slog package - log/slog - Go Packages

Package slog provides structured logging, in which log records include a message, a severity level, and various other attributes expressed as key-value pairs.

It defines a type, Logger, which provides several methods (such as Logger.Info and Logger.Error) for reporting events of interest.

Each Logger is associated with a Handler. A Logger output method creates a Record from the method arguments and passes it to the Handler, which decides how to handle it. There is a default Logger accessible through top-level functions (such as Info and Error) that call the corresponding Logger methods.

A log record consists of a time, a level, a message, and a set of key-value pairs, where the keys are strings and the values may be of any type. As an example,

slog.Info("hello", "count", 3)

creates a record containing the time of the call, a level of Info, the message "hello", and a single pair with key "count" and value 3.

The Info top-level function calls the Logger.Info method on the default Logger. In addition to Logger.Info, there are methods for Debug, Warn and Error levels. Besides these convenience methods for common levels, there is also a Logger.Log method which takes the level as an argument. Each of these methods has a corresponding top-level function that uses the default logger.

The default handler formats the log record's message, time, level, and attributes as a string and passes it to the log package.

2022/11/08 15:28:26 INFO hello count=3

For more control over the output format, create a logger with a different handler. This statement uses New to create a new logger with a TextHandler that writes structured records in text form to standard error:

logger := slog.New(slog.NewTextHandler(os.Stderr, nil))

TextHandler output is a sequence of key=value pairs, easily and unambiguously parsed by machine. This statement:

logger.Info("hello", "count", 3)

produces this output:

time=2022-11-08T15:28:26.000-05:00 level=INFO msg=hello count=3

The package also provides JSONHandler, whose output is line-delimited JSON:

logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
logger.Info("hello", "count", 3)

produces this output:

{"time":"2022-11-08T15:28:26.000000000-05:00","level":"INFO","msg":"hello","count":3}

Both TextHandler and JSONHandler can be configured with HandlerOptions. There are options for setting the minimum level (see Levels, below), displaying the source file and line of the log call, and modifying attributes before they are logged.

Setting a logger as the default with

slog.SetDefault(logger)

will cause the top-level functions like Info to use it. SetDefault also updates the default logger used by the log package, so that existing applications that use log.Printf and related functions will send log records to the logger's handler without needing to be rewritten.

Some attributes are common to many log calls. For example, you may wish to include the URL or trace identifier of a server request with all log events arising from the request. Rather than repeat the attribute with every log call, you can use Logger.With to construct a new Logger containing the attributes:

logger2 := logger.With("url", r.URL)

The arguments to With are the same key-value pairs used in Logger.Info. The result is a new Logger with the same handler as the original, but additional attributes that will appear in the output of every call.

Levels ¶

A Level is an integer representing the importance or severity of a log event. The higher the level, the more severe the event. This package defines constants for the most common levels, but any int can be used as a level.

In an application, you may wish to log messages only at a certain level or greater. One common configuration is to log messages at Info or higher levels, suppressing debug logging until it is needed. The built-in handlers can be configured with the minimum level to output by setting [HandlerOptions.Level]. The program's `main` function typically does this. The default value is LevelInfo.

Setting the [HandlerOptions.Level] field to a Level value fixes the handler's minimum level throughout its lifetime. Setting it to a LevelVar allows the level to be varied dynamically. A LevelVar holds a Level and is safe to read or write from multiple goroutines. To vary the level dynamically for an entire program, first initialize a global LevelVar:

var programLevel = new(slog.LevelVar) // Info by default

Then use the LevelVar to construct a handler, and make it the default:

```
h := slog.NewJSONHandler(os.Stderr, &slog.HandlerOptions{Level: programLevel})
slog.SetDefault(slog.New(h))
```

Now the program can change its logging level with a single statement:

```
programLevel.Set(slog.LevelDebug)
```

Groups ¶

Attributes can be collected into groups. A group has a name that is used to qualify the names of its attributes. How this qualification is displayed depends on the handler. TextHandler separates the group and attribute names with a dot. JSONHandler treats each group as a separate JSON object, with the group name as the key.

Use Group to create a Group attribute from a name and a list of key-value pairs:

```
slog.Group("request",
    "method", r.Method,
    "url", r.URL)
```

TextHandler would display this group as

```
request.method=GET request.url=http://example.com
```

JSONHandler would display it as

```
"request":{"method":"GET","url":"http://example.com"}
```

Use Logger.WithGroup to qualify all of a Logger's output with a group name. Calling WithGroup on a Logger results in a new Logger with the same Handler as the original, but with all its attributes qualified by the group name.

