Filesystem

`Base.Filesystem.pwd` — Function

```
pwd() -> AbstractString
```

Get the current working directory.

Examples

```
julia> pwd()
"/home/JuliaUser"

julia> cd("/home/JuliaUser/Projects/julia")

julia> pwd()
"/home/JuliaUser/Projects/julia"
```

`Base.Filesystem.cd` — Method

```
cd(dir::AbstractString=homedir())
```

Set the current working directory.

Examples

```
julia> cd("/home/JuliaUser/Projects/julia")

julia> pwd()
"/home/JuliaUser/Projects/julia"

julia> cd()

julia> pwd()
"/home/JuliaUser"
```

`Base.Filesystem.cd` — Method

```
cd(f::Function, dir::AbstractString=homedir())
```

Temporarily change the current working directory to `dir`, apply function `f` and finally return to the original directory.

Examples

```
julia> pwd()
"/home/JuliaUser"

julia> cd(readdir, "/home/JuliaUser/Projects/julia")
34-element Array{String,1}:
 ".circleci"
 ".freebsdci.sh"
 ".git"
 ".gitattributes"
 ".github"
 ⋮
 "test"
 "ui"
 "usr"
 "usr-staging"
```

```
julia> pwd()
"/home/JuliaUser"
```

`Base.Filesystem.readdir` — Function

```
readdir(dir::AbstractString=pwd();
    join::Bool = false,
    sort::Bool = true,
) -> Vector{String}
```

Return the names in the directory `dir` or the current working directory if not given. When `join` is false, `readdir` returns just the names in the directory as is; when `join` is true, it returns `joinpath(dir, name)` for each `name` so that the returned strings are full paths. If you want to get absolute paths back, call `readdir` with an absolute directory path and `join` set to true.

By default, `readdir` sorts the list of names it returns. If you want to skip sorting the names and get them in the order that the file system lists them, you can use `readir(dir, sort=false)` to opt out of sorting.

> **Julia 1.4**
>
> The `join` and `sort` keyword arguments require at least Julia 1.4.

Examples

```
julia> cd("/home/JuliaUser/dev/julia")

julia> readdir()
30-element Array{String,1}:
 ".appveyor.yml"
 ".git"
 ".gitattributes"
 ⋮
 "ui"
 "usr"
 "usr-staging"

julia> readdir(join=true)
```

```
30-element Array{String,1}:
 "/home/JuliaUser/dev/julia/.appveyor.yml"
 "/home/JuliaUser/dev/julia/.git"
 "/home/JuliaUser/dev/julia/.gitattributes"
 ⋮
 "/home/JuliaUser/dev/julia/ui"
 "/home/JuliaUser/dev/julia/usr"
 "/home/JuliaUser/dev/julia/usr-staging"

julia> readdir("base")
145-element Array{String,1}:
 ".gitignore"
 "Base.jl"
 "Enums.jl"
 ⋮
 "version_git.sh"
 "views.jl"
 "weakkeydict.jl"

julia> readdir("base", join=true)
145-element Array{String,1}:
 "base/.gitignore"
 "base/Base.jl"
 "base/Enums.jl"
 ⋮
 "base/version_git.sh"
 "base/views.jl"
 "base/weakkeydict.jl"```

julia> readdir(abspath("base"), join=true)
145-element Array{String,1}:
 "/home/JuliaUser/dev/julia/base/.gitignore"
 "/home/JuliaUser/dev/julia/base/Base.jl"
 "/home/JuliaUser/dev/julia/base/Enums.jl"
 ⋮
 "/home/JuliaUser/dev/julia/base/version_git.sh"
 "/home/JuliaUser/dev/julia/base/views.jl"
 "/home/JuliaUser/dev/julia/base/weakkeydict.jl"
```

`Base.Filesystem.walkdir` — Function

```
walkdir(dir; topdown=true, follow_symlinks=false, onerror=throw)
```

Return an iterator that walks the directory tree of a directory. The iterator returns a tuple containing `(rootpath, dirs, files)`. The directory tree can be traversed top-down or bottom-up. If `walkdir` encounters a `SystemError` it will rethrow the error by default. A custom error handling function can be provided through `onerror` keyword argument. `onerror` is called with a `SystemError` as argument.

