# pprof

package standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 20 | Imported by: 13,314

| | |
|---|---|
| **Details** | Valid go.mod file ? Redistributable license ? Tagged version ? Stable version ? Learn more |
| **Repository** | cs.opensource.google/go/go |
| **Links** | Report a Vulnerability |

## Documentation

Rendered for linux/amd64

### Overview

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

### Profiling a Go program

The first step to profiling a Go program is to enable profiling. Support for profiling benchmarks built with the standard testing package is built into go test. For example, the following command runs benchmarks in the current directory and writes the CPU and memory profiles to cpu.prof and mem.prof:

```
go test -cpuprofile cpu.prof -memprofile mem.prof -bench .
```

To add equivalent profiling support to a standalone program, add code like the following to your main function:

```
var cpuprofile = flag.String("cpuprofile", "", "write cpu profile to `file`")
var memprofile = flag.String("memprofile", "", "write memory profile to `file`")

func main() {
    flag.Parse()
    if *cpuprofile != "" {
        f, err := os.Create(*cpuprofile)
        if err != nil {
            log.Fatal("could not create CPU profile: ", err)
        }
        defer f.Close() // error handling omitted for example
        if err := pprof.StartCPUProfile(f); err != nil {
            log.Fatal("could not start CPU profile: ", err)
        }
        defer pprof.StopCPUProfile()
    }

    // ... rest of the program ...

    if *memprofile != "" {
        f, err := os.Create(*memprofile)
        if err != nil {
            log.Fatal("could not create memory profile: ", err)
        }
        defer f.Close() // error handling omitted for example
        runtime.GC() // get up-to-date statistics
        if err := pprof.WriteHeapProfile(f); err != nil {
            log.Fatal("could not write memory profile: ", err)
        }
    }
}
```

There is also a standard HTTP interface to profiling data. Adding the following line will install handlers under the /debug/pprof/ URL to download live profiles:

```
import _ "net/http/pprof"
```

See the net/http/pprof package for more details.

Profiles can then be visualized with the pprof tool:

```
go tool pprof cpu.prof
```

There are many commands available from the pprof command line. Commonly used commands include "top", which prints a summary of the top program hot-spots, and "web", which opens an interactive graph of hot-spots and their call graphs. Use "help" for information on all pprof commands.

For more information about pprof, see https://github.com/google/pprof/blob/master/doc/README.md.

## Index

## Constants

This section is empty.

## Variables

This section is empty.

## Functions

### func Do

added in go1.9

```
func Do(ctx context.Context, labels LabelSet, f func(context.Context))
```

Do calls f with a copy of the parent context with the given labels added to the parent's label map. Goroutines spawned while executing f will inherit the augmented label-set. Each key/value pair in labels is inserted into the label map in the order provided, overriding any previous value for the same key. The augmented label map will be set for the duration of the call to f and restored once f returns.

### func ForLabels

added in go1.9

```
func ForLabels(ctx context.Context, f func(key, value string) bool)
```

ForLabels invokes f with each label set on the context. The function f should return true to continue iteration or false to stop iteration early.

### func Label

added in go1.9

```
func Label(ctx context.Context, key string) (string, bool)
```

Label returns the value of the label with the given key on ctx, and a boolean indicating whether that label exists.

### func SetGoroutineLabels

added in go1.9

```
func SetGoroutineLabels(ctx context.Context)
```

SetGoroutineLabels sets the current goroutine's labels to match ctx. A new goroutine inherits the labels of the goroutine that created it. This is a lower-level API than Do, which should be used instead when possible.

### func StartCPUProfile

```
func StartCPUProfile(w io.Writer) error
```

StartCPUProfile enables CPU profiling for the current process. While profiling, the profile will be buffered and written to w. StartCPUProfile returns an error if profiling is already enabled.

On Unix-like systems, StartCPUProfile does not work by default for Go code built with -buildmode=c-archive or -buildmode=c-shared. StartCPUProfile relies on the SIGPROF signal, but that signal will be delivered to the main program's SIGPROF signal handler (if any) not to the one used by Go. To make it work, call os/signal.Notify for syscall.SIGPROF, but note that doing so may break any profiling being done by the main program.

### func StopCPUProfile

```
func StopCPUProfile()
```

StopCPUProfile stops the current CPU profile, if any. StopCPUProfile only returns after all the writes for the profile have completed.

### func WithLabels

added in go1.9

```
func WithLabels(ctx context.Context, labels LabelSet) context.Context
```

WithLabels returns a new context.Context with the given labels added. A label overwrites a prior label with the same key.

### func WriteHeapProfile

```
func WriteHeapProfile(w io.Writer) error
```

WriteHeapProfile is shorthand for Lookup("heap").WriteTo(w, 0). It is preserved for backwards compatibility.

## Types

### type LabelSet

added in go1.9

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

LabelSet is a set of labels.

