Environment Variables

Julia can be configured with a number of environment variables, set either in the usual way for each operating system, or in a portable way from within Julia. Supposing that you want to set the environment variable `JULIA_EDITOR` to `vim`, you can type `ENV["JULIA_EDITOR"] = "vim"` (for instance, in the REPL) to make this change on a case by case basis, or add the same to the user configuration file `~/.julia/config/startup.jl` in the user's home directory to have a permanent effect. The current value of the same environment variable can be determined by evaluating `ENV["JULIA_EDITOR"]`.

The environment variables that Julia uses generally start with `JULIA`. If `InteractiveUtils.versioninfo` is called with the keyword `verbose=true`, then the output will list any defined environment variables relevant for Julia, including those which include `JULIA` in their names.

Note

Some variables, such as `JULIA_NUM_THREADS` and `JULIA_PROJECT`, need to be set before Julia starts, therefore adding these to `~/.julia/config/startup.jl` is too late in the startup process. In Bash, environment variables can either be set manually by running, e.g., `export JULIA_NUM_THREADS=4` before starting Julia, or by adding the same command to `~/.bashrc` or `~/.bash_profile` to set the variable each time Bash is started.

File locations

JULIA_BINDIR

The absolute path of the directory containing the Julia executable, which sets the global variable `Sys.BINDIR`. If `$JULIA_BINDIR` is not set, then Julia determines the value `Sys.BINDIR` at run-time.

The executable itself is one of

```
$JULIA_BINDIR/julia
$JULIA_BINDIR/julia-debug
```

by default.

The global variable `Base.DATAROOTDIR` determines a relative path from `Sys.BINDIR` to the data directory associated with Julia. Then the path

```
$JULIA_BINDIR/$DATAROOTDIR/julia/base
```

determines the directory in which Julia initially searches for source files (via `Base.find_source_file()`).

Likewise, the global variable `Base.SYSCONFDIR` determines a relative path to the configuration file directory. Then Julia searches for a `startup.jl` file at

```
$JULIA_BINDIR/$SYSCONFDIR/julia/startup.jl
$JULIA_BINDIR/../etc/julia/startup.jl
```

by default (via `Base.load_julia_startup()`).

For example, a Linux installation with a Julia executable located at `/bin/julia`, a `DATAROOTDIR` of `../share`, and a `SYSCONFDIR` of `../etc` will have `JULIA_BINDIR` set to `/bin`, a source-file search path of

```
/share/julia/base
```

and a global configuration search path of

```
/etc/julia/startup.jl
```

JULIA_PROJECT

A directory path that indicates which project should be the initial active project. Setting this environment variable has the same effect as specifying the `--project` start-up option, but `--project` has higher precedence. If the variable is set to `@.` then Julia tries to find a project directory that contains `Project.toml` or `JuliaProject.toml` file from the current directory and its parents. See also the chapter on Code Loading.

Note

`JULIA_PROJECT` must be defined before starting julia; defining it in `startup.jl` is too late in the startup process.

JULIA_LOAD_PATH

The `JULIA_LOAD_PATH` environment variable is used to populate the global Julia `LOAD_PATH` variable, which determines which packages can be loaded via `import` and `using` (see Code Loading).

Unlike the shell `PATH` variable, empty entries in `JULIA_LOAD_PATH` are expanded to the default value of `LOAD_PATH`, `["@", "@v#.#", "@stdlib"]` when populating `LOAD_PATH`. This allows easy appending, prepending, etc. of the load path value in shell scripts regardless of whether `JULIA_LOAD_PATH` is already set or not. For example, to prepend the directory `/foo/bar` to `LOAD_PATH` just do

```
export JULIA_LOAD_PATH="/foo/bar:$JULIA_LOAD_PATH"
```

If the `JULIA_LOAD_PATH` environment variable is already set, its old value will be prepended with `/foo/bar`. On the other hand, if `JULIA_LOAD_PATH` is not set, then it will be set to `/foo/bar:` which will expand to a `LOAD_PATH` value of `["/foo/bar", "@", "@v#.#", "@stdlib"]`. If `JULIA_LOAD_PATH` is set to the empty string, it expands to an empty `LOAD_PATH` array. In other words, the empty string is interpreted as a zero-element array, not a one-element array of the empty string. This behavior was chosen so that it would be possible to set an empty load path via the environment variable. If you want the default load path, either unset the environment variable or if it must have a value, set it to the string `:`.

