Package ‘future’

March 14, 2026

Version 1.70.0

Title Unified Parallel and Distributed Processing in R for Everyone

Depends R (>= 3.2.0)

Imports digest, globals (>= 0.18.0), listenv (>= 0.8.0), parallel,
parallelly (>= 1.44.0), tools, utils

Suggests methods, RhpcBLASctl, R.rsp, markdown

VignetteBuilder R.rsp

Description The purpose of this package is to provide a lightweight and
unified Future API for sequential and parallel processing of R
expression via futures. The simplest way to evaluate an expression
in parallel is to use \`x %<-% { expression }\` with \`plan(multisession)\`.
This package implements sequential, multicore, multisession, and
cluster futures. With these, R expressions can be evaluated on the
local machine, in parallel a set of local machines, or distributed
on a mix of local and remote machines.
Extensions to this package implement additional backends for
processing futures via compute cluster schedulers, etc.
Because of its unified API, there is no need to modify any code in order
switch from sequential on the local machine to, say, distributed
processing on a remote compute cluster.
Another strength of this package is that global variables and functions
are automatically identified and exported as needed, making it
straightforward to tweak existing code to make use of futures.

License LGPL (>= 2.1)

LazyLoad TRUE

ByteCompile TRUE

URL `https://future.futureverse.org`,
`https://github.com/futureverse/future`

BugReports `https://github.com/futureverse/future/issues`

Language en-US

Encoding UTF-8

RoxygenNote 7.3.3

Collate '000.bquote.R' '000.import.R' '000.re-exports.R' '009.deprecation.R' '010.tweakable.R' '010.utils-parallelly.R' 'backend_api-01-FutureBackend-class.R' 'backend_api-03.MultiprocessFutureBackend-class.R' 'backend_api-11.ClusterFutureBackend-class.R' 'backend_api-11.MulticoreFutureBackend-class.R' 'backend_api-11.SequentialFutureBackend-class.R' 'backend_api-13.MultisessionFutureBackend-class.R' 'backend_api-ConstantFuture-class.R' 'backend_api-Future-class.R' 'backend_api-FutureRegistry.R' 'backend_api-UniprocessFuture-class.R' 'backend_api-evalFuture.R' 'core_api-cancel.R' 'core_api-future.R' 'core_api-reset.R' 'core_api-resolved.R' 'core_api-value.R' 'delayed_api-futureAssign.R' 'delayed_api-futureOf.R' 'demo_api-mandelbrot.R' 'infix_api-01-futureAssign_OP.R' 'infix_api-02-globals_OP.R' 'infix_api-03-seed_OP.R' 'infix_api-04-stdout_OP.R' 'infix_api-05-conditions_OP.R' 'infix_api-06-lazy_OP.R' 'infix_api-07-label_OP.R' 'infix_api-08-plan_OP.R' 'infix_api-09-tweak_OP.R' 'protected_api-FutureCondition-class.R' 'protected_api-FutureGlobals-class.R' 'protected_api-FutureResult-class.R' 'protected_api-futures.R' 'protected_api-globals.R' 'protected_api-journal.R' 'protected_api-resolve.R' 'protected_api-result.R' 'protected_api-signalConditions.R' 'testme.R' 'utils-basic.R' 'utils-conditions.R' 'utils-connections.R' 'utils-debug.R' 'utils-immediateCondition.R' 'utils-marshalling.R' 'utils-objectSize.R' 'utils-options.R' 'utils-prune_pkg_code.R' 'utils-registerClusterTypes.R' 'utils-rng_utils.R' 'utils-signalEarly.R' 'utils-stealth_sample.R' 'utils-sticky_globals.R' 'utils-tweakExpression.R' 'utils-uuid.R' 'utils-whichIndex.R' 'utils_api-backtrace.R' 'utils_api-capture_journals.R' 'utils_api-futureCall.R' 'utils_api-futureSessionInfo.R' 'utils_api-makeClusterFuture.R' 'utils_api-minifuture.R' 'utils_api-nbrOfWorkers.R' 'utils_api-plan.R' 'utils_api-plan-with.R' 'utils_api-sessionDetails.R' 'utils_api-tweak.R' 'zzz.R'

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph] (ORCID: <`https://orcid.org/0000-0002-7579-5165`>)

Maintainer Henrik Bengtsson <`henrikb@braju.com`>

Repository CRAN

Date/Publication 2026-03-14 06:10:29 UTC

Contents

`backtrace` *Back trace the expressions evaluated when an error was caught*

Description

Back trace the expressions evaluated when an error was caught

Usage

```
backtrace(future, envir = parent.frame(), ...)
```

Arguments

`future`	A future with a caught error.
`envir`	the environment where to locate the future.
`...`	Not used.

Value

A list with the future's call stack that led up to the error.

Examples

```
my_log <- function(x) log(x)
foo <- function(...) my_log(...)

f <- future({ foo("a") })
res <- tryCatch({
  v <- value(f)
}, error = function(ex) {
  t <- backtrace(f)
  print(t)
})
```

cancel *Cancel a future*

Description

Cancels futures, with the option to interrupt running ones.

Usage

```
cancel(x, interrupt = TRUE, ...)
```

Arguments

`x`	A Future.
`interrupt`	If TRUE, running futures are interrupted, if the future backend supports it.
`...`	All arguments used by the S3 methods.

Value

`cancel()` returns (invisibly) the canceled Futures after flagging them as "canceled" and possibly interrupting them as well.

Canceling a lazy or a finished future has no effect.

See Also

A canceled future can be `reset()` to a lazy, vanilla future such that it can be relaunched, possibly on another future backend.

Examples

```
## Set up two parallel workers
plan(multisession, workers = 2)

## Launch two long running futures
fs <- lapply(c(1, 2), function(duration) {
  future({
    Sys.sleep(duration)
    42
  })
})

## Wait until at least one of the futures is resolved
while (!any(resolved(fs))) Sys.sleep(0.1)

## Cancel the future that is not yet resolved
r <- resolved(fs)
cancel(fs[!r])

## Get the value of the resolved future
f <- fs[r]
v <- value(f)
message("Result: ", v)

## The value of the canceled future is an error
try(v <- value(fs[!r]))

## Shut down parallel workers
plan(sequential)
```

`cluster` *Create a cluster future whose value will be resolved asynchronously in a parallel process*

Description

WARNING: This function must never be called. It may only be used with `plan()`

Usage

```
cluster(
  ...,
  workers = availableWorkers(constraints = "connections"),
  persistent = FALSE
)
```

Arguments

`workers`	A `cluster` object, a character vector of host names, a positive numeric scalar, or a function. If a character vector or a numeric scalar, a `cluster` object is created using `makeClusterPSOCK(workers)`. If a function, it is called without arguments *when the future is created* and its value is used to configure the workers. The function should return any of the above types. If `workers == 1`, then all processing using done in the current/main R session and we therefore fall back to using a sequential future. To override this fallback, use `workers = I(1)`.
`persistent`	If FALSE, the evaluation environment is cleared from objects prior to the evaluation of the future.
`...`	Not used.

Details

A cluster future is a future that uses cluster evaluation, which means that its *value is computed and resolved in parallel in another process*.

This function is must *not* be called directly. Instead, the typical usages are:

```
# Evaluate futures via a single background R process on the local machine
plan(cluster, workers = I(1))

# Evaluate futures via two background R processes on the local machine
plan(cluster, workers = 2)

# Evaluate futures via a single R process on another machine on on the
# local area network (LAN)
plan(cluster, workers = "raspberry-pi")

# Evaluate futures via a single R process running on a remote machine
plan(cluster, workers = "pi.example.org")

# Evaluate futures via four R processes, one running on the local machine,
# two running on LAN machine 'n1' and one on a remote machine
plan(cluster, workers = c("localhost", "n1", "n1", "pi.example.org"))
```

See Also

For alternative future backends, see the 'A Future for R: Available Future Backends' vignette and https://www.futureverse.org/backends.html.

