Package 'JuliaConnectoR'

August 31, 2026

Type Package

Title A Functionally Oriented Interface for Integrating 'Julia' with R

Version 1.1.6

Maintainer Stefan Lenz <stefan-m-lenz@web.de>

Description Allows to import functions and whole packages from 'Julia' in R.
Imported 'Julia' functions can directly be called as R functions.
Data structures can be translated between 'Julia' and R.
More details can also be found in the corresponding article
<doi:10.18637/jss.v101.i06>.

License MIT + file LICENCE

URL https://github.com/stefan-m-lenz/JuliaConnectoR

BugReports https://github.com/stefan-m-lenz/JuliaConnectoR/issues

SystemRequirements Julia >= 1.0

Encoding UTF-8

RoxygenNote 7.3.3

Suggests testthat (>= 2.1.0), future, parallel, parallelly

NeedsCompilation no

Author Stefan Lenz [aut, cre] (ORCID: <https://orcid.org/0000-0001-9135-1743>),
Harald Binder [aut, ths] (ORCID:
<https://orcid.org/0000-0002-5666-8662>),
Angelo D'Ambrosio [ctb] (ORCID:
<https://orcid.org/0000-0002-2045-5155>),
June Choe [ctb] (ORCID: <https://orcid.org/0000-0002-0701-921X>),
Bjarke Hautop Kristensen [ctb] (ORCID:
<https://orcid.org/0009-0007-3733-1769>)

Repository CRAN

Date/Publication 2026-08-31 19:31:10 UTC

Contents

JuliaConnectoR-package . . . 2
AccessMutate.JuliaProxy . . . 5
as.data.frame.JuliaProxy . . . 8
EnvVars-JuliaConnectoR . . . 9
Julia-Setup . . . 10
juliaCall . . . 10
juliaEval . . . 11
juliaExpr . . . 12
juliaFun . . . 13
juliaGet . . . 13
juliaImport . . . 14
juliaLet . . . 16
juliaPut . . . 17
juliaSetupOk . . . 18
startJuliaServer . . . 18
stopJulia . . . 20

Index **21**

`JuliaConnectoR-package`

A Functionally Oriented Interface for Integrating Julia with R

Description

This package provides a functionally oriented interface between R and Julia. The goal is to call functions from Julia packages directly as R functions.

Details

This R-package provides a functionally oriented interface between R and Julia. The goal is to call functions from Julia packages directly as R functions. Julia functions imported via the **JuliaConnectoR** can accept and return R variables. It is also possible to pass R functions as arguments in place of Julia functions, which allows *callbacks* from Julia to R.

From a technical perspective, R data structures are serialized with an optimized custom streaming format, sent to a (local) Julia TCP server, and translated to Julia data structures by Julia. The results are returned back to R. Simple objects, which correspond to vectors in R, are directly translated. Complex Julia structures are by default transferred to R by reference via proxy objects. This enables an effective and intuitive handling of the Julia objects via R. It is also possible to fully translate Julia objects to R objects. These translated objects are annotated with information about the original Julia objects, such that they can be translated back to Julia. This makes it also possible to serialize them as R objects.

Setup

The package requires that Julia (Version $\geq$ 1.0) is installed separately from the package. The Julia installation is discovered via the system search `PATH` or the `JULIA_BINDIR` environment variable, which can be set to the `bin` directory of the Julia installation. If Julia is installed via the Julia installation manager `juliaup`, it should be discovered without requiring more configuration. For more details about the setup, see `Julia-Setup`.

Function overview

The function `juliaImport` makes functions and data types from Julia packages or modules available as R functions.

If only a single Julia function needs to be imported in R, `juliaFun` can do this. The simplest way to call a Julia function without any importing is to use `juliaCall` with the function name given as character string.

For evaluating expressions in Julia, `juliaEval` and `juliaLet` can be used. With `juliaLet` one can use R variables in a expression.

`juliaExpr` makes it possible use complex Julia syntax in R via R strings that contain Julia expressions.

With `juliaGet`, a full translation of a Julia proxy object into an R object is performed.