Examples

```
for (root, dirs, files) in walkdir(".")
    println("Directories in $root")
    for dir in dirs
        println(joinpath(root, dir)) # path to directories
    end
    println("Files in $root")
    for file in files
        println(joinpath(root, file)) # path to files
    end
end
```

```
julia> mkpath("my/test/dir");

julia> itr = walkdir("my");

julia> (root, dirs, files) = first(itr)
("my", ["test"], String[])

julia> (root, dirs, files) = first(itr)
("my/test", ["dir"], String[])

julia> (root, dirs, files) = first(itr)
("my/test/dir", String[], String[])
```

`Base.Filesystem.mkdir` — Function

```
mkdir(path::AbstractString; mode::Unsigned = 0o777)
```

Make a new directory with name `path` and permissions `mode`. `mode` defaults to `0o777`, modified by the current file creation mask. This function never creates more than one directory. If the directory already exists, or some intermediate directories do not exist, this function throws an error. See `mkpath` for a function which creates all required intermediate directories. Return `path`.

Examples

```
julia> mkdir("testingdir")
"testingdir"

julia> cd("testingdir")

julia> pwd()
"/home/JuliaUser/testingdir"
```

Base.Filesystem.mkpath — Function

```
mkpath(path::AbstractString; mode::Unsigned = 0o777)
```

Create all directories in the given `path`, with permissions `mode`. `mode` defaults to `0o777`, modified by the current file creation mask. Return `path`.

Examples

```
julia> mkdir("testingdir")
"testingdir"

julia> cd("testingdir")

julia> pwd()
"/home/JuliaUser/testingdir"

julia> mkpath("my/test/dir")
"my/test/dir"

julia> readdir()
1-element Array{String,1}:
 "my"

julia> cd("my")

julia> readdir()
1-element Array{String,1}:
 "test"

julia> readdir("test")
```

```
1-element Array{String,1}:
 "dir"
```

Base.Filesystem.symlink — Function

```
symlink(target::AbstractString, link::AbstractString)
```

Creates a symbolic link to `target` with the name `link`.

> **Note**
>
> This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

Base.Filesystem.readlink — Function

```
readlink(path::AbstractString) -> AbstractString
```

Return the target location a symbolic link `path` points to.

Base.Filesystem.chmod — Function

```
chmod(path::AbstractString, mode::Integer; recursive::Bool=false)
```

Change the permissions mode of `path` to `mode`. Only integer `mode`s (e.g. `0o777`) are currently supported. If `recursive=true` and the path is a directory all permissions in that directory will be recursively changed. Return `path`.

Base.Filesystem.chown — Function

```
chown(path::AbstractString, owner::Integer, group::Integer=-1)
```

Change the owner and/or group of `path` to `owner` and/or `group`. If the value entered for `owner` or `group` is `-1` the corresponding ID will not change. Only integer `owner`s and `group`s are currently supported. Return `path`.

`Base.Libc.RawFD` — Type

```
RawFD
```

Primitive type which wraps the native OS file descriptor. `RawFD`s can be passed to methods like `stat` to discover information about the underlying file, and can also be used to open streams, with the `RawFD` describing the OS file backing the stream.

`Base.stat` — Function

```
stat(file)
```

Returns a structure whose fields contain information about the file. The fields of the structure are:

Name	Description
size	The size (in bytes) of the file
device	ID of the device that contains the file
inode	The inode number of the file
mode	The protection mode of the file
nlink	The number of hard links to the file
uid	The user id of the owner of the file
gid	The group id of the file owner
rdev	If this file refers to a device, the ID of the device it refers to
blksize	The file-system preferred block size for the file
blocks	The number of such blocks allocated

mtime	Unix timestamp of when the file was last modified
ctime	Unix timestamp of when the file was created

`Base.Filesystem.lstat` — Function

```
lstat(file)
```

Like `stat`, but for symbolic links gets the info for the link itself rather than the file it refers to. This function must be called on a file path rather than a file object or a file descriptor.

`Base.Filesystem.ctime` — Function

```
ctime(file)
```

Equivalent to `stat(file).ctime`.

`Base.Filesystem.mtime` — Function

```
mtime(file)
```

Equivalent to `stat(file).mtime`.

`Base.Filesystem.filemode` — Function

```
filemode(file)
```

Equivalent to `stat(file).mode`.

`Base.Filesystem.filesize` — Function

```
filesize(path...)
```

Equivalent to `stat(file).size`.

`Base.Filesystem.uperm` — Function

```
uperm(file)
```

Get the permissions of the owner of the file as a bitfield of

Value	Description
01	Execute Permission
02	Write Permission
04	Read Permission

For allowed arguments, see `stat`.

`Base.Filesystem.gperm` — Function

```
gperm(file)
```

Like `uperm` but gets the permissions of the group owning the file.

