

# Package: juliabridge (via r-universe)

October 13, 2026

**Title** Bridge R Packages to Julia

**Version** 0.1.0

**Description** Helper utilities for R packages that call into a Julia simulation engine. Provides a small, opinionated API for finding the Julia binary, installing required Julia packages, loading '.jl' bridge files, managing lazy initialisation, describing Julia calls in R and rendering them to Julia source, and holding Julia objects from R between calls. Uses JuliaConnectoR (out-of-process Julia via TCP) under the hood, chosen over JuliaCall to structurally avoid the embedded-Julia segfaults that otherwise leak into R from libR/libjulia symbol interactions, environment poisoning, and Pkg operations.

**License** MIT + file LICENSE

**URL** <https://github.com/epiforecasts/juliabridge>,  
<https://epiforecasts.io/juliabridge/>

**BugReports** <https://github.com/epiforecasts/juliabridge/issues>

**Imports** JuliaConnectoR, stats

**Suggests** testthat (>= 3.2.0), withr

**Config/testthat/edition** 3

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.3

**Repository** <https://epiforecasts.r-universe.dev>

**Date/Publication** 2026-10-05 16:11:29 UTC

**RemoteUrl** <https://github.com/epiforecasts/juliabridge>

**RemoteRef** v0.1.0

**RemoteSha** a6697c243ea09975ad62ce4338e5e665d5ddf6c2

Contents

|                                  |           |
|----------------------------------|-----------|
| as_julia . . . . .               | 2         |
| as_julia_value . . . . .         | 3         |
| assert_role . . . . .            | 4         |
| assign_julia . . . . .           | 5         |
| call_julia . . . . .             | 5         |
| command_julia . . . . .          | 6         |
| ensure_julia . . . . .           | 6         |
| eval_julia . . . . .             | 7         |
| get_julia . . . . .              | 7         |
| import_julia . . . . .           | 8         |
| julia . . . . .                  | 8         |
| julia_alive . . . . .            | 9         |
| julia_bin . . . . .              | 10        |
| julia_handle . . . . .           | 10        |
| julia_handle_owned . . . . .     | 12        |
| julia_load_bridge . . . . .      | 13        |
| julia_ready . . . . .            | 13        |
| julia_release_pending . . . . .  | 15        |
| julia_spec . . . . .             | 16        |
| juliaup_julia . . . . .          | 17        |
| manifest_julia_version . . . . . | 18        |
| wrappers . . . . .               | 19        |
| <b>Index</b>                     | <b>20</b> |

---

|          |                                    |
|----------|------------------------------------|
| as_julia | <i>Render a spec as Julia code</i> |
|----------|------------------------------------|

---

Description

Render a spec as Julia code

Usage

```
as_julia(x, ascii = FALSE)
```

Arguments

|       |                                                                                                                                                                                                                                                                                                                                                               |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| x     | A spec from <code>julia_spec()</code> or <code>julia()</code> .                                                                                                                                                                                                                                                                                               |
| ascii | Logical. If TRUE, keyword names with non-ASCII characters are written with Unicode escapes, as string literals always are. The constructor name is inserted verbatim, as code supplied through <code>julia()</code> is, since Julia has no escape for an identifier in call position. The default gives the more readable form that can be pasted into Julia. |

**Value**

A character string of Julia code that constructs the spec.

**Examples**

```
as_julia(julia_spec("Solver", 10L, tol = 1e-8))

# A keyword name outside ASCII travels as an escape
greek <- list(1)
names(greek) <- "\u03f5_t"
as_julia(do.call(julia_spec, c("Process", greek)), ascii = TRUE)
```

---

as\_julia\_value

*Render a value of a class this package does not know*


---

**Description**

The extension point for a calling package with its own way of describing a value, such as a distribution object. Write a method returning either a spec from `julia_spec()` or a plain R value, and it renders wherever the value appears.

**Usage**

```
as_julia_value(x, ...)
```

**Arguments**

|     |                      |
|-----|----------------------|
| x   | The value to render. |
| ... | Passed to methods.   |

**Value**

A spec or an R value that renders on its own.

