

[Discover Packages](#) > [Standard library](#) > [runtime](#) 


# runtime

package

standard library

Version: [go1.20.1](#) **Latest** | Published: Feb 14, 2023 | License: [BSD-3-Clause](#) | Imports: [12](#) |Imported by: [218,778](#)

## 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

### [Environment Variables](#)

Package runtime contains operations that interact with Go's runtime system, such as functions to control goroutines. It also includes the low-level type information used by the reflect package; see reflect's documentation for the programmable interface to the run-time type system.

## Environment Variables

The following environment variables (`$name` or `%name%`, depending on the host operating system) control the run-time behavior of Go programs. The meanings and use may change from release to release.

The `GOGC` variable sets the initial garbage collection target percentage. A collection is triggered when the ratio of freshly allocated data to live data remaining after the previous collection reaches this percentage. The default is `GOGC=100`. Setting `GOGC=off` disables the garbage collector entirely. [runtime/debug.SetGCPercent](#) allows changing this percentage at run time.

The `GOMEMLIMIT` variable sets a soft memory limit for the runtime. This memory limit includes the Go heap and all other memory managed by the runtime, and excludes external memory sources such as mappings of the binary itself, memory managed in other languages, and memory held by the operating system on behalf of the Go program. `GOMEMLIMIT` is a numeric value in bytes with an optional unit suffix. The supported suffixes include B, KiB, MiB, GiB, and TiB. These suffixes represent quantities of bytes as defined by the IEC 80000-13 standard. That is, they are based on powers of two: KiB means

2<sup>10</sup> bytes, MiB means 2<sup>20</sup> bytes, and so on. The default setting is `math.MaxInt64`, which effectively disables the memory limit. [runtime/debug.SetMemoryLimit](#) allows changing this limit at run time.

The `GODEBUG` variable controls debugging variables within the runtime. It is a comma-separated list of `name=val` pairs setting these named variables:

`allocfreetrace`: setting `allocfreetrace=1` causes every allocation to be profiled and a stack trace printed on each object's allocation and free.

`clobberfree`: setting `clobberfree=1` causes the garbage collector to clobber the memory content of an object with bad content when it frees the object.

`cpu.*`: `cpu.all=off` disables the use of all optional instruction set extensions. `cpu.extension=off` disables use of instructions from the specified instruction set extension. `extension` is the lower case name for the instruction set extension such as `sse41` or `avx` as listed in `internal/cpu` package. As an example `cpu.avx=off` disables runtime detection and thereby use of AVX instructions.

`cgocheck`: setting `cgocheck=0` disables all checks for packages using `cgo` to incorrectly pass Go pointers to non-Go code. Setting `cgocheck=1` (the default) enables relatively cheap checks that may miss some errors. Setting `cgocheck=2` enables expensive checks that should not miss any errors, but will cause your program to run slower.

`efence`: setting `efence=1` causes the allocator to run in a mode where each object is allocated on a unique page and addresses are never recycled.

`gccheckmark`: setting `gccheckmark=1` enables verification of the garbage collector's concurrent mark phase by performing a second mark pass while the world is stopped. If the second pass finds a reachable object that was not found by concurrent mark, the garbage collector will panic.

`gcpacertrace`: setting `gcpacertrace=1` causes the garbage collector to print information about the internal state of the concurrent pacer.

`gcshrinkstackoff`: setting `gcshrinkstackoff=1` disables moving goroutines onto smaller stacks. In this mode, a goroutine's stack can only grow.

`gcstoptheworld`: setting `gcstoptheworld=1` disables concurrent garbage collection, making every garbage collection a stop-the-world event. Setting `gcstoptheworld=2` also disables concurrent sweeping after the garbage collection finishes.

`gctrace`: setting `gctrace=1` causes the garbage collector to emit a single line to standard error at each collection, summarizing the amount of memory collected and the length of the pause. The format of this line is subject to change. Currently, it is:

`gc # @#s #%: ##+## ms clock, ##+##/##+## ms cpu, #->#-># MB, # MB goal, # MB stacks, # MB`, where the fields are as follows:

```

gc #           the GC number, incremented at each GC
@#s           time in seconds since program start
#%           percentage of time spent in GC since program start
#+...+#       wall-clock/CPU times for the phases of the GC
#->#-># MB     heap size at GC start, at GC end, and live heap
# MB goal     goal heap size
# MB stacks   estimated scannable stack size
# MB globals  scannable global size
# P           number of processors used

```

The phases are stop-the-world (STW) sweep termination, concurrent mark and scan, and STW mark termination. The CPU times for mark/scan are broken down in to assist time (GC performed in line with allocation), background GC time, and idle GC time. If the line ends with "(forced)", this GC was forced by a `runtime.GC()` call.

`harddecommit`: setting `harddecommit=1` causes memory that is returned to the OS to also have protections removed on it. This is the only mode of operation on Windows, but is helpful in debugging scavenger-related issues on other platforms. Currently, only supported on Linux.

`inittrace`: setting `inittrace=1` causes the runtime to emit a single line to standard error for each package with init work, summarizing the execution time and memory allocation. No information is printed for inits executed as part of plugin loading and for packages without both user defined and compiler generated init work. The format of this line is subject to change. Currently, it is:

```
init # @#ms, # ms clock, # bytes, # allocs
```

where the fields are as follows:

```

init #       the package name
@# ms       time in milliseconds when the init started since program start
# clock     wall-clock time for package initialization work
# bytes     memory allocated on the heap
# allocs    number of heap allocations

```

`madvdontneed`: setting `madvdontneed=0` will use `MADV_FREE` instead of `MADV_DONTNEED` on Linux when returning memory to the kernel. This is more efficient, but means RSS numbers will drop only when the OS is under memory pressure. On the BSDs and Illumos/Solaris, setting `madvdontneed=1` will use `MADV_DONTNEED` instead of `MADV_FREE`. This is less efficient, but causes RSS numbers to drop more quickly.

`memprofilerate`: setting `memprofilerate=X` will update the value of `runtime.MemProfileRate`. When set to 0 memory profiling is disabled. Refer to the description of `MemProfileRate` for the default value.

`pagetrace`: setting `pagetrace=/path/to/file` will write out a trace of page events that can be viewed, analyzed, and visualized using the `x/debug/cmd/pagetrace` tool. Build your program with `GOEXPERIMENT=pagetrace` to enable this functionality. Do not enable this functionality if your program is a `setuid` binary as it introduces a security risk in that scenario. Currently not supported on Windows, plan9 or js/wasm. Setting the option for some applications can produce large traces, so use with care.

