

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

metrics


package

standard library

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

Imported by: 48

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

[Interface](#)[Metric key format](#)[A note about floats](#)[Supported metrics](#)

Package metrics provides a stable interface to access implementation-defined metrics exported by the Go runtime. This package is similar to existing functions like `runtime.ReadMemStats` and `debug.ReadGCStats`, but significantly more general.

The set of metrics defined by this package may evolve as the runtime itself evolves, and also enables variation across Go implementations, whose relevant metric sets may not intersect.

Interface

Metrics are designated by a string key, rather than, for example, a field name in a struct. The full list of supported metrics is always available in the slice of `Descriptions` returned by `All`. Each `Description` also includes useful information about the metric.

Thus, users of this API are encouraged to sample supported metrics defined by the slice returned by `All` to remain compatible across Go versions. Of course, situations arise where reading specific metrics is critical. For these cases, users are encouraged to use build tags, and although metrics may be deprecated and removed, users should consider this to be an exceptional and rare event, coinciding with a very large change in a particular Go implementation.

Each metric key also has a "kind" that describes the format of the metric's value. In the interest of not breaking users of this package, the "kind" for a given metric is guaranteed not to change. If it must change, then a new metric will be introduced with a new key and a new "kind."

Metric key format

As mentioned earlier, metric keys are strings. Their format is simple and well-defined, designed to be both human and machine readable. It is split into two components, separated by a colon: a rooted path and a unit. The choice to include the unit in the key is motivated by compatibility: if a metric's unit changes, its semantics likely did also, and a new key should be introduced.

For more details on the precise definition of the metric key's path and unit formats, see the documentation of the Name field of the Description struct.

A note about floats

This package supports metrics whose values have a floating-point representation. In order to improve ease-of-use, this package promises to never produce the following classes of floating-point values: NaN, infinity.

Supported metrics

Below is the full list of supported metrics, ordered lexicographically.

`/cgo/go-to-c-calls:calls`

Count of calls made from Go to C by the current process.

`/cpu/classes/gc/mark/assist:cpu-seconds`

Estimated total CPU time goroutines spent performing GC tasks to assist the GC and prevent it from falling behind the application. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics.

`/cpu/classes/gc/mark/dedicated:cpu-seconds`

Estimated total CPU time spent performing GC tasks on processors (as defined by `GOMAXPROCS`) dedicated to those tasks. This includes time spent with the world stopped due to the GC. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics.

`/cpu/classes/gc/mark/idle:cpu-seconds`

Estimated total CPU time spent performing GC tasks on spare CPU resources that the Go scheduler could not otherwise find a use for. This should be subtracted from the total GC CPU time to obtain a measure of compulsory GC CPU time. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics.

`/cpu/classes/gc/pause:cpu-seconds`

Estimated total CPU time spent with the application paused by the GC. Even if only one thread is running during the pause, this is computed as GOMAXPROCS times the pause latency because nothing else can be executing. This is the exact sum of samples in `/gc/pause:seconds` if each sample is multiplied by GOMAXPROCS at the time it is taken. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics.

`/cpu/classes/gc/total:cpu-seconds`

Estimated total CPU time spent performing GC tasks. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics. Sum of all metrics in `/cpu/classes/gc`.

`/cpu/classes/idle:cpu-seconds`

Estimated total available CPU time not spent executing any Go or Go runtime code. In other words, the part of `/cpu/classes/total:cpu-seconds` that was unused. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics.

`/cpu/classes/scavenge/assist:cpu-seconds`

Estimated total CPU time spent returning unused memory to the underlying platform in response eagerly in response to memory pressure. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics.

`/cpu/classes/scavenge/background:cpu-seconds`

Estimated total CPU time spent performing background tasks to return unused memory to the underlying platform. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics.

`/cpu/classes/scavenge/total:cpu-seconds`

Estimated total CPU time spent performing tasks that return unused memory to the underlying platform. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics. Sum of all metrics in `/cpu/classes/scavenge`.

