

Package ‘JuliaConnectoR’

April 23, 2020

Type Package

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

Version 0.6.0

Author Stefan Lenz [aut, cre] (<<https://orcid.org/0000-0001-9135-1743>>),
Harald Binder [aut] (<<https://orcid.org/0000-0002-5666-8662>>)

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.

License MIT + file LICENCE

SystemRequirements Julia >= 1.0

Encoding UTF-8

RoxygenNote 7.1.0

Suggests testthat (>= 2.1.0)

NeedsCompilation no

Repository CRAN

Date/Publication 2020-04-23 16:30:14 UTC

R topics documented:

JuliaConnectoR-package	2
AccessMutate.JuliaProxy	4
as.data.frame.JuliaProxy	6
juliaCall	8
juliaEval	8
juliaExpr	9
juliaFun	10
juliaGet	10
juliaImport	11
juliaLet	13
juliaPut	14
juliaSetupOk	15

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 ≥ 1.0) is installed** and that the Julia executable is in the system search PATH or that the JULIA_BINDIR environment variable is set to the bin directory of the Julia installation.

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, `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 an 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	→	double
Float16, Float32, UInt32	→	double with type attribute
Int64 that fits in 32 bits	→	integer

Int64 not fitting in 32 bits	→	double with type attribute
Int8, Int16, UInt16, Int32, Char	→	integer with type attribute
UInt8	→	raw
UInt64, Int128, UInt128, Ptr	→	raw with type attribute
Complex{Float64}	→	complex
Complex{Int X } with $X \leq 64$	→	complex with type attribute
Complex{Float X } with $X \leq 32$	→	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

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 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.

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

<code>x</code>	a Julia proxy object
<code>name</code>	the field of a struct type, the name of a member in a <code>NamedTuple</code> , or a key in a Julia dictionary (type <code>AbstractDict</code>)
<code>value</code>	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.
<code>i, j, k, ...</code>	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

```

if (juliaSetupOk()) {

  # (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"]

}

```

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 (juliaSetupOk()) {  
  
  # Demonstrate the usage with the Julia package "JuliaDB"  
  juliaEval('import Pkg; Pkg.add("JuliaDB")')  
  JuliaDB <- juliaImport("JuliaDB")  
  
  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 JuliaDB  
  mytbl <- JuliaDB$table(mydf)  
  
  # this table can, e g. be queried and  
  # the result can be translated to an R data frame  
  seltbl <- JuliaDB$select(mytbl, juliaExpr("(:x, :y)"))[1:2]  
  
  # translate selection of Julia table into R data frame  
  as.data.frame(seltbl)  
  
}
```

juliaCall	<i>Call a Julia function</i>
-----------	------------------------------

Description

Call a Julia function and get the translated result.

Usage

```
juliaCall(name, ...)
```

Arguments

name	name of the Julia function
...	parameters handed to the function. Will be 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.

juliaEval	<i>Evaluate a Julia expression</i>
-----------	------------------------------------

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 Pkg; Pkg.add("BoltzmannMachines")')  
  juliaEval('using Random; Random.seed!(5);')  
}
```

juliaExpr	<i>Mark a string as Julia expression</i>
-----------	--

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	<i>Wrap a Julia function in an R function</i>
----------	---

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	<i>Translate a Julia proxy object to an R object</i>
----------	--

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 be 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 <code>.MyModule</code> allows importing a module that does not correspond to a package, but has been loaded in the Main module, e. g. by <code>juliaCall("include", "path/to/MyModule.jl")</code> . Additionally, via a path such as <code>SomePkg.SubModule</code> , 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

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()
}

if (juliaSetupOk()) {

  # 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()  
}
```

`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

<code>expr</code>	Julia code, given as one-element character vector
<code>...</code>	arguments that will be introduced as variables in the <code>let</code> block. The values are transferred to Julia and assigned to the variables introduced in the <code>let</code> 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 creates a proxy object for a Julia object that would otherwise be translated to an R object. This is useful to prevent many translations of large objects if it is necessary performance reasons. To see which objects are translated by default, please see the [JuliaConnectoR-package](#) documentation.

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)  
  
}
```

<code>juliaSetupOk</code>	<i>Check Julia setup</i>
---------------------------	--------------------------

Description

Checks that Julia can be started and that the Julia version is at least 1.0.

Usage

`juliaSetupOk()`

Value

TRUE if the Julia setup is OK; otherwise FALSE

Index

[.JuliaProxy (AccessMutate.JuliaProxy),
4
[.JuliaSimpleArrayProxy
(AccessMutate.JuliaProxy), 4
[<-.JuliaProxy
(AccessMutate.JuliaProxy), 4
[[.JuliaArrayProxy
(AccessMutate.JuliaProxy), 4
[[.JuliaStructProxy
(AccessMutate.JuliaProxy), 4
[[<-.JuliaArrayProxy
(AccessMutate.JuliaProxy), 4
[[<-.JuliaStructProxy
(AccessMutate.JuliaProxy), 4
\$.JuliaStructProxy
(AccessMutate.JuliaProxy), 4
\$<-.JuliaStructProxy
(AccessMutate.JuliaProxy), 4

AccessMutate.JuliaProxy, 4, 4
as.data.frame.JuliaProxy, 2, 6

dim.JuliaArrayProxy
(AccessMutate.JuliaProxy), 4

juliaCall, 2, 8
JuliaConnectoR-package, 2, 14
juliaEval, 2, 8, 9
juliaExpr, 2, 3, 9
juliaFun, 2, 4, 10
juliaGet, 2, 4, 10
juliaImport, 2, 11
juliaLet, 2, 9, 13
juliaPut, 14
juliaSetupOk, 15

length.JuliaArrayProxy
(AccessMutate.JuliaProxy), 4