`as.data.frame` is overloaded (`as.data.frame.JuliaProxy`) for translating Julia objects that implement the `Tables` interface to R data frames.

Translation

Since Julia is more type-sensitive than R, and many Julia functions expect to be called using specific types, it is important to know the translations of the R data structures to Julia.

Translation from R to Julia: The type correspondences of the basic R data types in Julia are the following:

R		Julia
`integer`	→	`Int`
`double`	→	`Float64`
`logical`	→	`Bool`
`character`	→	`String`
`complex`	→	`Complex{Float64}`
`raw`	→	`UInt8`
`symbol`	→	`Symbol`

R vectors of length 1 of the types in the table above will be translated to the types shown.

R vectors or arrays with more than one element will be translated to Julia `Arrays` of the corresponding types. The dimensions of an R array, as returned by `dim()`, will also be respected. For example, the R integer vector `c(1L, 2L)` will be of type `Vector{Int}`, or `Array{Int,1}`, in Julia. A double matrix such as `matrix(c(1,2,3,4), nrow = 2)` will be of type `Array{Float64,2}`.

Missing values (NA) in R are translated to missing values in Julia. R vectors and arrays with missing values are converted to Julia arrays of type Array{Union{Missing, T}}, where T stands for the translated type in the table above.

R lists are translated as Vector{T} in Julia, with T being the most specific supertype of the list elements after translation to Julia.

An R function that is handed to Julia as argument in a function call is translated to a Julia callback function that will call the given R function.

Strings with attribute "JLEXPR" will be evaluated as Julia expressions, and the value is used in their place (see juliaExpr).

R data frames are translated to objects that implement the Julia Tables interface. Such objects can be used by functions of many different Julia packages that deal with table-like data structures.

Translation from Julia to R: The type system of Julia is richer than that of R. Therefore, to be able to turn the Julia data structures that have been translated to R back to the original Julia data structures, the original Julia types are added to the translated Julia objects in R via the attribute "JLTYPE". When passed to Julia, R variables with this attribute will be coerced to the respective type. This allows the reconstruction of the objects with their original type.

It should not be necessary to worry too much about the translations from Julia to R because the resulting R objects should be intuitive to handle.

The following table shows how basic R-compatible types of Julia are translated to R:

Julia		**R**
Float64	$\rightarrow$	double
Float16, Float32, UInt32	$\rightarrow$	double with type attribute
Int64 that fits in 32 bits	$\rightarrow$	integer
Int64 not fitting in 32 bits	$\rightarrow$	double with type attribute
Int8, Int16, UInt16, Int32, Char	$\rightarrow$	integer with type attribute
UInt8	$\rightarrow$	raw
UInt64, Int128, UInt128, Ptr	$\rightarrow$	raw with type attribute
Complex{Float64}	$\rightarrow$	complex
Complex{Int$X$} with $X \leq 64$	$\rightarrow$	complex with type attribute
Complex{Float$X$} with $X \leq 32$	$\rightarrow$	complex with type attribute

Julia Arrays of these types are translated to vectors or arrays of the corresponding types in R.

Julia functions are translated to R functions that call the Julia function. These functions can also be translated back to the corresponding Julia functions when used as argument of another function (see juliaFun).

Julia object of other types, in particular structs, Tuples, NamedTuples, and AbstractArrays of other types are transferred by reference in the form of proxy objects. Elements and properties of these proxy objects can be accessed and mutated via the operators `[[`, `[`, and `$` (see AccessMutate.JuliaProxy).

A full translation of the proxy objects into R objects, which also allows saving these objects in R, is possible via juliaGet.

Limitations

Possible inexactness when dealing with large 64 bit integers: Numbers of type Int64 that are too big to be expressed as 32-bit integer values in R will be translated to double numbers. This

may lead to a inaccurate results for very large numbers, when they are translated back to Julia, since, e. g., `(2^53 + 1) - 2^53 == 0` holds for double-precision floating point numbers.

