Package 'future'

July 14, 2015

Version 0.7.0

Title A Future API for R

Imports listenv, globals

Suggests parallel, R.rsp

VignetteBuilder R.rsp

Description A Future API for R is provided. In programming, a future is an abstraction for a value that may be available at some point in the future. The state of a future can either be unresolved or resolved. As soon as it is resolved, the value is available. Futures are useful constructs in for instance concurrent evaluation, e.g. multicore parallel processing and distributed processing on compute clusters. The purpose of this package is to provide a lightweight interface for using futures in R. Functions 'future()' and 'value()' exist for creating futures and requesting their values. An infix assignment operator '%<=%' exists for creating futures whose values are accessible by the assigned variables (as promises). This package implements the synchronous ``lazy'' and ``eager'' futures, and the asynchronous ``multicore'' future (not on Windows). Additional types of futures are provided by other packages enhancing this package.

License LGPL (>= 2.1)

LazyLoad TRUE

URL https://github.com/HenrikBengtsson/future

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

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph]

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2015-07-14 20:40:36

R topics documented:

`eager` *Create an eager future whose value will be resolved immediately*

Description

An eager future is a future that uses eager evaluation, which means that its *value is computed and resolved immediately*, which is how regular expressions are evaluated in R. This type of future exists mainly for the purpose of troubleshooting code that fails with other types of futures.

Usage

```
eager(expr, envir = parent.frame(), substitute = TRUE, local = TRUE, ...)
```

Arguments

`expr`	An R expression.
`envir`	The environment in which the evaluation is done (or inherits from if `local` is TRUE).
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`local`	If TRUE, the expression is evaluated such that all assignments are done to local temporary environment, otherwise the assignments are done in the calling environment.
`...`	Not used.

Details

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

Value

An EagerFuture.

Examples

```
## A global variable
a <- 0

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

## Since 'a' is a global variable in _eager_ future 'f',
## it already has been resolved, 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)
```

`future` *Create a future*

Description

Creates a future from an expression and returns it. The state of the future is either unresolved or resolved. When it becomes resolved, at some point in the future, its value can be retrieved.

Usage

```
future(expr, envir = parent.frame(), substitute = TRUE, ...,
  evaluator = plan())
```

Arguments

`expr`	An R expression.
`envir`	The environment from where global objects should be identified. Depending on "evaluator", it may also be the environment in which the expression is evaluated.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`...`	Additional arguments passed to the "evaluator".
`evaluator`	The actual function that evaluates `expr` and returns a future. The evaluator function should accept all the same arguments as this function (except `evaluator`).

Value

A Future.

See Also

It is recommended that the evaluator is *non-blocking* (returns immediately), but it is not required. The default evaluator function is `eager()`, but this can be changed via `plan()` function.

Examples

```
plan(lazy)

f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
})

print(resolved(f))
y <- value(f)
print(y)
```

`Future-class` *A future represents a value that will be available at some point in the future*

Description

A *future* is an abstraction for a *value* that may available at some point in the future. A future can either be `unresolved` or `resolved`, a state which can be checked with `resolved()`. As long as it is *unresolved*, the value is not available. As soon as it is *resolved*, the value is available via `value()`.

Usage

```
Future(expr = NULL, envir = parent.frame(), substitute = FALSE, ...)
```

Arguments

`expr`	An R expression.
`envir`	The environment in which the evaluation is done (or inherits from if `local` is TRUE).
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`...`	Additional named elements of the future.

Details

A Future object is itself an environment.

Value

An object of class `Future`.

See Also

One function that creates a Future is `future()`. It returns a Future that evaluates an R expression in the future. An alternative approach is to use the `%<=%` infix assignment operator, which creates a future from the right-hand-side (RHS) R expression and assigns its future value to a variable as a *promise*.

`futureAssign` *Create a future and assign its value to a variable as a promise*

Description

Method and infix operators for creating futures and assigning their values as variables using promises. Trying to access such a "future variable" will correspond to requesting the value of the underlying future. If the the future is already resolved at this time, then the value will be available instantaneously and the future variable will appear as any other variable. If the future is unresolved, then the current process will block until the future is resolved and the value is available.

Usage

```
futureAssign(name, value, envir = parent.frame(), assign.env = envir,
  substitute = TRUE)
```

Arguments

`name`	the name of the variable (and the future) to assign.
`value`	the expression to be evaluated in the future and whose value will be assigned to the variable.
`envir`	The environment from which global variables used by the expression should be search for.
`assign.env`	The environment to which the variable should be assigned.
`substitute`	Controls whether `expr` should be `substitute()`:d or not.

Details

This function creates a future and a corresponding *promise*, which hold the future's value. Both the future and the promise are assigned to environment `assign.env`. The name of the promise is given by `name` and the name of the future is `.future_<name>`. The future is also returned invisibly.

Value

A Future invisibly.

See Also

The `futureOf()` function can be used to get the Future object of a future variable.