`invalidptr`: `invalidptr=1` (the default) causes the garbage collector and stack copier to crash the program if an invalid pointer value (for example, 1) is found in a pointer-typed location. Setting `invalidptr=0` disables this check. This should only be used as a temporary workaround to diagnose buggy code. The real fix is to not store integers in pointer-typed locations.

`sbrk`: setting `sbrk=1` replaces the memory allocator and garbage collector with a trivial allocator that obtains memory from the operating system and never reclaims any memory.

`scavtrace`: setting `scavtrace=1` causes the runtime to emit a single line to standard error, roughly once per GC cycle, summarizing the amount of work done by the scavenger as well as the total amount of memory returned to the operating system and an estimate of physical memory utilization. The format of this line is subject to change, but currently it is:

```
scav # KiB work, # KiB total, #% util
```

where the fields are as follows:

```
# KiB work    the amount of memory returned to the OS since the last line
```

```
# KiB total   the total amount of memory returned to the OS
```

```
##% util      the fraction of all unscavenged memory which is in-use
```

If the line ends with "(forced)", then scavenging was forced by a `debug.FreeOSMemory()` call.

`scheddetail`: setting `schedtrace=X` and `scheddetail=1` causes the scheduler to emit detailed multiline info every X milliseconds, describing state of the scheduler, processors, threads and goroutines.

`schedtrace`: setting `schedtrace=X` causes the scheduler to emit a single line to standard error every X milliseconds, summarizing the scheduler state.

`tracebackancestors`: setting `tracebackancestors=N` extends tracebacks with the stacks at which goroutines were created, where N limits the number of ancestor goroutines to report. This also extends the information returned by `runtime.Stack`. Ancestor's goroutine IDs will refer to the ID of the goroutine at the time of creation; it's possible for the ID to be reused for another goroutine. Setting N to 0 will report no ancestry information.

`asyncpreemptoff`: `asyncpreemptoff=1` disables signal-based asynchronous goroutine preemption. This makes some loops non-preemptible for long periods, which may delay GC and goroutine scheduling. This is useful for debugging GC issues because it also disables the conservative stack scanning used for asynchronously preempted goroutines.

The `net` and `net/http` packages also refer to debugging variables in `GODEBUG`. See the documentation for those packages for details.

The `GOMAXPROCS` variable limits the number of operating system threads that can execute user-level Go code simultaneously. There is no limit to the number of threads that can be blocked in system calls on behalf of Go code; those do not count against the `GOMAXPROCS` limit. This package's `GOMAXPROCS` function queries and changes the limit.

The GORACE variable configures the race detector, for programs built using -race. See [https://golang.org/doc/articles/race\\_detector.html](https://golang.org/doc/articles/race_detector.html) for details.

The GOTRACEBACK variable controls the amount of output generated when a Go program fails due to an unrecovered panic or an unexpected runtime condition. By default, a failure prints a stack trace for the current goroutine, eliding functions internal to the run-time system, and then exits with exit code 2. The failure prints stack traces for all goroutines if there is no current goroutine or the failure is internal to the run-time. GOTRACEBACK=none omits the goroutine stack traces entirely. GOTRACEBACK=single (the default) behaves as described above. GOTRACEBACK=all adds stack traces for all user-created goroutines. GOTRACEBACK=system is like “all” but adds stack frames for run-time functions and shows goroutines created internally by the run-time. GOTRACEBACK=crash is like “system” but crashes in an operating system-specific manner instead of exiting. For example, on Unix systems, the crash raises SIGABRT to trigger a core dump. For historical reasons, the GOTRACEBACK settings 0, 1, and 2 are synonyms for none, all, and system, respectively. The runtime/debug package's SetTraceback function allows increasing the amount of output at run time, but it cannot reduce the amount below that specified by the environment variable. See <https://golang.org/pkg/runtime/debug/#SetTraceback>.

The GOARCH, GOOS, GOPATH, and GOROOT environment variables complete the set of Go environment variables. They influence the building of Go programs (see <https://golang.org/cmd/go> and <https://golang.org/pkg/go/build>). GOARCH, GOOS, and GOROOT are recorded at compile time and made available by constants or functions in this package, but they do not influence the execution of the run-time system.

## Index

### Constants

### Variables

func BlockProfile(p []BlockProfileRecord) (n int, ok bool)

func Breakpoint()

func CPUProfile() []byte DEPRECATED

func Caller(skip int) (pc uintptr, file string, line int, ok bool)

func Callers(skip int, pc []uintptr) int

func GC()

func GOMAXPROCS(n int) int

func GOROOT() string

func Goexit()

func GoroutineProfile(p []StackRecord) (n int, ok bool)

func Gosched()

func KeepAlive(x any)

func LockOSThread()

func MemProfile(p []MemProfileRecord, inuseZero bool) (n int, ok bool)

func MutexProfile(p []BlockProfileRecord) (n int, ok bool)

func NumCPU() int

func NumCgoCall() int64

func NumGoroutine() int

func ReadMemStats(m \*MemStats)

func ReadTrace() []byte

func SetBlockProfileRate(rate int)

```

func SetCPUProfileRate(hz int)
func SetCgoTraceback(version int, traceback, context, symbolizer unsafe.Pointer)
func SetFinalizer(obj any, finalizer any)
func SetMutexProfileFraction(rate int) int
func Stack(buf []byte, all bool) int
func StartTrace() error
func StopTrace()
func ThreadCreateProfile(p []StackRecord) (n int, ok bool)
func UnlockOSThread()
func Version() string
type BlockProfileRecord
type Error
type Frame
type Frames
    func CallersFrames(callers []uintptr) *Frames
    func (ci *Frames) Next() (frame Frame, more bool)
type Func
    func FuncForPC(pc uintptr) *Func
    func (f *Func) Entry() uintptr
    func (f *Func) FileLine(pc uintptr) (file string, line int)
    func (f *Func) Name() string
type MemProfileRecord
    func (r *MemProfileRecord) InUseBytes() int64
    func (r *MemProfileRecord) InUseObjects() int64
    func (r *MemProfileRecord) Stack() []uintptr
type MemStats
type StackRecord
    func (r *StackRecord) Stack() []uintptr
type TypeAssertionError
    func (e *TypeAssertionError) Error() string
    func (*TypeAssertionError) RuntimeError()

```

## Examples

Frames

## Constants

<a href="#">View Source</a>	
const	Compiler = "gc"

Compiler is the name of the compiler toolchain that built the running binary. Known toolchains are:

gc	Also known as cmd/compile.
gccgo	The gccgo front end, part of the GCC compiler suite.
<a href="#">View Source</a>	

```
const GOARCH string = goarch.GOARCH
```

GOARCH is the running program's architecture target: one of 386, amd64, arm, s390x, and so on.