Non-ASCII characters in variable names: Julia uses UTF-8 as default string encoding everywhere. In particular, Julia permits characters that are not expressible in encodings such as "Latin-1" in variable and function names. In R, the encoding of names in lists of environments depends on the platform. On locales without UTF-8 as native encoding, (i.e., mostly Windows), unexpected translations may happen when using UTF-8 characters in strings.

When using `juliaImport` for importing packages/modules, alternative names for variables using non-ASCII characters are added, which are compatible across different encodings. (For more information, see `juliaImport`.)

In other places, such as when evaluating code via `juliaEval` and `juliaLet`, the problem cannot be addressed. It should therefore be avoided to use non-ASCII characters if code should be portable across different platforms.

Author(s)

Maintainer: Stefan Lenz <stefan-m-lenz@web.de> (ORCID)

Authors:

- Harald Binder (ORCID) [thesis advisor]

Other contributors:

- Angelo D'Ambrosio (ORCID) [contributor]
- June Choe (ORCID) [contributor]
- Bjarke Hautop Kristensen (ORCID) [contributor]

See Also

Useful links:

- https://github.com/stefan-m-lenz/JuliaConnectoR
- Report bugs at https://github.com/stefan-m-lenz/JuliaConnectoR/issues

---

`AccessMutate.JuliaProxy` *Access or mutate Julia objects via proxy objects*

Description

Apply the R operators `$` and `$<-`, `[` and `[<-`, `[[` and `[[<-` to access or modify parts of Julia objects via their proxy objects. For an intuitive understanding, best see the examples below.

Usage

```
## S3 method for class 'JuliaStructProxy'
x$name

## S3 replacement method for class 'JuliaStructProxy'
x$name <- value

## S3 method for class 'JuliaProxy'
x[...]

## S3 replacement method for class 'JuliaProxy'
x[i, j, k] <- value

## S3 method for class 'JuliaSimpleArrayProxy'
x[...]

## S3 method for class 'JuliaArrayProxy'
x[[...]]

## S3 replacement method for class 'JuliaArrayProxy'
x[[i, j, k]] <- value

## S3 method for class 'JuliaStructProxy'
x[[name]]

## S3 replacement method for class 'JuliaStructProxy'
x[[name]] <- value

## S3 method for class 'JuliaArrayProxy'
length(x)

## S3 method for class 'JuliaArrayProxy'
dim(x)
```

Arguments

`x`	a Julia proxy object
`name`	the field of a struct type, the name of a member in a `NamedTuple`, or a key in a Julia dictionary (type `AbstractDict`)
`value`	a suitable replacement value. When replacing a range of elements in an array type, it is possible to replace multiple elements with single elements. In all other cases, the length of the replacement must match the number of elements to replace.
`i`, `j`, `k`, ...	index(es) for specifying the elements to extract or replace

Details

The operators `$` and `[[` allow to access properties of Julia `structs` and `NamedTuples` via their proxy objects. For dictionaries (Julia type `AbstractDict`), `$` and `[[` can also be used to look up string keys. Fields of `mutable structs` and dictionary elements with string keys can be set via `$<-` and `[[<-`.

For `AbstractArrays`, the `[`, `[<-`, `[[`, and `[[<-` operators relay to the `getindex` and `setindex!` Julia functions. The `[[` and `[[<-` operators are used to access or mutate a single element. With `[` and `[<-`, a range of objects is accessed or mutated. The elements of `Tuples` can also be accessed via `[` and `[[`.

The dimensions of proxy objects for Julia `AbstractArrays` and `Tuples` can be queried via `length` and `dim`.

