exec

package standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 15 | Imported by: 132,420

Details ✓ Valid go.mod file ? ✓ Redistributable license ? ✓ Tagged version ? ✓ Stable version ?

Learn more

Repository cs.opensource.google/go/go

Links Report a Vulnerability

Documentation

Documentation

Rendered for linux/amd64

Overview

Executables in the current directory

Package exec runs external commands. It wraps os.StartProcess to make it easier to remap stdin and stdout, connect I/O with pipes, and do other adjustments.

Unlike the "system" library call from C and other languages, the os/exec package intentionally does not invoke the system shell and does not expand any glob patterns or handle other expansions, pipelines, or redirections typically done by shells. The package behaves more like C's "exec" family of functions. To expand glob patterns, either call the shell directly, taking care to escape any dangerous input, or use the path/filepath package's Glob function. To expand environment variables, use package os's ExpandEnv.

Note that the examples in this package assume a Unix system. They may not run on Windows, and they do not run in the Go Playground used by golang.org and godoc.org.

Executables in the current directory

The functions Command and LookPath look for a program in the directories listed in the current path, following the conventions of the host operating system. Operating systems have for decades included the current directory in this search, sometimes implicitly and sometimes configured explicitly that way by default. Modern practice is that including the current directory is usually unexpected and often leads to security problems.

To avoid those security problems, as of Go 1.19, this package will not resolve a program using an implicit or explicit path entry relative to the current directory. That is, if you run exec.LookPath("go"), it will not successfully return ./go on Unix nor .\go.exe on Windows, no matter how the path is configured. Instead, if the usual path algorithms would result in that answer, these functions return an error err satisfying errors.Is(err, ErrDot).

For example, consider these two program snippets:

```
path, err := exec.LookPath("prog")
if err != nil {
    log.Fatal(err)
}
use(path)
```

and

```
cmd := exec.Command("prog")
if err := cmd.Run(); err != nil {
    log.Fatal(err)
}
```

These will not find and run ./prog or .\prog.exe, no matter how the current path is configured.

Code that always wants to run a program from the current directory can be rewritten to say "./prog" instead of "prog".

Code that insists on including results from relative path entries can instead override the error using an errors.Is check:

```
path, err := exec.LookPath("prog")
if errors.Is(err, exec.ErrDot) {
    err = nil
}
if err != nil {
    log.Fatal(err)
}
use(path)
```

and

```
cmd := exec.Command("prog")
if errors.Is(cmd.Err, exec.ErrDot) {
    cmd.Err = nil
}
if err := cmd.Run(); err != nil {
    log.Fatal(err)
}
```

Setting the environment variable GODEBUG=execerrdot=0 disables generation of ErrDot entirely, temporarily restoring the pre-Go 1.19 behavior for programs that are unable to apply more targeted fixes. A future version of Go may remove support for this variable.

Before adding such overrides, make sure you understand the security implications of doing so. See https://go.dev/blog/path-security for more information.

Index

Examples

Constants

This section is empty.

Variables

View Source

```
var ErrDot = errors.New("cannot run executable found relative to current directory")
```

ErrDot indicates that a path lookup resolved to an executable in the current directory due to '.' being in the path, either implicitly or explicitly. See the package documentation for details.

Note that functions in this package do not return ErrDot directly. Code should use errors.Is(err, ErrDot), not err == ErrDot, to test whether a returned error err is due to this condition.

View Source

```
var ErrNotFound = errors.New("executable file not found in $PATH")
```

ErrNotFound is the error resulting if a path search failed to find an executable file.

View Source

```
var ErrWaitDelay = errors.New("exec: WaitDelay expired before I/O complete")
```

ErrWaitDelay is returned by (*Cmd).Wait if the process exits with a successful status code but its output pipes are not closed before the command's WaitDelay expires.

Functions

func LookPath

```
func LookPath(file string) (string, error)
```

LookPath searches for an executable named file in the directories named by the PATH environment variable. If file contains a slash, it is tried directly and the PATH is not consulted. Otherwise, on success, the result is an absolute path.