[View Source](#)

```
const GOOS string = goos.GOOS
```

GOOS is the running program's operating system target: one of darwin, freebsd, linux, and so on. To view possible combinations of GOOS and GOARCH, run "go tool dist list".

## Variables

[View Source](#)

```
var MemProfileRate int = 512 * 1024
```

MemProfileRate controls the fraction of memory allocations that are recorded and reported in the memory profile. The profiler aims to sample an average of one allocation per MemProfileRate bytes allocated.

To include every allocated block in the profile, set MemProfileRate to 1. To turn off profiling entirely, set MemProfileRate to 0.

The tools that process the memory profiles assume that the profile rate is constant across the lifetime of the program and equal to the current value. Programs that change the memory profiling rate should do so just once, as early as possible in the execution of the program (for example, at the beginning of main).

## Functions

### func BlockProfile

added in go1.1

```
func BlockProfile(p []BlockProfileRecord) (n int, ok bool)
```

BlockProfile returns n, the number of records in the current blocking profile. If len(p) >= n, BlockProfile copies the profile into p and returns n, true. If len(p) < n, BlockProfile does not change p and returns n, false.

Most clients should use the runtime/pprof package or the testing package's -test.blockprofile flag instead of calling BlockProfile directly.

### func Breakpoint

```
func Breakpoint()
```

Breakpoint executes a breakpoint trap.

### func CPUProfile DEPRECATED [Show](#)



## func Caller

```
func Caller(skip int) (pc uintptr, file string, line int, ok bool)
```

Caller reports file and line number information about function invocations on the calling goroutine's stack. The argument skip is the number of stack frames to ascend, with 0 identifying the caller of Caller. (For historical reasons the meaning of skip differs between Caller and Callers.) The return values report the program counter, file name, and line number within the file of the corresponding call. The boolean ok is false if it was not possible to recover the information.

## func Callers

```
func Callers(skip int, pc []uintptr) int
```

Callers fills the slice pc with the return program counters of function invocations on the calling goroutine's stack. The argument skip is the number of stack frames to skip before recording in pc, with 0 identifying the frame for Callers itself and 1 identifying the caller of Callers. It returns the number of entries written to pc.

To translate these PCs into symbolic information such as function names and line numbers, use CallersFrames. CallersFrames accounts for inlined functions and adjusts the return program counters into call program counters. Iterating over the returned slice of PCs directly is discouraged, as is using FuncForPC on any of the returned PCs, since these cannot account for inlining or return program counter adjustment.

## func GC

```
func GC()
```

GC runs a garbage collection and blocks the caller until the garbage collection is complete. It may also block the entire program.

## func GOMAXPROCS

```
func GOMAXPROCS(n int) int
```

GOMAXPROCS sets the maximum number of CPUs that can be executing simultaneously and returns the previous setting. It defaults to the value of runtime.NumCPU. If  $n < 1$ , it does not change the current setting. This call will go away when the scheduler improves.

## func GOROOT

```
func GOROOT() string
```

GOROOT returns the root of the Go tree. It uses the GOROOT environment variable, if set at process start, or else the root used during the Go build.



## func Goexit

```
func Goexit()
```

Goexit terminates the goroutine that calls it. No other goroutine is affected. Goexit runs all deferred calls before terminating the goroutine. Because Goexit is not a panic, any recover calls in those deferred functions will return nil.

Calling Goexit from the main goroutine terminates that goroutine without func main returning. Since func main has not returned, the program continues execution of other goroutines. If all other goroutines exit, the program crashes.

## func GoroutineProfile

```
func GoroutineProfile(p []StackRecord) (n int, ok bool)
```

GoroutineProfile returns n, the number of records in the active goroutine stack profile. If len(p) >= n, GoroutineProfile copies the profile into p and returns n, true. If len(p) < n, GoroutineProfile does not change p and returns n, false.

Most clients should use the runtime/pprof package instead of calling GoroutineProfile directly.

## func Gosched

```
func Gosched()
```

Gosched yields the processor, allowing other goroutines to run. It does not suspend the current goroutine, so execution resumes automatically.

## func KeepAlive

added in go1.7

```
func KeepAlive(x any)
```

KeepAlive marks its argument as currently reachable. This ensures that the object is not freed, and its finalizer is not run, before the point in the program where KeepAlive is called.

A very simplified example showing where KeepAlive is required:

```
type File struct { d int }
d, err := syscall.Open("/file/path", syscall.O_RDONLY, 0)
// ... do something if err != nil ...
p := &File{d}
runtime.SetFinalizer(p, func(p *File) { syscall.Close(p.d) })
var buf [10]byte
n, err := syscall.Read(p.d, buf[:])
// Ensure p is not finalized until Read returns.
runtime.KeepAlive(p)
// No more uses of p after this point.
```

Without the `KeepAlive` call, the finalizer could run at the start of `syscall.Read`, closing the file descriptor before `syscall.Read` makes the actual system call.

Note: `KeepAlive` should only be used to prevent finalizers from running prematurely. In particular, when used with `unsafe.Pointer`, the rules for valid uses of `unsafe.Pointer` still apply.

## func `LockOSThread`

```
func LockOSThread()
```

`LockOSThread` wires the calling goroutine to its current operating system thread. The calling goroutine will always execute in that thread, and no other goroutine will execute in it, until the calling goroutine has made as many calls to `UnlockOSThread` as to `LockOSThread`. If the calling goroutine exits without unlocking the thread, the thread will be terminated.

All init functions are run on the startup thread. Calling `LockOSThread` from an init function will cause the main function to be invoked on that thread.

A goroutine should call `LockOSThread` before calling OS services or non-Go library functions that depend on per-thread state.

## func `MemProfile`

```
func MemProfile(p []MemProfileRecord, inuseZero bool) (n int, ok bool)
```

`MemProfile` returns a profile of memory allocated and freed per allocation site.

`MemProfile` returns `n`, the number of records in the current memory profile. If `len(p) >= n`, `MemProfile` copies the profile into `p` and returns `n, true`. If `len(p) < n`, `MemProfile` does not change `p` and returns `n, false`.

If `inuseZero` is true, the profile includes allocation records where `r.AllocBytes > 0` but `r.AllocBytes == r.FreeBytes`. These are sites where memory was allocated, but it has all been released back to the runtime.

The returned profile may be up to two garbage collection cycles old. This is to avoid skewing the profile toward allocations; because allocations happen in real time but frees are delayed until the garbage collector performs sweeping, the profile only accounts for allocations that have had a chance to be freed by the garbage collector.

Most clients should use the `runtime/pprof` package or the `testing` package's `-test.memprofile` flag instead of calling `MemProfile` directly.

