Package: XRJulia (via r-universe)

October 18, 2024

Type Package

Title Structured Interface to Julia

Version 0.9.0.1

Date 2019-05-01

Author John M. Chambers

Maintainer John Chambers <jmc@r-project.org>

Description A Julia interface structured according to the general form described in package 'XR' and in the book ``Extending R".

License GPL (>= 2)

Imports methods, XR

NeedsCompilation no

SystemRequirements Julia, v 1.0 or later

RoxygenNote 6.1.1

Suggests knitr, rmarkdown

VignetteBuilder knitr

Repository CRAN

Date/Publication 2024-04-20 09:58:28 UTC

Contents

`XRjulia-package` *Structured Interface to Julia*

Description

A Julia interface structured according to the general form described in package XR and the book "Extending R".

Author(s)

John Chambers

Maintainer: John Chambers <jmc@stat.stanford.edu>

References

Chambers, John M. *Extending R* Chapman & Hall/CRC 2016.

`asServerObjectMethods` *Julia methods for asServerObject()*

Description

The default method for JuliaObject is modelled on the overall default method in XR.

For arrays, the method uses the reshape() function in Julia to create a suitable multi-way array.

For "plain" lists, the method produces the Julia expression for a list or a dictionary; with other attributes, uses the .RClass form.

Usage

```
## S4 method for signature 'ANY,JuliaObject'
asServerObject(object, prototype)

## S4 method for signature 'array,JuliaObject'
asServerObject(object, prototype)

## S4 method for signature 'list,JuliaObject'
asServerObject(object, prototype)
```

Arguments

object	The R object.
prototype	The proxy for a prototype of the Julia object, supplied by the evaluator.

---

findJulia *Find a Julia Executable*

Description

This function looks for an executable Julia application in the local operating system. The location can be prespecified by setting environment variable JULIA_BIN; otherwise, the function looks in various conventional locations and if that doesn't work, runs a shell command to look for julia.

Usage

```
findJulia(test = FALSE)
```

Arguments

test	Should the function test for the existence of the application. Default FALSE. Calling with TRUE is useful to bullet-proof examples or tests for the absence of Julia. If the test search succeeds, the location is saved in environment variable JULIA_BIN.

Value

The location as a character string, unless test is TRUE, in which case success or failure is returned, and the location found (or the empty string) is saved as the environment variable. Note that in this case, FALSE is returned if the Julia package JSON has not been added.

If test is FALSE, failure to find a Julia in the current system is an error.

On Mac OS X

Installing Julia in the usual way does not put the command line version in a standard location, but instead in a folder under /Applications. Assuming one wants to have Julia available from the command line, creating a symbolic link to it in /usr/local/bin is a standard approach. If the current version of Julia is 0.6:

```
sudo ln -s /Applications/Julia-0.6.app/Contents/Resources/julia/bin/julia /usr/local/bin/julia
```

If for some reason you did not want this to be available, set the shell variable JULIA_BIN to the first file in the command, the one in /Applications.

from_Julia-class *Class for General Julia Composite Type Objects*

Description

The Julia side of the interface will return a general object from a composite type as an R object of class "from_Julia. its Julia fields (converted to R objects) can be accessed by the $ operator.

Slots

`serverClass` the Julia type.

`module` the Julia module, or ""

`fields` the converted versioin of the Julia fields; these are accessed by the $ operator.

functions *Function Versions of Methods for Julia Interface evaluators.*

Description

Function Versions of Methods for Julia Interface evaluators.