This can help prevent duplicate attribute keys in large systems, where subsystems might use the same keys. Pass each subsystem a different Logger with its own group name so that potential duplicates are qualified:

```
logger := slog.Default().With("id", systemID)
parserLogger := logger.WithGroup("parser")
parseInput(input, parserLogger)
```

When parseInput logs with parserLogger, its keys will be qualified with "parser", so even if it uses the common key "id", the log line will have distinct keys.

Contexts ¶

Some handlers may wish to include information from the context.Context that is available at the call site. One

example of such information is the identifier for the current span when tracing is enabled.

The Logger.Log and Logger.LogAttrs methods take a context as a first argument, as do their corresponding top-level functions.

Although the convenience methods on Logger (Info and so on) and the corresponding top-level functions do not take a context, the alternatives ending in "Context" do. For example,

slog.InfoContext(ctx, "message")

It is recommended to pass a context to an output method if one is available.

Attrs and Values ¶

An Attr is a key-value pair. The Logger output methods accept Attrs as well as alternating keys and values. The statement

slog.Info("hello", slog.Int("count", 3))

behaves the same as

slog.Info("hello", "count", 3)

There are convenience constructors for Attr such as Int, String, and Bool for common types, as well as the function Any for constructing Attrs of any type.

The value part of an Attr is a type called Value. Like an [any], a Value can hold any Go value, but it can represent typical values, including all numbers and strings, without an allocation.

For the most efficient log output, use Logger.LogAttrs. It is similar to Logger.Log but accepts only Attrs, not alternating keys and values; this allows it, too, to avoid allocation.

The call

logger.LogAttrs(ctx, slog.LevelInfo, "hello", slog.Int("count", 3))

is the most efficient way to achieve the same output as

slog.Info("hello", "count", 3)

Customizing a type's logging behavior ¶

If a type implements the LogValuer interface, the Value returned from its LogValue method is used for logging. You can use this to control how values of the type appear in logs. For example, you can redact secret information like passwords, or gather a struct's fields in a Group. See the examples under LogValuer for details.

A LogValue method may return a Value that itself implements LogValuer. The Value.Resolve method handles

these cases carefully, avoiding infinite loops and unbounded recursion. Handler authors and others may wish to use Value.Resolve instead of calling LogValue directly.

Wrapping output methods ¶

The logger functions use reflection over the call stack to find the file name and line number of the logging call within the application. This can produce incorrect source information for functions that wrap slog. For instance, if you define this function in file mylog.go:

```
func Infof(format string, args ...any) {
    slog.Default().Info(fmt.Sprintf(format, args...))
}
```

and you call it like this in main.go:

```
Infof(slog.Default(), "hello, %s", "world")
```

then slog will report the source file as mylog.go, not main.go.

A correct implementation of Infof will obtain the source location (pc) and pass it to NewRecord. The Infof function in the package-level example called "wrapping" demonstrates how to do this.

Working with Records ¶

Sometimes a Handler will need to modify a Record before passing it on to another Handler or backend. A Record contains a mixture of simple public fields (e.g. Time, Level, Message) and hidden fields that refer to state (such as attributes) indirectly. This means that modifying a simple copy of a Record (e.g. by calling Record.Add or Record.AddAttrs to add attributes) may have unexpected effects on the original. Before modifying a Record, use Record.Clone to create a copy that shares no state with the original, or create a new Record with NewRecord and build up its Attrs by traversing the old ones with Record.Attrs.

Performance considerations ¶

If profiling your application demonstrates that logging is taking significant time, the following suggestions may help.

If many log lines have a common attribute, use Logger.With to create a Logger with that attribute. The built-in handlers will format that attribute only once, at the call to Logger.With. The Handler interface is designed to allow that optimization, and a well-written Handler should take advantage of it.

The arguments to a log call are always evaluated, even if the log event is discarded. If possible, defer computation so that it happens only if the value is actually logged. For example, consider the call

slog.Info("starting request", "url", r.URL.String()) // may compute String unnecessarily

The URL.String method will be called even if the logger discards Info-level events. Instead, pass the URL directly:

slog.Info("starting request", "url", &r.URL) // calls URL.String only if needed

The built-in TextHandler will call its String method, but only if the log event is enabled. Avoiding the call to String also preserves the structure of the underlying value. For example JSONHandler emits the components of the parsed URL as a JSON object. If you want to avoid eagerly paying the cost of the String call without causing the handler to potentially inspect the structure of the value, wrap the value in a fmt.Stringer implementation that hides its Marshal methods.