## func `MutexProfile`

added in go1.8

```
func MutexProfile(p []BlockProfileRecord) (n int, ok bool)
```

`MutexProfile` returns `n`, the number of records in the current mutex profile. If `len(p) >= n`, `MutexProfile` copies the profile into `p` and returns `n, true`. Otherwise, `MutexProfile` does not change `p`, and returns `n,`

false.

Most clients should use the runtime/pprof package instead of calling MutexProfile directly.

## func NumCPU

```
func NumCPU() int
```

NumCPU returns the number of logical CPUs usable by the current process.

The set of available CPUs is checked by querying the operating system at process startup. Changes to operating system CPU allocation after process startup are not reflected.

## func NumCgoCall

```
func NumCgoCall() int64
```

NumCgoCall returns the number of cgo calls made by the current process.

## func NumGoroutine

```
func NumGoroutine() int
```

NumGoroutine returns the number of goroutines that currently exist.

## func ReadMemStats

```
func ReadMemStats(m *MemStats)
```

ReadMemStats populates m with memory allocator statistics.

The returned memory allocator statistics are up to date as of the call to ReadMemStats. This is in contrast with a heap profile, which is a snapshot as of the most recently completed garbage collection cycle.

## func ReadTrace

added in go1.5

```
func ReadTrace() []byte
```

ReadTrace returns the next chunk of binary tracing data, blocking until data is available. If tracing is turned off and all the data accumulated while it was on has been returned, ReadTrace returns nil. The caller must copy the returned data before calling ReadTrace again. ReadTrace must be called from one goroutine at a time.

## func SetBlockProfileRate

added in go1.1

```
func SetBlockProfileRate(rate int)
```

SetBlockProfileRate controls the fraction of goroutine blocking events that are reported in the blocking profile. The profiler aims to sample an average of one blocking event per rate nanoseconds spent blocked.

To include every blocking event in the profile, pass rate = 1. To turn off profiling entirely, pass rate <= 0.

## func SetCPUProfileRate

```
func SetCPUProfileRate(hz int)
```

SetCPUProfileRate sets the CPU profiling rate to hz samples per second. If hz <= 0, SetCPUProfileRate turns off profiling. If the profiler is on, the rate cannot be changed without first turning it off.

Most clients should use the runtime/pprof package or the testing package's -test.cpuprofile flag instead of calling SetCPUProfileRate directly.

## func SetCgoTraceback

added in go1.7

```
func SetCgoTraceback(version int, traceback, context, symbolizer unsafe.Pointer)
```

SetCgoTraceback records three C functions to use to gather traceback information from C code and to convert that traceback information into symbolic information. These are used when printing stack traces for a program that uses cgo.

The traceback and context functions may be called from a signal handler, and must therefore use only async-signal safe functions. The symbolizer function may be called while the program is crashing, and so must be cautious about using memory. None of the functions may call back into Go.

The context function will be called with a single argument, a pointer to a struct:

```
struct {
    Context uintptr
}
```

In C syntax, this struct will be

```
struct {
    uintptr_t Context;
};
```

If the Context field is 0, the context function is being called to record the current traceback context. It should record in the Context field whatever information is needed about the current point of execution to later produce a stack trace, probably the stack pointer and PC. In this case the context function will be called from C code.

If the Context field is not 0, then it is a value returned by a previous call to the context function. This case is called when the context is no longer needed; that is, when the Go code is returning to its C code caller. This permits the context function to release any associated resources.

While it would be correct for the context function to record a complete a stack trace whenever it is called, and simply copy that out in the traceback function, in a typical program the context function will be called many times without ever recording a traceback for that context. Recording a complete stack trace in a call to the context function is likely to be inefficient.

The traceback function will be called with a single argument, a pointer to a struct:

```
struct {
    Context      uintptr
    SigContext   uintptr
    Buf          *uintptr
    Max          uintptr
}
```

In C syntax, this struct will be

```
struct {
    uintptr_t    Context;
    uintptr_t    SigContext;
    uintptr_t*   Buf;
    uintptr_t    Max;
};
```

The Context field will be zero to gather a traceback from the current program execution point. In this case, the traceback function will be called from C code.

Otherwise Context will be a value previously returned by a call to the context function. The traceback function should gather a stack trace from that saved point in the program execution. The traceback function may be called from an execution thread other than the one that recorded the context, but only when the context is known to be valid and unchanging. The traceback function may also be called deeper in the call stack on the same thread that recorded the context. The traceback function may be called multiple times with the same Context value; it will usually be appropriate to cache the result, if possible, the first time this is called for a specific context value.

If the traceback function is called from a signal handler on a Unix system, SigContext will be the signal context argument passed to the signal handler (a C `ucontext_t*` cast to `uintptr_t`). This may be used to start tracing at the point where the signal occurred. If the traceback function is not called from a signal handler, SigContext will be zero.

Buf is where the traceback information should be stored. It should be PC values, such that Buf[0] is the PC of the caller, Buf[1] is the PC of that function's caller, and so on. Max is the maximum number of entries to store. The function should store a zero to indicate the top of the stack, or that the caller is on a different stack, presumably a Go stack.

Unlike `runtime.Callers`, the PC values returned should, when passed to the symbolizer function, return the file/line of the call instruction. No additional subtraction is required or appropriate.

On all platforms, the traceback function is invoked when a call from Go to C to Go requests a stack trace. On `linux/amd64`, `linux/ppc64le`, `linux/arm64`, and `freebsd/amd64`, the traceback function is also invoked

when a signal is received by a thread that is executing a cgo call. The traceback function should not make assumptions about when it is called, as future versions of Go may make additional calls.

The symbolizer function will be called with a single argument, a pointer to a struct:

```
struct {
    PC      uintptr // program counter to fetch information for
    File    *byte   // file name (NUL terminated)
    Lineno  uintptr // line number
    Func    *byte   // function name (NUL terminated)
    Entry   uintptr // function entry point
    More    uintptr // set non-zero if more info for this PC
    Data    uintptr // unused by runtime, available for function
}
```

In C syntax, this struct will be

```
struct {
    uintptr_t PC;
    char*      File;
    uintptr_t Lineno;
    char*      Func;
    uintptr_t Entry;
    uintptr_t More;
    uintptr_t Data;
};
```

The PC field will be a value returned by a call to the traceback function.

The first time the function is called for a particular traceback, all the fields except PC will be 0. The function should fill in the other fields if possible, setting them to 0/nil if the information is not available. The Data field may be used to store any useful information across calls. The More field should be set to non-zero if there is more information for this PC, zero otherwise. If More is set non-zero, the function will be called again with the same PC, and may return different information (this is intended for use with inlined functions). If More is zero, the function will be called with the next PC value in the traceback. When the traceback is complete, the function will be called once more with PC set to zero; this may be used to free any information. Each call will leave the fields of the struct set to the same values they had upon return, except for the PC field when the More field is zero. The function must not keep a copy of the struct pointer between calls.