`Base.Filesystem.operm` — Function

```
operm(file)
```

Like `uperm` but gets the permissions for people who neither own the file nor are a member of the group owning the file

Base.Filesystem.cp — Function

```
cp(src::AbstractString, dst::AbstractString; force::Bool=false, follow_symlinks
```

Copy the file, link, or directory from `src` to `dst`. `force=true` will first remove an existing `dst`.

If `follow_symlinks=false`, and `src` is a symbolic link, `dst` will be created as a symbolic link. If `follow_symlinks=true` and `src` is a symbolic link, `dst` will be a copy of the file or directory `src` refers to. Return `dst`.

Base.download — Function

```
download(url::AbstractString, [localfile::AbstractString])
```

Download a file from the given url, optionally renaming it to the given local file name. If no filename is given this will download into a randomly-named file in your temp directory. Note that this function relies on the availability of external tools such as `curl`, `wget` or `fetch` to download the file and is provided for convenience. For production use or situations in which more options are needed, please use a package that provides the desired functionality instead.

Returns the filename of the downloaded file.

Base.Filesystem.mv — Function

```
mv(src::AbstractString, dst::AbstractString; force::Bool=false)
```

Move the file, link, or directory from `src` to `dst`. `force=true` will first remove an existing `dst`. Return `dst`.

Examples

```
julia> write("hello.txt", "world");

julia> mv("hello.txt", "goodbye.txt")
"goodbye.txt"

julia> "hello.txt" in readdir()
```

```
false

julia> readline("goodbye.txt")
"world"

julia> write("hello.txt", "world2");

julia> mv("hello.txt", "goodbye.txt")
ERROR: ArgumentError: 'goodbye.txt' exists. `force=true` is required to remove
Stacktrace:
 [1] #checkfor_mv_cp_cptree#10(::Bool, ::Function, ::String, ::String, ::String
[...]

julia> mv("hello.txt", "goodbye.txt", force=true)
"goodbye.txt"

julia> rm("goodbye.txt");
```

`Base.Filesystem.rm` — Function

```
rm(path::AbstractString; force::Bool=false, recursive::Bool=false)
```

Delete the file, link, or empty directory at the given path. If `force=true` is passed, a non-existing path is not treated as error. If `recursive=true` is passed and the path is a directory, then all contents are removed recursively.

Examples

```
julia> mkpath("my/test/dir");

julia> rm("my", recursive=true)

julia> rm("this_file_does_not_exist", force=true)

julia> rm("this_file_does_not_exist")
ERROR: IOError: unlink: no such file or directory (ENOENT)
Stacktrace:
[...]
```

`Base.Filesystem.touch` — Function

```
touch(path::AbstractString)
```

Update the last-modified timestamp on a file to the current time.

If the file does not exist a new file is created.

Return `path`.

Examples

```
julia> write("my_little_file", 2);

julia> mtime("my_little_file")
1.5273815391135583e9

julia> touch("my_little_file");

julia> mtime("my_little_file")
1.527381559163435e9
```

We can see the `mtime` has been modified by `touch`.

`Base.Filesystem.tempname` — Function

```
tempname(parent=tempdir(); cleanup=true) -> String
```

Generate a temporary file path. This function only returns a path; no file is created. The path is likely to be unique, but this cannot be guaranteed due to the very remote posibility of two simultaneous calls to `tempname` generating the same file name. The name is guaranteed to differ from all files already existing at the time of the call to `tempname`.

When called with no arguments, the temporary name will be an absolute path to a temporary name in the system temporary directory as given by `tempdir()`. If a `parent` directory argument is given, the temporary path will be in that directory instead.

The `cleanup` option controls whether the process attempts to delete the returned path automatically when the process exits. Note that the `tempname` function does not create any file or

directory at the returned location, so there is nothing to cleanup unless you create a file or directory there. If you do and `clean` is `true` it will be deleted upon process termination.

> **Julia 1.4**
>
> The `parent` and `cleanup` arguments were added in 1.4. Prior to Julia 1.4 the path `tempname` would never be cleaned up at process termination.

> **Warning**
>
> This can lead to security holes if another process obtains the same file name and creates the file before you are able to. Open the file with `JL_O_EXCL` if this is a concern. Using `mktemp()` is also recommended instead.

`Base.Filesystem.tempdir` — Function

```
tempdir()
```

Gets the path of the temporary directory. On Windows, `tempdir()` uses the first environment variable found in the ordered list `TMP`, `TEMP`, `USERPROFILE`. On all other operating systems, `tempdir()` uses the first environment variable found in the ordered list `TMPDIR`, `TMP`, `TEMP`, and `TEMPDIR`. If none of these are found, the path `"/tmp"` is used.