Examples

```
## Use cluster futures
cl <- parallel::makeCluster(2, timeout = 60)
plan(cluster, workers = cl)

## A global variable
```

```
a <- 0

## Create future (explicitly)
f <- future({
  b <- 3
  c <- 2
  a * b * c
})

## A cluster future is evaluated in a separate process.
## Regardless, changing the value of a global variable will
## not affect the result of the future.
a <- 7
print(a)

v <- value(f)
print(v)
stopifnot(v == 0)

## CLEANUP
parallel::stopCluster(cl)
```

`future` *Create a future*

Description

Creates a future that evaluates an R expression or a future that calls an R function with a set of arguments. How, when, and where these futures are evaluated can be configured using `plan()` such that they are evaluated in parallel on, for instance, the current machine, on a remote machine, or via a job queue on a compute cluster. Importantly, any R code using futures remains the same regardless of these settings and there is no need to modify the code when switching from, say, sequential to parallel processing.

Usage

```
future(
  expr,
  envir = parent.frame(),
  substitute = TRUE,
  lazy = FALSE,
  seed = FALSE,
  globals = TRUE,
  packages = NULL,
  stdout = TRUE,
  conditions = "condition",
  label = NULL,
  ...
)

futureCall(
  FUN,
  args = list(),
  envir = parent.frame(),
  lazy = FALSE,
  seed = FALSE,
  globals = TRUE,
  packages = NULL,
  stdout = TRUE,
  conditions = "condition",
  label = NULL,
  ...
)

minifuture(
  expr,
  substitute = TRUE,
  globals = NULL,
  packages = NULL,
  stdout = NA,
  conditions = NULL,
  seed = NULL,
  ...,
  envir = parent.frame()
)
```

Arguments

`expr`	An R expression.
`envir`	The environment from where global objects should be identified.
`substitute`	If TRUE, argument `expr` is `substitute()`:d, otherwise not.
`lazy`	If FALSE (default), the future is resolved eagerly (starting immediately), otherwise not.
`seed`	(optional) If TRUE, the random seed, that is, the state of the random number generator (RNG) will be set such that statistically sound random numbers are produced (also during parallelization). If FALSE (default), it is assumed that the future expression neither needs nor uses random number generation. To use a fixed random seed, specify a L'Ecuyer-CMRG seed (seven integers) or a regular RNG seed (a single integer). If the latter, then a L'Ecuyer-CMRG seed will be automatically created based on the given seed. Furthermore, if FALSE, then the future will be monitored to make sure it does not use random numbers. If it does and depending on the value of option future.rng.onMisuse, the check is ignored, an informative warning, or error will be produced. If `seed` is NULL, then the effect is as with `seed = FALSE` but without the RNG check being performed.

`globals`	(optional) a logical, a character vector, or a named list to control how globals are handled. For details, see section 'Globals used by future expressions' in the help for `future()`.
`packages`	(optional) a character vector specifying packages to be attached in the R environment evaluating the future, *in addition to packages required by global variables* specified or identified via argument `globals`.
`stdout`	If TRUE (default), then the standard output is captured, and re-outputted when `value()` is called. If FALSE, any output is silenced (by sinking it to the null device as it is outputted). Using `stdout = structure(TRUE, drop = TRUE)` causes the captured standard output to be dropped from the future object as soon as it has been relayed. This can help decrease the overall memory consumed by captured output across futures. Using `stdout = NA` fully avoids intercepting the standard output; behavior of such unhandled standard output depends on the future backend.
`conditions`	A character string of condition classes to be captured and relayed. The default is to relay all conditions, including messages and warnings. To drop all conditions, use `conditions = character(0)`. Errors are always relayed. Attribute `exclude` can be used to ignore specific classes, e.g. `conditions = structure("condition", exclude = "message")` will capture all `condition` classes except those that inherit from the `message` class. Using `conditions = structure(..., drop = TRUE)` causes any captured conditions to be dropped from the future object as soon as they have been relayed, e.g. by `value(f)`. This can help decrease the overall memory consumed by captured conditions across futures. Using `conditions = NULL` (not recommended) avoids intercepting conditions, except from errors; behavior of such unhandled conditions depends on the future backend and the environment from which R runs.
`label`	A character string label attached to the future.
`FUN`	A function to be evaluated.
`args`	A list of arguments passed to function `FUN`.
`...`	Additional arguments passed to `Future()`.

Details

The state of a future is either unresolved or resolved. The value of a future can be retrieved using `v <- value(f)`. Querying the value of a non-resolved future will *block* the call until the future is resolved. It is possible to check whether a future is resolved or not without blocking by using `resolved(f)`. It is possible to `cancel()` a future that is being resolved. Failed, canceled, and interrupted futures can be `reset()` to a lazy, vanilla future that can be relaunched.

The `futureCall()` function works analogously to `do.call()`, which calls a function with a set of arguments. The difference is that `do.call()` returns the value of the call whereas `futureCall()` returns a future.

Value

`future()` returns Future that evaluates expression `expr`.

`futureCall()` returns a Future that calls function `FUN` with arguments `args`.

`minifuture(expr)` creates a future with minimal overhead, by disabling user-friendly behaviors, e.g. automatic identification of global variables and packages needed, and relaying of output. Unless you have good reasons for using this function, please use `future()` instead. This function exists mainly for the purpose of profiling and identifying which automatic features of `future()` introduce extra overhead.

Eager or lazy evaluation

By default, a future is resolved using *eager* evaluation (`lazy = FALSE`). This means that the expression starts to be evaluated as soon as the future is created.

As an alternative, the future can be resolved using *lazy* evaluation (`lazy = TRUE`). This means that the expression will only be evaluated when the value of the future is requested. *Note that this means that the expression may not be evaluated at all - it is guaranteed to be evaluated if the value is requested.*

Globals used by future expressions

Global objects (short *globals*) are objects (e.g. variables and functions) that are needed in order for the future expression to be evaluated while not being local objects that are defined by the future expression. For example, in

```
a <- 42
f <- future({ b <- 2; a * b })
```

variable `a` is a global of future assignment `f` whereas `b` is a local variable. In order for the future to be resolved successfully (and correctly), all globals need to be gathered when the future is created such that they are available whenever and wherever the future is resolved.

The default behavior (`globals = TRUE`), is that globals are automatically identified and gathered. More precisely, globals are identified via code inspection of the future expression `expr` and their values are retrieved with environment `envir` as the starting point (basically via `get(global, envir = envir, inherits = TRUE)`). *In most cases, such automatic collection of globals is sufficient and less tedious and error prone than if they are manually specified.*

However, for full control, it is also possible to explicitly specify exactly which globals are by providing their names as a character vector. In the above example, we could use

```
a <- 42
f <- future({ b <- 2; a * b }, globals = "a")
```

Yet another alternative is to explicitly also specify their values using a named list as in

```
a <- 42
f <- future({ b <- 2; a * b }, globals = list(a = a))
```

or

```
f <- future({ b <- 2; a * b }, globals = list(a = 42))
```

Specifying globals explicitly avoids the overhead added from automatically identifying the globals and gathering their values. Furthermore, if we know that the future expression does not make use of any global variables, we can disable the automatic search for globals by using

```
f <- future({ a <- 42; b <- 2; a * b }, globals = FALSE)
```

Future expressions often make use of functions from one or more packages. As long as these functions are part of the set of globals, the future package will make sure that those packages are attached when the future is resolved. Because there is no need for such globals to be frozen or exported, the future package will not export them, which reduces the amount of transferred objects. For example, in

```
x <- rnorm(1000)
f <- future({ median(x) })
```

variable `x` and `median()` are globals, but only `x` is exported whereas `median()`, which is part of the **stats** package, is not exported. Instead, it ensures that the **stats** package is on the search path when the future expression is evaluated. Effectively, the above becomes

```
x <- rnorm(1000)
f <- future({
  library(stats)
  median(x)
})
```

To manually specify this, one can either do

```
x <- rnorm(1000)
f <- future({
  median(x)
}, globals = list(x = x, median = stats::median)
```

or

```
x <- rnorm(1000)
f <- future({
  library(stats)
  median(x)
}, globals = list(x = x))
```

Both are effectively the same.

Although rarely needed, a combination of automatic identification and manual specification of globals is supported via attributes `add` (to add false negatives) and `ignore` (to ignore false positives) on value `TRUE`. For example, with `globals = structure(TRUE, ignore = "b", add = "a")` any globals automatically identified, except `b`, will be used, in addition to global `a`.