When calling SetCgoTraceback, the version argument is the version number of the structs that the functions expect to receive. Currently this must be zero.

The symbolizer function may be nil, in which case the results of the traceback function will be displayed as numbers. If the traceback function is nil, the symbolizer function will never be called. The context function may be nil, in which case the traceback function will only be called with the context field set to zero. If the context function is nil, then calls from Go to C to Go will not show a traceback for the C portion of the call stack.

SetCgoTraceback should be called only once, ideally from an init function.

## func SetFinalizer

```
func SetFinalizer(obj any, finalizer any)
```

SetFinalizer sets the finalizer associated with obj to the provided finalizer function. When the garbage collector finds an unreachable block with an associated finalizer, it clears the association and runs finalizer(obj) in a separate goroutine. This makes obj reachable again, but now without an associated finalizer. Assuming that SetFinalizer is not called again, the next time the garbage collector sees that obj is unreachable, it will free obj.

SetFinalizer(obj, nil) clears any finalizer associated with obj.

The argument obj must be a pointer to an object allocated by calling new, by taking the address of a composite literal, or by taking the address of a local variable. The argument finalizer must be a function that takes a single argument to which obj's type can be assigned, and can have arbitrary ignored return values. If either of these is not true, SetFinalizer may abort the program.

Finalizers are run in dependency order: if A points at B, both have finalizers, and they are otherwise unreachable, only the finalizer for A runs; once A is freed, the finalizer for B can run. If a cyclic structure includes a block with a finalizer, that cycle is not guaranteed to be garbage collected and the finalizer is not guaranteed to run, because there is no ordering that respects the dependencies.

The finalizer is scheduled to run at some arbitrary time after the program can no longer reach the object to which obj points. There is no guarantee that finalizers will run before a program exits, so typically they are useful only for releasing non-memory resources associated with an object during a long-running program. For example, an os.File object could use a finalizer to close the associated operating system file descriptor when a program discards an os.File without calling Close, but it would be a mistake to depend on a finalizer to flush an in-memory I/O buffer such as a bufio.Writer, because the buffer would not be flushed at program exit.

It is not guaranteed that a finalizer will run if the size of \*obj is zero bytes, because it may share same address with other zero-size objects in memory. See

[https://go.dev/ref/spec#Size\\_and\\_alignment\\_guarantees](https://go.dev/ref/spec#Size_and_alignment_guarantees).

It is not guaranteed that a finalizer will run for objects allocated in initializers for package-level variables. Such objects may be linker-allocated, not heap-allocated.

Note that because finalizers may execute arbitrarily far into the future after an object is no longer referenced, the runtime is allowed to perform a space-saving optimization that batches objects together in a single allocation slot. The finalizer for an unreferenced object in such an allocation may never run if it always exists in the same batch as a referenced object. Typically, this batching only happens for tiny (on the order of 16 bytes or less) and pointer-free objects.

A finalizer may run as soon as an object becomes unreachable. In order to use finalizers correctly, the program must ensure that the object is reachable until it is no longer required. Objects stored in global variables, or that can be found by tracing pointers from a global variable, are reachable. For other objects, pass the object to a call of the KeepAlive function to mark the last point in the function where the object must be reachable.



For example, if `p` points to a struct, such as `os.File`, that contains a file descriptor `d`, and `p` has a finalizer that closes that file descriptor, and if the last use of `p` in a function is a call to `syscall.Write(p.d, buf, size)`, then `p` may be unreachable as soon as the program enters `syscall.Write`. The finalizer may run at that moment, closing `p.d`, causing `syscall.Write` to fail because it is writing to a closed file descriptor (or, worse, to an entirely different file descriptor opened by a different goroutine). To avoid this problem, call `KeepAlive(p)` after the call to `syscall.Write`.

A single goroutine runs all finalizers for a program, sequentially. If a finalizer must run for a long time, it should do so by starting a new goroutine.

In the terminology of the Go memory model, a call `SetFinalizer(x, f)` “synchronizes before” the finalization call `f(x)`. However, there is no guarantee that `KeepAlive(x)` or any other use of `x` “synchronizes before” `f(x)`, so in general a finalizer should use a mutex or other synchronization mechanism if it needs to access mutable state in `x`. For example, consider a finalizer that inspects a mutable field in `x` that is modified from time to time in the main program before `x` becomes unreachable and the finalizer is invoked. The modifications in the main program and the inspection in the finalizer need to use appropriate synchronization, such as mutexes or atomic updates, to avoid read-write races.

## func `SetMutexProfileFraction`

added in go1.8

```
func SetMutexProfileFraction(rate int) int
```

`SetMutexProfileFraction` controls the fraction of mutex contention events that are reported in the mutex profile. On average  $1/\text{rate}$  events are reported. The previous rate is returned.

To turn off profiling entirely, pass rate 0. To just read the current rate, pass `rate < 0`. (For  $n > 1$  the details of sampling may change.)

## func `Stack`

```
func Stack(buf []byte, all bool) int
```

`Stack` formats a stack trace of the calling goroutine into `buf` and returns the number of bytes written to `buf`. If `all` is true, `Stack` formats stack traces of all other goroutines into `buf` after the trace for the current goroutine.

## func `StartTrace`

added in go1.5

```
func StartTrace() error
```

`StartTrace` enables tracing for the current process. While tracing, the data will be buffered and available via `ReadTrace`. `StartTrace` returns an error if tracing is already enabled. Most clients should use the `runtime/trace` package or the `testing` package's `-test.trace` flag instead of calling `StartTrace` directly.

## func `StopTrace`

added in go1.5

```
func StopTrace()
```

StopTrace stops tracing, if it was previously enabled. StopTrace only returns after all the reads for the trace have completed.

## func ThreadCreateProfile

```
func ThreadCreateProfile(p []StackRecord) (n int, ok bool)
```

ThreadCreateProfile returns n, the number of records in the thread creation profile. If len(p) >= n, ThreadCreateProfile copies the profile into p and returns n, true. If len(p) < n, ThreadCreateProfile does not change p and returns n, false.

Most clients should use the runtime/pprof package instead of calling ThreadCreateProfile directly.

## func UnlockOSThread

```
func UnlockOSThread()
```

UnlockOSThread undoes an earlier call to LockOSThread. If this drops the number of active LockOSThread calls on the calling goroutine to zero, it unwires the calling goroutine from its fixed operating system thread. If there are no active LockOSThread calls, this is a no-op.

