Package 'future'

November 3, 2020

Version 1.20.1

Title Unified Parallel and Distributed Processing in R for Everyone

Imports digest, globals (>= 0.13.1), listenv (>= 0.8.0), parallel, parallelly (>= 1.21.0), tools, utils

Suggests 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(multiprocess)`. 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://github.com/HenrikBengtsson/future

BugReports https://github.com/HenrikBengtsson/future/issues

RoxygenNote 7.1.1

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph]

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2020-11-03 06:40:10 UTC

R topics documented:

`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)
})
```

---

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

Description

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

Usage

```
cluster(..., workers = availableWorkers(), envir = parent.frame())
```

Arguments

...	Additional named elements passed to `ClusterFuture()`.
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.
envir	The environment from where global objects should be identified.

Details

This function will block if all available R cluster nodes are occupied and will be unblocked as soon as one of the already running cluster futures is resolved.

The preferred way to create an cluster future is not to call this function directly, but to register it via `plan(cluster)` such that it becomes the default mechanism for all futures. After this `future()` and `%<-%` will create *cluster futures*.

Value

A ClusterFuture.

Examples

```
## Use cluster futures
cl <- parallel::makeCluster(2L, 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)
```

---

clusterExportSticky *Export globals to the sticky-globals environment of the cluster nodes*

Description

Export globals to the sticky-globals environment of the cluster nodes

Usage

```
clusterExportSticky(cl, globals)
```

Arguments

cl	(cluster) A cluster object as returned by parallel::makeCluster().
globals	(list) A named list of sticky globals to be exported.

Details

This requires that the **future** package is installed on the cluster nodes.

Value

(invisible; cluster) The cluster object.

---

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 it is 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 on 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,
  label = NULL,
  gc = FALSE,
  ...
)

futureAssign(
  x,
  value,
  envir = parent.frame(),
```

```
  substitute = TRUE,
  lazy = FALSE,
  seed = NULL,
  globals = TRUE,
  ...,
  assign.env = envir
)

x %<-% value

futureCall(
  FUN,
  args = list(),
  envir = parent.frame(),
  lazy = FALSE,
  seed = NULL,
  globals = TRUE,
  packages = NULL,
  ...
)
```

Arguments

expr, value	An R expression.
envir	The environment from where global objects should be identified.
substitute	If TRUE, argument expr is substitute():ed, 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 does neither need nor use random numbers generation. To use a fixed random seed, specify a L'Ecuyer-CMRG seed (seven integer) or a regular RNG seed (a single integer). 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.
label	An optional character string label attached to the future.
gc	If TRUE, the garbage collector run (in the process that evaluated the future) only after the value of the future is collected. Exactly when the values are collected

	may depend on various factors such as number of free workers and whether `earlySignal` is TRUE (more frequently) or FALSE (less frequently). *Some types of futures ignore this argument.*
`...`	Additional arguments passed to `Future()`.
`x`	the name of a future variable, which will hold the value of the future expression (as a promise).
`assign.env`	The environment to which the variable should be assigned.
`FUN`	A function to be evaluated.
`args`	A list of arguments passed to function `FUN`.

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)`.

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 constructs 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 }`.

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

`f <-future(expr)` creates a Future `f` that evaluates expression `expr`, the value of the future is retrieved using `v <-value(f)`.

`x %<-% value` (a future assignment) and `futureAssign("x",value)` create a Future that evaluates expression `expr` and binds its value (as a promise) to a variable `x`. The value of the future is automatically retrieved when the assigned variable (promise) is queried. The future itself is returned invisibly, e.g. `f <-futureAssign("x",expr)` and `f <-(x %<-% expr)`. Alternatively, the future of a future variable `x` can be retrieved without blocking using `f <-futureOf(x)`. Both the future and the variable (promise) are assigned to environment `assign.env` where the name of the future is .future_<name>.

`f <-futureCall(FUN,args)` creates a Future `f` that calls function `FUN` with arguments `args`, where the value of the future is retrieved using `x <-value(f)`.

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.*

For future assignments, lazy evaluation can be controlled via the %lazy% operator, e.g. `x %<-% { expr } %lazy% TRUE`.

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 the 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 specify also 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 is made sure 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`.

When using future assignments, globals can be specified analogously using the `%globals%` operator, e.g.

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

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 given by the *future strategy*, which can be set by the end user using the `plan()` function. The future strategy must not be set by the developer, e.g. it must not be called within a package.