Author(s)

The future logo was designed by Dan LaBar and tweaked by Henrik Bengtsson.

See Also

How, when, and where futures are resolved is determined by the *future backend*, which can be set by the end user using the `plan()` function.

Examples

```
## Evaluate futures in parallel
plan(multisession)

## Data
x <- rnorm(100)
y <- 2 * x + 0.2 + rnorm(100)
w <- 1 + x ^ 2


## EXAMPLE: Regular assignments (evaluated sequentially)
fitA <- lm(y ~ x, weights = w)      ## with offset
fitB <- lm(y ~ x - 1, weights = w)  ## without offset
fitC <- {
  w <- 1 + abs(x)  ## Different weights
  lm(y ~ x, weights = w)
}
print(fitA)
print(fitB)
print(fitC)


## EXAMPLE: Future assignments (evaluated in parallel)
fitA %<-% lm(y ~ x, weights = w)      ## with offset
fitB %<-% lm(y ~ x - 1, weights = w)  ## without offset
fitC %<-% {
  w <- 1 + abs(x)
  lm(y ~ x, weights = w)
}
print(fitA)
print(fitB)
print(fitC)


## EXAMPLE: Explicitly create futures (evaluated in parallel)
## and retrieve their values
fA <- future( lm(y ~ x, weights = w) )
fB <- future( lm(y ~ x - 1, weights = w) )
fC <- future({
  w <- 1 + abs(x)
  lm(y ~ x, weights = w)
})
fitA <- value(fA)
fitB <- value(fB)
fitC <- value(fC)
print(fitA)
print(fitB)
print(fitC)


## EXAMPLE: futureCall() and do.call()
x <- 1:100
```

```
y0 <- do.call(sum, args = list(x))
print(y0)

f1 <- futureCall(sum, args = list(x))
y1 <- value(f1)
print(y1)
```

---

futureAssign *Create a future assignment*

Description

`x %<-% value` (also known as a "future assignment") and `futureAssign("x", value)` create a Future that evaluates the expression (`value`) and binds it to variable `x` (as a promise). The expression is evaluated in parallel in the background. Later on, when `x` is first queried, the value of the future is automatically retrieved as if it were a regular variable and `x` is materialized as a regular value.

Usage

```
futureAssign(
  x,
  value,
  envir = parent.frame(),
  substitute = TRUE,
  lazy = FALSE,
  seed = FALSE,
  globals = TRUE,
  packages = NULL,
  stdout = TRUE,
  conditions = "condition",
  label = NULL,
  ...,
  assign.env = envir
)

x %<-% value

fassignment %globals% globals
fassignment %packages% packages

fassignment %seed% seed

fassignment %stdout% capture

fassignment %conditions% capture

fassignment %lazy% lazy
fassignment %label% label

fassignment %plan% strategy

fassignment %tweak% tweaks
```

Arguments

`x`	the name of a future variable, which will hold the value of the future expression (as a promise).
`value`	An R expression.
`envir`	The environment from where global objects should be identified.
`substitute`	If TRUE, argument `expr` is `substitute()`:d, otherwise not.
`lazy`	If FALSE (default), the future is resolved eagerly (starting immediately), otherwise not.
`seed`	(optional) If TRUE, the random seed, that is, the state of the random number generator (RNG) will be set such that statistically sound random numbers are produced (also during parallelization). If FALSE (default), it is assumed that the future expression neither needs nor uses random number generation. To use a fixed random seed, specify a L'Ecuyer-CMRG seed (seven integers) or a regular RNG seed (a single integer). If the latter, then a L'Ecuyer-CMRG seed will be automatically created based on the given seed. Furthermore, if FALSE, then the future will be monitored to make sure it does not use random numbers. If it does and depending on the value of option future.rng.onMisuse, the check is ignored, an informative warning, or error will be produced. If `seed` is NULL, then the effect is as with `seed = FALSE` but without the RNG check being performed.
`globals`	(optional) a logical, a character vector, or a named list to control how globals are handled. For details, see section 'Globals used by future expressions' in the help for `future()`.
`packages`	(optional) a character vector specifying packages to be attached in the R environment evaluating the future.
`stdout`	If TRUE (default), then the standard output is captured, and re-outputted when `value()` is called. If FALSE, any output is silenced (by sinking it to the null device as it is outputted). Using `stdout = structure(TRUE, drop = TRUE)` causes the captured standard output to be dropped from the future object as soon as it has been relayed. This can help decrease the overall memory consumed by captured output across futures. Using `stdout = NA` fully avoids intercepting the standard output; behavior of such unhandled standard output depends on the future backend.
`conditions`	A character string of condition classes to be captured and relayed. The default is to relay all conditions, including messages and warnings. To drop all conditions, use `conditions = character(0)`. Errors are always relayed. Attribute `exclude` can be used to ignore specific classes, e.g. `conditions = structure("condition", exclude = "message")` will capture all `condition` classes except those that inherit from the `message` class. Using `conditions = structure(..., drop =`

	`TRUE`) causes any captured conditions to be dropped from the future object as soon as they have been relayed, e.g. by `value(f)`. This can help decrease the overall memory consumed by captured conditions across futures. Using `conditions = NULL` (not recommended) avoids intercepting conditions, except from errors; behavior of such unhandled conditions depends on the future backend and the environment from which R runs.
`label`	A character string label attached to the future.
`assign.env`	The environment to which the variable should be assigned.
`fassignment`	The future assignment, e.g. `x %<-% { expr }`.
`capture`	If TRUE, the standard output will be captured, otherwise not.
`strategy`	The backend controlling how the future is resolved. See `plan()` for further details.
`tweaks`	A named list (or vector) with arguments that should be changed relative to the current backend.
`...`	Additional arguments passed to `Future()`.

Details

For a future created via a future assignment, `x %<-% value` or `futureAssign("x", value)`, the value is bound to a promise, which when queried will internally call `value()` on the future and which will then be resolved into a regular variable bound to that value. For example, with future assignment `x %<-% value`, the first time variable `x` is queried the call blocks if, and only if, the future is not yet resolved. As soon as it is resolved, and any succeeding queries, querying `x` will immediately give the value.

The future assignment construct `x %<-% value` is not a formal assignment per se, but a binary infix operator on objects `x` and expression `value`. However, by using non-standard evaluation, this construct can emulate an assignment operator similar to `x <- value`. Due to R's precedence rules of operators, future expressions often need to be explicitly bracketed, e.g. `x %<-% { a + b }`.

Value

`futureAssign()` and `x %<-% expr` returns the Future invisibly, e.g. `f <- futureAssign("x", expr)` and `f <- (x %<-% expr)`.

Adjust future arguments of a future assignment

`future()` and `futureAssign()` take several arguments that can be used to explicitly specify what global variables and packages the future should use. They can also be used to override default behaviors of the future, e.g. whether output should be relayed or not. When using a future assignment, these arguments can be specified via corresponding assignment expression. For example, `x %<-% { rnorm(10) } %seed% TRUE` corresponds to `futureAssign("x", { rnorm(10) }, seed = TRUE)`. Here are several examples.

To explicitly specify variables and functions that a future assignment should use, use `%globals%`. To explicitly specify which packages need to be attached for the evaluation to succeed, use `%packages%`. For example,

```
> x <- rnorm(1000)
> y %<-% { median(x) } %globals% list(x = x) %packages% "stats"
> y
[1] -0.03956372
```

The `median()` function is part of the 'stats' package.