Before calling UnlockOSThread, the caller must ensure that the OS thread is suitable for running other goroutines. If the caller made any permanent changes to the state of the thread that would affect other goroutines, it should not call this function and thus leave the goroutine locked to the OS thread until the goroutine (and hence the thread) exits.

## func Version

```
func Version() string
```

Version returns the Go tree's version string. It is either the commit hash and date at the time of the build or, when possible, a release tag like "go1.3".

## Types

### type BlockProfileRecord

added in go1.1

```
type BlockProfileRecord struct {  
    Count  int64  
    Cycles int64  
    StackRecord  
}
```

BlockProfileRecord describes blocking events originated at a particular call sequence (stack trace).

### type Error

```
type Error interface {  
    error
```

```

// RuntimeError is a no-op function but
// serves to distinguish types that are run time
// errors from ordinary errors: a type is a
// run time error if it has a RuntimeError method.
RuntimeError()
}

```

The Error interface identifies a run time error.

## type **Frame**

added in go1.7

```

type Frame struct {
    // PC is the program counter for the location in this frame.
    // For a frame that calls another frame, this will be the
    // program counter of a call instruction. Because of inlining,
    // multiple frames may have the same PC value, but different
    // symbolic information.
    PC uintptr

    // Func is the Func value of this call frame. This may be nil
    // for non-Go code or fully inlined functions.
    Func *Func

    // Function is the package path-qualified function name of
    // this call frame. If non-empty, this string uniquely
    // identifies a single function in the program.
    // This may be the empty string if not known.
    // If Func is not nil then Function == Func.Name().
    Function string

    // File and Line are the file name and line number of the
    // location in this frame. For non-leaf frames, this will be
    // the location of a call. These may be the empty string and
    // zero, respectively, if not known.
    File string
    Line int

    // Entry point program counter for the function; may be zero
    // if not known. If Func is not nil then Entry ==
    // Func.Entry().
    Entry uintptr
    // contains filtered or unexported fields
}

```

Frame is the information returned by Frames for each call frame.

## type **Frames**

added in go1.7

```

type Frames struct {
    // contains filtered or unexported fields
}

```

```
}
```

Frames may be used to get function/file/line information for a slice of PC values returned by Callers.

► [Example](#)

## func [CallersFrames](#)

added in go1.7

```
func CallersFrames(callers []uintptr) *Frames
```

CallersFrames takes a slice of PC values returned by Callers and prepares to return function/file/line information. Do not change the slice until you are done with the Frames.

## func (\*Frames) [Next](#)

added in go1.7

```
func (ci *Frames) Next() (frame Frame, more bool)
```

Next returns a Frame representing the next call frame in the slice of PC values. If it has already returned all call frames, Next returns a zero Frame.

The more result indicates whether the next call to Next will return a valid Frame. It does not necessarily indicate whether this call returned one.

See the Frames example for idiomatic usage.

## type [Func](#)

```
type Func struct {  
    // contains filtered or unexported fields  
}
```

A Func represents a Go function in the running binary.

## func [FuncForPC](#)

```
func FuncForPC(pc uintptr) *Func
```

FuncForPC returns a \*Func describing the function that contains the given program counter address, or else nil.

If pc represents multiple functions because of inlining, it returns the \*Func describing the innermost function, but with an entry of the outermost function.

## func (\*Func) [Entry](#)

```
func (f *Func) Entry() uintptr
```

Entry returns the entry address of the function.

## func (\*Func) FileLine

```
func (f *Func) FileLine(pc uintptr) (file string, line int)
```

FileLine returns the file name and line number of the source code corresponding to the program counter pc. The result will not be accurate if pc is not a program counter within f.

## func (\*Func) Name

```
func (f *Func) Name() string
```

Name returns the name of the function.

## type MemProfileRecord

```
type MemProfileRecord struct {
    AllocBytes, FreeBytes      int64      // number of bytes allocated, freed
    AllocObjects, FreeObjects  int64      // number of objects allocated, freed
    Stack0                     [32]uintptr // stack trace for this record; ends at first
}
```

A MemProfileRecord describes the live objects allocated by a particular call sequence (stack trace).

## func (\*MemProfileRecord) InUseBytes

```
func (r *MemProfileRecord) InUseBytes() int64
```

InUseBytes returns the number of bytes in use (AllocBytes - FreeBytes).

## func (\*MemProfileRecord) InUseObjects

```
func (r *MemProfileRecord) InUseObjects() int64
```

InUseObjects returns the number of objects in use (AllocObjects - FreeObjects).

## func (\*MemProfileRecord) Stack

```
func (r *MemProfileRecord) Stack() []uintptr
```

Stack returns the stack trace associated with the record, a prefix of r.Stack0.

## type MemStats

```
type MemStats struct {

    // Alloc is bytes of allocated heap objects.
    //
    // This is the same as HeapAlloc (see below).
```

Alloc [uint64](#)

```
// TotalAlloc is cumulative bytes allocated for heap objects.
//
// TotalAlloc increases as heap objects are allocated, but
// unlike Alloc and HeapAlloc, it does not decrease when
// objects are freed.
```

TotalAlloc [uint64](#)

```
// Sys is the total bytes of memory obtained from the OS.
//
// Sys is the sum of the XSys fields below. Sys measures the
// virtual address space reserved by the Go runtime for the
// heap, stacks, and other internal data structures. It's
// likely that not all of the virtual address space is backed
// by physical memory at any given moment, though in general
// it all was at some point.
```

Sys [uint64](#)

```
// Lookups is the number of pointer lookups performed by the
// runtime.
```

```
//
```

```
// This is primarily useful for debugging runtime internals.
```

Lookups [uint64](#)

```
// Mallocs is the cumulative count of heap objects allocated.
// The number of live objects is Mallocs - Frees.
```

Mallocs [uint64](#)

```
// Frees is the cumulative count of heap objects freed.
```

Frees [uint64](#)

```
// HeapAlloc is bytes of allocated heap objects.
```

```
//
```

```
// "Allocated" heap objects include all reachable objects, as
// well as unreachable objects that the garbage collector has
// not yet freed. Specifically, HeapAlloc increases as heap
// objects are allocated and decreases as the heap is swept
// and unreachable objects are freed. Sweeping occurs
// incrementally between GC cycles, so these two processes
// occur simultaneously, and as a result HeapAlloc tends to
// change smoothly (in contrast with the sawtooth that is
// typical of stop-the-world garbage collectors).
```