**Examples**

```
# A package with a value class of its own renders it like this:
as_julia_value.my_interval <- function(x, ...) {
  julia_spec("Interval", x$lower, x$upper)
}
```

---

|             |                                                             |
|-------------|-------------------------------------------------------------|
| assert_role | <i>Check that an argument is a spec of an accepted role</i> |
|-------------|-------------------------------------------------------------|

---

## Description

Composition errors are worth catching in R, where the argument can be named, rather than in Julia, where the error arrives from inside a constructor. A `julia()` expression with no role is accepted for any role, since the caller has said what it is by writing the code.

## Usage

```
assert_role(
  x,
  roles,
  null_ok = FALSE,
  arg_name = deparse(substitute(x)),
  labels = NULL
)
```

## Arguments

|                       |                                                                                                                                                                                                           |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>x</code>        | Object to check.                                                                                                                                                                                          |
| <code>roles</code>    | Character vector of accepted roles.                                                                                                                                                                       |
| <code>null_ok</code>  | Logical. Whether NULL is accepted.                                                                                                                                                                        |
| <code>arg_name</code> | Name used in error messages.                                                                                                                                                                              |
| <code>labels</code>   | Optional named character vector describing each role, used in the error message, for instance <code>c(prior = "a prior (e.g. Normal())")</code> . Roles without an entry are described by their own name. |

## Value

Invisibly TRUE.

## Examples

```
transform <- julia_spec("LogTransform", .role = "transform")
assert_role(transform, "transform")
try(assert_role(transform, "model"))
```

---

|              |                                                           |
|--------------|-----------------------------------------------------------|
| assign_julia | <i>Assign an R value to a name in Julia's Main module</i> |
|--------------|-----------------------------------------------------------|

---

**Description**

Equivalent in spirit to `JuliaCall::julia_assign(name, value)`. Useful when porting code that previously used the assign-then-eval pattern. Idiomatic JuliaConnector code prefers passing values directly via `call_julia()`; this is provided for migration convenience.

**Usage**

```
assign_julia(name, value)
```

**Arguments**

|       |                                |
|-------|--------------------------------|
| name  | Variable name to bind in Main. |
| value | R value to convert and assign. |

**Details**

Internally defines a small Julia helper `__juliabridge_assign__!` on first use and calls it with the (Symbol, value) pair.

**Value**

Invisibly NULL.

---

|            |                                                  |
|------------|--------------------------------------------------|
| call_julia | <i>Call a Julia function by (qualified) name</i> |
|------------|--------------------------------------------------|

---

**Description**

The function name may be module-qualified (e.g. `"Distributions.mean"`). Qualifying the name after using makes constructor names resolve unambiguously.

**Usage**

```
call_julia(name, ...)
```

**Arguments**

|      |                                             |
|------|---------------------------------------------|
| name | Function name, optionally module-qualified. |
| ...  | Arguments passed to the Julia function.     |

**Value**

The Julia function's return value, converted to R where reasonable.

---

|               |                                                 |
|---------------|-------------------------------------------------|
| command_julia | <i>Run a Julia command for its side effects</i> |
|---------------|-------------------------------------------------|

---

**Description**

Equivalent in spirit to `JuliaCall::julia_command(code)`: evaluate Julia code without using the return value. Provided for migration convenience; functionally identical to `eval_julia()` called for its side effects.

**Usage**

```
command_julia(code)
```

**Arguments**

|      |                         |
|------|-------------------------|
| code | A string of Julia code. |
|------|-------------------------|

**Value**

Invisibly the result of `juliaEval`.

---

|              |                                        |
|--------------|----------------------------------------|
| ensure_julia | <i>Lazy-init guard for Julia setup</i> |
|--------------|----------------------------------------|

---

**Description**

Convenience wrapper for the common "call my package's `setup_*`() function the first time any Julia-using function is called" pattern. Idempotent: once `state_env$ready` is `TRUE`, this is a no-op.

