing context

When quoted expressions are passed to the `cond` or `value` argument of workflow functions they are evaluated in a special environment which we call the "tracing context".

The tracing context is "data-masked" (see `rlang::eval_tidy()`), and exposes an internal variable called `._counter_` which increments every time a function/method has been called by the ggplot object supplied to the `x` argument of workflow functions. For example, `cond = quote(._counter_ == 1L)` is evaluated as `TRUE` when the method is called for the first time. The `cond` argument also supports numeric shorthands like `cond = 1L` which evaluates to `quote(._counter_ == 1L)`, and this is the default value of `cond` for all workflow functions that only return one value (e.g., `ggtrace_capture_fn()`). It is recommended to consult the output of `ggtrace_inspect_n()` and `ggtrace_inspect_which()` to construct expressions that condition on `._counter_`.

For highjack functions like `ggtrace_highjack_return()`, the value about to be returned by the function/method can be accessed with `returnValue()` in the `value` argument. By default, `value` is set to `quote(returnValue())` which simply evaluates to the return value, but directly computing on `returnValue()` to derive a different return value for the function/method is also possible.

Examples

```
set.seed(1116)
library(ggplot2)
library(dplyr)
```

```
p1 <- ggplot(diamonds, aes(cut)) +
  geom_bar(aes(fill = cut)) +
  facet_wrap(~ clarity)

p1

# Highjack `Stat$compute_panel` at the first panel
# to return higher values for `count`
ggtrace_highjack_return(
  x = p1, method = Stat$compute_panel,
  value = quote({
    returnValue() %>%
      mutate(count = count * 100)
  })
)

# Highjack `Stat$compute_panel` at the fourth panel
# to shuffle bars in the x-axis
ggtrace_highjack_return(
  x = p1, method = Stat$compute_panel,
  cond = quote(data$PANEL[1] == 4),
  value = quote({
    returnValue() %>%
      mutate(x = sample(x))
  })
)

# Bars get a black outline and get darker from left-to-right, but only for second panel
ggtrace_highjack_return(
  x = p1, method = GeomBar$draw_panel,
  cond = quote(data$PANEL[1] == 2),
  value = quote({
    editGrob(returnValue(), gp = gpar(
      col = "black", alpha = seq(0.2, 1, length.out = nrow(data)
    )))
  })
)
```

`ggtrace_inspect_args` *Inspect the arguments passed into a method*

Description

Inspect the arguments passed into a method

Usage

```
ggtrace_inspect_args(
  x,
  method,
  cond = 1L,
  hoist_dots = TRUE,
  ...,
  error = FALSE
)

inspect_args(x, method, cond = 1L, hoist_dots = TRUE, ..., error = FALSE)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`cond`	When the arguments should be inspected. Defaults to `1L`.
`hoist_dots`	Whether treat arguments passed to `...` like regular arguments. If `FALSE`, the `...` is treated as an argument
`...`	Unused.
`error`	If `TRUE`, continues inspecting the method until the ggplot errors. This is useful for debugging but note that it can sometimes return incomplete output.

Value

A list of argument-value pairs from the `method` when it is called.

Tracing context

When quoted expressions are passed to the `cond` or `value` argument of workflow functions they are evaluated in a special environment which we call the "tracing context".

The tracing context is "data-masked" (see `rlang::eval_tidy()`), and exposes an internal variable called `._counter_` which increments every time a function/method has been called by the ggplot object supplied to the `x` argument of workflow functions. For example, `cond = quote(._counter_ == 1L)` is evaluated as `TRUE` when the method is called for the first time. The `cond` argument also supports numeric shorthands like `cond = 1L` which evaluates to `quote(._counter_ == 1L)`, and this is the default value of `cond` for all workflow functions that only return one value (e.g., `ggtrace_capture_fn()`). It is recommended to consult the output of `ggtrace_inspect_n()` and `ggtrace_inspect_which()` to construct expressions that condition on `._counter_`.

For highjack functions like `ggtrace_highjack_return()`, the value about to be returned by the function/method can be accessed with `returnValue()` in the `value` argument. By default, `value` is set to `quote(returnValue())` which simply evaluates to the return value, but directly computing on `returnValue()` to derive a different return value for the function/method is also possible.

Examples