Usage

```
juliaSource(..., evaluator = RJulia())

juliaAddToPath(directory = "julia",
  package = utils::packageName(topenv(parent.frame())), pos = NA,
  evaluator = RJulia(.makeNew = FALSE), where = topenv(parent.frame()))

juliaUsing(module, evaluator)

juliaImport(..., evaluator)

juliaSend(object, evaluator = XR::getInterface(.JuliaInterfaceClass))

juliaGet(object, evaluator = XR::getInterface(.JuliaInterfaceClass))

juliaPrint(object, ..., evaluator = XRJulia::RJulia())

juliaEval(expr, ..., evaluator = XR::getInterface(.JuliaInterfaceClass))

juliaCommand(expr, ...,
  evaluator = XR::getInterface(.JuliaInterfaceClass))
```

```
juliaCall(expr, ..., evaluator = XR::getInterface(.JuliaInterfaceClass))

juliaSerialize(object, file, append = FALSE,
  evaluator = XR::getInterface(.JuliaInterfaceClass))

juliaUnserialize(file, all = FALSE,
  evaluator = XR::getInterface(.JuliaInterfaceClass))

juliaName(object)

juliaImport(..., evaluator)
```

Arguments

`...`	arguments to the corresponding method for an evaluator object.
`evaluator`	The evaluator object to use. By default, and usually, the current evaluator is used, and one is started if none has been. But see the note under `juliaImport` for the load actions created in special cases.
`directory`	the directory to add, defaults to "julia"
`package`, `pos`	arguments to the method, usually omitted.
`where`	for the load action, omitted if called from a package source file. Otherwise, must be the environment in which a load action can take place.
`module`	String identifying a Julia module.
`object`	A proxy in R for a Julia object.
`expr`	A string that should be legal when parsed and evaluated in Julia.
`file`, `append`, `all`	Arguments to the evalutor's serialize and unserialize methods. See the reference, Chapter 10.

Functions

- `juliaSource`: evaluate the file of Julia source.
- `juliaAddToPath`: adds the directory specified to the search path for Julia modules. If called from the source directory of a package during installation, sets up a load action for that package. If you want to add the path to all evaluators in *this* session, call the function before creating an evaluator. Otherwise, the action applies only to the specified evaluator or, by default, to the current evaluator.
- `juliaUsing`: the "using" form of Julia imports: the module is imported with all exports exposed.
- `juliaImport`: adds the module information specified to the modules imported for future Julia evaluator objects.

  Add the module to the table of imports for Julia evaluators, and import it to the current evaluator if there is one. If called from the source directory of a package during installation, both `juliaImport` and `juliaAddToPath()` set up a load action for that package. The functional versions, not the methods themselves, should be called from package source files to ensure that the load actions are created. Note that calling either function before any evaluator has been generated will install that call as a setup action for all XRJulia evaluators.

- `juliaSend`: sends the `object` to Julia, converting it via methods for `asServerObject` and returns a proxy for the converted object.
- `juliaGet`: converts the proxy object that is its argument to an R object.
- `juliaPrint`: Print an object in Julia. Either one object or several arguments as would be given to the Eval() method.
- `juliaEval`: evaluates the `expr` string subsituting the arguments. See the corresponding evaluator method for details.
- `juliaCommand`: evaluates the `expr` string subsituting the arguments; used for a command that is not an expression.
- `juliaCall`: call the function in Julia, with arguments given; expr is the string name of the function
- `juliaSerialize`: serialize the `object` in Julia
- `juliaUnserialize`: unserialize the file in Julia
- `juliaName`: return the name by which this proxy object was assigned in Julia
- `juliaImport`: Import a Julia module or add a directory to the Julia Search Path If called from the source directory of a package during installation, both `juliaImport` and `juliaAddToPath()` also set up a load action for that package. The functional versions, not the methods themselves, should be called from package source files to ensure that the load actions are created.

`juliaClassDef` *Information about a Julia Class*

Description

The Julia class definition information is computed, and converted to R.

Usage

```
juliaClassDef(Class, module = "", ..., .ev = RJulia())
```

Arguments

`Class`, `module`	Strings identifying the Julia composite type and optionally, the module containing it.
`...`, `.ev`	Don't supply these, `.ev` defaults to the current Julia interface evaluator.

Value

the Julia definition of the specified class, optionally from the module.

JuliaFunction-class *Proxy Objects in R for Julia Functions*

Description

A class and generator function for proxies in R for Julia functions.