**Usage**

```
ensure_julia(state_env, init_fn)
```

**Arguments**

|           |                                                                                                                                                           |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| state_env | An environment shared with <code>julia_ready()</code> used to track init state. The caller's R package typically owns this environment.                   |
| init_fn   | A zero-argument function that performs the package's one-time setup (typically calling <code>julia_ready()</code> and <code>julia_load_bridge()</code> ). |

**Value**

Invisibly `TRUE`.

**Examples**

```
## Not run:
# Inside the consuming package:
.my_pkg_env <- new.env(parent = emptyenv())
setup_my_pkg <- function() {
  julia_ready(packages = "MyJuliaPkg", state_env = .my_pkg_env)
  julia_load_bridge("mypkg", "bridge.jl")
}
my_function <- function() {
  ensure_julia(.my_pkg_env, setup_my_pkg)
  call_julia("MyJuliaPkg.do_something")
}

## End(Not run)
```

eval\_julia

*Evaluate Julia code***Description**

Evaluate Julia code

**Usage**

```
eval_julia(code)
```

**Arguments**

code                      A string of Julia code.

**Value**

The result, converted to R where reasonable; otherwise a JuliaProxy object.

get\_julia

*Fully translate a Julia value into R***Description**

`eval_julia()` and `call_julia()` return composite Julia values (structs, Tuples, NamedTuples, Dicts) as proxy objects that reference the value inside the Julia session. This function translates such a proxy into a plain R object, e.g. a NamedTuple into a named list. Values that are already plain R objects pass through unchanged.

**Usage**

```
get_julia(x)
```

**Arguments**

x                      A Julia proxy object (or an already-translated R value).

**Value**

The value fully translated into R data structures.

---

|              |                              |
|--------------|------------------------------|
| import_julia | <i>Import a Julia module</i> |
|--------------|------------------------------|

---

**Description**

Returns a list/environment-like object of the module's exported names, callable as R functions. Equivalent to `juliaImport`.

**Usage**

```
import_julia(module)
```

**Arguments**

module                Name of the Julia module, e.g. "Distributions".

**Value**

A `JuliaModuleImport` object.

---

|       |                                   |
|-------|-----------------------------------|
| julia | <i>Embed Julia code in a spec</i> |
|-------|-----------------------------------|

---

**Description**

Marks a string as Julia source to be inserted verbatim when a spec is rendered, for arguments that cannot be expressed in R, such as functions or objects from other Julia packages.

**Usage**

```
julia(code, .role = NULL)
```

**Arguments**

code                    Character string of Julia code.  
 .role                   Optional role (see `julia_spec()`). Without one the expression is accepted wherever a spec is expected.

**Value**

An object of class `julia_code`, which is also a `julia_spec` and holds the code in `$code`. It has no `$fn`, `$args` or `$kwargs`, so a package walking a call branches on `inherits(x, "julia_code")` first.

**Examples**

```
# A Julia function, which has no R equivalent to render
julia_spec("Sampler", transform = julia("identity"))
```

---

julia\_alive

*Is the Julia session still usable?*


---

**Description**

`julia_ready()` records that setup has run and returns immediately when called again. That leaves a session stuck once its Julia process goes away, whether it was stopped, interrupted or died: the packages are no longer loaded, while the flag still says they are.

**Usage**

```
julia_alive(state_env, probe = NULL)
```

**Arguments**

|                        |                                                                                                                                                                                                                                                                                                       |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>state_env</code> | The environment given to <code>julia_ready()</code> .                                                                                                                                                                                                                                                 |
| <code>probe</code>     | Julia code returning a <code>Bool</code> , evaluated to decide whether the session still holds what the caller needs. The default checks that this is the server <code>state_env</code> was set up in. A caller can ask for something more specific, e.g. <code>"isdefined(Main, :MyBridge)"</code> . |

**Details**

This checks the running session and clears the flag when the check fails. The next `julia_ready()` call then sets Julia up again. The environment itself is kept, because callers may use its identity.

The check itself may start Julia, because `JuliaConnector` starts a new server whenever its connection has gone. The default probe is `FALSE` in such a server: only the server `julia_ready()` ran in holds the token it recorded.