```
library(ggplot2)

p1 <- ggplot(diamonds, aes(cut)) +
  geom_bar(aes(fill = cut)) +
  facet_wrap(~ clarity)

p1

# Argument value of `Stat$compute_panel` for the first panel
compute_panel_args_1 <- ggtrace_inspect_args(x = p1, method = Stat$compute_panel)
names(ggformals(Stat$compute_panel))
names(compute_panel_args_1)
table(compute_panel_args_1$data$fill)

# `hoist_dots` preserves information about which arguments were passed to `...`
with_dots <- ggtrace_inspect_args(p1, Stat$compute_panel, hoist_dots = FALSE)
names(with_dots)
with_dots$`...`
```

`ggtrace_inspect_n` *Inspect how many times a method was called*

Description

Inspect how many times a method was called

Usage

```
ggtrace_inspect_n(x, method, ..., error = FALSE)

inspect_n(x, method, ..., error = FALSE)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`...`	Unused.
`error`	If `TRUE`, continues inspecting the method until the ggplot errors. This is useful for debugging but note that it can sometimes return incomplete output.

Value

The number of times `method` was called in the evaluation of `x`

Examples

```
library(ggplot2)

p1 <- ggplot(diamonds, aes(cut)) +
  geom_bar(aes(fill = cut)) +
  facet_wrap(~ clarity)

p1

# 1 call to Stat$compute_layer
ggtrace_inspect_n(p1, Stat$compute_layer)

# 8 calls to Stat$compute_panel
ggtrace_inspect_n(p1, Stat$compute_panel)

# Note that there are 0 calls to Stat$compute_group ...
ggtrace_inspect_n(p1, Stat$compute_group)

# because StatCount has its own "compute_group" method defined
ggtrace_inspect_n(p1, StatCount$compute_group)

# How about if we add a second layer that uses StatCount?
p2 <- p1 + geom_text(
  aes(label = after_stat(count)),
  stat = StatCount, position = position_nudge(y = 500)
)

p2

# Now there are double the calls to Stat/StatCount methods
ggtrace_inspect_n(p2, Stat$compute_layer)
ggtrace_inspect_n(p2, Stat$compute_panel)
ggtrace_inspect_n(p2, StatCount$compute_group)

# But separate calls to each layer's respective Geoms
ggtrace_inspect_n(p2, GeomBar$draw_panel)
ggtrace_inspect_n(p2, GeomText$draw_panel)
```

`ggtrace_inspect_on_error` *Get information about a ggproto method on error*

Description

Get information about a ggproto method on error

Usage

```
ggtrace_inspect_on_error(x, method, ...)

inspect_on_error(x, method, ...)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`...`	Unused.

Value

A list of three elements: `counter`, `args`, and `env`.

Examples

```
library(ggplot2)
erroring_barplot <- ggplot(mtcars, aes(mpg, hp)) +
  stat_summary() +
  geom_bar()
ggtrace_inspect_on_error(erroring_barplot, StatCount$setup_params)
ggtrace_inspect_on_error(erroring_barplot, ggplot2:::Layer$compute_statistic)
```

`ggtrace_inspect_return` *Inspect the return value of a method*

Description

Inspect the return value of a method

Usage

```
ggtrace_inspect_return(x, method, cond = 1L, ..., error = FALSE)

inspect_return(x, method, cond = 1L, ..., error = FALSE)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`cond`	When the return value should be inspected. Defaults to `1L`.
`...`	Unused.
`error`	If `TRUE`, continues inspecting the method until the ggplot errors. This is useful for debugging but note that it can sometimes return incomplete output.

Value

The return value from `method` when it is called.

Tracing context

When quoted expressions are passed to the `cond` or `value` argument of workflow functions they are evaluated in a special environment which we call the "tracing context".

The tracing context is "data-masked" (see `rlang::eval_tidy()`), and exposes an internal variable called `._counter_` which increments every time a function/method has been called by the ggplot object supplied to the `x` argument of workflow functions. For example, `cond = quote(._counter_ == 1L)` is evaluated as `TRUE` when the method is called for the first time. The `cond` argument also supports numeric shorthands like `cond = 1L` which evaluates to `quote(._counter_ == 1L)`, and this is the default value of `cond` for all workflow functions that only return one value (e.g., `ggtrace_capture_fn()`). It is recommended to consult the output of `ggtrace_inspect_n()` and `ggtrace_inspect_which()` to construct expressions that condition on `._counter_`.

