Package 'rscala'

April 11, 2018

Type Package

Title Bi-Directional Interface Between 'R' and 'Scala' with Callbacks

Version 2.5.3

Date 2018-04-10

URL https://github.com/dbdahl/rscala

BugReports https://github.com/dbdahl/rscala/issues

Imports utils

SystemRequirements Scala (>= 2.11)

Description The 'Scala' <http://www.scala-lang.org/> interpreter is embedded in 'R' and callbacks to 'R' from the embedded interpreter are supported. Conversely, the 'R' interpreter is embedded in 'Scala'. 'Scala' versions in the 2.11.x and 2.12.x series are supported.

License GPL (>= 2) | BSD_3_clause + file LICENSE

VignetteBuilder knitr

Suggests knitr, digest, microbenchmark, rJava, xtable

NeedsCompilation no

Author David B. Dahl [aut, cre]

Maintainer David B. Dahl <dahl@stat.byu.edu>

Repository CRAN

Date/Publication 2018-04-11 15:08:02 UTC

R topics documented:

`rscala-package` *Bi-Directional Interface Between* R *and Scala with Callbacks*

Description

This package embeds: i. the R interpreter in Scala, and ii. the Scala interpreter/compiler in R (with callbacks to R).

Author(s)

David B. Dahl <dahl@stat.byu.edu>

See Also

`scala`

`.rscalaOptions` *Set global options*

Description

This function can be used to set **rscala**'s global options before any packages that depend on **rscala** are loaded. Using the function is equivalent to using the `options` function with the arguments `rscala.scala.home`, `rscala.heap.maximum`, and/or `rscala.command.line.options`.

Usage

```
.rscalaOptions(scala.home=NULL, heap.maximum=NULL, command.line.options=NULL)
```

Arguments

`scala.home`	See the `scala` function.
`heap.maximum`	See the `scala` function.
`command.line.options`	See the `scala` function.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

See Also

scala

Examples

```
.rscalaOptions(heap.maximum="2G")
```

---

close *Closes the Resources of a Scala Interpreter*

Description

This function closes (i.e., frees) the resources associated with an interpreter. Subsequent uses of the interpreter object will fail.

Usage

```
## S3 method for class 'ScalaInterpreter'
close(con, ...)
```

Arguments

con	An interpreter from an interpreter constructor (i.e. scala).
...	Currently ignored.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

---

is.scalaReference *Tests whether an object is a Scala reference.*

Description

This function returns TRUE if and only if the object is a Scala reference.

Usage

```
is.scalaReference(x)
```

Arguments

x	An R object.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

Examples

```
is.scalaReference(pi)
```

---

print *String Representation of a Scala Interpreter, Reference, or Function*

Description

These functions print or return a string representation of a Scala interpreter, a Scala reference, or a Scala function.

Usage

```
## S3 method for class 'ScalaInterpreter'
print(x, ...)
## S3 method for class 'ScalaInterpreter'
toString(x, ...)
## S3 method for class 'ScalaInterpreterReference'
print(x, ...)
## S3 method for class 'ScalaInterpreterReference'
toString(x, ...)
## S3 method for class 'ScalaInterpreterItem'
print(x, ...)
## S3 method for class 'ScalaInterpreterItem'
toString(x, ...)
```

Arguments

x	An interpreter from the scala function, a Scala reference, or a Scala function.
...	Currently ignored.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

scala *Create an Instance of an Embedded Scala Interpreter*

Description

The function scala creates an instance of an embedded Scala interpreter/compiler and binds a Scala object named R to permit callbacks to R. Options are available to customize where Scala is found and how it is invoked (e.g., setting the classpath and maximum heap size). Multiple interpreters can be created and each runs independently with its own memory. Each interpreter can use multiple threads/cores, but the bridge between R and Scala is not thread-safe. As such, multiple R threads/cores should not simultaneously access the same interpreter.

The functions scala2 and scala3 is equivalent to scala except they set mode="lazy" and mode="serial", respectively.