To declare that you will generate random numbers, use `%seed%`, e.g.

```
> x %<-% { rnorm(3) } %seed% TRUE
> x
[1] -0.2590562 -1.2262495  0.8858702
```

To disable relaying of standard output (e.g. `print()`, `cat()`, and `str()`), while keeping relaying of conditions (e.g. `message()` and

```
> x %<-% { cat("Hello\n"); message("Hi there"); 42 } %stdout% FALSE
> y <- 13
> z <- x + y
Hi there
> z
[1] 55
```

To disable relaying of conditions, use `%conditions%`, e.g.

```
> x %<-% { cat("Hello\n"); message("Hi there"); 42 } %conditions% character(0)
> y <- 13
> z <- x + y
Hello
> z
[1] 55
```

```
> x %<-% { print(1:10); message("Hello"); 42 } %stdout% FALSE
> y <- 13
> z <- x + y
Hello
> z
[1] 55
```

To create a future without launching it such that it will only be processed if the value is really needed, use `%lazy%`, e.g.

```
> x %<-% { Sys.sleep(5); 42 } %lazy% TRUE
> y <- sum(1:10)
> system.time(z <- x + y)
  user  system elapsed
  0.004   0.000   5.008
> z
[1] 97
```

Error handling

Because future assignments are promises, errors produced by the future expression will not be signaled until the value of the future is requested. For example, if you create a future assignment that produces an error, you will not be affected by the error until you "touch" the future-assignment variable. For example,

```
> x %<-% { stop("boom") }
> y <- sum(1:10)
> z <- x + y
Error in eval(quote({ : boom
```

Use alternative future backend for future assignment

Futures are evaluated on the future backend that the user has specified by `plan()`. With regular futures, we can temporarily use another future backend by wrapping our code in `with(plan(...), { ... })`, or temporarily inside a function using `with(plan(...), local = TRUE)`. To achieve the same for a specific future assignment, use `%plan%`, e.g.

```
> plan(multisession)
> x %<-% { 42 }
> y %<-% { 13 } %plan% sequential
> z <- x + y
> z
[1] 55
```

Here `x` is resolved in the background via the multisession backend, whereas `y` is resolved sequentially in the main R session.

Getting the future object of a future assignment

The underlying Future of a future variable `x` can be retrieved without blocking using `f <- futureOf(x)`, e.g.

```
> x %<-% { stop("boom") }
> f_x <- futureOf(x)
> resolved(f_x)
[1] TRUE
> x
Error in eval(quote({ : boom
> value(f_x)
Error in eval(quote({ : boom
```

Technically, both the future and the variable (promise) are assigned at the same time to environment `assign.env` where the name of the future is `.future_<name>`.

futureOf — *Get the future of a future variable*

Description

Get the future of a future variable that has been created directly or indirectly via `future()`.

Usage

```
futureOf(
  var = NULL,
  envir = parent.frame(),
  mustExist = TRUE,
  default = NA,
  drop = FALSE
)
```

Arguments

`var`	the variable. If NULL, all futures in the environment are returned.
`envir`	the environment where to search from.
`mustExist`	If TRUE and the variable does not exist, then an informative error is thrown, otherwise `default` is returned.
`default`	the default value if future was not found.
`drop`	if TRUE and `var` is NULL, then returned list only contains futures, otherwise also `default` values.

Value

A Future (or `default`). If `var` is NULL, then a named list of Future:s is returned.

Examples

```
a %<-% { 1 }

f <- futureOf(a)
print(f)

b %<-% { 2 }

f <- futureOf(b)
print(f)

## All futures
fs <- futureOf()
print(fs)
```

```
## Futures part of environment
env <- new.env()
env$c %<-% { 3 }

f <- futureOf(env$c)
print(f)

f2 <- futureOf(c, envir = env)
print(f2)

f3 <- futureOf("c", envir = env)
print(f3)

fs <- futureOf(envir = env)
print(fs)
```

`futures` *Get all futures in a container*

Description

Gets all futures in an environment, a list, or a list environment and returns an object of the same class (and dimensions). Non-future elements are returned as is.

Usage

```
futures(x, ...)
```

Arguments

`x`	An environment, a list, or a list environment.
`...`	Not used.

Details

This function is useful for retrieve futures that were created via future assignments (`%<-%`) and therefore stored as promises. This function turns such promises into standard `Future` objects.

Value

An object of same type as `x` and with the same names and/or dimensions, if set.

`futureSessionInfo` *Get future-specific session information and validate current backend*

Description

Get future-specific session information and validate current backend

Usage

```
futureSessionInfo(test = TRUE, anonymize = TRUE)
```

Arguments

`test`	If TRUE, one or more futures are created to query workers and validate their information.
`anonymize`	If TRUE, user names and host names are anonymized.

Value

Nothing.

Examples

```
plan(multisession, workers = 2)
futureSessionInfo()
plan(sequential)
```

`multicore` *Create a multicore future whose value will be resolved asynchronously in a forked parallel process*

Description

WARNING: This function must never be called. It may only be used with `plan()`

Usage

```
multicore(..., workers = availableCores(constraints = "multicore"))
```

Arguments

`workers`	The number of parallel processes to use. If a function, it is called without arguments *when the future is created* and its value is used to configure the workers. If `workers` == 1, then all processing using done in the current/main R session and we therefore fall back to using a sequential future. To override this fallback, use `workers = I(1)`.
`...`	Not used.

Details

A multicore future is a future that uses multicore evaluation, which means that its *value is computed and resolved in parallel in another process*.

This function is must *not* be called directly. Instead, the typical usages are:

```
# Evaluate futures in parallel on the local machine via as many forked
# processes as available to the current R process
plan(multicore)

# Evaluate futures in parallel on the local machine via two forked processes
plan(multicore, workers = 2)
```

Support for forked ("multicore") processing

Not all operating systems support process forking and thereby not multicore futures. For instance, forking is not supported on Microsoft Windows. Moreover, process forking may break some R environments such as RStudio. Because of this, the future package disables process forking also in such cases. See `parallelly::supportsMulticore()` for details. Trying to create multicore futures on non-supported systems or when forking is disabled will result in multicore futures falling back to becoming sequential futures. If used in RStudio, there will be an informative warning:

```
> plan(multicore)
Warning message:
In supportsMulticoreAndRStudio(...) :
  [ONE-TIME WARNING] Forked processing ('multicore') is not supported when
running R from RStudio because it is considered unstable. For more details,
how to control forked processing or not, and how to silence this warning in
future R sessions, see ?parallelly::supportsMulticore
```

Why not forked processing?

One reason for *not* using forked parallel processing in R is that is not guaranteed to be stable in all environments or in all contexts, which also depends on which functions are called in the forked processes. Here is what R Core developer of the `parallel::mclapply()` family of functions, which `multicore` rely on, said on R-devel (2020-04-29):

"Do NOT use `mcparallel()` *in packages except as a non-default option that user can set ... Multicore is intended for HPC applications that need to use many cores for computing-heavy jobs, but it does not play well with RStudio and more importantly you [as the developer] don't know the resource available so only the user can tell you when it's safe to use."*

See Also

For processing in multiple background R sessions, see multisession futures.

For alternative future backends, see the 'A Future for R: Available Future Backends' vignette and `https://www.futureverse.org/backends.html`.

Use `parallelly::availableCores()` to see the total number of cores that are available for the current R session. Use `availableCores("multicore") > 1L` to check whether multicore futures are supported or not on the current system.

Examples

```
## Use multicore futures
plan(multicore)

## A global variable
a <- 0

## Create future (explicitly)
f <- future({
  b <- 3
  c <- 2
  a * b * c
})

## A multicore future is evaluated in a separate forked
## process.  Changing the value of a global variable
## will not affect the result of the future.
a <- 7
print(a)

v <- value(f)
print(v)
stopifnot(v == 0)
```

---

multisession — *Create a multisession future whose value will be resolved asynchronously in a parallel R session*

Description

WARNING: This function must never be called. It may only be used with `plan()`

Usage

```
multisession(
  ...,
  workers = availableCores(constraints = "connections-16"),
  rscript_libs = .libPaths()
)
```

Arguments

`workers`	The number of parallel processes to use. If a function, it is called without arguments *when the future is created* and its value is used to configure the workers. If `workers == 1`, then all processing using done in the current/main R session and we therefore fall back to using a sequential future. To override this fallback, use `workers = I(1)`.

`rscript_libs`	A character vector of R package library folders that the workers should use. The default is `.libPaths()` so that multisession workers inherits the same library path as the main R session. To avoid this, use `plan(multisession, ..., rscript_libs = NULL)`. *Important: Note that the library path is set on the workers when they are created, i.e. when* `plan(multisession)` *is called. Any changes to* `.libPaths()` *in the main R session after the workers have been created will have no effect.* This is passed down as-is to `parallelly::makeClusterPSOCK()`.
`...`	Additional arguments passed to `Future()`.