**Value**

`TRUE` when the session is usable, otherwise `FALSE`.

**Examples**

```
## Not run:
if (!julia_alive(.my_pkg_env, "isdefined(Main, :MyBridge)")) {
  setup_my_pkg()
}

## End(Not run)
```

---

julia\_bin

*Find the Julia binary*


---

**Description**

Resolves the Julia binary path, used by `julia_subprocess()` to run subprocess work (installing packages into the default depot) before starting the JuliaConnector server.

**Usage**

```
julia_bin()
```

**Details**

Detection order:

1. JULIACONNECTOR\_JULIABIN env var (JuliaConnector's preferred mechanism).
2. JULIA\_BINDIR env var (Julia's own; `joinpath(JULIA_BINDIR, "julia")`).
3. The `julia` on the PATH (`Sys.which("julia")`).

**Value**

Absolute path to the Julia executable, or "" if not found.

---

julia\_handle

*Hold a Julia object from R*


---

**Description**

Some Julia results are too big, or too Julia-shaped, to convert: an MCMC chain a later call will sample from, a simulation state later summarised. The Julia side keeps such an object in a registry and returns an integer handle. R holds the handle in an environment whose finaliser queues it for release once nothing refers to it.

**Usage**

```
julia_handle(handle, session, state_env)
```

**Arguments**

|           |                                                       |
|-----------|-------------------------------------------------------|
| handle    | Integer handle returned by the Julia side.            |
| session   | Token identifying the Julia session that made it.     |
| state_env | The environment given to <code>julia_ready()</code> . |

**Details**

These functions deal with two hazards of that arrangement.

Handles are numbered per session. After a restart, an old handle names whatever object the new session gave that number. Each handle is therefore paired with a token for the session that made it. Both sides compare tokens before acting.

Saving an R object copies the environment by value but not its finaliser. A reloaded object then points at a Julia object it does not own. `julia_handle_owned()` detects this by comparing the state environment recorded in the handle with the current one.

A consumer package defines three things on the Julia side, in its own module where they precompile:

```
const HANDLES = Dict{Int, Any}()
const NEXT_HANDLE = Ref{Int}()
const SESSION = Ref{String}()

function __init__()
    SESSION[] = string(rand(Random.RandomDevice(), UInt128); base = 16)
    return nothing
end

function keep!(x)
    NEXT_HANDLE[] += 1
    HANDLES[NEXT_HANDLE[]] = x
    return NEXT_HANDLE[]
end

function release!(handle::Integer, session::AbstractString)
    session == SESSION[] && delete!(HANDLES, Int(handle))
    return nothing
end
```

A function returning a handle also returns `SESSION[]`. A function acting on a handle takes that token too and refuses a mismatch.

**Value**

An environment holding handle, session and the owning state\_env.

**Examples**

```
## Not run:
result <- call_julia("MyBridge.fit", data)
fit <- list(
  draws = result$draws,
  julia = julia_handle(result$handle, result$session, .my_pkg_env)
)

## End(Not run)
```

---

|                    |                                                                  |
|--------------------|------------------------------------------------------------------|
| julia_handle_owned | <i>Does this R session own the Julia object behind a handle?</i> |
|--------------------|------------------------------------------------------------------|

---

**Description**

Returns FALSE for a handle that arrived by saving and reloading, whose Julia object belongs to the session that created it. It also returns FALSE for a handle created before `julia_ready()` last set Julia up, whose object went with the old Julia server. Callers use this to tell the user the object came from disk or an earlier session.

**Usage**

```
julia_handle_owned(x, state_env)
```

**Arguments**

|           |                                                       |
|-----------|-------------------------------------------------------|
| x         | A handle from <code>julia_handle()</code> .           |
| state_env | The environment given to <code>julia_ready()</code> . |

**Value**

TRUE when the current setup of state\_env created the handle.