You can also use the LogValuer interface to avoid unnecessary work in disabled log calls. Say you need to log some expensive value:

slog.Debug("frobbing", "value", computeExpensiveValue(arg))

Even if this line is disabled, computeExpensiveValue will be called. To avoid that, define a type implementing LogValuer:

```
type expensive struct { arg int }

func (e expensive) LogValue() slog.Value {
    return slog.AnyValue(computeExpensiveValue(e.arg))
}
```

Then use a value of that type in log calls:

slog.Debug("frobbing", "value", expensive{arg})

Now computeExpensiveValue will only be called when the line is enabled.

The built-in handlers acquire a lock before calling io.Writer.Write to ensure that each record is written in one piece. User-defined handlers are responsible for their own locking.

Writing a handler ¶

For a guide to writing a custom handler, see https://golang.org/s/slog-handler-guide.

▸ Details

Constants

func Debug(msg string, args ...any)

func DebugContext(ctx context.Context, msg string, args ...any)

func (l *Logger) Debug(msg string, args ...any)

func (l *Logger) DebugContext(ctx context.Context, msg string, args ...any)

func (l *Logger) Enabled(ctx context.Context, level Level) bool

func (l *Logger) Error(msg string, args ...any)

func (l *Logger) ErrorContext(ctx context.Context, msg string, args ...any)

func (l *Logger) Handler() Handler

func (l *Logger) Info(msg string, args ...any)

func (l *Logger) InfoContext(ctx context.Context, msg string, args ...any)

func (l *Logger) Log(ctx context.Context, level Level, msg string, args ...any)

func (l *Logger) LogAttrs(ctx context.Context, level Level, msg string, attrs ...Attr)

func (l *Logger) Warn(msg string, args ...any)

func (l *Logger) WarnContext(ctx context.Context, msg string, args ...any)

func (l *Logger) With(args ...any) *Logger

func (l *Logger) WithGroup(name string) *Logger

type Record

func NewRecord(t time.Time, level Level, msg string, pc uintptr) Record

func (r *Record) Add(args ...any)

func (r *Record) AddAttrs(attrs ...Attr)

func (r Record) Attrs(f func(Attr) bool)

func (r Record) Clone() Record

func (r Record) NumAttrs() int

type Source

type TextHandler

func NewTextHandler(w io.Writer, opts *HandlerOptions) *TextHandler

func (h *TextHandler) Enabled(_ context.Context, level Level) bool

func (h *TextHandler) Handle(_ context.Context, r Record) error

Package (Wrapping)

Group

Handler (LevelHandler)

HandlerOptions (CustomLevels)

LogValuer (Group)

LogValuer (Secret)

View Source

```
const (

	TimeKey = "time"


	LevelKey = "level"


	MessageKey = "msg"


	SourceKey = "source"
)
```

Keys for "built-in" attributes.

This section is empty.

Debug calls Logger.Debug on the default logger.

DebugContext calls Logger.DebugContext on the default logger.

Error calls Logger.Error on the default logger.

ErrorContext calls Logger.ErrorContext on the default logger.

Info calls Logger.Info on the default logger.

InfoContext calls Logger.InfoContext on the default logger.

Log calls Logger.Log on the default logger.

LogAttrs calls Logger.LogAttrs on the default logger.

NewLogLogger returns a new log.Logger such that each call to its Output method dispatches a Record to the specified handler. The logger acts as a bridge from the older log API to newer structured logging handlers.

func SetDefault(l *Logger)

SetDefault makes l the default Logger. After this call, output from the log package's default Logger (as with log.Print, etc.) will be logged at LevelInfo using l's Handler.

Warn calls Logger.Warn on the default logger.

WarnContext calls Logger.WarnContext on the default logger.

```
type Attr struct {
    Key   string
    Value Value
}
```

An Attr is a key-value pair.

Any returns an Attr for the supplied value. See AnyValue for how values are treated.

Bool returns an Attr for a bool.

Duration returns an Attr for a time.Duration.

Float64 returns an Attr for a floating-point number.

Group returns an Attr for a Group Value. The first argument is the key; the remaining arguments are converted to Attrs as in Logger.Log.