## func Labels

added in go1.9

```
func Labels(args ...string) LabelSet
```

Labels takes an even number of strings representing key-value pairs and makes a LabelSet containing them. A label overwrites a prior label with the same key. Currently only the CPU and goroutine profiles utilize any labels information. See https://golang.org/issue/23458 for details.

## type Profile

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

A Profile is a collection of stack traces showing the call sequences that led to instances of a particular event, such as allocation. Packages can create and maintain their own profiles; the most common use is for tracking resources that must be explicitly closed, such as files or network connections.

A Profile's methods can be called from multiple goroutines simultaneously.

Each Profile has a unique name. A few profiles are predefined:

```
goroutine    - stack traces of all current goroutines
heap         - a sampling of memory allocations of live objects
allocs       - a sampling of all past memory allocations
threadcreate - stack traces that led to the creation of new OS threads
block        - stack traces that led to blocking on synchronization primitives
mutex        - stack traces of holders of contended mutexes
```

These predefined profiles maintain themselves and panic on an explicit Add or Remove method call.

The heap profile reports statistics as of the most recently completed garbage collection; it elides more recent allocation to avoid skewing the profile away from live data and toward garbage. If there has been no garbage collection at all, the heap profile reports all known allocations. This exception helps mainly in programs running without garbage collection enabled, usually for debugging purposes.

The heap profile tracks both the allocation sites for all live objects in the application memory and for all objects allocated since the program start. Pprof's -inuse_space, -inuse_objects, -alloc_space, and -alloc_objects flags select which to display, defaulting to -inuse_space (live objects, scaled by size).

The allocs profile is the same as the heap profile but changes the default pprof display to -alloc_space, the total number of bytes allocated since the program began (including garbage-collected bytes).

The CPU profile is not available as a Profile. It has a special API, the StartCPUProfile and StopCPUProfile functions, because it streams output to a writer during profiling.

## func Lookup

```
func Lookup(name string) *Profile
```

Lookup returns the profile with the given name, or nil if no such profile exists.

### func NewProfile

```
func NewProfile(name string) *Profile
```

NewProfile creates a new profile with the given name. If a profile with that name already exists, NewProfile panics. The convention is to use a 'import/path.' prefix to create separate name spaces for each package. For compatibility with various tools that read pprof data, profile names should not contain spaces.

### func Profiles

```
func Profiles() []*Profile
```

Profiles returns a slice of all the known profiles, sorted by name.

### func (*Profile) Add

```
func (p *Profile) Add(value any, skip int)
```

Add adds the current execution stack to the profile, associated with value. Add stores value in an internal map, so value must be suitable for use as a map key and will not be garbage collected until the corresponding call to Remove. Add panics if the profile already contains a stack for value.

The skip parameter has the same meaning as runtime.Caller's skip and controls where the stack trace begins. Passing skip=0 begins the trace in the function calling Add. For example, given this execution stack:

```
Add
called from rpc.NewClient
called from mypkg.Run
called from main.main
```

Passing skip=0 begins the stack trace at the call to Add inside rpc.NewClient. Passing skip=1 begins the stack trace at the call to NewClient inside mypkg.Run.

### func (*Profile) Count

```
func (p *Profile) Count() int
```

Count returns the number of execution stacks currently in the profile.

### func (*Profile) Name

```
func (p *Profile) Name() string
```

Name returns this profile's name, which can be passed to Lookup to reobtain the profile.

### func (*Profile) Remove

```
func (p *Profile) Remove(value any)
```

Remove removes the execution stack associated with value from the profile. It is a no-op if the value is not in the profile.

### func (*Profile) WriteTo

```
func (p *Profile) WriteTo(w io.Writer, debug int) error
```

WriteTo writes a pprof-formatted snapshot of the profile to w. If a write to w returns an error, WriteTo returns that error. Otherwise, WriteTo returns nil.

The debug parameter enables additional output. Passing debug=0 writes the gzip-compressed protocol buffer described in https://github.com/google/pprof/tree/master/proto#overview. Passing debug=1 writes the legacy text format with comments translating addresses to function names and line numbers, so that a programmer can read the profile without tools.

The predefined profiles may assign meaning to other debug values; for example, when printing the "goroutine" profile, debug=2 means to print the goroutine stacks in the same form that a Go program uses when dying due to an unrecovered panic.

## Notes

## Bugs

- Profiles are only as good as the kernel support used to generate them. See https://golang.org/issue/13841 for details about known problems.

## Source Files

View all

| | | |
|---|---|---|
| elf.go | pprof.go | protobuf.go |
| label.go | pprof_rusage.go | protomem.go |
| map.go | proto.go | runtime.go |
| pe.go | proto_other.go | |

Stack Overflow

Help

Issue Tracker

Release Notes

Brand Guidelines

Code of Conduct

## Connect

Twitter

GitHub

Slack

r/golang

Meetup

Golang Weekly