trace

package | standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 7 | Imported by: 1,740

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

Overview

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

Tracing runtime activities

The execution trace captures a wide range of execution events such as goroutine creation/blocking/unblocking, syscall enter/exit/block, GC-related events, changes of heap size, processor start/stop, etc. When CPU profiling is active, the execution tracer makes an effort to include those samples as well. A precise nanosecond-precision timestamp and a stack trace is captured for most events. The generated trace can be interpreted using `go tool trace`.

Support for tracing tests and benchmarks built with the standard testing package is built into `go test`. For example, the following command runs the test in the current directory and writes the trace file (trace.out).

```
go test -trace=trace.out
```

This runtime/trace package provides APIs to add equivalent tracing support to a standalone program. See the Example that demonstrates how to use this API to enable tracing.

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

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

See the net/http/pprof package for more details about all of the debug endpoints installed by this import.

User annotation

Package trace provides user annotation APIs that can be used to log interesting events during execution.

There are three types of user annotations: log messages, regions, and tasks.

Log emits a timestamped message to the execution trace along with additional information such as the category of the message and which goroutine called Log. The execution tracer provides UIs to filter and group goroutines using the log category and the message supplied in Log.

A region is for logging a time interval during a goroutine's execution. By definition, a region starts and ends in the same goroutine. Regions can be nested to represent subintervals. For example, the following code records four regions in the execution trace to trace the durations of sequential steps in a cappuccino making operation.

```
trace.WithRegion(ctx, "makeCappuccino", func() {

   // orderID allows to identify a specific order
   // among many cappuccino order region records.
   trace.Log(ctx, "orderID", orderID)

   trace.WithRegion(ctx, "steamMilk", steamMilk)
   trace.WithRegion(ctx, "extractCoffee", extractCoffee)
   trace.WithRegion(ctx, "mixMilkCoffee", mixMilkCoffee)
})
```

A task is a higher-level component that aids tracing of logical operations such as an RPC request, an HTTP request, or an interesting local operation which may require multiple goroutines working together. Since tasks can involve multiple goroutines, they are tracked via a context.Context object. NewTask creates a new task and embeds it in the returned context.Context object. Log messages and regions are attached to the task, if any, in the Context passed to Log and WithRegion.

For example, assume that we decided to froth milk, extract coffee, and mix milk and coffee in separate goroutines. With a task, the trace tool can identify the goroutines involved in a specific cappuccino order.

```
ctx, task := trace.NewTask(ctx, "makeCappuccino")
trace.Log(ctx, "orderID", orderID)

milk := make(chan bool)
espresso := make(chan bool)

go func() {
        trace.WithRegion(ctx, "steamMilk", steamMilk)
        milk <- true
}()
go func() {
```

```
        trace.WithRegion(ctx, "extractCoffee", extractCoffee)
        espresso <- true
}()
go func() {
        defer task.End() // When assemble is done, the order is complete.
        <-espresso
        <-milk
        trace.WithRegion(ctx, "mixMilkCoffee", mixMilkCoffee)
}()
```

The trace tool computes the latency of a task by measuring the time between the task creation and the task end and provides latency distributions for each task type found in the trace.

▶ Example

Index

Examples

Package

Constants

This section is empty.

Variables

This section is empty.

Functions

func IsEnabled

added in go1.11

```
func IsEnabled() bool
```

IsEnabled reports whether tracing is enabled. The information is advisory only. The tracing status may have changed by the time this function returns.

func Log

added in go1.11

```
func Log(ctx context.Context, category, message string)
```

Log emits a one-off event with the given category and message. Category can be empty and the API assumes there are only a handful of unique categories in the system.

func Logf

added in go1.11

```
func Logf(ctx context.Context, category, format string, args ...any)
```

Logf is like Log, but the value is formatted using the specified format spec.

func Start

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

Start enables tracing for the current program. While tracing, the trace will be buffered and written to w. Start returns an error if tracing is already enabled.

func Stop

```
func Stop()
```

Stop stops the current tracing, if any. Stop only returns after all the writes for the trace have completed.

func WithRegion

added in go1.11

```
func WithRegion(ctx context.Context, regionType string, fn func())
```

WithRegion starts a region associated with its calling goroutine, runs fn, and then ends the region. If the context carries a task, the region is associated with the task. Otherwise, the region is attached to the background task.

The regionType is used to classify regions, so there should be only a handful of unique region types.

Types

type Region

added in go1.11

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

Region is a region of code whose execution time interval is traced.

func StartRegion

added in go1.11

```
func StartRegion(ctx context.Context, regionType string) *Region
```

StartRegion starts a region and returns a function for marking the end of the region. The returned Region's End function must be called from the same goroutine where the region was started. Within each goroutine, regions must nest. That is, regions started after this region must be ended before this region can be ended. Recommended usage is

```
defer trace.StartRegion(ctx, "myTracedRegion").End()
```

func (*Region) End

added in go1.11

```
func (r *Region) End()
```

End marks the end of the traced code region.

type Task

added in go1.11

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

Task is a data type for tracing a user-defined, logical operation.

func NewTask

added in go1.11

```
func NewTask(pctx context.Context, taskType string) (ctx context.Context, task *Task)
```

NewTask creates a task instance with the type taskType and returns it along with a Context that carries the task. If the input context contains a task, the new task is its subtask.

The taskType is used to classify task instances. Analysis tools like the Go execution tracer may assume there are only a bounded number of unique task types in the system.

The returned end function is used to mark the task's end. The trace tool measures task latency as the time between task creation and when the end function is called, and provides the latency distribution per task type. If the end function is called multiple times, only the first call is used in the latency measurement.

```
ctx, task := trace.NewTask(ctx, "awesomeTask")
trace.WithRegion(ctx, "preparation", prepWork)
// preparation of the task
go func() {  // continue processing the task in a separate goroutine.
    defer task.End()
    trace.WithRegion(ctx, "remainingWork", remainingWork)
}()
```

func (*Task) End

added in go1.11

```
func (t *Task) End()
```

End marks the end of the operation represented by the Task.

Source Files

View all

annotation.go

trace.go