Details

A multisession future is a future that uses multisession evaluation, which means that its *value is computed and resolved in parallel in another* R *session*.

This function is must *not* be called directly. Instead, the typical usages are:

```
# Evaluate futures in parallel on the local machine via as many background
# processes as available to the current R process
plan(multisession)

# Evaluate futures in parallel on the local machine via two background
# processes
plan(multisession, workers = 2)
```

The background R sessions (the "workers") are created using `makeClusterPSOCK()`.

For the total number of R sessions available including the current/main R process, see `parallelly::availableCores()`.

A multisession future is a special type of cluster future.

Value

A MultisessionFuture. If `workers == 1`, then all processing is done in the current/main R session and we therefore fall back to using a lazy future. To override this fallback, use `workers = I(1)`.

See Also

For processing in multiple forked R sessions, see multicore futures.

Use `parallelly::availableCores()` to see the total number of cores that are available for the current R session.

Examples

```
## Use multisession futures
plan(multisession)

## A global variable
a <- 0

## Create future (explicitly)
```

```
f <- future({
  b <- 3
  c <- 2
  a * b * c
})

## A multisession future is evaluated in a separate R session.
## Changing the value of a global variable will not affect
## the result of the future.
a <- 7
print(a)

v <- value(f)
print(v)
stopifnot(v == 0)

## Explicitly close multisession workers by switching plan
plan(sequential)
```

`nbrOfWorkers` *Get the number of workers available*

Description

Get the number of workers available

Usage

```
nbrOfWorkers(evaluator = NULL)

nbrOfFreeWorkers(evaluator = NULL, background = FALSE, ...)
```

Arguments

`evaluator`	A future evaluator function. If NULL (default), the current evaluator as returned by `plan()` is used.
`background`	If TRUE, only workers that can process a future in the background are considered. If FALSE, also workers running in the main R process are considered, e.g. when using the 'sequential' backend.
`...`	Not used; reserved for future use.

Value

`nbrOfWorkers()` returns a positive number in $1, 2, 3, ...$, which for some future backends may also be `+Inf`.

`nbrOfFreeWorkers()` returns a non-negative number in $0, 1, 2, 3, ...$ which is less than or equal to `nbrOfWorkers()`.

Examples

```
plan(multisession)
nbrOfWorkers()  ## == availableCores()

plan(sequential)
nbrOfWorkers()  ## == 1
```

---

plan — *Plan how to resolve a future*

Description

This function allows *the user* to plan the future, more specifically, it specifies how `future()`s are resolved, e.g. sequentially or in parallel.

Usage

```
plan(
  strategy = NULL,
  ...,
  substitute = TRUE,
  .skip = FALSE,
  .call = TRUE,
  .cleanup = NA,
  .init = TRUE
)

## S3 method for class 'FutureStrategyList'
with(data, expr, ..., local = FALSE, envir = parent.frame(), .cleanup = NA)

tweak(strategy, ..., penvir = parent.frame())
```

Arguments

`strategy`	A future backend or the name of one.
`substitute`	If `TRUE`, the `strategy` expression is `substitute()`:d, otherwise not.
`.skip`	(internal) If `TRUE`, then attempts to set a future backend that is the same as what is currently in use, will be skipped.
`.call`	(internal) Used for recording the call to this function.
`.cleanup`	(internal) Used to stop implicitly started clusters.
`.init`	(internal) Used to initiate workers.
`data`	The future plan to use temporarily, e.g. `plan(multisession)`.
`expr`	The R expression to be evaluated.
`local`	If TRUE, then the future plan specified by `data` is applied temporarily in the calling frame. Argument `expr` must not be specified if `local = TRUE`.

`envir`	The environment where the future plan should be set and the expression evaluated.
`penvir`	The environment used when searching for a future function by its name.
`...`	Additional arguments overriding the default arguments of the evaluation function. Which additional arguments are supported depends on which future backend is used, e.g. several support argument `workers` but not all. For details, see the individual backends of which some are linked to below.

Details

The default backend is `sequential`, but another one can be set using `plan()`, e.g. `plan(multisession)` will launch parallel workers running in the background, which then will be used to resolve futures. To shut down background workers launched this way, call `plan(sequential)`.

Value

`plan()` returns the previous plan invisibly if a new future backend is chosen, otherwise it returns the current one visibly.

The value of the expression evaluated (invisibly).

a future function.

Built-in evaluation strategies

The **future** package provides the following built-in backends:

`sequential`**:** Resolves futures sequentially in the current R process, e.g. `plan(sequential)`.

`multisession`**:** Resolves futures asynchronously (in parallel) in separate R sessions running in the background on the same machine, e.g. `plan(multisession)` and `plan(multisession, workers = 2)`.

`multicore`**:** Resolves futures asynchronously (in parallel) in separate *forked* R processes running in the background on the same machine, e.g. `plan(multicore)` and `plan(multicore, workers = 2)`. This backend is not supported on Windows.

`cluster`**:** Resolves futures asynchronously (in parallel) in separate R sessions running typically on one or more machines, e.g. `plan(cluster)`, `plan(cluster, workers = 2)`, and `plan(cluster, workers = c("n1", "n1", "n2", "server.remote.org"))`.

Other evaluation strategies available

In addition to the built-in ones, additional parallel backends are implemented in future-backend packages **future.callr** and **future.mirai** that leverage R package **callr** and **mirai**:

`callr`**:** Similar to `multisession`, this resolves futures in parallel in background R sessions on the local machine via the **callr** package, e.g. `plan(future.callr::callr)` and `plan(future.callr::callr, workers = 2)`. The difference is that each future is processed in a fresh parallel R worker, which is automatically shut down as soon as the future is resolved. This can help decrease the overall memory usage. Moreover, contrary to `multisession`, `callr` does not rely on socket connections, which means it is not limited by the number of connections that R can have open at any time.

`mirai_multisession`**:** Similar to `multisession`, this resolves futures in parallel in background R sessions on the local machine via the **mirai** package, e.g. `plan(future.mirai::mirai_multisession)` and `plan(future.mirai::mirai_multisession, workers = 2)`.

`mirai_cluster`**:** Similar to `cluster`, this resolves futures in parallel via pre-configured R **mirai** daemon processes, e.g. `plan(future.mirai::mirai_cluster)`.

Another example is the **future.batchtools** package, which leverages **batchtools** package, to resolve futures via high-performance compute (HPC) job schedulers, e.g. LSF, Slurm, TORQUE/PBS, Grid Engine, and OpenLava;

`batchtools_slurm`**:** The backend resolves futures via the Slurm scheduler, e.g. `plan(future.batchtools::batchtools_`

`batchtools_torque`**:** The backend resolves futures via the TORQUE/PBS scheduler, e.g. `plan(future.batchtools::bat`

`batchtools_sge`**:** The backend resolves futures via the Grid Engine (SGE, AGE) scheduler, e.g. `plan(future.batchtools::batchtools_sge)`.

`batchtools_lsf`**:** The backend resolves futures via the Load Sharing Facility (LSF) scheduler, e.g. `plan(future.batchtools::batchtools_lsf)`.

`batchtools_openlava`**:** The backend resolves futures via the OpenLava scheduler, e.g. `plan(future.batchtools::batch`

For package developers

Please refrain from modifying the future backend inside your packages / functions, i.e. do not call `plan()` in your code. Instead, leave the control on what backend to use to the end user. This idea is part of the core philosophy of the future framework—as a developer you can never know what future backends the user have access to. Moreover, by not making any assumptions about what backends are available, your code will also work automatically with any new backends developed after you wrote your code.

If you think it is necessary to modify the future backend within a function, then make sure to undo the changes when exiting the function. This can be achieved by using `with(plan(...), local = TRUE)`, e.g.

```
  my_fcn <- function(x) {
    with(plan(multisession), local = TRUE)
    y <- analyze(x)
    summarize(y)
  }
```

This is important because the end-user might have already set the future strategy elsewhere for other purposes and will most likely not know that calling your function will break their setup. *Remember, your package and its functions might be used in a greater context where multiple packages and functions are involved and those might also rely on the future framework, so it is important to avoid stepping on others' toes.*

Using plan() in scripts and vignettes

When writing scripts or vignettes that use futures, try to place any call to `plan()` as far up (i.e. as early on) in the code as possible. This will help users to quickly identify where the future plan is set up and allow them to modify it to their computational resources. Even better is to leave it to the user to set the plan() prior to source():ing the script or running the vignette. If a '.future.R' exists in the current directory and / or in the user's home directory, it is sourced when the **future** package is *loaded*. Because of this, the '.future.R' file provides a convenient place for users to set the plan(). This behavior can be controlled via an R option—see future options for more details.