The functions scalaInfo and .rscalaJar provide file paths to JAR files, installation directories, the Scala executable, and this package. Note that if you only want to embed R in a Scala application, you do not need to install the package. Simply add the following line to the your SBT build.sbt file: 'libraryDependencies += "org.ddahl" "_VERSION_"', where _VERSION_ is the rscala version number (i.e., 2.5.3).

scalaInstall downloads and installs Scala in "~/.rscala" in the user's home directory. System administrators can install Scala globally as described here: http://www.scala-lang.org/download/install.html. In short, simply download the archive, unpack it, and add the "scala" script to the path.

The function .rscalaPackage should be called in the .onLoad function of a package that wishes to depend on this package. The function should not be called elsewhere. It sets the classpath to the JAR files in the 'java' directory of the package and passes the ... arguments to the scala function. This instance of Scala is available as the object s in the namespace of the package (thereby making it available to the package's function) but it is not exported from the namespace. The object s is only initialized on its first usage. The function .rscalaPackageUnload is deprecated and is no longer needed for packages that depend on rscala.

Usage

```
scala(classpath=character(), classpath.packages=character(),
      serialize.output=.Platform$OS.type=="windows", scala.home=NULL,
      heap.maximum=NULL, command.line.options=NULL, row.major=TRUE,
      debug=FALSE, stdout=TRUE, stderr=TRUE, port=0,
      scala.info=NULL, major.release=c("2.11","2.12"),
      mode="parallel", assign.name="s", assign.env=parent.frame(),
      callback=function(s) {}, snippet=character())

scala2(...)

scala3(...)

scalaInfo(scala.home=NULL, major.release=c("2.11","2.12"),
```

```
          verbose=FALSE)

scalaInstall(major.release=c("2.11","2.12"), global=FALSE)

.rscalaJar(major.release=c("2.11","2.12"))

.rscalaPackage(pkgname, ..., classpath.packages=character(),
               mode="parallel", assign.name = "s")

.rscalaPackageUnload()
```

Arguments

`classpath`	A character vector whose elements are paths to JAR files or directories which specify the classpath for the Scala compiler/interpreter.
`classpath.packages`	A character vector giving names of other installed packages whose JAR files should be appended to the classpath.
`serialize.output`	Should standard output (stdout) and standard error (stderr) be captured and serialized back to R? The default is `TRUE` on Windows and `FALSE` on other operating systems. `FALSE` requires less computing and is usually not necessary on Linux and Mac OS X. Depending on the environment and operating system in which R is run, `TRUE` may be needed to see output and error messages.
`scala.home`	A character vector of length one giving the path where Scala is installed. When set to `NULL` (the default), the function sequentially tries to find the Scala home by: i. querying the global option *rscala.scala.home*, ii. using the environment variable `SCALA_HOME`, iii. querying the operating system search path, and iv. looking in subdirectories of `~/.rscala`. If all these fail, the function displaces a message to help the user install Scala. Alternatively, in the class of `scalaInfo`, `scala.home` may also be the result of the `scala` function. Using the function `.rscalaOptions`, the global option *rscala.scala.home* is set and effects future **rscala** function calls.
`heap.maximum`	A character vector of length one used to specify the maximum heap size in the JVM. If `NULL`, the global option *rscala.heap.maximum* is queried and, if that is also `NULL`, Scala's default value is used. This option is ignored if *command.line.options* is not `NULL` or if the global options *rscala.command.line.options* is not `NULL`. Using the function `.rscalaOptions`, the global option *rscala.heap.maximum* is set and effects future Scala instances.
`command.line.options`	A character vector whose elements are passed as command line arguments when invoking Scala. If `NULL`, the global option *rscala.command.line.options* is queried and, if that is also `NULL`, the value is set to `NULL`. A value of `NULL` means no extra arguments are provided. If you simply want to add to the classpath and/or set the maximum heap size, use the *classpath* and *heap.maximum* arguments. Using the function `.rscalaOptions`, the global option *rscala.command.line.options* is set and effects future Scala instances.
`row.major`	Should matrices in Scala be row major?