JULIA_DEPOT_PATH

The `JULIA_DEPOT_PATH` environment variable is used to populate the global Julia `DEPOT_PATH` variable, which controls where the package manager, as well as Julia's code loading mechanisms, look for package registries, installed packages, named environments, repo clones, cached compiled package images, configuration files, and the default location of the REPL's history file.

Unlike the shell `PATH` variable but similar to `JULIA_LOAD_PATH`, empty entries in `JULIA_DEPOT_PATH` are expanded to the default value of `DEPOT_PATH`. This allows easy appending, prepending, etc. of the depot path value in shell scripts regardless of whether `JULIA_DEPOT_PATH` is already set or not. For example, to prepend the directory `/foo/bar` to `DEPOT_PATH` just do

```
export JULIA_DEPOT_PATH="/foo/bar:$JULIA_DEPOT_PATH"
```

If the `JULIA_DEPOT_PATH` environment variable is already set, its old value will be prepended with `/foo/bar`. On the other hand, if `JULIA_DEPOT_PATH` is not set, then it will be set to `/foo/bar:` which will have the effect of prepending `/foo/bar` to the default depot path. If `JULIA_DEPOT_PATH` is set to the empty string, it expands to an empty `DEPOT_PATH` array. In other words, the empty string is interpreted as a zero-element array, not a one-element array of the empty string. This behavior was chosen so that it would be possible to set an empty depot path via the environment variable. If you want the default depot path, either unset the environment variable or if it must have a value, set it to the string `:`.

JULIA_HISTORY

The absolute path `REPL.find_hist_file()` of the REPL's history file. If `$JULIA_HISTORY` is not set, then `REPL.find_hist_file()` defaults to

```
$(DEPOT_PATH[1])/logs/repl_history.jl
```

External applications

JULIA_SHELL

The absolute path of the shell with which Julia should execute external commands (via `Base.repl_cmd()`). Defaults to the environment variable `$SHELL`, and falls back to `/bin/sh` if `$SHELL` is unset.

Note

On Windows, this environment variable is ignored, and external commands are executed directly.

JULIA_EDITOR

The editor returned by `InteractiveUtils.editor()` and used in, e.g., `InteractiveUtils.edit`, referring to the command of the preferred editor, for instance `vim`.

`$JULIA_EDITOR` takes precedence over `$VISUAL`, which in turn takes precedence over `$EDITOR`. If none of these environment variables is set, then the editor is taken to be `open` on Windows and OS X, or `/etc/alternatives/editor` if it exists, or `emacs` otherwise.

Parallelization

JULIA_CPU_THREADS

Overrides the global variable `Base.Sys.CPU_THREADS`, the number of logical CPU cores available.

JULIA_WORKER_TIMEOUT

A `Float64` that sets the value of `Distributed.worker_timeout()` (default: `60.0`). This function gives the number of seconds a worker process will wait for a master process to establish a connection before dying.

JULIA_NUM_THREADS

An unsigned 64-bit integer (`uint64_t`) that sets the maximum number of threads available to Julia. If `$JULIA_NUM_THREADS` exceeds the number of available physical CPU cores, then the number of threads is set to the number of cores. If `$JULIA_NUM_THREADS` is not positive or is not set, or if the number of CPU cores cannot be determined through system calls, then the number of threads is set to `1`.

> Note
>
> `JULIA_NUM_THREADS` must be defined before starting julia; defining it in `startup.jl` is too late in the startup process.

> Julia 1.5
>
> In Julia 1.5 and above the number of threads can also be specified on startup using the `-t`/`--threads` command line argument.

JULIA_THREAD_SLEEP_THRESHOLD

If set to a string that starts with the case-insensitive substring `"infinite"`, then spinning threads never sleep. Otherwise, `$JULIA_THREAD_SLEEP_THRESHOLD` is interpreted as an unsigned 64-bit integer (`uint64_t`) and gives, in nanoseconds, the amount of time after which spinning threads should sleep.