See Also

Use plan() to set a future to become the new default strategy.

Examples

```
a <- b <- c <- NA_real_

# An sequential future
plan(sequential)
f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
})
y <- value(f)
print(y)
str(list(a = a, b = b, c = c)) ## All NAs


# A sequential future with lazy evaluation
plan(sequential)
f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
}, lazy = TRUE)
y <- value(f)
print(y)
str(list(a = a, b = b, c = c)) ## All NAs


# A multicore future (specified as a string)
plan("multicore")
f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
})
y <- value(f)
print(y)
str(list(a = a, b = b, c = c)) ## All NAs

## Multisession futures gives an error on R CMD check on
## Windows (but not Linux or macOS) for unknown reasons.
## The same code works in package tests.


# A multisession future (specified via a string variable)
plan("future::multisession")
f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
})
y <- value(f)
print(y)
str(list(a = a, b = b, c = c)) ## All NAs




## Explicitly specifying number of workers
## (default is parallelly::availableCores())
plan(multicore, workers = 2)
message("Number of parallel workers: ", nbrOfWorkers())


## Explicitly close multisession workers by switching plan
plan(sequential)
# Evaluate a future using the 'multisession' plan
with(plan(multisession, workers = 2), {
  f <- future(Sys.getpid())
  w_pid <- value(f)
})
print(c(main = Sys.getpid(), worker = w_pid))



# Evaluate a future locally using the 'multisession' plan
local({
  with(plan(multisession, workers = 2), local = TRUE)

  f <- future(Sys.getpid())
  w_pid <- value(f)
  print(c(main = Sys.getpid(), worker = w_pid))
})
```

`reset` *Reset a finished, failed, canceled, or interrupted future to a lazy future*

Description

A future that has successfully completed, was canceled or interrupted, or has failed due to an error, can be relaunched after resetting it.

Usage

```
reset(x, ...)
```

Arguments

`x`	A Future.
`...`	Not used.

Details

A lazy, vanilla Future can be reused in another R session. For instance, if we do:

```
library(future)
a <- 2
f <- future(42 * a, lazy = TRUE)
saveRDS(f, "myfuture.rds")
```

Then we can read and evaluate the future in another R session using:

```
library(future)
f <- readRDS("myfuture.rds")
v <- value(f)
print(v)
#> [1] 84
```

Value

`reset()` returns a lazy, vanilla Future that can be relaunched. Resetting a running future results in a FutureError.

Examples

```
## Like mean(), but fails 90% of the time
shaky_mean <- function(x) {
  if (as.double(Sys.time()) %% 1 < 0.90) stop("boom")
  mean(x)
}

x <- rnorm(100)

## Calculate the mean of 'x' with a risk of failing randomly
f <- future({ shaky_mean(x) })

## Relaunch until success
```

```
repeat({
  v <- tryCatch(value(f), error = identity)
  if (!inherits(v, "error")) break
  message("Resetting failed future, and retry in 0.1 seconds")
  f <- reset(f)
  Sys.sleep(0.1)
})
cat("mean:", v, "\n")
```

`resolve` *Resolve one or more futures synchronously*

Description

This function provides an efficient mechanism for waiting for multiple futures in a container (e.g. list or environment) to be resolved while in the meanwhile retrieving values of already resolved futures.

Usage

```
resolve(
  x,
  idxs = NULL,
  recursive = 0,
  result = FALSE,
  stdout = FALSE,
  signal = FALSE,
  force = FALSE,
  sleep = getOption("future.wait.interval", 0.01),
  ...
)
```

Arguments

`x`	A Future to be resolved, or a list, an environment, or a list environment of futures to be resolved.
`idxs`	(optional) integer or logical index specifying the subset of elements to check.
`recursive`	A non-negative number specifying how deep of a recursion should be done. If TRUE, an infinite recursion is used. If FALSE or zero, no recursion is performed.
`result`	(internal) If TRUE, the results are *retrieved*, otherwise not. Note that this only collects the results from the parallel worker, which can help lower the overall latency if there are multiple concurrent futures. This does *not* return the collected results.
`stdout`	(internal) If TRUE, captured standard output is relayed, otherwise not.
`signal`	(internal) If TRUE, captured conditions are relayed, otherwise not.

`force`	(internal) If TRUE, captured standard output and captured conditions already relayed are relayed again, otherwise not.
`sleep`	Number of seconds to wait before checking if futures have been resolved since last time.
`...`	Not used.

Details

This function is resolves synchronously, i.e. it blocks until `x` and any containing futures are resolved.

Value

Returns `x` (regardless of subsetting or not). If `signal` is TRUE and one of the futures produces an error, then that error is produced.

See Also

To resolve a future *variable*, first retrieve its Future object using `futureOf()`, e.g. `resolve(futureOf(x))`.

`resolved.ClusterFuture` *Check whether a future is resolved or not*

Description

Check whether a future is resolved or not

Usage

```
## S3 method for class 'ClusterFuture'
resolved(x, timeout = NULL, ...)

## S3 method for class 'MulticoreFuture'
resolved(x, timeout = NULL, ...)

resolved(x, ...)

## Default S3 method:
resolved(x, ...)

## S3 method for class 'list'
resolved(x, ...)

## S3 method for class 'environment'
resolved(x, ...)

## S3 method for class 'Future'
resolved(x, ...)
```

Arguments

`x`	A Future, a list, or an environment (which also includes list environment).
`timeout`	(numeric) The maximum time (in seconds) for polling the worker for a response. If no response is available within this time limit, FALSE is returned assuming the future is still being processed. If NULL, the value defaults to `getOption("future.<type>.resolved.timeout")`, then `getOption("future.resolved.timeout")` and finally 0.01 (seconds), where <type> corresponds to the type of future, e.g. `cluster` and `multicore`.
`...`	Not used.

Details

`resolved()` attempts to launch a lazy future, if there is an available worker, otherwise not.

`resolved()` methods must always return `TRUE` or `FALSE` values, must always launch lazy futures, and must never block indefinitely. This is because it should always be possible to poll futures until they are resolved using `resolved()`, e.g. `while (!all(resolved(futures))) Sys.sleep(5)`.

Each future backend must implement a `resolved()` method. It should return either TRUE or FALSE, or throw a FutureError (which indicates a significant, often unrecoverable infrastructure problem, or an interrupt).

Value

A logical vector of the same length and dimensions as `x`. Each element is TRUE unless the corresponding element is a non-resolved future in case it is FALSE. It never signals an error.

The default method always returns TRUE.

Behavior of cluster and multisession futures

If all worker slots are occupied, `resolved()` for `ClusterFuture` and `MultisessionFuture` will attempt to free one up by checking whether one of the futures is *resolved*. If there is one, then its result is collected in order to free up one worker slot.

`resolved()` for `ClusterFuture` may receive immediate condition objects, rather than a FutureResult, when polling the worker for results. In such cases, the condition object is collected and another poll it performed. Up to 100 immediate conditions may be collected this way per `resolved()` call, before considering the future non-resolved and FALSE being returned.

Behavior of multicore futures

`resolved()` for `MulticoreFuture` may receive immediate condition objects, rather than a FutureResult, when polling the worker for results. In such cases, *all* such condition objects are collected, before considering the future non-resolved and FALSE being returned.

sequential — *Create a sequential future whose value will be in the current R session*

Description

WARNING: This function must never be called. It may only be used with `plan()`

Usage

```
sequential(..., envir = parent.frame())
```

Arguments

`envir`	The environment from where global objects should be identified.
`...`	Not used.

Details

A sequential future is a future that is evaluated sequentially in the current R session similarly to how R expressions are evaluated in R. The only difference to R itself is that globals are validated by default just as for all other types of futures in this package.