HeapAlloc [uint64](#)

```
// HeapSys is bytes of heap memory obtained from the OS.
```

```
//
```

```
// HeapSys measures the amount of virtual address space
// reserved for the heap. This includes virtual address space
// that has been reserved but not yet used, which consumes no
// physical memory, but tends to be small, as well as virtual
// address space for which the physical memory has been
// returned to the OS after it became unused (see HeapReleased
```

```
// for a measure of the latter).
//
// HeapSys estimates the largest size the heap has had.
HeapSys uint64

// HeapIdle is bytes in idle (unused) spans.
//
// Idle spans have no objects in them. These spans could be
// (and may already have been) returned to the OS, or they can
// be reused for heap allocations, or they can be reused as
// stack memory.
//
// HeapIdle minus HeapReleased estimates the amount of memory
// that could be returned to the OS, but is being retained by
// the runtime so it can grow the heap without requesting more
// memory from the OS. If this difference is significantly
// larger than the heap size, it indicates there was a recent
// transient spike in live heap size.
HeapIdle uint64

// HeapInuse is bytes in in-use spans.
//
// In-use spans have at least one object in them. These spans
// can only be used for other objects of roughly the same
// size.
//
// HeapInuse minus HeapAlloc estimates the amount of memory
// that has been dedicated to particular size classes, but is
// not currently being used. This is an upper bound on
// fragmentation, but in general this memory can be reused
// efficiently.
HeapInuse uint64

// HeapReleased is bytes of physical memory returned to the OS.
//
// This counts heap memory from idle spans that was returned
// to the OS and has not yet been reacquired for the heap.
HeapReleased uint64

// HeapObjects is the number of allocated heap objects.
//
// Like HeapAlloc, this increases as objects are allocated and
// decreases as the heap is swept and unreachable objects are
// freed.
HeapObjects uint64

// StackInuse is bytes in stack spans.
//
// In-use stack spans have at least one stack in them. These
// spans can only be used for other stacks of the same size.
//
// There is no StackIdle because unused stack spans are
// returned to the heap (and hence counted toward HeapIdle).
```



StackInuse uint64

```
// StackSys is bytes of stack memory obtained from the OS.
//
// StackSys is StackInuse, plus any memory obtained directly
// from the OS for OS thread stacks (which should be minimal).
StackSys uint64
```

```
// MSpanInuse is bytes of allocated mspan structures.
MSpanInuse uint64
```

```
// MSpanSys is bytes of memory obtained from the OS for mspan
// structures.
MSpanSys uint64
```

```
// MCacheInuse is bytes of allocated mcache structures.
MCacheInuse uint64
```

```
// MCacheSys is bytes of memory obtained from the OS for
// mcache structures.
MCacheSys uint64
```

```
// BuckHashSys is bytes of memory in profiling bucket hash tables.
BuckHashSys uint64
```

```
// GCSys is bytes of memory in garbage collection metadata.
GCSys uint64
```

```
// OtherSys is bytes of memory in miscellaneous off-heap
// runtime allocations.
OtherSys uint64
```

```
// NextGC is the target heap size of the next GC cycle.
//
// The garbage collector's goal is to keep HeapAlloc ≤ NextGC.
// At the end of each GC cycle, the target for the next cycle
// is computed based on the amount of reachable data and the
// value of GOGC.
NextGC uint64
```

```
// LastGC is the time the last garbage collection finished, as
// nanoseconds since 1970 (the UNIX epoch).
LastGC uint64
```

```
// PauseTotalNs is the cumulative nanoseconds in GC
// stop-the-world pauses since the program started.
//
// During a stop-the-world pause, all goroutines are paused
// and only the garbage collector can run.
PauseTotalNs uint64
```

```
// PauseNs is a circular buffer of recent GC stop-the-world
// pausetimes in nanoseconds.
```

```

//
// The most recent pause is at PauseNs[(NumGC+255)%256]. In
// general, PauseNs[N%256] records the time paused in the most
// recent N%256th GC cycle. There may be multiple pauses per
// GC cycle; this is the sum of all pauses during a cycle.
PauseNs [256]uint64

// PauseEnd is a circular buffer of recent GC pause end times,
// as nanoseconds since 1970 (the UNIX epoch).
//
// This buffer is filled the same way as PauseNs. There may be
// multiple pauses per GC cycle; this records the end of the
// last pause in a cycle.
PauseEnd [256]uint64

// NumGC is the number of completed GC cycles.
NumGC uint32

// NumForcedGC is the number of GC cycles that were forced by
// the application calling the GC function.
NumForcedGC uint32

// GCCPUFraction is the fraction of this program's available
// CPU time used by the GC since the program started.
//
// GCCPUFraction is expressed as a number between 0 and 1,
// where 0 means GC has consumed none of this program's CPU. A
// program's available CPU time is defined as the integral of
// GOMAXPROCS since the program started. That is, if
// GOMAXPROCS is 2 and a program has been running for 10
// seconds, its "available CPU" is 20 seconds. GCCPUFraction
// does not include CPU time used for write barrier activity.
//
// This is the same as the fraction of CPU reported by
// GODEBUG=gctrace=1.
GCCPUFraction float64

// EnableGC indicates that GC is enabled. It is always true,
// even if GOGC=off.
EnableGC bool

// DebugGC is currently unused.
DebugGC bool

// BySize reports per-size class allocation statistics.
//
// BySize[N] gives statistics for allocations of size S where
// BySize[N-1].Size < S ≤ BySize[N].Size.
//
// This does not report allocations larger than BySize[60].Size.
BySize [61]struct {
    // Size is the maximum byte size of an object in this
    // size class.

```

Size `uint32`

```
// Mallocs is the cumulative count of heap objects
// allocated in this size class. The cumulative bytes
// of allocation is Size*Mallocs. The number of live
// objects in this size class is Mallocs - Frees.
Mallocs uint64

// Frees is the cumulative count of heap objects freed
// in this size class.
Frees uint64
}
```

A MemStats records statistics about the memory allocator.

### type `StackRecord`

```
type StackRecord struct {
    Stack0 [32]uintptr // stack trace for this record; ends at first 0 entry
}
```

A StackRecord describes a single execution stack.

### func (\*StackRecord) `Stack`

```
func (r *StackRecord) Stack() []uintptr
```

Stack returns the stack trace associated with the record, a prefix of r.Stack0.

### type `TypeAssertionError`

```
type TypeAssertionError struct {
    // contains filtered or unexported fields
}
```

A TypeAssertionError explains a failed type assertion.