JULIA_EXCLUSIVE

If set to anything besides `0`, then Julia's thread policy is consistent with running on a dedicated machine: the master thread is on proc 0, and threads are affinitized. Otherwise, Julia lets the operating system handle thread policy.

REPL formatting

Environment variables that determine how REPL output should be formatted at the terminal. Generally, these variables should be set to ANSI terminal escape sequences. Julia provides a high-level interface with much of the same functionality; see the section on The Julia REPL.

`JULIA_ERROR_COLOR`

The formatting `Base.error_color()` (default: light red, `"\033[91m"`) that errors should have at the terminal.

`JULIA_WARN_COLOR`

The formatting `Base.warn_color()` (default: yellow, `"\033[93m"`) that warnings should have at the terminal.

`JULIA_INFO_COLOR`

The formatting `Base.info_color()` (default: cyan, `"\033[36m"`) that info should have at the terminal.

`JULIA_INPUT_COLOR`

The formatting `Base.input_color()` (default: normal, `"\033[0m"`) that input should have at the terminal.

`JULIA_ANSWER_COLOR`

The formatting `Base.answer_color()` (default: normal, `"\033[0m"`) that output should have at the terminal.

`JULIA_STACKFRAME_LINEINFO_COLOR`

The formatting `Base.stackframe_lineinfo_color()` (default: bold, `"\033[1m"`) that line info should have during a stack trace at the terminal.

`JULIA_STACKFRAME_FUNCTION_COLOR`

The formatting `Base.stackframe_function_color()` (default: bold, `"\033[1m"`) that function calls should have during a stack trace at the terminal.

Debugging and profiling

`JULIA_DEBUG`

Enable debug logging for a file or module, see `Logging` for more information.

`JULIA_GC_ALLOC_POOL`, `JULIA_GC_ALLOC_OTHER`, `JULIA_GC_ALLOC_PRINT`

If set, these environment variables take strings that optionally start with the character `'r'`, followed by a string interpolation of a colon-separated list of three signed 64-bit integers (`int64_t`). This triple of integers `a:b:c` represents the arithmetic sequence `a`, `a + b`, `a + 2*b`, ... `c`.

- If it's the `n`th time that `jl_gc_pool_alloc()` has been called, and `n` belongs to the arithmetic sequence represented by `$JULIA_GC_ALLOC_POOL`, then garbage collection is forced.
- If it's the `n`th time that `maybe_collect()` has been called, and `n` belongs to the arithmetic sequence represented by `$JULIA_GC_ALLOC_OTHER`, then garbage collection is forced.
- If it's the `n`th time that `jl_gc_collect()` has been called, and `n` belongs to the arithmetic sequence represented by `$JULIA_GC_ALLOC_PRINT`, then counts for the number of calls to `jl_gc_pool_alloc()` and `maybe_collect()` are printed.

If the value of the environment variable begins with the character `'r'`, then the interval between garbage collection events is randomized.

Note

These environment variables only have an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to `1` in the build configuration).

`JULIA_GC_NO_GENERATIONAL`

If set to anything besides `0`, then the Julia garbage collector never performs "quick sweeps" of memory.

Note

This environment variable only has an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to `1` in the build configuration).

`JULIA_GC_WAIT_FOR_DEBUGGER`

If set to anything besides `0`, then the Julia garbage collector will wait for a debugger to attach instead of aborting whenever there's a critical error.

Note

This environment variable only has an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to `1` in the build configuration).

`ENABLE_JITPROFILING`

If set to anything besides `0`, then the compiler will create and register an event listener for just-in-time (JIT) profiling.

Note

This environment variable only has an effect if Julia was compiled with JIT profiling support, using either

- Intel's VTune™ Amplifier (`USE_INTEL_JITEVENTS` set to `1` in the build configuration), or
- OProfile (`USE_OPROFILE_JITEVENTS` set to `1` in the build configuration).

`JULIA_LLVM_ARGS`

Arguments to be passed to the LLVM backend.

« Handling Operating System Variation Embedding Julia »