This function is must *not* be called directly. Instead, the typical usages are:

```
# Evaluate futures sequentially in the current R process
plan(sequential)
```

Examples

```
## Use sequential futures
plan(sequential)

## A global variable
a <- 0

## Create a sequential future
f <- future({
  b <- 3
  c <- 2
  a * b * c
})

## Since 'a' is a global variable in future 'f' which
## is eagerly resolved (default), this global has already
## been resolved / incorporated, and any changes to 'a'
## at this point will _not_ affect the value of 'f'.
a <- 7
print(a)
```

```
v <- value(f)
print(v)
stopifnot(v == 0)
```

`value` *The value of a future or the values of all elements in a container*

Description

Gets the value of a future or the values of all elements (including futures) in a container such as a list, an environment, or a list environment. If one or more futures are unresolved, then this function blocks until all queried futures are resolved.

Usage

```
value(...)

## S3 method for class 'Future'
value(future, stdout = TRUE, signal = TRUE, drop = FALSE, ...)

## S3 method for class 'list'
value(
  x,
  idxs = NULL,
  recursive = 0,
  reduce = NULL,
  stdout = TRUE,
  signal = TRUE,
  cancel = TRUE,
  interrupt = cancel,
  inorder = TRUE,
  drop = FALSE,
  force = TRUE,
  sleep = getOption("future.wait.interval", 0.01),
  ...
)

## S3 method for class 'listenv'
value(
  x,
  idxs = NULL,
  recursive = 0,
  reduce = NULL,
  stdout = TRUE,
  signal = TRUE,
  cancel = TRUE,
  interrupt = cancel,
  inorder = TRUE,
  drop = FALSE,
  force = TRUE,
  sleep = getOption("future.wait.interval", 0.01),
  ...
)

## S3 method for class 'environment'
value(x, ...)
```

Arguments

`future, x`	A Future, an environment, a list, or a list environment.
`stdout`	If TRUE, standard output captured while resolving futures is relayed, otherwise not.
`signal`	If TRUE, conditions captured while resolving futures are relayed, otherwise not.
`drop`	If TRUE, resolved futures are minimized in size and invalidated as soon as their values have been collected and any output and conditions have been relayed. Combining `drop = TRUE` with `inorder = FALSE` reduces the memory use sooner, especially avoiding the risk of holding on to future values until the very end.
`idxs`	(optional) integer or logical index specifying the subset of elements to check.
`recursive`	A non-negative number specifying how deep of a recursion should be done. If TRUE, an infinite recursion is used. If FALSE or zero, no recursion is performed.
`reduce`	An optional function for reducing all the values. Optional attribute `init` can be used to set initial value for the reduction. If not specified, the first value will be used as the initial value. Reduction of values is done as soon as possible, but always in the same order as `x`, unless `inorder` is FALSE.
`cancel, interrupt`	If TRUE and `signal` is TRUE, non-resolved futures are canceled as soon as an error is detected in one of the futures, before signaling the error. Argument `interrupt` is passed to `cancel()` controlling whether non-resolved futures should also be interrupted.
`inorder`	If TRUE, then standard output and conditions are relayed, and value reduction is done in the order the futures occur in `x`, but always as soon as possible. This is achieved by buffering the details until they can be released. By setting `inorder = FALSE`, no buffering takes place and everything is relayed and reduced as soon as a new future is resolved. Regardless, the values are always returned in the same order as `x`.
`force`	(internal) If TRUE, captured standard output and captured conditions already relayed are relayed again, otherwise not.
`sleep`	Number of seconds to wait before checking if futures have been resolved since last time.
`...`	All arguments used by the S3 methods.

Value

`value()` of a Future object returns the value of the future, which can be any type of R object.

`value()` of a list, an environment, or a list environment returns an object with the same number of elements and of the same class. Names and dimension attributes are preserved, if available. All future elements are replaced by their corresponding `value()` values. For all other elements, the existing object is kept as-is.

If `signal` is TRUE and one of the futures produces an error, then that error is relayed. Any remaining, non-resolved futures in `x` are canceled, prior to signaling such an error. If the future was interrupted, canceled, or the parallel worker terminated abruptly ("crashed"), then a FutureInterruptError is signaled.

Examples

```
## -------------------------------------------------------
## A single future
## -------------------------------------------------------
x <- sample(100, size = 50)
f <- future(mean(x))
v <- value(f)
message("The average of 50 random numbers in [1,100] is: ", v)



## -------------------------------------------------------
## Ten futures
## -------------------------------------------------------
xs <- replicate(10, { list(sample(100, size = 50)) })
fs <- lapply(xs, function(x) { future(mean(x)) })

## The 10 values as a list (because 'fs' is a list)
vs <- value(fs)
message("The ten averages are:")
str(vs)

## The 10 values as a vector (by manually unlisting)
vs <- value(fs)
vs <- unlist(vs)
message("The ten averages are: ", paste(vs, collapse = ", "))

## The values as a vector (by reducing)
vs <- value(fs, reduce = c)
message("The ten averages are: ", paste(vs, collapse = ", "))

## Calculate the sum of the averages (by reducing)
total <- value(fs, reduce = `+`)
message("The sum of the ten averages is: ", total)
```

zzz-future.options *Options used for futures*

Description

Below are the R options and environment variables that are used by the **future** package and packages enhancing it.

WARNING: Note that the names and the default values of these options may change in future versions of the package. Please use with care until further notice.

Packages must not change future options

Just like for other R options, as a package developer you must *not* change any of the below `future.*` options. Only the end-user should set these. If you find yourself having to tweak one of the options, make sure to undo your changes immediately afterward. For example, if you want to bump up the `future.globals.maxSize` limit when creating a future, use something like the following inside your function:

```
oopts <- options(future.globals.maxSize = 1.0 * 1e9)  ## 1.0 GB
on.exit(options(oopts))
f <- future({ expr })  ## Launch a future with large objects
```

Options for controlling futures

'`future.plan`': (character string or future function) Default future backend used unless otherwise specified via `plan()`. This will also be the future plan set when calling `plan("default")`. If not specified, this option may be set when the **future** package is *loaded* if command-line option `--parallel=ncores` (short `-p ncores`) is specified; if `ncores` > 1, then option '`future.plan`' is set to `multisession` otherwise `sequential` (in addition to option '`mc.cores`' being set to `ncores`, if `ncores` >= 1). (Default: `sequential`)

'`future.globals.maxSize`': (numeric) Maximum allowed total size (in bytes) of global variables identified. This is used to protect against exporting too large objects to parallel workers by mistake. Transferring large objects over a network, or over the internet, can be slow and therefore introduce a large bottleneck that increases the overall processing time. It can also result in large egress or ingress costs, which may exist on some systems. If set of `+Inf`, then the check for large globals is skipped. (Default: `500 * 1024 ^ 2` = 500 MiB)

'`future.globals.onReference`': (*beta feature - may change*) (character string) Controls whether the identified globals should be scanned for so called *references* (e.g. external pointers and connections) or not. It is unlikely that another R process ("worker") can use a global that uses a internal reference of the master R process—we call such objects *non-exportable globals*. If this option is `"error"`, an informative error message is produced if a non-exportable global is detected. If `"warning"`, a warning is produced, but the processing will continue; it is likely that the future will be resolved with a run-time error unless processed in the master R process (e.g. `plan(sequential)` and `plan(multicore)`). If `"ignore"`, no scan is performed. (Default: `"ignore"` but may change)

'`future.resolve.recursive`': (integer) An integer specifying the maximum recursive depth to which futures should be resolved. If negative, nothing is resolved. If `0`, only the future itself is resolved. If `1`, the future and any of its elements that are futures are resolved, and so on. If `+Inf`, infinite search depth is used. (Default: `0`)

'`future.onFutureCondition.keepFuture`': (logical) If `TRUE`, a `FutureCondition` keeps a copy of the `Future` object that triggered the condition. If `FALSE`, it is dropped. (Default: `TRUE`)

'`future.wait.timeout`': (numeric) Maximum waiting time (in seconds) for a future to resolve or for a free worker to become available before a timeout error is generated. (Default: `30 * 24 * 60 * 60` (= 30 days))

'`future.wait.interval`': (numeric) Initial interval (in seconds) between polls. This controls the polling frequency for finding an available worker when all workers are currently busy. It also controls the polling frequency of `resolve()`. (Default: `0.01` = 1 ms)

'`future.wait.alpha`': (numeric) Positive scale factor used to increase the interval after each poll. (Default: `1.01`)

Options for built-in sanity checks

Ideally, the evaluation of a future should have no side effects. To protect against unexpected side effects, the future framework comes with a set of built-in tools for checking against this. Below R options control these built-in checks and what should happen if they fail. You may modify them for troubleshooting purposes, but please refrain from disabling these checks when there is an underlying problem that should be fixed.