`Base.Filesystem.mktemp` — Method

```
mktemp(parent=tempdir(); cleanup=true) -> (path, io)
```

Return `(path, io)`, where `path` is the path of a new temporary file in `parent` and `io` is an open file object for this path. The `cleanup` option controls whether the temporary file is automatically deleted when the process exits.

`Base.Filesystem.mktemp` — Method

```
mktemp(f::Function, parent=tempdir())
```

Apply the function `f` to the result of `mktemp(parent)` and remove the temporary file upon completion.

`Base.Filesystem.mktempdir` — Method

```
mktempdir(parent=tempdir(); prefix="jl_", cleanup=true) -> path
```

Create a temporary directory in the `parent` directory with a name constructed from the given prefix and a random suffix, and return its path. Additionally, any trailing `X` characters may be replaced with random characters. If `parent` does not exist, throw an error. The `cleanup` option controls whether the temporary directory is automatically deleted when the process exits.

`Base.Filesystem.mktempdir` — Method

```
mktempdir(f::Function, parent=tempdir(); prefix="jl_")
```

Apply the function `f` to the result of `mktempdir(parent; prefix)` and remove the temporary directory all of its contents upon completion.

`Base.Filesystem.isblockdev` — Function

```
isblockdev(path) -> Bool
```

Return `true` if `path` is a block device, `false` otherwise.

`Base.Filesystem.ischardev` — Function

```
ischardev(path) -> Bool
```

Return `true` if `path` is a character device, `false` otherwise.

Base.Filesystem.isdir — Function

```
isdir(path) -> Bool
```

Return `true` if `path` is a directory, `false` otherwise.

Examples

```
julia> isdir(homedir())
true

julia> isdir("not/a/directory")
false
```

See also: `isfile` and `ispath`.

Base.Filesystem.isfifo — Function

```
isfifo(path) -> Bool
```

Return `true` if `path` is a FIFO, `false` otherwise.

Base.Filesystem.isfile — Function

```
isfile(path) -> Bool
```

Return `true` if `path` is a regular file, `false` otherwise.

Examples

```
julia> isfile(homedir())
false

julia> f = open("test_file.txt", "w");

julia> isfile(f)
true
```

```
julia> close(f); rm("test_file.txt")
```

See also: `isdir` and `ispath`.

`Base.Filesystem.islink` — Function

```
islink(path) -> Bool
```

Return `true` if `path` is a symbolic link, `false` otherwise.

`Base.Filesystem.ismount` — Function

```
ismount(path) -> Bool
```

Return `true` if `path` is a mount point, `false` otherwise.

`Base.Filesystem.ispath` — Function

```
ispath(path) -> Bool
```

Return `true` if a valid filesystem entity exists at `path`, otherwise returns `false`. This is the generalization of `isfile`, `isdir` etc.

`Base.Filesystem.issetgid` — Function

```
issetgid(path) -> Bool
```

Return `true` if `path` has the setgid flag set, `false` otherwise.

`Base.Filesystem.issetuid` — Function

```
issetuid(path) -> Bool
```

Return `true` if `path` has the setuid flag set, `false` otherwise.

`Base.Filesystem.issocket` — Function

```
issocket(path) -> Bool
```

Return `true` if `path` is a socket, `false` otherwise.

`Base.Filesystem.issticky` — Function

```
issticky(path) -> Bool
```

Return `true` if `path` has the sticky bit set, `false` otherwise.

`Base.Filesystem.homedir` — Function

```
homedir() -> String
```

Return the current user's home directory.

> **Note**
>
> `homedir` determines the home directory via `libuv`'s `uv_os_homedir`. For details (for example on how to specify the home directory via environment variables), see the `uv_os_homedir` documentation.

`Base.Filesystem.dirname` — Function

```
dirname(path::AbstractString) -> AbstractString
```

Get the directory part of a path. Trailing characters ('/' or '\') in the path are counted as part of the path.

Examples

```
julia> dirname("/home/myuser")
"/home"

julia> dirname("/home/myuser/")
"/home/myuser"
```

See also: `basename`

`Base.Filesystem.basename` — Function

```
basename(path::AbstractString) -> AbstractString
```

Get the file name part of a path.

Examples

```
julia> basename("/home/myuser/example.jl")
"example.jl"
```

See also: `dirname`

`Base.@__FILE__` — Macro

```
@__FILE__ -> AbstractString
```

Expand to a string with the path to the file containing the macrocall, or an empty string if evaluated by `julia -e <expr>`. Return `nothing` if the macro was missing parser source information. Alternatively see `PROGRAM_FILE`.