Examples

```
## Not run: #'
# (Mutable) struct
juliaEval("mutable struct MyStruct
             x::Int
           end")

MyStruct <- juliaFun("MyStruct")
s <- MyStruct(1L)
s$x
s$x <- 2
s[["x"]]

# Array
x <- juliaCall("map", MyStruct, c(1L, 2L, 3L))
x
length(x)
x[[1]]
x[[1]]$x
x[[1]] <- MyStruct(2L)
x[2:3]
x[2:3] <- MyStruct(2L)
x

# Tuple
x <- juliaEval("(1, 2, 3)")
x[[1]]
x[1:2]
length(x)

# NamedTuple
x <- juliaEval("(a=1, b=2)")
x$a

# Dictionary
strDict <- juliaEval('Dict("hi" => 1, "hello" => 2)')
strDict
strDict$hi
```

```
strDict$hi <- 0
strDict[["hi"]] <- 2
strDict["howdy", "greetings"] <- c(2, 3)
strDict["hi", "howdy"]


## End(Not run)
```

`as.data.frame.JuliaProxy` *Coerce a Julia Table to a Data Frame*

Description

Get the data from a Julia proxy object that implements the Julia `Tables` interface, and create an R data frame from it.

Usage

```
## S3 method for class 'JuliaProxy'
as.data.frame(x, ...)
```

Arguments

`x`	a proxy object pointing to a Julia object that implements the interface of the package Julia package `Tables`
`...`	(not used)

Details

Strings are not converted to factors.

Examples

```
if (Sys.getenv("NOT_CRAN") == "true" && juliaSetupOk()) {
   # (This example is not run on CRAN as it takes a little too long.)

   # Demonstrate the usage with the Julia package "IndexedTables" (v1.0)

   # Install the package first if it is not installed:
   # juliaEval('import Pkg; Pkg.add("IndexedTables")')

   # Import "IndexedTables" package
   IndexedTables <- juliaImport("IndexedTables")

   mydf <- data.frame(x = c(1, 2, 3),
                      y = c("a", "b", "c"),
                      z = c(TRUE, FALSE, NA),
                      stringsAsFactors = FALSE)
```

```
    # Create a table in Julia, e. g. via IndexedTables
    mytbl <- IndexedTables$table(mydf)

    # This table can, e g. be queried and
    # the result can be translated to an R data frame.
    seltbl <- IndexedTables$select(mytbl, juliaExpr("(:x, :y)"))[1:2]

    # Translate selection of Julia table into R data frame
    as.data.frame(seltbl)

}
```

EnvVars-JuliaConnectoR

Environment variables used by the **JuliaConnectoR**

Description

There are some environment variables which can be used to deviate from the default behavior of the package. To have an effect, these environment variables must be set before a Julia connection is established, i.e., before the first call to Julia or before a call to `startJuliaServer`. All the variables are optional.

Details

The environment variables that are used in the package are listed below:

`JULIA_BINDIR`**:** If this variable is set to the path of the Julia `bin` directory before connecting to Julia, the corresponding Julia installation will be used. By using this variable, it is possible to use a different Julia version than the one in the system `PATH`. (You can find the correct path to the `bin` directory of Julia by evaluating the expression `Sys.BINDIR` within Julia.)

`JULIACONNECTOR_JULIAENV`**:** Specify environment variables only for Julia. (This does not work on Windows and the variable is ignored there.) This allows, e.g., to set the LD_LIBRARY_PATH variable to a different value for Julia than for R. The value can be any R code that defines variables, e.g., `"LD_LIBRARY_PATH=''"` is a valid value. On Linux, Julia is started with an empty `LD_LIBRARY_PATH` by default as the `LD_LIBRARY_PATH` required by R may be incompatible with Julia. If the `LD_LIBRARY_PATH` needs to be set to a different value, this can be done via the `JULIACONNECTOR_JULIAENV` variable.

`JULIACONNECTOR_JULIAOPTS`**:** Set start-up options for Julia. As an example, consider specifying the project environment and enabling code coverage when starting Julia. This can be achieved by setting the environment variable to `"--project=/path/to/project --code-coverage"`.

`JULIACONNECTOR_SERVER`**:** Specifies the server address of a (running) Julia server that the R process can connect to. A possible example value is "localhost:11980", specifying host and port. The function `startJuliaServer` sets this variable and communicates the location of

the server to child processes with it. Due to security concerns, the Julia server accepts only connections from the same machine and connecting to remote machines is currently not possible.

`Julia-Setup` *Julia setup*

Description

Julia must be installed separately in order for the **JuliaConnectoR** package to work. You can download and install Julia from https://julialang.org/downloads/.