Usage

```
JuliaFunction(...)

## S4 method for signature 'JuliaFunction'
initialize(name, module = "", evaluator =
RJulia(, ...))
```

Arguments

name, module	The name and module of the Julia function.
evaluator	The evaluator object to use. By default, and usually, the current evaluator is used, and one is started if none has been.
...	For RJulia, the arguments as interpreted by the initialize method, so typically name and optionally module. Remaining arguments are passed along to the next method.

Details

An object from this class is an R function that is a proxy for a function in Julia. Calls to the R function evaluate a call to the Julia function. The arguments in the call are converted to equivalent Julia objects; these typically include proxy objects for results previously computed through the XRJulia interface.

Slots

name the name of the server language function

module the name of the module, if that needs to be imported

evaluatorClass the class for the evaluator, by default and usually, JuliaInterface

Examples

```
if(findJulia(test = TRUE) ) {
  ## even so, the Julia may not be valid
  ## so we catch any errors in the example, mainly to keep CRAN quiet
  tryCatch( {set.seed(228)
    x <- matrix(rnorm(1000),20,5)
    xm <- juliaSend(x)
    juliaCommand("using LinearAlgebra")
    svdJ <- JuliaFunction("svd")
```

```
  sxm <- svdJ(xm)
  sxm
}, error = function(e) message("Julia Example error: " ,e$message))}
```

`JuliaInterface-class` *An Interface to Julia*

Description

The JuliaInterface class provides an evaluator for computations in Julia, following the structure in the XR package. Proxy functions and classes allow use of the interface with no explicit reference to the evaluator. The function `RJulia()` returns an evaluator object.

Fields

`host` The remote host, as a character string. By default this will be the local host, and initializing the evalutor will set the field to "localhost".

`port` The port number for commuicating to Julia from this evalutor. By default, the port is set by adding the evaluator number-1 to a base port number. By default the base port is randomly chosen at package load time (this strategy may change).

The port may be controlled in two ways. If you know a good range or set of ports, it will be preferrable to supply unique port values (integer) in the initialization call. A less direct way is set the R option "JuliaBasePort", which will then be used as the base port. Since evaluator numbers are used to increment the port, the call to `options` should normally come before initializing the first Julia evaluator.

`julia_bin` The location for an executable version of the Julia interpreter. By default, this assumes there is a file named `"julia"` on the command-line search path. If Julia is not usable from the command line or if you want to run with a different version, supply the executable file name as this argument. It is also possible to set the location for all evaluators by setting the shell variable `JULIA_BIN` to this location *before* starting R.

`connection` The connection object through which commands are sent to Julia. Normally will be created by the initialization of the evaluator. Should only be supplied as a currently open socket on which to communicate with the Julia interpreter.

`serverWrapup` a vector of actions for the ServerEval to take after evaluation. Used to clean up after special operations, such as sending large objects to Julia.

`largeObject` Vectors with length bigger than this will be handled specially. See largeVectors. Default currently 1000. To change this, call `juliaOptions()` to set option `largeObject`.

`fileBase` a pattern for file names that the evaluator will use in Julia for various data transfer and other purposes. The evaluator appends "_1", "_2", etc. To change this, call `juliaOptions()` to set option `fileBase`. It is initiaized to an R tempfile with pattern `"Julia"`.

Methods

`Import(module, ...)` Import the module. The "Interface" method assumes a command "import" in the server language and does not handle any extra arguments (e.g., for importing specific members).

`initialize(...)` initializes the evaluator in a language-independent sense.

`ProxyClassName(serverClass)` If there is a proxy class defined corresponding to this serverClass, return the name of that class (typically pasted with the server langauge, separated by underscore). If no such class is defined, return NA.

`ServerClassDef(Class, module, ...)` Individual interface packages will define this to return a named list or other object such that value$fields and value$methods are the server fields and methods, character vectors of names or named objects whose elements give further information. This default version returns NULL, indicating that no metadata is available.