Beta features: Please consider these checks to be "under construction".

'`future.connections.onMisuse`': (character string) A future must close any connections it opens and must not close connections it did not open itself. If such misuse is detected and this option is set to `"error"`, then an informative error is produced. If it is set to `"warning"`, a warning is produced. If`"ignore"`, no check is performed. (Default: `"warning"`)

'`future.defaultDevice.onMisuse`': (character string) A future must open graphics devices explicitly, if it creates new plots. It should not rely on the default graphics device that is given by R option `"default"`, because that rarely does what is intended. If such misuse is detected and this option is set to `"error"`, then an informative error is produced. If it is set to `"warning"`, a warning is produced. If`"ignore"`, no check is performed. (Default: `"warning"`)

'`future.devices.onMisuse`': (character string) A future must close any graphics devices it opens and must not close devices it did not open itself. If such misuse is detected and this option is set to `"error"`, then an informative error is produced. If it is set to `"warning"`, a warning is produced. If`"ignore"`, no check is performed. (Default: `"warning"`)

'`future.globalenv.onMisuse`': (character string) Assigning variables to the global environment for the purpose of using the variable at a later time makes no sense with futures, because the next the future may be evaluated in different R process. To protect against mistakes, the future framework attempts to detect when variables are added to the global environment. If this is detected, and this option is set to `"error"`, then an informative error is produced. If `"warning"`, then a warning is produced. If `"ignore"`, no check is performed. (Default: `"ignore"`)

'`future.rng.onMisuse`': (character string) If random numbers are used in futures, then parallel RNG should be *declared* in order to get statistical sound RNGs. You can declare this by specifying future argument `seed = TRUE`. The defaults in the future framework assume that *no* random number generation (RNG) is taken place in the future expression because L'Ecuyer-CMRG RNGs come with an unnecessary overhead if not needed. To protect against mistakes of not declaring use of the RNG, the future framework detects when random numbers were used despite not declaring such use. If this is detected, and this options is set `"error"`, then an informative error is produced. If `"warning"`, then a warning is produced. If `"ignore"`, no check is performed. (Default: `"warning"`)

Options for debugging futures

'`future.debug`': (logical) If `TRUE`, extensive debug messages are generated. (Default: `FALSE`)

Options for controlling package startup

'`future.startup.script`': (character vector or a logical) Specifies zero of more future startup scripts to be sourced when the **future** package is *attached*. It is only the first existing script that is sourced. If none of the specified files exist, nothing is sourced—there will be neither a warning nor an error. If this option is not specified, environment variable `R_FUTURE_STARTUP_SCRIPT` is considered, where multiple scripts may be separated by either a colon (`:`) or a semicolon (`;`). If neither is set, or either is set to `TRUE`, the default is to look for a '`.future.R`' script in the current directory and then in the user's home directory. To disable future startup scripts, set the option or the environment variable to `FALSE`. *Importantly*, this option is *always* set to `FALSE` if the **future** package is loaded as part of a future expression being evaluated, e.g. in a background process. In other words, they are sourced in the main R process but not in future processes. (Default: `TRUE` in main R process and `FALSE` in future processes / during future evaluation)

'`future.cmdargs`': (character vector) Overrides `commandArgs()` when the **future** package is *loaded*.

Options for configuring low-level system behaviors

'`future.fork.multithreading.enable`' (*beta feature - may change*): (logical) Enable or disable *multi-threading* while using *forked* parallel processing. If `FALSE`, different multi-thread library settings are overridden such that they run in single-thread mode. Specifically, multi-threading will be disabled for OpenMP (which requires the **RhpcBLASctl** package) and for **RcppParallel**. If `TRUE`, or not set (the default), multi-threading is allowed. Parallelization via multi-threaded processing (done in native code by some packages and external libraries) while at the same time using forked (aka "multicore") parallel processing is known to unstable. Note that this is not only true when using `plan(multicore)` but also when using, for instance, `mclapply()` of the **parallel** package. (Default: not set)

'`future.output.windows.reencode`': (logical) Enable or disable re-encoding of UTF-8 symbols that were incorrectly encoded while captured. In R (< 4.2.0) and on older versions of MS Windows, R cannot capture UTF-8 symbols as-is when they are captured from the standard output. For examples, a UTF-8 check mark symbol (`"\u2713"`) would be relayed as `"<U+2713>"` (a string with eight ASCII characters). Setting this option to `TRUE` will cause `value()` to attempt to recover the intended UTF-8 symbols from <U+nnnn> string components, if, and only if, the string was captured by a future resolved on MS Windows. (Default: `TRUE`)

Options for demos

'`future.demo.mandelbrot.region`': (integer) Either a named list of `mandelbrot()` arguments or an integer in {1, 2, 3} specifying a predefined Mandelbrot region. (Default: `1L`)

'`future.demo.mandelbrot.nrow`': (integer) Number of rows and columns of tiles. (Default: `3L`)

Deprecated or for internal prototyping

The following options exists only for troubleshooting purposes and must not be used in production. If used, there is a risk that the results are non-reproducible if processed elsewhere. To lower the risk of them being used by mistake, they are marked as deprecated and will produce warnings if set.

'`future.globals.onMissing`': (character string) Action to take when non-existing global variables ("globals" or "unknowns") are identified when the future is created. If `"error"`, an error is generated immediately. If `"ignore"`, no action is taken and an attempt to evaluate the future expression will be made. The latter is useful when there is a risk for false-positive globals being identified, e.g. when future expression contains non-standard evaluation (NSE). (Default: `"ignore"`)

'`future.globals.method`': (character string) Method used to identify globals. For details, see `globalsOf()`. (Default: `"ordered"`)

'`future.globals.resolve`': (logical) If `TRUE`, globals that are `Future` objects (typically created as *explicit* futures) will be resolved and have their values (using `value()`) collected. Because searching for unresolved futures among globals (including their content) can be expensive, the default is not to do it and instead leave it to the run-time checks that assert proper ownership when resolving futures and collecting their values. (Default: `FALSE`)

Environment variables that set R options

All of the above R '`future.*`' options can be set by corresponding environment variable `R_FUTURE_*` *when the* **future** *package is loaded*. This means that those environment variables must be set before the **future** package is loaded in order to have an effect. For example, if `R_FUTURE_RNG_ONMISUSE="ignore"`, then option '`future.rng.onMisuse`' is set to `"ignore"` (character string). Similarly, if `R_FUTURE_GLOBALS_MAXSIZE="5000` then option '`future.globals.maxSize`' is set to `50000000` (numeric).

Options moved to the 'parallelly' package

Several functions have been moved to the **parallelly** package:

- `parallelly::availableCores()`
- `parallelly::availableWorkers()`
- `parallelly::makeClusterMPI()`
- `parallelly::makeClusterPSOCK()`
- `parallelly::makeNodePSOCK()`
- `parallelly::supportsMulticore()`

The options and environment variables controlling those have been adjusted accordingly to have different prefixes. For example, option '`future.fork.enable`' has been renamed to '`parallelly.fork.enable`' and the corresponding environment variable `R_FUTURE_FORK_ENABLE` has been renamed to `R_PARALLELLY_FORK_ENABLE`. For backward compatibility reasons, the **parallelly** package will support both versions for a long foreseeable time. See the parallelly::parallelly.options page for the settings.

See Also

To set R options or environment variables when R starts (even before the **future** package is loaded), see the Startup help page. The **startup** package provides a friendly mechanism for configurating R's startup process.

Examples

```
# Allow at most 5 MB globals per futures
options(future.globals.maxSize = 5e6)

# Be strict; catch all RNG mistakes
options(future.rng.onMisuse = "error")
```

Index