debug	An option meant only for developers of the package itself and not intended for users of the package.
stdout, stderr	Where standard output and standard error results that are not serialized should be sent. TRUE (the default) or "" sends output to the R console (although that may not work on Windows). FALSE or NULL discards the output. Otherwise, this is the name of the file that receives the output.
port	If 0, two random ports are selected. Otherwise, port and port+1 are used to the TCP/IP connections.
scala.info	The result of a previous call to scalaInfo.
verbose	A logical vector of length one indicating whether information regarding the search for the Scala installation should be displayed.
major.release	The character vector giving acceptable major.release numbers (e.g., c("2.11","2.12")), or NA in which case the system picks the appropriate version.
mode	One of "parallel", "lazy", "serial", or an arbitrary string to indicate how the R/Scala bridge is established. "serial" starts Scala in the background, blocks until the bridge is established, and causes the function to return the interpreter. "parallel" starts Scala in the background whereas "lazy" only starts Scala if and when the interpreter is first used. If mode is some other string, it is considered to be the name of an already-existing Scala instance (e.g., to use the Scala instance of the package sdols, use "sdols:::s"). When using an arbitrary string, many arguments are honored including those specifying the class path, but some arguments only make sense for a new instance (e.g., ignored arguments include serialize.output, heap.maximum, row.major, and command.line.options). All options except "serial" cause the function to return NULL and makes the interpreter (with the identifier given by assign.name) available as a promise. "serial" is not supported in the .rscalaPackage function.
assign.name	If mode != "serial", the name of the (promise of the) interpreter.
assign.env	If mode != "serial", the environment in which the (promise of the) interpreter is assigned.
callback	A function taking a Scala interpreter as its only argument. This function is called immediately after the R/Scala bridge is established.
snippet	A character vector providing Scala code that will be evaluated when the interpreter in the package namespace is first used.
global	Should the Scala installation be global, specifically inside the package's directory? This may require administrator privileges.
pkgname	A character string giving the name of the package (as provided the second argument of the .onLoad function) that wishes to depend on this package.
...	These arguments are passed by the .rscalaPackage function to the scala function.

Value

link{scala} and link{scala2} return NULL but have the side effect of creating (a promise for) an R object representing an embedded Scala interpreter.

scala3 returns an R object representing an embedded Scala interpreter.

scalaInfo returns a list detailing the Scala executable, version, jars, etc.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

See Also

scalaSettings,

strintrplt, rscala-package

Examples

```
# Uncomment the next line to download and install Scala
# scalaInstall()

.rscalaJar()
scalaInfo(verbose=TRUE)


# Make an instance of the Scala interpreter 's' and see how its output is captured.
scala(serialize.output=TRUE)
capture.output(s %~% 'println("This is Scala "+scala.util.Properties.versionString)')
scalaSettings()

# Demonstrate convenient notation and string interpolation
stringFromScala <- s %~% '"Hello @{Sys.getenv("USER")} from @{R.Version()$nickname}" + "!"*10'
stringFromScala

# Set and get variables
s$rPi <- pi
s$rPi
s$val("rPi")
s$.val("rPi")

s$rPi <- I(pi)     # Now rPi is an array of length one.
s$rPi              # It doesn't matter to R...
s$.val("rPi")      # ... but it does to Scala.

# Convenient notation
a1 <- s %~%  "rPi(0)/2"   # As an R value
a2 <- s %.~% "rPi(0)/2"   # As a Scala reference

# References can be set
s$foo <- a2
s$foo

# Instantiate an object
seed <- 2349234L
rng <- s$.scala.util.Random$new(seed)  # Scala equivalent: new scala.util.Random(seed)
```

```
# Call method of a reference
system.time(rng$nextInt(100L))   # Scala equivalent: rng.nextInt(100)
system.time(rng$nextInt(100L))  # Notice it runs much faster the second time due to caching

rInt <- rng$nextInt(100L,.EVALUATE=FALSE)  # Define function to call quickly later without ...
rInt(100)                                  # ... needing to protect scalars and ensure type.

# Call method of companion object and call methods of a reference
# Scala equivalent: (scala.math.BigInt("777",8) - 500).intValue
s$.scala.math.BigInt$apply("777",8L)$'-'(500L)$intValue()

# Example showing callback functionality
f <- function(func=NULL, data=numeric(), quiet=TRUE) s %!% '
  if ( ! quiet ) println("Here I am in Scala.")
  R.invokeD1(func, data.map(2*_), "verbose" -> !quiet ).sum
'


cube <- function(x, ignored.argument, verbose=TRUE) {
  if ( verbose ) cat("Here I am in R.\n")
  x^3
}

identical( f(cube,1:4,FALSE), sum((2*(1:4))^3) )
identical( f(cube,1:4,TRUE),  sum((2*(1:4))^3) )

# Longer example showing more flexible than '%~%'
drawGaussian <- function(mean=0.0, sd=1.0, rng=scalaNull("scala.util.Random")) s %!% '
  mean+sd*rng.nextDouble
'
drawGaussian(3,0.1,rng)  # No scalar protection or casting is needed.
n.draws <- 100L
s$random <- rng
system.time({
  draws <- s %~% '
    val result = new Array[Double](@{n.draws})
    result(0) = random.nextGaussian
    for ( i <- 1 until @{n.draws} ) {
      result(i) = 0.5*result(i-1) + random.nextGaussian
    }
    result
  '
  acf(draws,plot=FALSE)
})
sampler <- function(nDraws, rho, rng) s %!% '
  val result = new Array[Double](nDraws)
  result(0) = rng.nextGaussian
  for ( i <- 1 until nDraws ) {
    result(i) = rho*result(i-1) + rng.nextGaussian
  }
  result
'
system.time(acf(sampler(n.draws,0.5,rng),plot=FALSE))
system.time(acf(sampler(n.draws,0.5,rng),plot=FALSE))
```

```
close(s)
```

