Diagnostics

Basics

The Go ecosystem provides a large suite of APIs and tools to diagnose logic and performance problems in Go programs. This page summarizes the available tools and helps Go users pick the right one for their specific problem.

Diagnostics solutions can be categorized into the following groups:

- **Profiling**: Profiling tools analyze the complexity and costs of a Go program such as its memory usage and frequently called functions to identify the expensive sections of a Go program.
- **Tracing**: Tracing is a way to instrument code to analyze latency throughout the lifecycle of a call or user request. Traces provide an overview of how much latency each component contributes to the overall latency in a system. Traces can span multiple Go processes.
- **Debugging**: Debugging allows us to pause a Go program and examine its execution. Program state and flow can be verified with debugging.
- **Runtime statistics and events**: Collection and analysis of runtime stats and events provides a high-level overview of the health of Go programs. Spikes/dips of metrics helps us to identify changes in throughput, utilization, and performance.

Note: Some diagnostics tools may interfere with each other. For example, precise memory profiling skews CPU profiles and goroutine blocking profiling affects scheduler trace. Use tools in isolation to get more precise info.

Profiling

Profiling is useful for identifying expensive or frequently called sections of code. The Go runtime provides profiling data in the format expected by the pprof visualization tool. The profiling data can be collected during testing via `go test` or endpoints made available from the net/http/pprof package. Users need to collect the profiling data and use pprof tools to filter and visualize the top code paths.

Predefined profiles provided by the runtime/pprof package:

- **cpu**: CPU profile determines where a program spends its time while actively consuming CPU cycles (as opposed to while sleeping or waiting for I/O).

- **heap**: Heap profile reports memory allocation samples; used to monitor current and historical memory usage, and to check for memory leaks.
- **threadcreate**: Thread creation profile reports the sections of the program that lead the creation of new OS threads.
- **goroutine**: Goroutine profile reports the stack traces of all current goroutines.
- **block**: Block profile shows where goroutines block waiting on synchronization primitives (including timer channels). Block profile is not enabled by default; use `runtime.SetBlockProfileRate` to enable it.
- **mutex**: Mutex profile reports the lock contentions. When you think your CPU is not fully utilized due to a mutex contention, use this profile. Mutex profile is not enabled by default, see `runtime.SetMutexProfileFraction` to enable it.

Tracing

Tracing is a way to instrument code to analyze latency throughout the lifecycle of a chain of calls. Go provides golang.org/x/net/trace package as a minimal tracing backend per Go node and provides a minimal instrumentation library with a simple dashboard. Go also provides an execution tracer to trace the runtime events within an interval.

Tracing enables us to:

- Instrument and analyze application latency in a Go process.
- Measure the cost of specific calls in a long chain of calls.
- Figure out the utilization and performance improvements. Bottlenecks are not always obvious without tracing data.

Distributed tracing is a way to instrument code to analyze latency throughout the lifecycle of a user request. When a system is distributed and when conventional profiling and debugging tools don't scale, you might want to use distributed tracing tools to analyze the performance of your user requests and RPCs.

Runtime statistics and events

The runtime provides stats and reporting of internal events for users to diagnose performance and utilization problems at the runtime level.

Users can monitor these stats to better understand the overall health and performance of Go programs. Some frequently monitored stats and states:

- `runtime.ReadMemStats` reports the metrics related to heap allocation and garbage collection. Memory stats are useful for monitoring how much memory resources a process is consuming, whether the process can utilize memory well, and to catch memory leaks.
- `debug.ReadGCStats` reads statistics about garbage collection. It is useful to see how much of the resources are spent on GC pauses. It also reports a timeline of garbage collector pauses and pause time percentiles.
- `debug.Stack` returns the current stack trace. Stack trace is useful to see how many goroutines are currently running, what they are doing, and whether they are blocked or not.
- `debug.WriteHeapDump` suspends the execution of all goroutines and allows you to dump the heap to a file. A heap dump is a snapshot of a Go process' memory at a given time. It contains all allocated objects as well as goroutines, finalizers, and more.
- `runtime.NumGoroutine` returns the number of current goroutines. The value can be monitored to see whether enough goroutines are utilized, or to detect goroutine leaks.