In older versions of Go, LookPath could return a path relative to the current directory. As of Go 1.19, LookPath will instead return that path along with an error satisfying errors.Is(err, ErrDot). See the package documentation for more details.

▶ Example

Types

type Cmd

```
type Cmd struct {
	// Path is the path of the command to run.
	//
	// This is the only field that must be set to a non-zero
	// value. If Path is relative, it is evaluated relative
	// to Dir.
	Path string
```

```
    // Args holds command line arguments, including the command as Args[0].
    // If the Args field is empty or nil, Run uses {Path}.
    //
    // In typical use, both Path and Args are set by calling Command.
    Args []string

    // Env specifies the environment of the process.
    // Each entry is of the form "key=value".
    // If Env is nil, the new process uses the current process's
    // environment.
    // If Env contains duplicate environment keys, only the last
    // value in the slice for each duplicate key is used.
    // As a special case on Windows, SYSTEMROOT is always added if
    // missing and not explicitly set to the empty string.
    Env []string

    // Dir specifies the working directory of the command.
    // If Dir is the empty string, Run runs the command in the
    // calling process's current directory.
    Dir string

    // Stdin specifies the process's standard input.
    //
    // If Stdin is nil, the process reads from the null device (os.DevNull).
    //
    // If Stdin is an *os.File, the process's standard input is connected
    // directly to that file.
    //
    // Otherwise, during the execution of the command a separate
    // goroutine reads from Stdin and delivers that data to the command
    // over a pipe. In this case, Wait does not complete until the goroutine
    // stops copying, either because it has reached the end of Stdin
    // (EOF or a read error), or because writing to the pipe returned an error,
    // or because a nonzero WaitDelay was set and expired.
    Stdin io.Reader

    // Stdout and Stderr specify the process's standard output and error.
    //
    // If either is nil, Run connects the corresponding file descriptor
    // to the null device (os.DevNull).
    //
    // If either is an *os.File, the corresponding output from the process
    // is connected directly to that file.
    //
    // Otherwise, during the execution of the command a separate goroutine
    // reads from the process over a pipe and delivers that data to the
    // corresponding Writer. In this case, Wait does not complete until the
    // goroutine reaches EOF or encounters an error or a nonzero WaitDelay
    // expires.
    //
    // If Stdout and Stderr are the same writer, and have a type that can
    // be compared with ==, at most one goroutine at a time will call Write.
    Stdout io.Writer
```

```
	Stderr io.Writer

	// ExtraFiles specifies additional open files to be inherited by the
	// new process. It does not include standard input, standard output, or
	// standard error. If non-nil, entry i becomes file descriptor 3+i.
	//
	// ExtraFiles is not supported on Windows.
	ExtraFiles []*os.File

	// SysProcAttr holds optional, operating system-specific attributes.
	// Run passes it to os.StartProcess as the os.ProcAttr's Sys field.
	SysProcAttr *syscall.SysProcAttr

	// Process is the underlying process, once started.
	Process *os.Process

	// ProcessState contains information about an exited process.
	// If the process was started successfully, Wait or Run will
	// populate its ProcessState when the command completes.
	ProcessState *os.ProcessState

	Err error // LookPath error, if any.

	// If Cancel is non-nil, the command must have been created with
	// CommandContext and Cancel will be called when the command's
	// Context is done. By default, CommandContext sets Cancel to
	// call the Kill method on the command's Process.
	//
	// Typically a custom Cancel will send a signal to the command's
	// Process, but it may instead take other actions to initiate cancellation,
	// such as closing a stdin or stdout pipe or sending a shutdown request on a
	// network socket.
	//
	// If the command exits with a success status after Cancel is
	// called, and Cancel does not return an error equivalent to
	// os.ErrProcessDone, then Wait and similar methods will return a non-nil
	// error: either an error wrapping the one returned by Cancel,
	// or the error from the Context.
	// (If the command exits with a non-success status, or Cancel
	// returns an error that wraps os.ErrProcessDone, Wait and similar methods
	// continue to return the command's usual exit status.)
	//
	// If Cancel is set to nil, nothing will happen immediately when the command's
	// Context is done, but a nonzero WaitDelay will still take effect. That may
	// be useful, for example, to work around deadlocks in commands that do not
	// support shutdown signals but are expected to always finish quickly.
	//
	// Cancel will not be called if Start returns a non-nil error.
	Cancel func() error

	// If WaitDelay is non-zero, it bounds the time spent waiting on two sources
	// of unexpected delay in Wait: a child process that fails to exit after the
	// associated Context is canceled, and a child process that exits but leaves
	// its I/O pipes unclosed.
	//
	// The WaitDelay timer starts when either the associated Context is done or a
	// call to Wait observes that the child process has exited, whichever occurs
	// first. When the delay has elapsed, the command shuts down the child process
	// and/or its I/O pipes.
	//
	// If the child process has failed to exit — perhaps because it ignored or
	// failed to receive a shutdown signal from a Cancel function, or because no
	// Cancel function was set — then it will be terminated using os.Process.Kill.
	//
	// Then, if the I/O pipes communicating with the child process are still open,
	// those pipes are closed in order to unblock any goroutines currently blocked
	// on Read or Write calls.
	//
	// If pipes are closed due to WaitDelay, no Cancel call has occurred,
	// and the command has otherwise exited with a successful status, Wait and
	// similar methods will return ErrWaitDelay instead of nil.
	//
	// If WaitDelay is zero (the default), I/O pipes will be read until EOF,
	// which might not occur until orphaned subprocesses of the command have
	// also closed their descriptors for the pipes.
	WaitDelay time.Duration
	// contains filtered or unexported fields
}
```