`ServerEval(expr, key, get)` Must be defined by the server language interface: evaluates ‘expr‘(a text string). If ‘key‘ is an empty string, ‘expr‘ is treated as a directive, with no defined value. Otherwise, ‘key‘ is a non-empty string, and the server object should be assigned with this name. The value returned is the R result, which may be an AssignedProxy() object. If ‘get‘ is TRUE or the value judged simple enough, it will be converted to an ordinary R object instead.

`ServerRemove(key)` Should be defined by the server language interface: The reference previously created for ‘key‘ should be removed. What happens has no effect on the client side; the intent is to potentially recover memory.

`ServerTask(task, expr, key = "", get = NA)` Call the task operation in the Julia code for the interface; the arguments must be the simple strings or logical value expected.

`Source(filename)` Parse and evaluate the contents of the file. This method is likely to be overriden for particular langauges with a directive to include the contents of the file. The ‘XR‘ version reads the file and processes the entire contents as a single string, newlines inserted between lines of the file.

`Using(...)` The Julia "using" form of importing. Arguments are module names. All the exported members of these modules will then be available, without prefix.

JuliaObject-class *Proxy Objects in R for Julia Objects*

Description

This is a class for all proxy objects from a Julia class with an R proxy class definition. Objects will normally be from a subclass of this class, for the specific Julia class.

Details

Proxy objects returned from the Julia interface will be promoted to objects from a specific R proxy class for their Julia class, if such a class has been defined.

juliaOptions *Get and/or Set Internal Option Parameters in the Julia Evaluator*

Description

The Julia code for an evaluator maintains a dictionary, `RJuliaParams`, of named parameters used to control various evaluation details. These and any other desired options can be queried and/or set by calls to `juliaOptions`.

Usage

```
juliaOptions(..., .ev = XRJulia::RJulia())
```

Arguments

`...`	arguments to the corresponding method for an evaluator object.
`.ev`	The evaluator object to use. By default, and usually, the current evaluator.

Details

The function behaves essentially like the `options()` function in R itself, returning a list of the current entries corresponding to unnamed character arguments and setting the parameters named to the value in the corresponding named argument to `juliaOptions`. If no parameter corresponding to a name has been set, requesting the corresponding returned value is `nothing`, `NULL` in R.

Value

A named list of those parameters requested (as unnamed character string arguments). If none, an empty list. Note that options are always returned converted to R, not as proxyies.

juliaVersion *Get or test the Julia Version information*

Description

The Julia constant structure `VERSION` is returned. If `test` is `TRUE`, only returns a logical testing whether this version is compatible with `XRJulia`.

Usage

```
juliaVersion(test = FALSE, .ev = RJulia())
```

Arguments

`test`	If `TRUE`, tests compatibility (currently that the major number is at least `1`). Default `FALSE`
`.ev`	The evaluator object to use. By default, and usually, the current evaluator.

Value

A named list with the members of the Julia object, the usually relevant ones being `"major"`, `"minor"` and `"patch"`. `test=TRUE` overrides as described.

---

largeVectors *Internal Computations for Large Vectors*

Description

Internal Computations for Large Vectors

Sending Large Vectors between R and Julia

Large vectors will be slow to transfer as JSON, and may fail in Julia. Internal computations have been added to transfer vectors of types real, integer, logical and character by more direct computations when they are large. The computations and their implementation are described here.

R and Julia both have the concept of numeric (floating point) and integer arrays whose elements have a consistent type and both implement these (following Fortran) as contiguous blocks in memory, augmented by length or dimension information. They also both have a mechanism for arrays of character strings, class `"character"` in R and array type `Array{String, 1}` in Julia. Julia has arrays for boolean data; R stores the corresponding `logical` as integers.

JSON has no such concepts, so interface evaluators using the standard JSON form provided by 'XR' must send such data as a JSON list. This will become inefficient for very large data from these classes. Users have reported failure by Julia to parse the corresponding JSON.

The 'XRJulia' package (as of version 0.7.9) implements special code to send vectors to Julia, by writing an intermediate file that Julia reads. The actual text sent to Julia is a call to the relevant Julia function. The code is triggered within the methods for the `asServerObject` function, so vectors should be transferred this way whether on their own or as part of a larger structure, such as an array or the column of a data frame.