### func (\*TypeAssertionError) `Error`

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

### func (\*TypeAssertionError) `RuntimeError`

```
func (*TypeAssertionError) RuntimeError()
```



## Source Files

[View all](#)

<a href="#">alg.go</a>	<a href="#">map_faststr.go</a>	<a href="#">profbuf.go</a>
<a href="#">arena.go</a>	<a href="#">mbarrier.go</a>	<a href="#">proflabel.go</a>
<a href="#">asan0.go</a>	<a href="#">mbitmap.go</a>	<a href="#">race0.go</a>
<a href="#">atomic_pointer.go</a>	<a href="#">mcache.go</a>	<a href="#">rdebug.go</a>
<a href="#">cgo.go</a>	<a href="#">mcentral.go</a>	<a href="#">relax_stub.go</a>
<a href="#">cgo_mmap.go</a>	<a href="#">mcheckmark.go</a>	<a href="#">retry.go</a>
<a href="#">cgo_sigaction.go</a>	<a href="#">mem.go</a>	<a href="#">runtime.go</a>
<a href="#">cgocall.go</a>	<a href="#">mem_linux.go</a>	<a href="#">runtime1.go</a>
<a href="#">cgocallback.go</a>	<a href="#">metrics.go</a>	<a href="#">runtime2.go</a>
<a href="#">cgocheck.go</a>	<a href="#">mfinal.go</a>	<a href="#">runtime_boring.go</a>
<a href="#">chan.go</a>	<a href="#">mfixalloc.go</a>	<a href="#">rwmutex.go</a>
<a href="#">checkptr.go</a>	<a href="#">mgc.go</a>	<a href="#">select.go</a>
<a href="#">compiler.go</a>	<a href="#">mgclimit.go</a>	<a href="#">sema.go</a>
<a href="#">complex.go</a>	<a href="#">mgcmark.go</a>	<a href="#">signal_amd64.go</a>
<a href="#">covercounter.go</a>	<a href="#">mgcpacer.go</a>	<a href="#">signal_linux_amd64.go</a>
<a href="#">covermeta.go</a>	<a href="#">mgcscavenge.go</a>	<a href="#">signal_unix.go</a>
<a href="#">cpuflags.go</a>	<a href="#">mgcstack.go</a>	<a href="#">sigqueue.go</a>
<a href="#">cpuflags_amd64.go</a>	<a href="#">mgcsweep.go</a>	<a href="#">sigqueue_note.go</a>
<a href="#">cpuprof.go</a>	<a href="#">mgcwork.go</a>	<a href="#">sigtab_linux_generic.go</a>
<a href="#">cputicks.go</a>	<a href="#">mheap.go</a>	<a href="#">sizeclasses.go</a>
<a href="#">create_file_unix.go</a>	<a href="#">mpagealloc.go</a>	<a href="#">slice.go</a>
<a href="#">debug.go</a>	<a href="#">mpagealloc_64bit.go</a>	<a href="#">softfloat64.go</a>
<a href="#">debugcall.go</a>	<a href="#">mpagecache.go</a>	<a href="#">stack.go</a>
<a href="#">debuglog.go</a>	<a href="#">mpallocbits.go</a>	<a href="#">stkframe.go</a>
<a href="#">debuglog_off.go</a>	<a href="#">mprof.go</a>	<a href="#">string.go</a>
<a href="#">defs_linux_amd64.go</a>	<a href="#">mranges.go</a>	<a href="#">stubs.go</a>
<a href="#">env_posix.go</a>	<a href="#">msan0.go</a>	<a href="#">stubs2.go</a>
<a href="#">error.go</a>	<a href="#">msize.go</a>	<a href="#">stubs3.go</a>
<a href="#">exithook.go</a>	<a href="#">mspanset.go</a>	<a href="#">stubs_amd64.go</a>
<a href="#">extern.go</a>	<a href="#">mstats.go</a>	<a href="#">stubs_linux.go</a>
<a href="#">fastlog2.go</a>	<a href="#">mwbbuf.go</a>	<a href="#">symtab.go</a>
<a href="#">fastlog2table.go</a>	<a href="#">nbpipeline2.go</a>	<a href="#">sys_nonppc64x.go</a>
<a href="#">float.go</a>	<a href="#">netpoll.go</a>	<a href="#">sys_x86.go</a>
<a href="#">hash64.go</a>	<a href="#">netpoll_epoll.go</a>	<a href="#">time.go</a>
<a href="#">heapdump.go</a>	<a href="#">os_linux.go</a>	<a href="#">time_nofake.go</a>
<a href="#">histogram.go</a>	<a href="#">os_linux_generic.go</a>	<a href="#">timeasm.go</a>
<a href="#">iface.go</a>	<a href="#">os_linux_noauxv.go</a>	<a href="#">tls_stub.go</a>
<a href="#">lfstack.go</a>	<a href="#">os_linux_x86.go</a>	<a href="#">trace.go</a>
<a href="#">lfstack_64bit.go</a>	<a href="#">os_nonopenbsd.go</a>	<a href="#">traceback.go</a>
<a href="#">lock_futex.go</a>	<a href="#">pagetrace_off.go</a>	<a href="#">type.go</a>
<a href="#">lockrank.go</a>	<a href="#">panic.go</a>	<a href="#">typekind.go</a>
<a href="#">lockrank_off.go</a>	<a href="#">plugin.go</a>	<a href="#">unsafe.go</a>
<a href="#">malloc.go</a>	<a href="#">preempt.go</a>	<a href="#">utf8.go</a>
<a href="#">map.go</a>	<a href="#">preempt_nonwindows.go</a>	<a href="#">vdso_elf64.go</a>
<a href="#">map_fast32.go</a>	<a href="#">print.go</a>	<a href="#">vdso_linux.go</a>
<a href="#">map_fast64.go</a>	<a href="#">proc.go</a>	<a href="#">vdso_linux_amd64.go</a>

## Directories

[Expand all](#)

### [cgo](#)

Package cgo contains runtime support for code generated by the cgo tool.

### [coverage](#)

### [debug](#)

Package debug contains facilities for programs to debug themselves while they are running.

### [metrics](#)

Package metrics provides a stable interface to access implementation-defined metrics exported by the Go runtime.

### [pprof](#)

Package pprof writes runtime profiling data in the format expected by the pprof visualization tool.

### [race](#)

Package race implements data race detection logic.

### [trace](#)

Package trace contains facilities for programs to generate traces for the Go execution tracer.

► [internal](#)

## Why Go

[Use Cases](#)

[Case Studies](#)

## Get Started

[Playground](#)

[Tour](#)

[Stack Overflow](#)

[Help](#)

## Packages

[Standard Library](#)

[About Go Packages](#)

## About

[Download](#)

[Blog](#)

[Issue Tracker](#)

[Release Notes](#)

[Brand Guidelines](#)

[Code of Conduct](#)

## Connect

[Twitter](#)

[GitHub](#)

[Slack](#)

[r/golang](#)

[Meetup](#)

[Golang Weekly](#)

[Copyright](#)

[Terms of Service](#)

[Privacy Policy](#)

[Report an Issue](#)



Google