Cmd represents an external command being prepared or run.

A Cmd cannot be reused after calling its Run, Output or CombinedOutput methods.

func Command

```
func Command(name string, arg ...string) *Cmd
```

Command returns the Cmd struct to execute the named program with the given arguments.

It sets only the Path and Args in the returned structure.

If name contains no path separators, Command uses LookPath to resolve name to a complete path if possible. Otherwise it uses name directly as Path.

The returned Cmd's Args field is constructed from the command name followed by the elements of arg, so arg should not include the command name itself. For example, Command("echo", "hello"). Args[0] is always name, not the possibly resolved Path.

On Windows, processes receive the whole command line as a single string and do their own parsing. Command combines and quotes Args into a command line string with an algorithm compatible with applications using CommandLineToArgvW (which is the most common way). Notable exceptions are msiexec.exe and cmd.exe (and thus, all batch files), which have a different unquoting algorithm. In these or other similar cases, you can do the quoting yourself and provide the full command line in SysProcAttr.CmdLine, leaving Args empty.

▶ Example

▶ Example (Environment)

func CommandContext

added in go1.7

```
func CommandContext(ctx context.Context, name string, arg ...string) *Cmd
```

CommandContext is like Command but includes a context.

The provided context is used to interrupt the process (by calling cmd.Cancel or os.Process.Kill) if the context becomes done before the command completes on its own.

CommandContext sets the command's Cancel function to invoke the Kill method on its Process, and leaves its WaitDelay unset. The caller may change the cancellation behavior by modifying those fields before starting the command.

▶ Example

func (*Cmd) CombinedOutput

```
func (c *Cmd) CombinedOutput() ([]byte, error)
```

CombinedOutput runs the command and returns its combined standard output and standard error.

▶ Example

func (*Cmd) Environ

added in go1.19

```
func (c *Cmd) Environ() []string
```

Environ returns a copy of the environment in which the command would be run as it is currently configured.

▶ Example

func (*Cmd) Output

```
func (c *Cmd) Output() ([]byte, error)
```

Output runs the command and returns its standard output. Any returned error will usually be of type *ExitError. If c.Stderr was nil, Output populates ExitError.Stderr.

▶ Example

func (*Cmd) Run

```
func (c *Cmd) Run() error
```

Run starts the specified command and waits for it to complete.