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)
```

`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 exists, then an informative error is thrown, otherwise NA 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 are 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.

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

Description

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

Usage

```
multicore(
  ...,
  workers = availableCores(constraints = "multicore"),
  envir = parent.frame()
)
```

Arguments

`...`	Additional arguments passed to `Future()`.
`workers`	A positive numeric scalar or a function specifying the maximum number of parallel futures that can be active at the same time before blocking. 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 a numeric scalar.
`envir`	The environment from where global objects should be identified.

Details

This function will block if all cores are occupied and will be unblocked as soon as one of the already running multicore futures is resolved. For the total number of cores available including the current/main R process, see `availableCores()`.

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 `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.

The preferred way to create an multicore future is not to call this function directly, but to register it via `plan(multicore)` such that it becomes the default mechanism for all futures. After this `future()` and `%<-%` will create *multicore futures*.

Value

A MulticoreFuture If `workers == 1`, then all processing using done in the current/main R session and we therefore fall back to using an sequential future. This is also the case whenever multicore processing is not supported, e.g. on Windows.

See Also

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

Use `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

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*.

Usage

```
multisession(
  ...,
  workers = availableCores(),
  lazy = FALSE,
  rscript_libs = .libPaths(),
  envir = parent.frame()
)
```

Arguments

`...`	Additional arguments passed to `Future()`.
`workers`	A positive numeric scalar or a function specifying the maximum number of parallel futures that can be active at the same time before blocking. 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 a numeric scalar.
`lazy`	If FALSE (default), the future is resolved eagerly (starting immediately), otherwise not.
`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()`.
`envir`	The environment from where global objects should be identified.

Details

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

The `multisession()` function will block if all available R session are occupied and will be unblocked as soon as one of the already running multisession futures is resolved. 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.

The preferred way to create an multisession future is not to call this function directly, but to register it via `plan(multisession)` such that it becomes the default mechanism for all futures. After this `future()` and `%<-%` will create *multisession futures*.

Value

A MultisessionFuture. If `workers == 1`, then all processing using done in the current/main R session and we therefore fall back to using a lazy future.

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)
```

Arguments

evaluator	A future evaluator function. If NULL (default), the current evaluator as returned by plan() is used.

Value

A positive number in $1, 2, 3, \ldots$. Note, it may also be `+Inf` for certain types of backends.

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 = TRUE,
  .init = TRUE
)
```

Arguments

`strategy`	The evaluation function (or name of it) to use for resolving a future. If NULL, then the current strategy is returned.
`...`	Additional arguments overriding the default arguments of the evaluation function. Which additional arguments are supported depends on what evaluation function is used, e.g. several support argument `workers` but not all. For details, see the individual functions of which some are linked to below.
`substitute`	If TRUE, the `strategy` expression is `substitute()`:d, otherwise not.
`.skip`	(internal) If `TRUE`, then attempts to set a strategy that is the same as what is currently in use, will 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.

Details

The default strategy is `sequential`, but the default can be configured by option '`future.plan`' and, if that is not set, system environment variable `R_FUTURE_PLAN`. To reset the strategy back to the default, use `plan("default")`.

Value

If a new strategy is chosen, then the previous one is returned (invisible), otherwise the current one is returned (visibly).

Implemented evaluation strategies

- `sequential`: Resolves futures sequentially in the current R process.
- `transparent`: Resolves futures sequentially in the current R process and assignments will be done to the calling environment. Early stopping is enabled by default.
- `multisession`: Resolves futures asynchronously (in parallel) in separate R sessions running in the background on the same machine.
- `multicore`: Resolves futures asynchronously (in parallel) in separate *forked* R processes running in the background on the same machine. Not supported on Windows.
- `multiprocess`:(DEPRECATED) If multicore evaluation is supported, that will be used, otherwise multisession evaluation will be used.
- `cluster`: Resolves futures asynchronously (in parallel) in separate R sessions running typically on one or more machines.
- `remote`: Resolves futures asynchronously in a separate R session running on a separate machine, typically on a different network.

Other package may provide additional evaluation strategies. Notably, the **future.batchtools** package implements a type of futures that will be resolved via job schedulers that are typically available on high-performance compute (HPC) clusters, e.g. LSF, Slurm, TORQUE/PBS, Sun Grid Engine, and OpenLava.

To "close" any background workers (e.g. `multisession`), change the plan to something different; `plan(sequential)` is recommended for this.

For package developers

Please refrain from modifying the future strategy 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 strategy within a function, then make sure to undo the changes when exiting the function. This can be done using:

```
oplan <- plan(new_set_of_strategies)
on.exit(plan(oplan), add = TRUE)
[...]
```

This is important because the end-user might have already set the future strategy elsewhere for other purposes and will most likely not known 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 uses futures, try to place any call to `plan()` as far up (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.

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

---

re-exports *Functions Moved to 'parallelly'*

Description

The following function used to be part of **future** but has since been migrated to **parallelly**. The migration started with **future** 1.20.0 (November 2020). They were moved because they are also useful outside of the **future** framework.

Details

- parallelly::as.cluster()
- parallelly::autoStopCluster()
- parallelly::availableCores()
- parallelly::availableWorkers()
- parallelly::makeClusterMPI()
- parallelly::makeClusterPSOCK()
- parallelly::makeNodePSOCK()

- parallelly::supportsMulticore()

For backward-compatible reasons, these functions remain available as exact copies also from this package (as re-exports). For example,

```
cl <- parallelly::makeClusterPSOCK(2)
```

can still be accessed as:

```
cl <- future::makeClusterPSOCK(2)
```

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

Description

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

Usage

```
remote(
  ...,
  workers = NULL,
  revtunnel = TRUE,
  myip = NULL,
  persistent = TRUE,
  envir = parent.frame()
)
```

Arguments

`...`	Additional named elements passed to `Future()`.
`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.
`revtunnel`	If TRUE, reverse SSH tunneling is used for the PSOCK cluster nodes to connect back to the master R process. This avoids the hassle of firewalls, port forwarding and having to know the internal / public IP address of the master R session.
`myip`	The external IP address of this machine. If NULL, then it is inferred using an online service (default).
`persistent`	If FALSE, the evaluation environment is cleared from objects prior to the evaluation of the future.
`envir`	The environment from where global objects should be identified.

Value

A ClusterFuture.

'remote' versus 'cluster'

The `remote` plan is a very similar to the `cluster` plan, but provides more convenient default argument values when connecting to remote machines. Specifically, `remote` uses `persistent = TRUE` by default, and it sets `homogeneous`, `revtunnel`, and `myip` "wisely" depending on the value of `workers`. See below for example on how `remote` and `cluster` are related.

Examples

```
## Not run: \donttest{

## Use a remote machine
plan(remote, workers = "remote.server.org")

## Evaluate expression remotely
host %<-% { Sys.info()["nodename"] }
host
[1] "remote.server.org"


## The following setups are equivalent:
plan(remote, workers = "localhost")
plan(cluster, workers = "localhost", persistent = TRUE)
plan(cluster, workers = 1L, persistent = TRUE)
plan(multisession, workers = 1L, persistent = TRUE)

## The following setups are equivalent:
plan(remote, workers = "remote.server.org")
plan(cluster, workers = "remote.server.org", persistent = TRUE, homogeneous = FALSE)

## The following setups are equivalent:
cl <- makeClusterPSOCK("remote.server.org")
plan(remote, workers = cl)
plan(cluster, workers = cl, persistent = TRUE)

}
## End(Not run)
```

`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 = 1,
  value = result,
  ...
)
```

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.
stdout	(internal) If TRUE, captured standard output is relayed, otherwise note.
signal	(internal) If TRUE, captured conditions are relayed, otherwise not.
force	(internal) If TRUE, captured standard output and captured conditions already relayed is relayed again, otherwise not.
sleep	Number of seconds to wait before checking if futures have been resolved since last time.
value	(DEPRECATED) Use argument result instead.
...	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 *Check whether a future is resolved or not*

Description

Check whether a future is resolved or not

Usage

```
resolved(x, ...)
```

Arguments

x	A Future, a list, or an environment (which also includes list environment.
...	Not used.

Details

This method needs to be implemented by the class that implement the Future API. The implementation should return either TRUE or FALSE and must never throw an error (except for FutureError:s which indicate significant, often unrecoverable infrastructure problems). It should also be possible to use the method for polling the future until it is resolved (without having to wait infinitely long), e.g. `while (!resolved(future)) Sys.sleep(5)`.

Value

A logical 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.

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

Description

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.

Usage

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

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

Arguments

...	Additional arguments passed to Future().
envir	The environment from where global objects should be identified.