For highjack functions like `ggtrace_highjack_return()`, the value about to be returned by the function/method can be accessed with `returnValue()` in the `value` argument. By default, `value` is set to `quote(returnValue())` which simply evaluates to the return value, but directly computing on `returnValue()` to derive a different return value for the function/method is also possible.

Examples

```
library(ggplot2)

p1 <- ggplot(diamonds, aes(cut)) +
  geom_bar(aes(fill = cut)) +
  facet_wrap(~ clarity)

p1

# Return value of `Stat$compute_panel` for the first panel
ggtrace_inspect_return(x = p1, method = Stat$compute_panel)

# Return value for 4th panel
```

```
ggtrace_inspect_return(x = p1, method = Stat$compute_panel,
                       cond = 4L)

# Return value for 4th panel, 2nd group (bar)
ggtrace_inspect_return(
  x = p1, method = StatCount$compute_group,
  cond = quote(data$PANEL[1] == 4 && data$group[1] == 2)
)
```

ggtrace_inspect_vars *Inspect the value of variables from a method*

Description

Inspect the value of variables from a method

Usage

```
ggtrace_inspect_vars(
  x,
  method,
  cond = 1L,
  at = "all",
  vars,
  by_var = TRUE,
  ...,
  error = FALSE
)

inspect_vars(
  x,
  method,
  cond = 1L,
  at = "all",
  vars,
  by_var = TRUE,
  ...,
  error = FALSE
)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method`

	• `namespace::ggproto$method` • `namespace:::ggproto$method`
`cond`	When the return value should be inspected. Defaults to `1L`.
`at`	Which steps in the method body the values of `vars` should be retrieved. Defaults to a special value `all` which is evaluated to all steps in the method body.
`vars`	A character vector of variable names
`by_var`	Boolean that controls the format of the output: • `TRUE` (default): returns a list of variables, with their values at each step. This also drops steps within a variable where the variable value has not changed from a previous step specified by `at`. • `FALSE`: returns a list of steps, where each element holds the value of `vars` at each step of `at`. Unchanged variable values are not dropped.
`...`	Unused.
`error`	If `TRUE`, continues inspecting the method until the ggplot errors. This is useful for debugging but note that it can sometimes return incomplete output.

Value

A list of values of `vars` at each step `at`. Simplifies if `vars` and/or `at` is length-1.

Tracing context

When quoted expressions are passed to the `cond` or `value` argument of workflow functions they are evaluated in a special environment which we call the "tracing context".

The tracing context is "data-masked" (see `rlang::eval_tidy()`), and exposes an internal variable called `._counter_` which increments every time a function/method has been called by the ggplot object supplied to the `x` argument of workflow functions. For example, `cond = quote(._counter_ == 1L)` is evaluated as `TRUE` when the method is called for the first time. The `cond` argument also supports numeric shorthands like `cond = 1L` which evaluates to `quote(._counter_ == 1L)`, and this is the default value of `cond` for all workflow functions that only return one value (e.g., `ggtrace_capture_fn()`). It is recommended to consult the output of `ggtrace_inspect_n()` and `ggtrace_inspect_which()` to construct expressions that condition on `._counter_`.

For highjack functions like `ggtrace_highjack_return()`, the value about to be returned by the function/method can be accessed with `returnValue()` in the `value` argument. By default, `value` is set to `quote(returnValue())` which simply evaluates to the return value, but directly computing on `returnValue()` to derive a different return value for the function/method is also possible.

Examples

```
library(ggplot2)

p1 <- ggplot(mtcars[1:10,], aes(mpg, hp)) +
  geom_smooth()
p1

# The `data` variable is bound to two unique values in `compute_group` method:
ggtrace_inspect_vars(p1, StatSmooth$compute_group, vars = "data")
```