The returned error is nil if the command runs, has no problems copying stdin, stdout, and stderr, and exits with a zero exit status.

If the command starts but does not complete successfully, the error is of type *ExitError. Other error types may be returned for other situations.

If the calling goroutine has locked the operating system thread with runtime.LockOSThread and modified any inheritable OS-level thread state (for example, Linux or Plan 9 name spaces), the new process will inherit the caller's thread state.

▶ Example

func (*Cmd) Start

```
func (c *Cmd) Start() error
```

Start starts the specified command but does not wait for it to complete.

If Start returns successfully, the c.Process field will be set.

After a successful call to Start the Wait method must be called in order to release associated system resources.

▶ Example

func (*Cmd) StderrPipe

```
func (c *Cmd) StderrPipe() (io.ReadCloser, error)
```

StderrPipe returns a pipe that will be connected to the command's standard error when the command starts.

Wait will close the pipe after seeing the command exit, so most callers need not close the pipe themselves. It is thus incorrect to call Wait before all reads from the pipe have completed. For the same reason, it is incorrect to use Run when using StderrPipe. See the StdoutPipe example for idiomatic usage.

▶ Example

func (*Cmd) StdinPipe

```
func (c *Cmd) StdinPipe() (io.WriteCloser, error)
```

StdinPipe returns a pipe that will be connected to the command's standard input when the command starts. The pipe will be closed automatically after Wait sees the command exit. A caller need only call Close to force the pipe to close sooner. For example, if the command being run will not exit until standard input is closed, the caller must close the pipe.

▶ Example

func (*Cmd) StdoutPipe

```
func (c *Cmd) StdoutPipe() (io.ReadCloser, error)
```

StdoutPipe returns a pipe that will be connected to the command's standard output when the command starts.

Wait will close the pipe after seeing the command exit, so most callers need not close the pipe themselves. It is thus incorrect to call Wait before all reads from the pipe have completed. For the same reason, it is incorrect to call Run when using StdoutPipe. See the example for idiomatic usage.

▶ Example

func (*Cmd) String

added in go1.13

```
func (c *Cmd) String() string
```

String returns a human-readable description of c. It is intended only for debugging. In particular, it is not suitable for use as input to a shell. The output of String may vary across Go releases.

func (*Cmd) Wait

```
func (c *Cmd) Wait() error
```

Wait waits for the command to exit and waits for any copying to stdin or copying from stdout or stderr to complete.

The command must have been started by Start.

The returned error is nil if the command runs, has no problems copying stdin, stdout, and stderr, and exits with a zero exit status.

If the command fails to run or doesn't complete successfully, the error is of type *ExitError. Other error types may be returned for I/O problems.

If any of c.Stdin, c.Stdout or c.Stderr are not an *os.File, Wait also waits for the respective I/O loop copying to or from the process to complete.

Wait releases any resources associated with the Cmd.

type Error

```
type Error struct {
    // Name is the file name for which the error occurred.
    Name string
    // Err is the underlying error.
    Err error
}
```

Error is returned by LookPath when it fails to classify a file as an executable.

func (*Error) Error

```
func (e *Error) Error() string
```

func (*Error) Unwrap

added in go1.13

```
func (e *Error) Unwrap() error
```

type ExitError

```
type ExitError struct {
    *os.ProcessState

    // Stderr holds a subset of the standard error output from the
    // Cmd.Output method if standard error was not otherwise being
    // collected.
    //
    // If the error output is long, Stderr may contain only a prefix
    // and suffix of the output, with the middle replaced with
    // text about the number of omitted bytes.
    //
    // Stderr is provided for debugging, for inclusion in error messages.
    // Users with other needs should redirect Cmd.Stderr as needed.
    Stderr []byte
}
```

An ExitError reports an unsuccessful exit by a command.

func (*ExitError) Error

```
func (e *ExitError) Error() string
```

Source Files

View all

exec.go   exec_unix.go   lp_unix.go

Directories

Expand all

- internal

Google