`/cpu/classes/total:cpu-seconds`

Estimated total available CPU time for user Go code or the Go runtime, as defined by GOMAXPROCS. In other words, GOMAXPROCS integrated over the wall-clock duration this process has been executing for. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics. Sum of all metrics in `/cpu/classes`.

`/cpu/classes/user:cpu-seconds`

Estimated total CPU time spent running user Go code. This may also include some small amount of time spent in the Go runtime. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with other `/cpu/classes` metrics.

`/gc/cycles/automatic:gc-cycles`

Count of completed GC cycles generated by the Go runtime.

`/gc/cycles/forced:gc-cycles`

Count of completed GC cycles forced by the application.

`/gc/cycles/total:gc-cycles`

Count of all completed GC cycles.

`/gc/heap/allocs-by-size:bytes`

Distribution of heap allocations by approximate size.

Note that this does not include tiny objects as defined by `/gc/heap/tiny/allocs:objects`, only tiny blocks.

`/gc/heap/allocs:bytes`

Cumulative sum of memory allocated to the heap by the application.

`/gc/heap/allocs:objects`

Cumulative count of heap allocations triggered by the application.

Note that this does not include tiny objects as defined by `/gc/heap/tiny/allocs:objects`, only tiny blocks.

`/gc/heap/frees-by-size:bytes`

Distribution of freed heap allocations by approximate size.

Note that this does not include tiny objects as defined by `/gc/heap/tiny/allocs:objects`, only tiny blocks.

`/gc/heap/frees:bytes`

Cumulative sum of heap memory freed by the garbage collector.

`/gc/heap/frees:objects`

Cumulative count of heap allocations whose storage was freed by the garbage collector.

Note that this does not include tiny objects as defined by `/gc/heap/tiny/allocs:objects`, only tiny blocks.

`/gc/heap/goal:bytes`

Heap size target for the end of the GC cycle.

`/gc/heap/objects:objects`

Number of objects, live or unswept, occupying heap memory.

`/gc/heap/tiny/allocs:objects`

Count of small allocations that are packed together into blocks.

These allocations are counted separately from other allocations because each individual allocation is not tracked by the runtime, only their block. Each block is already accounted for in

allocs-by-size and frees-by-size.

/gc/limiter/last-enabled:gc-cycle

GC cycle the last time the GC CPU limiter was enabled.

This metric is useful for diagnosing the root cause of an out-of-memory error, because the limiter trades memory for CPU time when the GC's CPU time gets too high. This is most likely to occur with use of SetMemoryLimit.

The first GC cycle is cycle 1, so a value of 0 indicates that it was never enabled.

/gc/pauses:seconds

Distribution individual GC-related stop-the-world pause latencies.

/gc/stack/starting-size:bytes

The stack size of new goroutines.

/memory/classes/heap/free:bytes

Memory that is completely free and eligible to be returned to the underlying system, but has not been. This metric is the runtime's estimate of free address space that is backed by physical memory.

/memory/classes/heap/objects:bytes

Memory occupied by live objects and dead objects that have not yet been marked free by the garbage collector.

/memory/classes/heap/released:bytes

Memory that is completely free and has been returned to the underlying system. This metric is the runtime's estimate of free address space that is still mapped into the process, but is not backed by physical memory.

/memory/classes/heap/stacks:bytes

Memory allocated from the heap that is reserved for stack space, whether or not it is currently in-use.

/memory/classes/heap/unused:bytes

Memory that is reserved for heap objects but is not currently used to hold heap objects.

/memory/classes/metadata/mcache/free:bytes

Memory that is reserved for runtime mcache structures, but not in-use.

/memory/classes/metadata/mcache/inuse:bytes

Memory that is occupied by runtime mcache structures that are currently being used.

/memory/classes/metadata/mspan/free:bytes

Memory that is reserved for runtime mspan structures, but not in-use.