Similarly, large arrays to be retrieved in R by the `Get()` method or the optional argument `.get = TRUE` will be written to an intermediate file by Julia and read by R.

As vectors become large, direct transfer becomes *much* faster. On a not-very-powerful laptop, vectors of length `10^7` transfer in an elapsed time of a few seconds. Character vectors are slightly slower than numeric, as explained below, but in all cases it would be hard to do much computation with the data that did not swamp the cost of transfer. That said, as always it's more sensible to transfer data once and then use the corresponding proxy object in later calls.

Details

For all vectors, the method uses binary writes and reads, which are defined in both R and Julia. No special computationss are needed for numeric, integer, complex and raw. For these, the R binary representation corresponds to array types in Julia. The special pseudo-value `NA` is defined for vectors in R, but no corresponding concept exists in Julia. For numeric and complex vectors, the floating-point pattern `NaN` is used. For all other vectors, a warning is issued and either a numeric object or a special character string is used instead.

For logicals, the internal representation in R uses integers. The Julia code when data is sent from R casts the integer array to a boolean array. On the return side, the Julia boolean array is converted to integer before writing.

Character vectors take a little more work, partly because of a weirdness in binary writes for string arrays in Julia. Where R character vectors can be written in binary form and then read back in, writing a `String` array in Julia omits the end-of-string character, effectively writing a single string, from which the array cannot be recovered. Communicating the entire vector to Julia requires that the Julia side uses this information to split the single string resulting from the R binary write by matching the end-of-string character explicitly For sending back to R, the Julia code appends an end-of-string character to each string before writing the array to a file. This produces the R format for a binary read of a character vector.

Two fields in the evaluator object control details. A large object is defined as a vector of length greater than the integer field `largeObject`. Julia creates intermediate files for sending large arrays to R by appending sequenctial numbers to a character field `fileBase`. By default, `largeObject` and `fileBase` is obtained from `tempfile()` with pattern `"Julia"`. Note that all the files are removed at the end of the evaluation of the expression sending or getting the relevant objects.

Since these fields must be known to the Julia evaluator, they should *not* be set directly—this will have no effect. Instead call the function `juliaOptions()` with these parameter names.

`noScalar` *Send a Non-scalar Version of an Object*

Description

Ensures that an object is interpreted as a vector (array) when sent to the server language. The default strategy is to send length-1 vectors as scalars.

Usage

```
noScalar(object)
```

Arguments

`object`	A vector object. Calling with a non-vector is an error.

Value

the object, but with the S4 bit turned on. Relies on the convention that XR interfaces leave S4 objects as vectors, not scalars, even when they are of length 1

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall/CRC. (Chapter 12, discussing this package, is included in the package: `../doc/Chapter_XR.pdf`.)

proxyJuliaObjects *Proxy Objects in R for Julia Objects*

Description

Proxy Objects in R for Julia Objects

RJulia *An Evaluator for the Julia Interface.*

Description

Returns an evaluator for the Julia interface. Starts one on the first call, or if arguments are provided; providing argument `.makeNew = TRUE` will force a new evaluator. Otherwise, the current evaluator is returned.

Usage

```
RJulia(...)
```

Arguments

`...`	Arguments passed to `getInterface()` but none usually required. See `JuliaInterface` for details of the evaluator.

setJuliaClass *Define a Proxy Julia Class (Composite Type)*

Description

Given the name and optionally the module for a Julia composite type, defines an R proxy class with the same fields as the Julia type. By default, uses metadata from Julia to find the fields. If the call supplies the desired field names explicitly, metadata is not used.

Usage

```
setJuliaClass(juliaType, module = "", fields = character(),
  where = topenv(parent.frame()), proxyObjectClass = "JuliaObject",
  ...)
```

Arguments

`juliaType`, `module`
: Strings identifying the composite type and optionally the module containing it. In normal use, metadata from Julia is used to find the definition of the type.

`fields`, `where`, `proxyObjectClass`, `...`
: Overriding arguments that should not be used by direct calls from package source code.

Index