**Examples**

```
## Not run:
if (!julia_handle_owned(fit$julia, .my_pkg_env)) {
  stop("This fit was loaded from disk; refit it to continue.")
}

## End(Not run)
```

---

|                   |                                                                             |
|-------------------|-----------------------------------------------------------------------------|
| julia_load_bridge | <i>Load Julia bridge (.jl) files from a package's inst/julia/ directory</i> |
|-------------------|-----------------------------------------------------------------------------|

---

**Description**

Reads each file as a string and evaluates it via `JuliaConnector::juliaEval()` in the running Julia server. Bridge files typically define helper functions used by the wrapping R package; loading them once after package setup makes them available to subsequent `juliaCall()` invocations.

**Usage**

```
julia_load_bridge(package, files, verbose = FALSE)
```

**Arguments**

|         |                                                                                                                                                                  |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| package | Name of the calling R package (used as the package argument to <code>system.file()</code> ). Files are looked up under <code>inst/julia/</code> of that package. |
| files   | Character vector of .jl filenames (no directory).                                                                                                                |
| verbose | If TRUE, print a message per loaded file.                                                                                                                        |

**Value**

Invisibly the character vector of paths that were loaded.

**Examples**

```
## Not run:
julia_load_bridge("ringbpjl",
  c("dist_lookup.jl", "simulate.jl", "generation_time.jl"))

## End(Not run)
```

---

|             |                                                           |
|-------------|-----------------------------------------------------------|
| julia_ready | <i>Ensure Julia and required Julia packages are ready</i> |
|-------------|-----------------------------------------------------------|

---

**Description**

Performs the steps needed to make Julia and a set of Julia packages callable from R via `JuliaConnector`:

**Usage**

```
julia_ready(
  packages,
  github = character(),
  state_env = new.env(parent = emptyenv()),
  install = TRUE,
  project = NULL,
  match_manifest = TRUE,
  verbose = TRUE
)
```

**Arguments**

|                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| packages       | Character vector of Julia package names to ensure are loaded (e.g. <code>c("EpiBranch", "Distributions")</code> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| github         | Named character vector of GitHub URLs for packages not in the General registry. Names must match entries in packages. Values may be a full URL, an "owner/repo" shorthand, or "owner/repo:subdir".                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| state_env      | An environment used to track initialisation state. The caller (typically a wrapping R package) supplies its own environment. Several consuming packages then keep separate state.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| install        | If FALSE, a missing package is an error.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| project        | Optional path to a Julia project directory containing a <code>Project.toml</code> (and ideally a <code>Manifest.toml</code> ). When supplied, the project is activated and instantiated in a subprocess. Setting <code>JULIA_PROJECT</code> before <code>JuliaConnector</code> starts makes the server use the project. Use this when your package ships a pinned Julia environment under <code>inst/julia/</code> . <code>Pkg.instantiate()</code> fetches everything in the project's manifest; packages then only names the packages to load with using.                                                                                                                            |
| match_manifest | If TRUE and project is supplied, read the Julia version its <code>Manifest.toml</code> was resolved with and use that version, installing it with <code>juliaup</code> where available. Instantiating a manifest under a different Julia can fail, because it pins standard-library versions that exist only on the version that resolved it. Ignored when the user has chosen a binary through <code>JULIACONNECTOR_JULIABIN</code> or <code>JULIA_BINDIR</code> . A <code>JuliaConnector</code> server that is already running, for instance one another package started, keeps its Julia: the project is then instantiated under the matched version but loaded in the running one. |
| verbose        | If TRUE, print progress messages.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

**Details**

1. Locates the Julia binary (see `julia_bin()`).
2. Without project, checks in a Julia subprocess that each required package loads. If a package is missing and `install = TRUE`, installs it (from a GitHub URL if listed in `github`, otherwise from the General registry). With project, instantiates that project instead. Both run in a subprocess, apart from the `JuliaConnector` server.
3. Starts (or attaches to) the `JuliaConnector` server.

4. Loads each package with `juliaEval("using <pkg>")`. Dotted constructor names such as `EpiBranch.NegBin` then resolve.

Idempotent: if `state_env$ready` is already `TRUE`, returns immediately.

### Value

Invisibly `TRUE`.