Examples

```
## Future assignment via "assign" function
futureAssign("A", {
  x <- 3
  x^2
})
message("Value 'A': ", A)


## Equivalent via infix "assign" operator
A %<=% {
  x <- 3
  x^2
}
message("Value 'A': ", A)


## A global variable
a <- 1

## Three future evaluations
A %<=% { 0.1 }
B %<=% { 0.2 }
C %<=% { z <- a+0.3 }

## Sleep until 'C' is available
message("Value 'C': ", C)

## Sleep until 'A' is available
message("Value 'A': ", A)

## Sleep until 'C' is available
message("Value 'B': ", B)
```

---

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_character_)
```

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.

Value

A Future (or NA). 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)
```

`lazy` *Create a lazy future whose value will be resolved at the time when requested*

Description

A lazy future is a future that uses lazy evaluation, which means that its *value is only computed and resolved at the time when the value is requested*. This means that the future will not be resolved if the value is never requested.

Usage

```
lazy(expr, envir = parent.frame(), substitute = TRUE, globals = TRUE,
  local = TRUE, ...)
```

Arguments

`expr`	An R expression.
`envir`	The environment in which the evaluation is done (or inherits from if `local` is TRUE) and from which globals are obtained.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`globals`	If TRUE, global objects are resolved ("frozen") at the point of time when the future is created, otherwise they are resolved when the future is resolved.
`local`	If TRUE, the expression is evaluated such that all assignments are done to local temporary environment, otherwise the assignments are done in the calling environment.
`...`	Not used.

Details

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

Value

A LazyFuture.

Examples

```
## A global variable
a <- 0

## Create lazy future (explicitly)
f <- lazy({
  b <- 3
  c <- 2
```

```
  a * b * c
})

## Although 'f' is a _lazy_ future and therefore
## resolved/evaluates the future expression only
## when the value is requested, any global variables
## identified in the expression (here 'a') are
## "frozen" at the time point when the future is
## created.  Because of this, the 'a' in the
## the future expression preserved the zero value
## although we reassign it in the global environment
a <- 7
print(a)

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

`multicore` *Create a multicore future whose value will be resolved asynchroneously in a 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(expr, envir = parent.frame(), substitute = TRUE,
  maxCores = availableCores(), ...)
```

Arguments

`expr`	An R expression.
`envir`	The environment in which the evaluation is done and from which globals are obtained.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`maxCores`	The maximum number of CPU cores that can be active at the same time before blocking.
`...`	Not used.

Details

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

Not all systems support multicore futures. For instance, it is not supported on Microsoft Windows. Trying to create multicore futures on non-supported systems will silently fall back to using eager futures, which effectively corresponds to a multicore future that can handle one parallel process (the current one) before blocking.

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 (or a EagerFuture if multicore futures are not supported).

See Also

`availableCores() > 1L` to check whether multicore futures are supported or not.

Examples

```
## A global variable
a <- 0

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

## A multicore future is evaluated in a separated
## 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)
```

---

plan *Plan how to resolve a future*

Description

This function allows you to plan the future, more specifically, it specifies how `future()`:s are resolved, e.g. by eager or by lazy evaluation.

Usage

```
plan(strategy = NULL, ..., substitute = TRUE, .call = TRUE)
```

Arguments

strategy	The evaluation function to use for resolving a future. If NULL, then the current strategy is returned.
...	Additional arguments overriding the default arguments of the evaluation function.
substitute	If TRUE, the strategy expression is substitute():d, otherwise not.
.call	(internal) Used to record the call to this function.

Value

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

See Also

Evaluation functions provided by this package are eager(), lazy() and multicore(). Other package may provide additional evaluation strategies/functions.

Examples

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

# A lazy future (evaluated in a local environment)
plan(lazy)
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


# An eager future (evaluated in a local environment)
plan(eager)
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 multicore future evaluated in a local environment
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


# An eager future evaluated in the global environment
plan(eager, local=FALSE)
f <- future({
  a <- 7
  b <- 3
  c <- 2
  a * b * c
})
y <- value(f)
print(y)
str(list(a=a, b=b, c=c)) ## Assigned the new values
```

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

Description

Check whether a future is resolved or not

Usage

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

Arguments

`future`	A Future.
`...`	Not used

Details

This method needs to be implemented by the class that implement the Future API. The implementation must never throw an error, but only return either TRUE or FALSE. It should also be possible to use the method for polling the future until it is resolved (without having to wait infinitly long), e.g. `while (!resolved(future)) Sys.sleep(5)`.

Value

TRUE if future is resolved and FALSE if unresolved.

`supportsMulticore` *Check whether multicore processing is supported or not*

Description

Multicore futures are only supported on systems supporting multicore processing. R supports this on most systems, except on the Microsoft Windows.

Usage

```
supportsMulticore()
```

Value

TRUE if multicore processing is supported, otherwise FALSE.

See Also

To use multicore futures, use `plan(multicore)`.

`value.Future` *The value of a future*

Description

Gets the value of a future. If the future is unresolved, then the evaluation blocks until the future is resolved.

Usage

```
## S3 method for class 'Future'
value(future, onError = c("signal", "return"), ...)
```

Arguments

`future`	A Future.
`onError`	A character string specifying how errors (conditions) should be handled in case they occur. If `"signal"`, the error is signalled, e.g. captured and re-thrown. If instead `"return"`, they are *returned* as is.
`...`	Not used.

Details

This method needs to be implemented by the class that implement the Future API.

Value

An R object of any data type.

`%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.

Index