```
# Note that elements of this list capture the method's state upon entering a step,
# so "Step1" and "Step5" should be interpreted as the value of `data` at the start
# the method's execution (before "Step1") and its value as a result of running Step4
# (before "Step5"). Indeed, we see that the `weight` column is defined in Step4, so
# the data is flagged as changed at the start of Step5
ggbody(StatSmooth$compute_group)[[4]]


# Comparing the "Steps" themselves can be useful
p2 <- p1 +
  scale_x_continuous(trans = "log") +
  scale_y_continuous(trans = "log")
p2

# Comparing the original plot to one with log-transformed scales reveals a change
# in data detected at the beginning of Step 14
names(ggtrace_inspect_vars(p1, ggplot2:::ggplot_build.ggplot, vars = "data"))
names(ggtrace_inspect_vars(p2, ggplot2:::ggplot_build.ggplot, vars = "data"))

# We can pinpoint the calculation of scale transformations to Step 13:
ggbody(ggplot2:::ggplot_build.ggplot)[[13]]


# With `by_vars = FALSE`, elements of the returned list are steps instead of values.
# Note that this does not drop unchanged values:
ggtrace_inspect_vars(p1, StatSmooth$compute_group, vars = "data", at = 1:6, by_var = FALSE)
```

`ggtrace_inspect_which` *Inspect which calls to a ggproto method met a particular condition*

Description

Inspect which calls to a ggproto method met a particular condition

Usage

```
ggtrace_inspect_which(x, method, cond, ..., error = FALSE)

inspect_which(x, method, cond, ..., error = FALSE)
```

Arguments

`x`	A ggplot object
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method`

	• `namespace::ggproto$method` • `namespace:::ggproto$method`
`cond`	Expression evaluating to a logical inside `method` when `x` is evaluated.
`...`	Unused.
`error`	If `TRUE`, continues inspecting the method until the ggplot errors. This is useful for debugging but note that it can sometimes return incomplete output.

Value

The values of the tracing context variable `._counter_` when `cond` is evaluated as `TRUE`.

Tracing context

When quoted expressions are passed to the `cond` or `value` argument of workflow functions they are evaluated in a special environment which we call the "tracing context".

The tracing context is "data-masked" (see `rlang::eval_tidy()`), and exposes an internal variable called `._counter_` which increments every time a function/method has been called by the ggplot object supplied to the `x` argument of workflow functions. For example, `cond = quote(._counter_ == 1L)` is evaluated as `TRUE` when the method is called for the first time. The `cond` argument also supports numeric shorthands like `cond = 1L` which evaluates to `quote(._counter_ == 1L)`, and this is the default value of `cond` for all workflow functions that only return one value (e.g., `ggtrace_capture_fn()`). It is recommended to consult the output of `ggtrace_inspect_n()` and `ggtrace_inspect_which()` to construct expressions that condition on `._counter_`.

For highjack functions like `ggtrace_highjack_return()`, the value about to be returned by the function/method can be accessed with `returnValue()` in the `value` argument. By default, `value` is set to `quote(returnValue())` which simply evaluates to the return value, but directly computing on `returnValue()` to derive a different return value for the function/method is also possible.

Examples

```
library(ggplot2)

p1 <- ggplot(diamonds, aes(cut)) +
  geom_bar(aes(fill = cut)) +
  facet_wrap(~ clarity)
p1


# Values of `._counter_` when `compute_group` is called for groups in the second panel:
ggtrace_inspect_which(p1, StatCount$compute_group, quote(data$PANEL[1] == 2))


# How about if we add a second layer that uses StatCount?
p2 <- p1 + geom_text(
  aes(label = after_stat(count)),
  stat = StatCount, position = position_nudge(y = 500)
)
p2
```

```
ggtrace_inspect_which(p2, StatCount$compute_group, quote(data$PANEL[1] == 2))


# Behaves like `base::which()` and returns `integer(0)` when no matches are found
ggtrace_inspect_which(p2, StatBoxplot$compute_group, quote(data$PANEL[1] == 2))
```

gguntrace *Remove any existing traces*

Description

Used for explicitly calling `untrace()` on a ggproto object.

Usage