---

scalaConvert *Convert an R object to a Scala reference.*

Description

These functions convert R objects to Scala references and vice versa. The function `scalaConvert` supports copyable types whereas `scalaConvert.data.frame` supports data frames. These functions provide prototypes for others to write their own conversions for other types. Factors in data frames are converted to strings when converting to Scala.

Usage

```
scalaConvert(x, interpreter = findScalaInstance())

scalaConvert.data.frame(x, interpreter = findScalaInstance())
```

Arguments

`x`	An R object to convert to a Scala reference.
`interpreter`	An instance of a Scala interpreter.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

Examples

```
scala(serialize.output=TRUE)

identical(pi, scalaConvert(scalaConvert(pi)))

identical(mtcars, scalaConvert.data.frame(scalaConvert.data.frame(mtcars)))

dfInScala <- scalaConvert.data.frame(ToothGrowth)
dfInScala$keys()
dfInScala$apply("supp")$"_2"()
dfInScala$apply("supp")$"_1"()
```

scalaRequire *Add a JAR to the Classpath of an Existing Scala Instance*

Description

This function adds the supplied JAR file to the classpath of an existing Scala instance.

Usage

```
scalaRequire(jar.file, interpreter = findScalaInstance())
```

Arguments

jar.file	A string giving the path to a JAR file.
interpreter	An instance of a Scala interpreter.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

scalaSettings *Settings of a Scala Instance*

Description

This function retrieves information about a Scala instance when using the default values of function arguments. Otherwise, the code interpolation and show snippet settings can be modified.

Usage

```
scalaSettings(interpolate=NULL, show.snippet=NULL, info=NULL,
              interpreter=findScalaInstance())
```

Arguments

interpolate	If TRUE, the interpreter will call strintrplt before running code.
show.snippet	If TRUE, the code for the conversion of Scala function arguments will be displayed (for debugging purposes).
info	An argument that cannot be set by the user.
interpreter	A Scala instance resulting from or side effect of scala, scala2, or scala3.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

See Also

scala, strintrplt

---

strintrplt *String Interpolation with Arbitrary* R *Code*

Description

This function allows a character vector to have arbitrary R code embedded within it. The embedded R code — placed between '@{' and '}' — is evaluated by default in the environment from which this function is called (not defined). The result of the function is a character vector in which the embedded R code is replaced by character representations of the results of evaluating the embedded R code.