Details

Setup via the Juliaup installation manager: If you have installed Julia via Juliaup, the Julia installation should be discovered by the **JuliaConnectoR**.

Juliaup on Windows: If you have freshly installed Juliaup, start Julia once on the command line. This will do the actual installation of the current Julia version. Juliaup puts the Julia executable on the system `PATH`. This way, the Julia installation can be detected by the **JuliaConnectoR**.

Juliaup on Mac: After the installation of Juliaup, Julia might not be on the system `PATH` but it should be discovered automatically if it is installed in the default location, i.e., the `.juliaup` folder in your home directory.

Setup via Julia binaries: If you have installed Julia via a binary package or any other method, the simplest way to make Julia discoverable is by adding the directory containing the Julia executable to the `PATH` environment variable.

Alternatively, you can set the `JULIA_BINDIR` environment variable to specify the exact directory containing the Julia binary. (You can find the correct path to this directory by evaluating the expression `Sys.BINDIR` within Julia.)

If the `JULIA_BINDIR` variable is set, it takes precedence over looking up the system `PATH`. This makes it easy to use a different Julia version than the one in your system `PATH`.

`juliaCall` *Call a Julia function by name*

Description

Call a Julia function via specifying the name as string and get the translated result. It is also possible to use a dot at the end of the function name for applying the function in a vectorized manner via "broadcasting" in Julia.

Usage

```
juliaCall(...)
```

Arguments

...	the name of the Julia function as first argument, followed by the parameters handed to the function. All arguments to the Julia function are translated to Julia data structures.

Value

The value returned from Julia, translated to an R data structure. If Julia returns `nothing`, an invisible `NULL` is returned.

Examples

```
if (juliaSetupOk()) {

   juliaCall("/", 4, 2)
   juliaCall("Base.div", 4, 2)
   juliaCall("sin.", c(1,2,3))
   juliaCall("Base.cos.", c(1,2,3))

}
```

---

juliaEval *Evaluate a Julia expression*

Description

This function evaluates Julia code, given as a string, in Julia, and translates the result back to R.

Usage

```
juliaEval(expr)
```

Arguments

expr	Julia code, given as a one-element character vector

Details

If the code needs to use R variables, consider using `juliaLet` instead.

Value

The value returned from Julia, translated to an R data structure. If Julia returns `nothing`, an invisible `NULL` is returned. This is also the case if the last non-whitespace character of `expr` is a semicolon.

Examples

```
if (juliaSetupOk()) {
   juliaEval("1 + 2")
   juliaEval('using Random; Random.seed!(5);')

## Not run:
   juliaEval('using Pkg; Pkg.add("BoltzmannMachines")')

## End(Not run)
}
```

`juliaExpr` *Mark a string as Julia expression*

Description

A given R character vector is marked as a Julia expression. It will be executed and evaluated when passed to Julia. This allows to pass a Julia object that is defined by complex Julia syntax as an argument without needing the round-trip to R via `juliaEval` or `juliaLet`.

Usage

```
juliaExpr(expr)
```

Arguments

`expr`	a character vector which should contain one string

Examples

```
if (juliaSetupOk()) {

   # Create complicated objects like version strings in Julia, and compare them
   v1 <- juliaExpr('v"1.0.1"')
   v2 <- juliaExpr('v"1.2.0"')
   juliaCall("<", v1, v2)

}
```

juliaFun *Wrap a Julia function in an R function*

Description

Creates an R function that will call the Julia function with the given name when it is called. Like any R function, the returned function can also be passed as a function argument to Julia functions.

Usage

```
juliaFun(name, ...)
```

Arguments

name	the name of the Julia function
...	optional arguments for currying: The resulting function will be called using these arguments.