```
gguntrace(method, ...)
```

Arguments

`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`...`	Ignored. Designed for the ease of calling this function by modifying the call to an earlier `{ggtrace}` function in interactive contexts.

Details

Unlike `base::untrace()`, there is no adverse side effect to repeatedly calling `gguntrace()` on a ggproto method. `gguntrace()` will only throw an error if the method cannot be found.

If the method is valid, `gguntrace()` will do one of two things:

- Inform that it has successfully removed the trace (after untracing)
- Inform that the there isn't an existing trace (after doing nothing)

See Also

`ggtrace()`, `ggedit()`

Examples

```
library(ggplot2)

gguntrace(Stat$compute_layer)

is_traced(Stat$compute_layer)

ggtrace(Stat$compute_layer, 1)

is_traced(Stat$compute_layer)

gguntrace(Stat$compute_layer)

is_traced(Stat$compute_layer)

gguntrace(Stat$compute_layer)
```

`is_traced` *Check if a method is being traced*

Description

Check if a method is being traced

Usage

```
is_traced(method)
```

Arguments

`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`

Value

logical

Examples

```
library(ggplot2)

gguntrace(Stat$compute_layer)

is_traced(Stat$compute_layer)
```

```
ggtrace(Stat$compute_layer, 1)

is_traced(Stat$compute_layer)

gguntrace(Stat$compute_layer)

is_traced(Stat$compute_layer)

gguntrace(Stat$compute_layer)
```

`last_layer_errorcontext` *Get the internal context of the last (sub-)layer error*

Description

- `last_layer_errorcontext()` returns the error context at the level of the `Layer` ggproto.
- `last_sublayer_errorcontext()` (EXPERIMENTAL) returns the error context at the sub-`Layer` level (e.g., `Stat` or `Geom`).

Usage

```
last_layer_errorcontext(reprint_error = FALSE, ggtrace_notes = TRUE)

last_sublayer_errorcontext(reprint_error = FALSE, ggtrace_notes = TRUE)
```

Arguments

`reprint_error`	Re-prints the original error message to the console. Defaults to `FALSE`.
`ggtrace_notes`	Prints the `ggtrace_inspect_args()` call used to inspect the error context. Defaults to `TRUE`.

Value

An dynamically constructed and evaluated call to `ggtrace_inspect_args()`. Prioritizes showing the state of layer data whenever possible (by extracting the `data` argument).

Scope

These functions can only retrieve information from errors propagating from `Layer` ggproto methods. In non-technical terms, they only work for errors with a **"Error occured in the Nth layer"** message (as of `{ggplot2}` >= 3.4.0).

The scope of `last_sublayer_errorcontext()` is narrower, since not all `Layer` methods call a sub-Layer method. This function is intended for developers - in most cases users can get all the information necessary to debug layer code from `last_layer_errorcontext()` (there are only so many ways to break a ggplot from user-facing code).

See Also

ggtrace_inspect_on_error()

Examples

```
## Not run:
library(ggplot2)
erroring_barplot1 <- ggplot(mtcars, aes(mpg, hp)) +
  stat_summary(fun.data = "mean_se") +
  geom_bar()

# Render to trigger error
erroring_barplot1

# Both return the same snapshot of layer data
# but at different levels of specificity
last_layer_errorcontext()
last_sublayer_errorcontext()

erroring_barplot2 <- ggplot(mtcars, aes(mpg, hp)) +
  stat_summary() +
  geom_bar(aes(y = c(1, 2)))
erroring_barplot2

# This works:
last_layer_errorcontext()
# This doesn't: there's no sub-layer ggproto involved in this error
last_sublayer_errorcontext()

library(ggforce)
erroring_sina <- ggplot(mtcars, aes(mpg)) +
  geom_bar() +
  geom_sina()
erroring_barplot1

# The two return different snapshots of layer data here -
# see `ggplot2:::Layer$compute_statistic` for why.
last_layer_errorcontext()
last_sublayer_errorcontext()


## End(Not run)
```

---

with_ggtrace *Generic workflow function which localizes a ggtrace call to a single ggplot object*

Description

`with_ggtrace()` provides a functional interface to `ggtrace()`. It takes a ggplot object and parameters passed to `ggtrace()` and returns the immediate tracedump and/or graphical output without side effects.

Usage