Usage

```
strintrplt(snippet,envir=parent.frame())
```

Arguments

snippet	A character vector of length one which may contain multiple '@{RCODE}' substrings which will be replaced by a character representation of the result of evaluating 'RCODE'. This character vector may contain many R statements and span multiple lines.
envir	The environment in which to evaluate the embedded R code.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

Examples

```
strintrplt('val msg = "Current R version is @{paste0(R.Version()$major,".",R.Version()$minor)}"')
index <- 5
strintrplt('var i = @{index-1}')
```

`%\textasciitilde%` *Execute Code, Set Values, and Get Values in a Scala Interpreter*

Description

These functions define the package's interface to an embedded Scala interpreter/compiler. Code is executed in Scala and data is passed between R and Scala. Callbacks to the original R interpreter are supported. R functions can be implemented in Scala. The result of a Scala execution can be a Scala reference or an R object.

Usage

```
interpreter %@% snippet

interpreter %~% snippet
interpreter %.~% snippet

interpreter %!% snippet
interpreter %.!% snippet

scalaNull(type)
II(x)

## S3 method for class 'ScalaInterpreter'
interpreter$identifier
## S3 replacement method for class 'ScalaInterpreter'
interpreter$identifier <- value
```

Arguments

`interpreter`	An interpreter from the `scala` function.
`snippet`	A character vector of arbitrary length to be evaluated by the interpreter. Multi-line elements are accepted.
`type`	A string containing a valid Scala type, e.g., `"List[Double]"`.
`x`	A expression whose value will be passed to a Scala function as an `EphemeralReference` even if it is a copyable type.
`identifier`	A character vector of length one containing a valid Scala identifier.
`value`	Either: i. a vector or matrix of integers, doubles, logicals, characters, or raw, or ii. a Scala reference.

Details

`interpreter %@% snippet` evaluates the expression *snippet*, whereas `interpreter %~% snippet` both evaluates the expression *snippet* and attempts to return a vector or matrix of integers, doubles, logicals, characters, or raw or — if this is not possible — `NULL` is returned. `interpreter %.~% snippet` has the same behavior except it always returns a Scala reference. References may be used later for

assignment and Scala function arguments. Note that memory allocated as a result of assignment or executing `snippet` will never be garbage collected due to a limitation of Scala's REPL. (See the Scala bug detailed here: https://issues.scala-lang.org/browse/SI-4331.)

`interpreter %!% snippet` and `interpreter %.!% snippet` are the analog to `interpreter %~% snippet` and `interpreter %~% snippet`, respectively, that should only be used within a function to define a Scala function. Memory associated with the return values by function defined by `interpreter %!% snippet` and `interpreter %.!% snippet` is garbage collected. Thus, heavy usage of `interpreter %!% snippet` and `interpreter %.!% snippet` functions is encouraged for memory intensive applications. Importantly, these functions have much less invocation latency than the equivalent code using `%\textasciitilde%` and `%.\textasciitilde%`. The speed difference in invocation is especially noticeable for quick functions.

`interpreter$identifier` and `interpreter$identifier <- value` get and set variables in the interpreter.

Value

`s$x` tries to convert the Scala object `x` to a vector or matrix of integers, doubles, logicals, characters, or raw. If a conversion is not possible, a Scala reference is returned. `s$.x` always returns a Scala reference for the Scala object `x`.

`s$x <- value` always returns `value` after having set the Scala variable `x` to it.

`function(...) s %!% snippet` returns a Scala function where `...` are arguments that can be used in the Scala code given in `snippet`. `function(...) s %.!% snippet` is the same, except a Scala reference is always returned.

`scalaNull(type)` returns a null reference whose Scala type is `type`. This can be helpful in defining Scala functions.

`s$.CLASSNAME$METHODNAME(...)` returns the result of calling `METHODNAME` in the class `CLASSNAME` using the unnamed, positional arguments in `....` `ref$METHODNAME(...)` returns the result of calling `METHODNAME` on the Scala reference `ref` using the unnamed, positional arguments in `....` Adding `.EVALUATE=FALSE` after `...` will yield an optimized Scala function instead of evaluating the function. Adding `.AS.REFERENCE=TRUE` after `...` will always yield a Scala reference.

Author(s)

David B. Dahl <dahl@stat.byu.edu>

See Also

scala, scalaSettings, strintrplt

Index