Examples

```
if (juliaSetupOk()) {

   # Wrap a Julia function and use it
   juliaSqrt <- juliaFun("sqrt")
   juliaSqrt(2)
   # In the following call, the sqrt function is called without
   # a callback to R because the linked function object is used.
   juliaCall("map", juliaSqrt, c(1,4,9))

   # may also be used with arguments
   plus1 <- juliaFun("+", 1)
   plus1(2)
   # Results in an R callback (calling Julia again)
   # because there is no linked function object in Julia.
   juliaCall("map", plus1, c(1,2,3))

}
```

juliaGet *Translate a Julia proxy object to an R object*

Description

R objects of class JuliaProxy are references to Julia objects in the Julia session. These R objects are also called "proxy objects". With this function it is possible to translate these objects into R objects.

Usage

```
juliaGet(x)
```

Arguments

x	a reference to a Julia object

Details

If the corresponding Julia objects do not contain external references, translated objects can also saved in R and safely be restored in Julia.

Modifying objects is possible and changes in R will be translated back to Julia.

The following table shows the translation of Julia objects into R objects.

Julia		R
`struct`	→	`list` with the named struct elements
`Array` of `struct` type	→	`list` (of `lists`)
`Tuple`	→	`list`
`NamedTuple`	→	`list` with the named elements
`AbstractDict`	→	`list` with two sub-lists: "`keys`" and "`values`"
`AbstractSet`	→	`list`

Note

Objects containing circular references cannot be translated back to Julia.

It is safe to translate objects that contain external references from Julia to R. The pointers will be copied as values and the finalization of the translated Julia objects is prevented. The original objects are garbage collected after all direct or indirect copies are garbage collected. Note, however, that these translated objects cannot be translated back to Julia after the Julia process has been stopped and restarted.

`juliaImport` *Load and import a Julia package via* `import` *statement*

Description

The specified package/module is loaded via `import` in Julia. Its functions and type constructors are wrapped into R functions. The return value is an environment containing all these R functions.

Usage

```
juliaImport(modulePath, all = TRUE)
```

Arguments

modulePath	a module path or a module object. A module path may simply be the name of a package but it may also be a relative module path. Specifying a relative Julia module path like `.MyModule` allows importing a module that does not correspond to a package, but has been loaded in the `Main` module, e. g. by `juliaCall("include", "path/to/MyModule.jl")`. Additionally, via a path such as `SomePkg.SubModule`, a submodule of a package can be imported.
all	`logical` value, default `TRUE`. Specifies whether all functions and types shall be imported or only those exported explicitly.

Value

an environment containing all functions and type constructors from the specified module as R functions

Note

If a package or module contains functions or types with names that contain non-ASCII characters, (additional) alternatives names are provided if there are LaTeX-like names for the characters available in Julia. In the alternative names of the variables, the LaTeX-like names of the characters surrounded by <...> replace the original characters. (See example below.) For writing platform independent code, it is recommended to use those alternative names. (See also JuliaConnectoR-package under "Limitations".)

Examples

```
if (juliaSetupOk()) {

   # Importing a package and using one of its exported functions
   UUIDs <- juliaImport("UUIDs")
   juliaCall("string", UUIDs$uuid4())


   # Importing a module without a package
   testModule <- system.file("examples", "TestModule1.jl",
                             package = "JuliaConnectoR")
   # take a look at the file
   writeLines(readLines(testModule))
   # load in Julia
   juliaCall("include", testModule)
   # import in R via relative module path
   TestModule1 <- juliaImport(".TestModule1")
   TestModule1$test1()

   # Importing a local module is also possible in one line,
   # by directly using the module object returned by "include".
   TestModule1 <- juliaImport(juliaCall("include", testModule))
   TestModule1$test1()
```

```
   # Importing a submodule
   testModule <- system.file("examples", "TestModule1.jl",
                             package = "JuliaConnectoR")
   juliaCall("include", testModule)
   # load sub-module via module path
   SubModule1 <- juliaImport(".TestModule1.SubModule1")
   # call function of submodule
   SubModule1$test2()


   # Functions using non-ASCII characters
   greekModule <- system.file("examples", "GreekModule.jl",
                             package = "JuliaConnectoR")
   suppressWarnings({ # importing gives a warning on non-UTF-8 locales
      GreekModule <- juliaImport(juliaCall("include", greekModule))
   })
   # take a look at the file
   cat(readLines(greekModule, encoding = "UTF-8"), sep = "\n")
   # use alternative names
   GreekModule$`<sigma>`(1)
   GreekModule$`log<sigma>`(1)
}
```

`juliaLet` *Evaluate Julia code in a* `let` *block using values of R variables*

Description

R variables can be passed as named arguments, which are inserted for those variables in the Julia expression that have the same name as the named arguments. The given Julia code is executed in Julia inside a `let` block and the result is translated back to R.