/memory/classes/metadata/mspan/inuse:bytes

Memory that is occupied by runtime mspan structures that are

currently being used.

`/memory/classes/metadata/other:bytes`

Memory that is reserved for or used to hold runtime metadata.

`/memory/classes/os-stacks:bytes`

Stack memory allocated by the underlying operating system.

`/memory/classes/other:bytes`

Memory used by execution trace buffers, structures for debugging the runtime, finalizer and profiler specials, and more.

`/memory/classes/profiling/buckets:bytes`

Memory that is used by the stack trace hash map used for profiling.

`/memory/classes/total:bytes`

All memory mapped by the Go runtime into the current process as read-write. Note that this does not include memory mapped by code called via `cgo` or via the `syscall` package.
Sum of all metrics in `/memory/classes`.

`/sched/gomaxprocs:threads`

The current `runtime.GOMAXPROCS` setting, or the number of operating system threads that can execute user-level Go code simultaneously.

`/sched/goroutines:goroutines`

Count of live goroutines.

`/sched/latencies:seconds`

Distribution of the time goroutines have spent in the scheduler in a runnable state before actually running.

`/sync/mutex/wait/total:seconds`

Approximate cumulative time goroutines have spent blocked on a `sync.Mutex` or `sync.RWMutex`. This metric is useful for identifying global changes in lock contention. Collect a mutex or block profile using the `runtime/pprof` package for more detailed contention data.

Index

[func Read\(m \[\]Sample\)](#)

[type Description](#)

[func All\(\) \[\]Description](#)

[type Float64Histogram](#)

[type Sample](#)

[type Value](#)

[func \(v Value\) Float64\(\) float64](#)

```
func (v Value) Float64Histogram() *Float64Histogram
func (v Value) Kind() ValueKind
func (v Value) Uint64() uint64
type ValueKind
```

Examples

[Read \(ReadingAllMetrics\)](#)
[Read \(ReadingOneMetric\)](#)

Constants

This section is empty.

Variables

This section is empty.

Functions

func [Read](#)

```
func Read(m []Sample)
```

`Read` populates each `Value` field in the given slice of metric samples.

Desired metrics should be present in the slice with the appropriate name. The user of this API is encouraged to re-use the same slice between calls for efficiency, but is not required to do so.

Note that re-use has some caveats. Notably, `Values` should not be read or manipulated while a `Read` with that value is outstanding; that is a data race. This property includes pointer-typed `Values` (for example, `Float64Histogram`) whose underlying storage will be reused by `Read` when possible. To safely use such values in a concurrent setting, all data must be deep-copied.

It is safe to execute multiple `Read` calls concurrently, but their arguments must share no underlying memory. When in doubt, create a new `[]Sample` from scratch, which is always safe, though may be inefficient.

Sample values with names not appearing in `All` will have their `Value` populated as `KindBad` to indicate that the name is unknown.

► [Example \(ReadingAllMetrics\)](#)

► [Example \(ReadingOneMetric\)](#)

Types

type [Description](#)

```

type Description struct {
    // Name is the full name of the metric which includes the unit.
    //
    // The format of the metric may be described by the following regular expression.
    //
    // ^(?P<name>/[^\:]+):(?P<unit>[^\:*/]+(?:[*/*][^\:*/]+)*)$
    //
    // The format splits the name into two components, separated by a colon: a path which
    // starts with a /, and a machine-parseable unit. The name may contain any valid Unicode
    // codepoint in between / characters, but by convention will try to stick to lowercase
    // characters and hyphens. An example of such a path might be "/memory/heap/free".
    //
    // The unit is by convention a series of lowercase English unit names (singular or plural)
    // without prefixes delimited by '*' or '/'. The unit names may contain any valid Unicode
    // codepoint that is not a delimiter.
    // Examples of units might be "seconds", "bytes", "bytes/second", "cpu-seconds",
    // "byte*cpu-seconds", and "bytes/second/second".
    //
    // For histograms, multiple units may apply. For instance, the units of the buckets
    // the count. By convention, for histograms, the units of the count are always "samples"
    // with the type of sample evident by the metric's name, while the unit in the name
    // specifies the buckets' unit.
    //
    // A complete name might look like "/memory/heap/free:bytes".
    Name string

    // Description is an English language sentence describing the metric.
    Description string

    // Kind is the kind of value for this metric.
    //
    // The purpose of this field is to allow users to filter out metrics whose values are of
    // types which their application may not understand.
    Kind ValueKind

    // Cumulative is whether or not the metric is cumulative. If a cumulative metric is
    // a single number, then it increases monotonically. If the metric is a distribution
    // then each bucket count increases monotonically.
    //
    // This flag thus indicates whether or not it's useful to compute a rate from this metric.
    Cumulative bool
}