### Examples

```
## Not run:
.my_pkg_env <- new.env(parent = emptyenv())
julia_ready(
  packages = c("EpiBranch", "Distributions", "Random"),
  github   = c(EpiBranch = "epiforecasts/EpiBranch.jl"),
  state_env = .my_pkg_env
)

## End(Not run)
```

---

`julia_release_pending` *Release the Julia objects of collected handles*

---

### Description

Because a finaliser can run partway through another Julia call, it only adds the handle to a queue. This function flushes the queue. Call it immediately before a Julia call of your own.

### Usage

```
julia_release_pending(state_env, release)
```

### Arguments

|                        |                                                                                         |
|------------------------|-----------------------------------------------------------------------------------------|
| <code>state_env</code> | The environment given to <code>julia_ready()</code> .                                   |
| <code>release</code>   | Name of the Julia function releasing a handle, taking the handle and the session token. |

### Details

Only handles from the current setup of `state_env` are released. The queue is dropped when `state_env` is not set up. Handles from an earlier setup are dropped too, because their objects went with the old Julia server. A release that fails stays queued for the next call, up to three attempts, after which it is dropped: a release that keeps failing usually means the server was replaced without `julia_alive()` noticing.

### Value

Invisibly the number of handles released.

## Examples

```
## Not run:
julia_release_pending(.my_pkg_env, "MyBridge.release!")
call_julia("MyBridge.forecast", handle, horizon)

## End(Not run)
```

---

julia\_spec

*Describe a Julia constructor call*


---

## Description

Records a call to a Julia constructor as an R object, rendered to Julia source only when it is run. A package wrapping a Julia library builds its own constructors on top of this, and exposes it directly so that users can reach anything it has not wrapped.

## Usage

```
julia_spec(.fn, ..., .role = NULL)
```

## Arguments

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>.fn</code>   | Character string. Name of the Julia constructor, which may be qualified by its module and may hold a parameter list, as in <code>"Vector{Float64}"</code> . A parameter list may not hold a parenthesis or a quote, so a name such as <code>"NamedTuple{(:a, :b), Tuple{Int, Int}}"</code> has to be written with <code>julia()</code> . It is spelled with a dot so that a Julia keyword named <code>f</code> or <code>fn</code> still reaches <code>...</code> rather than being taken for this argument. |
| <code>...</code>   | Arguments to the constructor. Unnamed arguments are positional and named arguments become keyword arguments. A keyword name is refused if it begins with a digit, holds a space or comma, holds a character that would take it out of the call, or repeats another; beyond that, which names Julia accepts is left to Julia, and a name may hold non-ASCII characters. Every keyword name is checked, including one whose value is <code>NULL</code> and so is dropped.                                     |
| <code>.role</code> | Optional character string naming what the spec is, in whatever vocabulary the calling package uses (for example <code>"prior"</code> or <code>"model"</code> ). It becomes a class, so <code>assert_role()</code> can check that specs are composed sensibly. <code>NULL</code> leaves the spec untyped, which every role accepts. It is spelled with a dot for the same reason as <code>.fn</code> , so that a Julia keyword named <code>role</code> reaches <code>...</code>                              |

## Details

Arguments are rendered to Julia as follows: specs and `julia()` expressions are inserted as code, numeric vectors of length one become scalars and longer ones become vectors, unnamed lists become vectors, `NA` becomes missing, and character strings become Julia strings. Integers (e.g. `2L`) render as Julia integers and doubles as floats. A `NULL` keyword argument is dropped, so the Julia

default applies; a NULL positional argument is an error, since dropping it would renumber the arguments that follow. A matrix or array keeps its shape, arriving in Julia as a `Matrix` or `Array` of the same dimensions. A named vector, a named list and an empty value are all refused, since Julia would read them as something the R value did not say.

## Value

An object of class `julia_spec`, holding the constructor name in `$fn`, the positional arguments in `$args` and the keyword arguments in `$kwargs`. A package may read those to inspect or rewrite a call.