`Base.@__DIR__` — Macro

```
@__DIR__ -> AbstractString
```

Expand to a string with the absolute path to the directory of the file containing the macrocall. Return the current working directory if run from a REPL or if evaluated by `julia -e <expr>`.

`Base.@__LINE__` — Macro

```
@__LINE__ -> Int
```

Expand to the line number of the location of the macrocall. Return `0` if the line number could not be determined.

`Base.Filesystem.isabspath` — Function

```
isabspath(path::AbstractString) -> Bool
```

Determine whether a path is absolute (begins at the root directory).

Examples

```
julia> isabspath("/home")
true

julia> isabspath("home")
false
```

`Base.Filesystem.isdirpath` — Function

```
isdirpath(path::AbstractString) -> Bool
```

Determine whether a path refers to a directory (for example, ends with a path separator).

Examples

```
julia> isdirpath("/home")
false

julia> isdirpath("/home/")
true
```

Base.Filesystem.joinpath — Function

```
joinpath(parts::AbstractString...) -> String
```

Join path components into a full path. If some argument is an absolute path or (on Windows) has a drive specification that doesn't match the drive computed for the join of the preceding paths, then prior components are dropped.

Note on Windows since there is a current directory for each drive, `joinpath("c:", "foo")` represents a path relative to the current directory on drive "c:" so this is equal to "c:foo", not "c:\foo". Furthermore, `joinpath` treats this as a non-absolute path and ignores the drive letter casing, hence `joinpath("C:\A","c:b") = "C:\A\b"`.

Examples

```
julia> joinpath("/home/myuser", "example.jl")
"/home/myuser/example.jl"
```

Base.Filesystem.abspath — Function

```
abspath(path::AbstractString) -> String
```

Convert a path to an absolute path by adding the current directory if necessary. Also normalizes the path as in `normpath`.

```
abspath(path::AbstractString, paths::AbstractString...) -> String
```

Convert a set of paths to an absolute path by joining them together and adding the current directory if necessary. Equivalent to `abspath(joinpath(path, paths...))`.

Base.Filesystem.normpath — Function

```
normpath(path::AbstractString) -> String
```

Normalize a path, removing "." and ".." entries.

Examples

```
julia> normpath("/home/myuser/../example.jl")
"/home/example.jl"
```

```
normpath(path::AbstractString, paths::AbstractString...) -> String
```

Convert a set of paths to a normalized path by joining them together and removing "." and ".." entries. Equivalent to `normpath(joinpath(path, paths...))`.

Base.Filesystem.realpath — Function

```
realpath(path::AbstractString) -> String
```

Canonicalize a path by expanding symbolic links and removing "." and ".." entries. On case-insensitive case-preserving filesystems (typically Mac and Windows), the filesystem's stored case for the path is returned.

(This function throws an exception if `path` does not exist in the filesystem.)

Base.Filesystem.relpath — Function

```
relpath(path::AbstractString, startpath::AbstractString = ".") -> AbstractStrin
```

Return a relative filepath to `path` either from the current directory or from an optional start directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of `path` or `startpath`.

Base.Filesystem.expanduser — Function

```
expanduser(path::AbstractString) -> AbstractString
```

On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

Base.Filesystem.splitdir — Function

```
splitdir(path::AbstractString) -> (AbstractString, AbstractString)
```

Split a path into a tuple of the directory name and file name.

Examples

```
julia> splitdir("/home/myuser")
("/home", "myuser")
```

Base.Filesystem.splitdrive — Function

```
splitdrive(path::AbstractString) -> (AbstractString, AbstractString)
```

On Windows, split a path into the drive letter part and the path part. On Unix systems, the first component is always the empty string.

Base.Filesystem.splitext — Function

```
splitext(path::AbstractString) -> (AbstractString, AbstractString)
```

If the last component of a path contains a dot, split the path into everything before the dot and everything including and after the dot. Otherwise, return a tuple of the argument unmodified and the empty string.

Examples

```
julia> splitext("/home/myuser/example.jl")
("/home/myuser/example", ".jl")

julia> splitext("/home/myuser/example")
("/home/myuser/example", "")
```

Base.Filesystem.splitpath — Function

```
splitpath(path::AbstractString) -> Vector{String}
```

Split a file path into all its path components. This is the opposite of `joinpath`. Returns an array of substrings, one for each directory or file in the path, including the root directory if present.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

Examples

```
julia> splitpath("/home/myuser/example.jl")
4-element Array{String,1}:
 "/"
 "home"
 "myuser"
 "example.jl"
```

« Constants I/O and Network »