Use Group to collect several key-value pairs under a single key on a log line, or as the result of LogValue in order to log a single value as multiple Attrs.

▸ Details

Int converts an int to an int64 and returns an Attr with that value.

Int64 returns an Attr for an int64.

String returns an Attr for a string value.

Time returns an Attr for a time.Time. It discards the monotonic portion.

Uint64 returns an Attr for a uint64.

func (a Attr) Equal(b Attr) bool

Equal reports whether a and b have equal keys and values.

type Handler ¶

A Handler handles log records produced by a Logger..

A typical handler may print log records to standard error, or write them to a file or database, or perhaps augment them with additional attributes and pass them on to another handler.

Any of the Handler's methods may be called concurrently with itself or with other methods. It is the responsibility of the Handler to manage this concurrency.

Users of the slog package should not invoke Handler methods directly. They should use the methods of Logger instead.

▶ Details

type HandlerOptions ¶

type HandlerOptions struct {

AddSource bool

Level Leveler

ReplaceAttr func(groups []string, a Attr) Attr
}

HandlerOptions are options for a TextHandler or JSONHandler. A zero HandlerOptions consists entirely of default values.

▸ Details

type JSONHandler ¶

type JSONHandler struct {

}

JSONHandler is a Handler that writes Records to an io.Writer as line-delimited JSON objects.

func NewJSONHandler ¶

NewJSONHandler creates a JSONHandler that writes to w, using the given options. If opts is nil, the default options are used.

func (*JSONHandler) Enabled ¶

Enabled reports whether the handler handles records at the given level. The handler ignores records whose level is lower.

func (*JSONHandler) Handle ¶

Handle formats its argument Record as a JSON object on a single line.

If the Record's time is zero, the time is omitted. Otherwise, the key is "time" and the value is output as with json.Marshal.

If the Record's level is zero, the level is omitted. Otherwise, the key is "level" and the value of Level.String is output.

If the AddSource option is set and source information is available, the key is "source", and the value is a record of type Source.

The message's key is "msg".

To modify these or other attributes, or remove them from the output, use [HandlerOptions.ReplaceAttr].

Values are formatted as with an encoding/json.Encoder with SetEscapeHTML(false), with two exceptions.

First, an Attr whose Value is of type error is formatted as a string, by calling its Error method. Only errors in Attrs receive this special treatment, not errors embedded in structs, slices, maps or other data structures that are processed by the encoding/json package.

Second, an encoding failure does not cause Handle to return an error. Instead, the error message is formatted as a string.

Each call to Handle results in a single serialized call to io.Writer.Write.

func (*JSONHandler) WithAttrs ¶

func (h *JSONHandler) WithAttrs(attrs []Attr) Handler

WithAttrs returns a new JSONHandler whose attributes consists of h's attributes followed by attrs.

Kind is the kind of a Value.

```
const (
    KindAny Kind = iota
    KindBool
    KindDuration
    KindFloat64
    KindInt64
    KindString
    KindTime
    KindUint64
```

```
    KindGroup
    KindLogValuer
)
```

A Level is the importance or severity of a log event. The higher the level, the more important or severe the event.

```
const (
    LevelDebug Level = -4
    LevelInfo  Level = 0
    LevelWarn  Level = 4
    LevelError Level = 8
)
```

Level numbers are inherently arbitrary, but we picked them to satisfy three constraints. Any system can map them to another numbering scheme if it wishes.

First, we wanted the default level to be Info, Since Levels are ints, Info is the default value for int, zero.

Second, we wanted to make it easy to use levels to specify logger verbosity. Since a larger level means a more severe event, a logger that accepts events with smaller (or more negative) level means a more verbose logger. Logger verbosity is thus the negation of event severity, and the default verbosity of 0 accepts all events at least as severe as INFO.

Third, we wanted some room between levels to accommodate schemes with named levels between ours. For example, Google Cloud Logging defines a Notice level between Info and Warn. Since there are only a few of these intermediate levels, the gap between the numbers need not be large. Our gap of 4 matches OpenTelemetry's mapping. Subtracting 9 from an OpenTelemetry level in the DEBUG, INFO, WARN and ERROR ranges converts it to the corresponding slog Level range. OpenTelemetry also has the names TRACE and FATAL, which slog does not. But those OpenTelemetry levels can still be represented as slog Levels by using the appropriate integers.

Names for common levels.

func (l Level) Level() Level

Level returns the receiver. It implements Leveler.