Usage

```
juliaLet(expr, ...)
```

Arguments

`expr`	Julia code, given as one-element character vector
`...`	arguments that will be introduced as variables in the `let` block. The values are transferred to Julia and assigned to the variables introduced in the `let` block.

Details

A simple, nonsensical example for explaining the principle:

```
juliaLet('println(x)', x = 1)
```

This is the same as

`juliaEval('let x = 1.0; println(x) end')`

More complex objects cannot be simply represented in a string like in this simple example any more. That is the problem that `juliaLet` solves.

Note that the evaluation is done in a `let` block. Therefore, changes to global variables in the Julia session are only possible by using the keyword `global` in front of the Julia variables (see examples).

Value

The value returned from Julia, translated to an R data structure. If Julia returns `nothing`, an invisible `NULL` is returned.

Examples

```
if (juliaSetupOk()) {

   # Intended use: Create a complex Julia object
   # using Julia syntax and data from the R workspace
   juliaLet('[1 => x, 17 => y]', x = rnorm(1), y = rnorm(2))

   # Assign a global variable
   # (although not recommended for a functional style)
   juliaLet("global x = xval", xval = rnorm(10))
   juliaEval("x")

}
```

---

juliaPut *Create a Julia proxy object from an R object*

Description

This function can be used to copy R vectors and matrices to Julia and keep them there. The returned proxy object can be used in place of the original vector or matrix. This is useful to prevent that large R vectors / matrices are repeatedly translated when using an object in multiple calls to Julia.

Usage

```
juliaPut(x)
```

Arguments

`x`	an R object (can also be a translated Julia object)

Examples

```
if (juliaSetupOk()) {

   # Transfer a large vector to Julia and use it in multiple calls
   x <- juliaPut(rnorm(100))
   # x is just a reference to a Julia vector now
   juliaEval("using Statistics")
   juliaCall("mean", x)
   juliaCall("var", x)

}
```

`juliaSetupOk` *Check Julia setup*

Description

Checks that Julia can be started and that the Julia version is at least 1.0. For more information about the setup and discovery of Julia, see JuliaConnectoR-package, section "Setup".

Usage

```
juliaSetupOk()
```

Value

`TRUE` if the Julia setup is OK; otherwise `FALSE`

`startJuliaServer` *Start a Julia server that may serve multiple clients (R processes)*

Description

Starting a Julia server allows that different R processes may connect to the the same Julia server and share a single session. This can be useful for saving start-up/precompilation time when starting additional processes or when sharing global variables between processes. A usage scenario for this is a web server where reloading Julia for each new session should be avoided. *For the standard way of starting Julia, this function is not needed. It is also not needed if child processes should use separate Julia sessions.*

Usage

```
startJuliaServer(port = 11980)
```

Arguments

port	a hint for the port that is used by the server. If it is not available, a different port is used. The final port is returned (invisibly).

Details

The functions communicates the server address via setting the JULIACONNECTOR_SERVER environment variable. A possible value for the variable is "localhost:11980". The JULIACONNECTOR_SERVER variable is communicated automatically via the system environment to child processes that are started after this function has been called. The child processes will then connect to the same Julia server if the variable is set. The variable can also be set explicitly in child processes before connecting to Julia to connect to a running server. Unsetting the variable will result in a normal Julia start-up in the first call to Julia, using a single-client Julia session.

For security reasons, the Julia server accepts only connections from localhost.

For using Julia with multiple clients, it can be good to advise Julia to use multiple threads via setting the JULIA_NUM_THREADS environment variable before starting Julia.