## Examples

```
# Any Julia constructor, by name. R types map across: `10L` is a Julia
# integer, `1e-8` a float and `"BFGS"` a Julia string.
julia_spec("Solver", 10L, tol = 1e-8, method = "BFGS")

# A name may be qualified by its module, and a spec may nest in another
julia_spec("MyModule.Problem", julia_spec("Solver", 10L), verbose = TRUE)

# A matrix keeps its shape, reaching Julia as a Matrix
as_julia(julia_spec("Weights", matrix(1:4, nrow = 2)))

# A role lets the calling package check how specs are composed
julia_spec("Normal", 0, 1, .role = "prior")
```

---

juliaup\_julia

---

*Install a Julia version with juliaup and return its binary*


---

## Description

The user's default channel stays as it is. The caller uses the returned binary, typically through `JULIACONNECTOR_JULIABIN`.

## Usage

```
juliaup_julia(version, verbose = TRUE)
```

## Arguments

|                      |                                        |
|----------------------|----------------------------------------|
| <code>version</code> | Julia version to install, e.g. "1.12". |
| <code>verbose</code> | If TRUE, print progress messages.      |

## Value

Path to the Julia executable, or NULL when juliaup is absent or the installation failed.

### Examples

```
## Not run:  
juliaup_julia("1.12")  
  
## End(Not run)
```

---

```
manifest_julia_version
```

*Julia version a Manifest was resolved with*

---

### Description

A `Manifest.toml` records the Julia version that resolved it and pins standard-library versions that only exist on that version. A package shipping a pinned project needs that Julia version to instantiate cleanly.

### Usage

```
manifest_julia_version(project)
```

### Arguments

|         |                                                                                                                                        |
|---------|----------------------------------------------------------------------------------------------------------------------------------------|
| project | Path to a Julia project directory containing a <code>Manifest.toml</code> or <code>Manifest-v&lt;major&gt;.&lt;minor&gt;.toml</code> . |
|---------|----------------------------------------------------------------------------------------------------------------------------------------|

### Details

A versioned manifest (`Manifest-v1.12.toml`, supported from Julia 1.11) takes precedence over `Manifest.toml` on the Julia version it names. When any exist, the highest-versioned one is read. If it records no Julia version, its file name supplies one.

### Value

The major and minor version as a string (e.g. `"1.12"`), or `NULL` when there is no manifest or it records no version.

### Examples

```
## Not run:  
manifest_julia_version(system.file("julia", package = "MyPkg"))  
  
## End(Not run)
```

---

wrappers*Backend-agnostic wrappers around Julia eval / call / import*

---

**Description**

These thin wrappers let consumer packages call into Julia through `juliabridge` alone. If `juliabridge` ever switches backend, consumer code keeps working.

# Index

as\_julia, [2](#)  
as\_julia\_value, [3](#)  
assert\_role, [4](#)  
assert\_role(), [16](#)  
assign\_julia, [5](#)  
  
call\_julia, [5](#)  
call\_julia(), [5](#), [7](#)  
command\_julia, [6](#)  
  
ensure\_julia, [6](#)  
eval\_julia, [7](#)  
eval\_julia(), [6](#), [7](#)  
  
get\_julia, [7](#)  
  
import\_julia, [8](#)  
  
julia, [8](#)  
julia(), [2](#), [4](#), [16](#)  
julia\_alive, [9](#)  
julia\_alive(), [15](#)  
julia\_bin, [10](#)  
julia\_bin(), [14](#)  
julia\_handle, [10](#)  
julia\_handle(), [12](#)  
julia\_handle\_owned, [12](#)  
julia\_handle\_owned(), [11](#)  
julia\_load\_bridge, [13](#)  
julia\_load\_bridge(), [6](#)  
julia\_ready, [13](#)  
julia\_ready(), [6](#), [9](#), [11](#), [12](#), [15](#)  
julia\_release\_pending, [15](#)  
julia\_spec, [16](#)  
julia\_spec(), [2](#), [3](#), [8](#)  
juliaup\_julia, [17](#)  
  
manifest\_julia\_version, [18](#)  
  
wrappers, [19](#)