String returns a name for the level. If the level has a name, then that name in uppercase is returned. If the level is between named values, then an integer is appended to the uppercased name. Examples:

```
LevelWarn.String() => "WARN"
(LevelInfo+2).String() => "INFO+2"
```

UnmarshalJSON implements encoding/json.Unmarshaler It accepts any string produced by Level.MarshalJSON, ignoring case. It also accepts numeric offsets that would result in a different string on output. For example, "Error-8" would marshal as "INFO".

UnmarshalText implements encoding.TextUnmarshaler. It accepts any string produced by Level.MarshalText, ignoring case. It also accepts numeric offsets that would result in a different string on output. For example, "Error-8" would marshal as "INFO".

A LevelVar is a Level variable, to allow a Handler level to change dynamically. It implements Leveler as well as a Set method, and it is safe for use by multiple goroutines. The zero LevelVar corresponds to LevelInfo.

func (v *LevelVar) Level() Level

Level returns v's level.

func (v *LevelVar) Set(l Level)

Set sets v's level to l.

```
type Leveler interface {
    Level() Level
}
```

A Leveler provides a Level value.

As Level itself implements Leveler, clients typically supply a Level value wherever a Leveler is needed, such as in HandlerOptions. Clients who need to vary the level dynamically can provide a more complex Leveler implementation such as *LevelVar.

```
type LogValuer interface {
    LogValue() Value
}
```

A LogValuer is any Go value that can convert itself into a Value for logging.

This mechanism may be used to defer expensive operations until they are needed, or to expand a single value into a sequence of components.

▸ Details

▸ Details

A Logger records structured information about each call to its Log, Debug, Info, Warn, and Error methods. For each call, it creates a Record and passes it to a Handler.

To create a new Logger, call New or a Logger method that begins "With".

Default returns the default Logger.

func New(h Handler) *Logger

New creates a new Logger with the given non-nil Handler.

func With(args ...any) *Logger

With calls Logger.With on the default logger.

Debug logs at LevelDebug.

DebugContext logs at LevelDebug with the given context.

Enabled reports whether l emits log records at the given context and level.

Error logs at LevelError.

ErrorContext logs at LevelError with the given context.

func (*Logger) Handler ¶

func (l *Logger) Handler() Handler

Handler returns l's Handler.

InfoContext logs at LevelInfo with the given context.

Log emits a log record with the current time and the given level and message. The Record's Attrs consist of the Logger's attributes followed by the Attrs specified by args.

The attribute arguments are processed as follows:

If an argument is an Attr, it is used as is.

If an argument is a string and this is not the last argument, the following argument is treated as the value and the two are combined into an Attr.

Otherwise, the argument is treated as a value with key "!BADKEY".

LogAttrs is a more efficient version of Logger.Log that accepts only Attrs.

WarnContext logs at LevelWarn with the given context.

func (l *Logger) With(args ...any) *Logger

With returns a Logger that includes the given attributes in each output operation. Arguments are converted to attributes as if by Logger.Log.

WithGroup returns a Logger that starts a group, if name is non-empty. The keys of all attributes added to the Logger will be qualified by the given name. (How that qualification happens depends on the

[Handler.WithGroup] method of the Logger's Handler.)

If name is empty, WithGroup returns the receiver.

A Record holds information about a log event. Copies of a Record share state. Do not modify a Record after handing out a copy to it. Call NewRecord to create a new Record. Use Record.Clone to create a copy with no shared state.

NewRecord creates a Record from the given arguments. Use Record.AddAttrs to add attributes to the Record.

NewRecord is intended for logging APIs that want to support a Handler as a backend.

func (r *Record) Add(args ...any)

Add converts the args to Attrs as described in Logger.Log, then appends the Attrs to the Record's list of Attrs. It omits empty groups.

func (r *Record) AddAttrs(attrs ...Attr)

AddAttrs appends the given Attrs to the Record's list of Attrs. It omits empty groups.

func (r Record) Attrs(f func(Attr) bool)

Attrs calls f on each Attr in the Record. Iteration stops if f returns false.

func (r Record) Clone() Record

Clone returns a copy of the record with no shared state. The original record and the clone can both be modified without interfering with each other.

func (r Record) NumAttrs() int

NumAttrs returns the number of attributes in the Record.

type Source struct {

    Function string `json:"function"`

    File string `json:"file"`
    Line int   `json:"line"`
}

Source describes the location of a line of source code.

type TextHandler ¶

type TextHandler struct {

}

TextHandler is a Handler that writes Records to an io.Writer as a sequence of key=value pairs separated by spaces and followed by a newline.

func NewTextHandler ¶

NewTextHandler creates a TextHandler that writes to w, using the given options. If opts is nil, the default options are used.

func (*TextHandler) Enabled ¶

Enabled reports whether the handler handles records at the given level. The handler ignores records whose level is lower.

func (*TextHandler) Handle ¶

Handle formats its argument Record as a single line of space-separated key=value items.