```

Description describes a runtime metric.

func All

```
func All() []Description
```

All returns a slice of containing metric descriptions for all supported metrics.

type **Float64Histogram**

```
type Float64Histogram struct {
    // Counts contains the weights for each histogram bucket.
    //
    // Given N buckets, Count[n] is the weight of the range
    // [bucket[n], bucket[n+1]), for 0 <= n < N.
    Counts []uint64

    // Buckets contains the boundaries of the histogram buckets, in increasing order.
    //
    // Buckets[0] is the inclusive lower bound of the minimum bucket while
    // Buckets[len(Buckets)-1] is the exclusive upper bound of the maximum bucket.
    // Hence, there are len(Buckets)-1 counts. Furthermore, len(Buckets) != 1, always,
    // since at least two boundaries are required to describe one bucket (and 0
    // boundaries are used to describe 0 buckets).
    //
    // Buckets[0] is permitted to have value -Inf and Buckets[len(Buckets)-1] is
    // permitted to have value Inf.
    //
    // For a given metric name, the value of Buckets is guaranteed not to change
    // between calls until program exit.
    //
    // This slice value is permitted to alias with other Float64Histograms' Buckets
    // fields, so the values within should only ever be read. If they need to be
    // modified, the user must make a copy.
    Buckets []float64
}
```

Float64Histogram represents a distribution of float64 values.

type **Sample**

```
type Sample struct {
    // Name is the name of the metric sampled.
    //
    // It must correspond to a name in one of the metric descriptions
    // returned by All.
    Name string

    // Value is the value of the metric sample.
    Value Value
}
```

Sample captures a single metric sample.

type **Value**

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

```
}
```

Value represents a metric value returned by the runtime.

func (Value) Float64

```
func (v Value) Float64() float64
```

Float64 returns the internal float64 value for the metric.

If v.Kind() != KindFloat64, this method panics.

func (Value) Float64Histogram

```
func (v Value) Float64Histogram() *Float64Histogram
```

Float64Histogram returns the internal *Float64Histogram value for the metric.

If v.Kind() != KindFloat64Histogram, this method panics.

func (Value) Kind

```
func (v Value) Kind() ValueKind
```

Kind returns the tag representing the kind of value this is.

func (Value) Uint64

```
func (v Value) Uint64() uint64
```

Uint64 returns the internal uint64 value for the metric.

If v.Kind() != KindUint64, this method panics.

type ValueKind

```
type ValueKind int
```

ValueKind is a tag for a metric Value which indicates its type.

```
const (  
    // KindBad indicates that the Value has no type and should not be used.  
    KindBad ValueKind = iota  
  
    // KindUint64 indicates that the type of the Value is a uint64.  
    KindUint64  
  
    // KindFloat64 indicates that the type of the Value is a float64.  
    KindFloat64
```

```
// KindFloat64Histogram indicates that the type of the Value is a *Float64Histogram
KindFloat64Histogram
)
```



Source Files

[View all](#) 

[description.go](#)
[doc.go](#)

[histogram.go](#)
[sample.go](#)

[value.go](#)

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