```
with_ggtrace(x, method, ..., out = c("tracedump", "gtable", "both"))
```

Arguments

`x`	A ggplot object whose evaluation triggers the trace as specified by the `...`
`method`	A function or a ggproto method. The ggproto method may be specified using any of the following forms: • `ggproto$method` • `namespace::ggproto$method` • `namespace:::ggproto$method`
`...`	Arguments passed on to `ggtrace` `trace_steps` A sorted numeric vector of positions in the method's body to trace. Negative indices reference steps from the last, where `-1` references the last step in the body. Special value `"all"` traces all steps of the method body. `trace_exprs` A list of expressions to evaluate at each position specified in `trace_steps`. If a single expression is provided, it is recycled to match the length of `trace_steps`. To simply run a step and return its output, you can use the `~step` keyword. If the step is an assign expression, the value of the assigned variable is returned. If `trace_exprs` is not provided, `ggtrace()` is called with `~step` by default. `once` Whether to `untrace()` the method on exit. If `FALSE`, creates a persistent trace which is active until `gguntrace()` is called on the method. Defaults to `TRUE`. `use_names` Whether the trace dump should use the names from `trace_exprs`. If `trace_exprs` is not specified, whether to use the method steps as names. Defaults to `TRUE`. `print_output` Whether to `print()` the output of each expression to the console. Defaults to `TRUE`. `verbose` Whether logs should be printed when trace is triggered. Encompasses `print_output`, meaning that `verbose = FALSE` also triggers the effect of `print_output = FALSE` by consequence. Defaults to `FALSE`.
`out`	Whether the function should return the output of triggered traces ("tracedump"), or the resulting graphical object from evaluating the ggplot ("gtable"), or "both", which returns the tracedump but also renders the resulting plot as a side effect. Partial matching is supported, so these options could also be specified as "t", "g", or "b". Defaults to "tracedump".

Value

A list or gtable object of class <ggtrace_highjacked>

Note

To trigger evaluation of `x`, the function `ggeval_silent(x)` is called internally.

See Also

`ggtrace()`, `ggeval_silent()`

Examples

```
library(ggplot2)

# Long-form `ggtrace()` method:
boxplot_plot <- ggplot(diamonds[1:500,], aes(cut, depth)) + geom_boxplot()
ggtrace(
 method = StatBoxplot$compute_group,
 trace_steps = -1, trace_exprs = quote(~step)
)
boxplot_plot
first_tracedump <- last_ggtrace()

# Short-form functional `with_ggtrace()` method:
second_tracedump <- with_ggtrace(
  x = boxplot_plot,
  method = StatBoxplot$compute_group,
  trace_steps = -1, trace_exprs = quote(~step)
)

identical(first_tracedump, second_tracedump)


# An example with `out = "gtable"` (or `"g"`)
grid_plot <- ggplot(mtcars, aes(mpg, hp)) +
  geom_point() +
  facet_grid(am ~ cyl)
grid_plot

outline <- grid::rectGrob(
  x = 0.5, y = 0.5, width = 1, height = 1,
  gp = grid::gpar(col = "red", lwd = 5, fill = NA)
)

with_ggtrace(
  x = grid_plot,
  method = Layout$render,
  trace_steps = 5,
  trace_exprs = rlang::expr({
    panels[c(3, 5)] <- lapply(panels[c(3, 5)], function(panel) {
      gTree(children = gList(panel, !!outline))
    })
  }),
  out = "gtable" # or "g"
)


# With `once = FALSE` for persistent tracing (still cleaned up after)
lm_plot <- ggplot(mpg, aes(displ, hwy, color = drv)) +
  geom_point() +
  geom_smooth(method = "lm")
lm_plot

with_ggtrace(
  x = lm_plot,
  method = StatSmooth$compute_group,
  trace_steps = c(1, 11),
  trace_exprs = list(
    group = quote(data$group[1]),
    coef = quote(model$coef)
  )
)

with_ggtrace(
  x = lm_plot,
  method = StatSmooth$compute_group,
  trace_steps = 1,
  trace_exprs = quote(method <- c("loess", "lm", "loess")[data$group[1]]),
  out = "g" # or "gtable"
)
```

Index