If the Record's time is zero, the time is omitted. Otherwise, the key is "time" and the value is output in RFC3339 format with millisecond precision.

If the Record's level is zero, the level is omitted. Otherwise, the key is "level" and the value of Level.String is output.

If the AddSource option is set and source information is available, the key is "source" and the value is output as FILE:LINE.

The message's key is "msg".

To modify these or other attributes, or remove them from the output, use [HandlerOptions.ReplaceAttr].

If a value implements encoding.TextMarshaler, the result of MarshalText is written. Otherwise, the result of fmt.Sprint is written.

Keys and values are quoted with strconv.Quote if they contain Unicode space characters, non-printing characters, "" or '='.

Keys inside groups consist of components (keys or group names) separated by dots. No further escaping is performed. Thus there is no way to determine from the key "a.b.c" whether there are two groups "a" and "b" and

a key "c", or a single group "a.b" and a key "c", or single group "a" and a key "b.c". If it is necessary to reconstruct the group structure of a key even in the presence of dots inside components, use [HandlerOptions.ReplaceAttr] to encode that information in the key.

Each call to Handle results in a single serialized call to io.Writer.Write.

func (*TextHandler) WithAttrs ¶

func (h *TextHandler) WithAttrs(attrs []Attr) Handler

WithAttrs returns a new TextHandler whose attributes consists of h's attributes followed by attrs.

A Value can represent any Go value, but unlike type any, it can represent most small values without an allocation. The zero Value corresponds to nil.

func AnyValue(v any) Value

AnyValue returns a Value for the supplied value.

If the supplied value is of type Value, it is returned unmodified.

Given a value of one of Go's predeclared string, bool, or (non-complex) numeric types, AnyValue returns a Value of kind String, Bool, Uint64, Int64, or Float64. The width of the original numeric type is not preserved.

Given a time.Time or time.Duration value, AnyValue returns a Value of kind KindTime or KindDuration. The monotonic time is not preserved.

For nil, or values of all other types, including named types whose underlying type is numeric, AnyValue returns a value of kind KindAny.

func BoolValue(v bool) Value

BoolValue returns a Value for a bool.

DurationValue returns a Value for a time.Duration.

Float64Value returns a Value for a floating-point number.

func GroupValue(as ...Attr) Value

GroupValue returns a new Value for a list of Attrs. The caller must not subsequently mutate the argument slice.

Int64Value returns a Value for an int64.

func IntValue(v int) Value

IntValue returns a Value for an int.

StringValue returns a new Value for a string.

TimeValue returns a Value for a time.Time. It discards the monotonic portion.

Uint64Value returns a Value for a uint64.

Any returns v's value as an any.

Bool returns v's value as a bool. It panics if v is not a bool.

Duration returns v's value as a time.Duration. It panics if v is not a time.Duration.

Equal reports whether v and w represent the same Go value.

Float64 returns v's value as a float64. It panics if v is not a float64.

func (v Value) Group() []Attr

Group returns v's value as a []Attr. It panics if v's Kind is not KindGroup.

Int64 returns v's value as an int64. It panics if v is not a signed integer.

func (v Value) Kind() Kind

Kind returns v's Kind.

func (v Value) LogValuer() LogValuer

LogValuer returns v's value as a LogValuer. It panics if v is not a LogValuer.

func (v Value) Resolve() (rv Value)

Resolve repeatedly calls LogValue on v while it implements LogValuer, and returns the result. If v resolves to a group, the group's attributes' values are not recursively resolved. If the number of LogValue calls exceeds a threshold, a Value containing an error is returned. Resolve's return value is guaranteed not to be of Kind KindLogValuer.

String returns Value's value as a string, formatted like fmt.Sprint. Unlike the methods Int64, Float64, and so on, which panic if v is of the wrong kind, String never panics.

Time returns v's value as a time.Time. It panics if v is not a time.Time.

Uint64 returns v's value as a uint64. It panics if v is not an unsigned integer.