Details

The preferred way to create a sequential future is not to call these functions directly, but to register them via plan(sequential) such that it becomes the default mechanism for all futures. After this future() and %<-% will create *sequential futures*.

Value

A SequentialFuture.

transparent futures

Transparent futures are sequential futures configured to emulate how R evaluates expressions as far as possible. For instance, errors and warnings are signaled immediately and assignments are done to the calling environment (without local() as default for all other types of futures). This makes transparent futures ideal for troubleshooting, especially when there are errors.

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)
```

signalConditions *Signals Captured Conditions*

Description

Captured conditions that meet the `include` and `exclude` requirements are signaled *in the order as they were captured*.

Usage

```
signalConditions(
  future,
  include = "condition",
  exclude = NULL,
  resignal = TRUE,
  ...
)
```

Arguments

`future`	A resolved Future.
`include`	A character string of condition classes to signal.
`exclude`	A character string of condition classes *not* to signal.
`resignal`	If TRUE, then already signaled conditions are signaled again, otherwise not.
`...`	Not used.

Value

Returns the Future where conditioned that were signaled have been flagged to have been signaled.

See Also

Conditions are signaled by `signalCondition()`.

tweak *Tweak a future function by adjusting its default arguments*

Description

Tweak a future function by adjusting its default arguments

Usage

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

Arguments

strategy	An existing future function or the name of one.
...	Named arguments to replace the defaults of existing arguments.
penvir	The environment used when searching for a future function by its name.

Value

a future function.

See Also

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

---

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 is unresolved, then this function blocks until all queried futures are resolved.

Usage

```
value(...)

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

## S3 method for class 'list'
value(x, stdout = TRUE, signal = TRUE, ...)

## S3 method for class 'listenv'
value(x, stdout = TRUE, signal = TRUE, ...)

## S3 method for class 'environment'
value(x, stdout = TRUE, signal = TRUE, ...)
```

Arguments

...	All arguments used by the S3 methods.
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.

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 produced.

%conditions% *Control whether standard output should be captured or not*

Description

Control whether standard output should be captured or not

Usage

```
fassignment %conditions% capture
```

Arguments

`fassignment`	The future assignment, e.g. `x %<-% { expr }`.
`capture`	If TRUE, the standard output will be captured, otherwise not.

%globals% *Specify globals and packages for a future assignment*

Description

Specify globals and packages for a future assignment

Usage

```
fassignment %globals% globals
fassignment %packages% packages
```

Arguments

`fassignment`	The future assignment, e.g. `x %<-% { expr }`.
`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.

%label% *Specify label for a future assignment*

Description

Specify label for a future assignment

Usage

```
fassignment %label% label
```

Arguments

fassignment	The future assignment, e.g. x %<-% { expr }.
label	An optional character string label attached to the future.

%lazy% *Control lazy / eager evaluation for a future assignment*

Description

Control lazy / eager evaluation for a future assignment

Usage

```
fassignment %lazy% lazy
```

Arguments

fassignment	The future assignment, e.g. x %<-% { expr }.
lazy	If FALSE (default), the future is resolved eagerly (starting immediately), otherwise not.

%plan% *Use a specific plan for a future assignment*

Description

Use a specific plan for a future assignment

Usage

```
fassignment %plan% strategy
```

Arguments

fassignment	The future assignment, e.g. x %<-% { expr }.
strategy	The mechanism for how the future should be resolved. See plan() for further details.

See Also

The plan() function sets the default plan for all futures.

%seed% *Set random seed for future assignment*

Description

Set random seed for future assignment

Usage

```
fassignment %seed% seed
```

Arguments

fassignment	The future assignment, e.g. x %<-% { expr }.
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 does neither need nor use random numbers generation. To use a fixed random seed, specify a L'Ecuyer-CMRG seed (seven integer) or a regular RNG seed (a single integer). 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.

%stdout% *Control whether standard output should be captured or not*

Description

Control whether standard output should be captured or not

Usage

```
fassignment %stdout% capture
```

Arguments

fassignment	The future assignment, e.g. x %<-% { expr }.
capture	If TRUE, the standard output will be captured, otherwise not.

%tweak% *Temporarily tweaks the arguments of the current strategy*

Description

Temporarily tweaks the arguments of the current strategy

Usage

```
fassignment %tweak% tweaks
```

Arguments

fassignment	The future assignment, e.g. x %<-% { expr }.
tweaks	A named list (or vector) with arguments that should be changed relative to the current strategy.

Index