Value

the port number (invisibly)

Note

The standard (error) output from Julia (printing and warnings) can currently only be forwarded to one client. This is currently the last client that has connected but this may be subject to change.

See Also

JULIACONNECTOR_SERVER

Examples

```
if (juliaSetupOk() && Sys.getenv("NOT_CRAN") == "true") {
   library(JuliaConnectoR)
   library(future)

  # 1) Start a shared Julia TCP server in the main R session, which may use all CPU threads
   Sys.setenv("JULIA_NUM_THREADS" = parallel::detectCores())
   stopJulia() # stop current Julia connection before starting server
   startJuliaServer()

   # 2) Create R workers
   cl <- parallelly::makeClusterPSOCK(workers = 2)

   # 3) Warm-up R workers, load Julia and execute first Julia function
   # (In practice, you could, e.g., trigger Julia pre-compilation here.)
   parallel::clusterEvalQ(cl, {
      library(JuliaConnectoR)
      JuliaConnectoR::juliaEval("1")  # open Julia TCP connection on each worker
   })
```

```
  ## 4) Re-Use those warmed workers for use via futures
  plan(cluster, workers = cl)

  ## 5) Create a global variable for demonstration purposes (and stay thread safe)
  juliaEval("global const counter = Threads.Atomic{Int}(0)")

  incrementCounter <- function() {
     JuliaConnectoR::juliaEval("Threads.atomic_add!(counter, 1)")
  }

  ## 6) Execute futures via worker-pool
  f1 <- future({incrementCounter()})
  f2 <- future({incrementCounter()})
  value(f1)
  value(f2)

  # Reading the value should demonstrate that the
  # global variable is shared across workers.
  juliaEval("counter[]")

  ## 7) Cleanup
  plan(sequential)
  parallel::stopCluster(cl)
  stopJulia()

}
```

`stopJulia` *Stop the connection to Julia*

Description

This ends the connection to Julia. Julia terminates if no R process is connected any more.

Usage

```
stopJulia()
```

Index

[.JuliaProxy *(*AccessMutate.JuliaProxy*)*, 5
[.JuliaSimpleArrayProxy *(*AccessMutate.JuliaProxy*)*, 5
[<-.JuliaProxy *(*AccessMutate.JuliaProxy*)*, 5
[[.JuliaArrayProxy *(*AccessMutate.JuliaProxy*)*, 5
[[.JuliaStructProxy *(*AccessMutate.JuliaProxy*)*, 5
[[<-.JuliaArrayProxy *(*AccessMutate.JuliaProxy*)*, 5
[[<-.JuliaStructProxy *(*AccessMutate.JuliaProxy*)*, 5
$.JuliaStructProxy *(*AccessMutate.JuliaProxy*)*, 5
$<-.JuliaStructProxy *(*AccessMutate.JuliaProxy*)*, 5

AccessMutate.JuliaProxy, *4*, 5
as.data.frame.JuliaProxy, *3*, 8

dim.JuliaArrayProxy *(*AccessMutate.JuliaProxy*)*, 5

EnvVars-JuliaConnectoR, 9

Julia-Setup, 10
JULIA_BINDIR *(*EnvVars-JuliaConnectoR*)*, 9
juliaCall, *3*, 10
JuliaConnectoR *(*JuliaConnectoR-package*)*, 2
JuliaConnectoR-package, 2, *15*, *18*
JULIACONNECTOR_JULIAENV *(*EnvVars-JuliaConnectoR*)*, 9
JULIACONNECTOR_SERVER, *19*
JULIACONNECTOR_SERVER *(*EnvVars-JuliaConnectoR*)*, 9
juliaEval, *3*, *5*, 11, *12*
juliaExpr, *3*, *4*, 12
juliaFun, *3*, *4*, 13
juliaGet, *3*, *4*, 13
juliaImport, *3*, *5*, 14
juliaLet, *3*, *5*, *12*, 16
juliaPut, 17
juliaSetupOk, 18

length.JuliaArrayProxy *(*AccessMutate.JuliaProxy*)*, 5

startJuliaServer, *9*, 18
stopJulia, 20