Structured Logging with slog - The Go Programming Language

Jonathan Amsterdam 22 August 2023

The Go Blog

The new `log/slog` package in Go 1.21 brings structured logging to the standard library. Structured logs use key-value pairs so they can be parsed, filtered, searched, and analyzed quickly and reliably. For servers, logging is an important way for developers to observe the detailed behavior of the system, and often the first place they go to debug it. Logs therefore tend to be voluminous, and the ability to search and filter them quickly is essential.

The standard library has had a logging package, `log`, since Go's initial release over a decade ago. Over time, we've learned that structured logging is important to Go programmers. It has consistently ranked high in our annual survey, and many packages in the Go ecosystem provide it. Some of these are quite popular: one of the first structured logging packages for Go, logrus, is used in over 100,000 other packages.

With many structured logging packages to choose from, large programs will often end up including more than one through their dependencies. The main program might have to configure each of these logging packages so that the log output is consistent: it all goes to the same place, in the same format. By including structured logging in the standard library, we can provide a common framework that all the other structured logging packages can share.

A tour of `slog`

Here is the simplest program that uses `slog`:

```
package main

import "log/slog"

func main() {
    slog.Info("hello, world")
}
```

As of this writing, it prints:

```
2023/08/04 16:09:19 INFO hello, world
```

The `Info` function prints a message at the Info log level using the default logger, which in this case is the default logger from the `log` package—the same logger you get when you write `log.Printf`. That explains why the output looks so similar: only the "INFO" is new. Out of the box, `slog` and the original `log` package work together to make it easy to get started.

Besides `Info`, there are functions for three other levels—`Debug`, `Warn`, and `Error`—as well as a more general `Log` function that takes the level as an argument. In `slog`, levels are just integers, so you aren't limited to the four named levels. For example, `Info` is zero and `Warn` is 4, so if your logging system has a level in between those, you can use 2 for it.

Unlike with the `log` package, we can easily add key-value pairs to our output by writing them after the message:

```
slog.Info("hello, world", "user", os.Getenv("USER"))
```

The output now looks like this:

```
2023/08/04 16:27:19 INFO hello, world user=jba
```

As we mentioned, `slog`'s top-level functions use the default logger. We can get this logger explicitly, and call its methods:

```
logger := slog.Default()
logger.Info("hello, world", "user", os.Getenv("USER"))
```

Every top-level function corresponds to a method on a `slog.Logger`. The output is the same as before.

Initially, slog's output goes through the default `log.Logger`, producing the output we've seen above. We can change the output by changing the *handler* used by the logger. `slog` comes with two built-in handlers. A `TextHandler` emits all log information in the form `key=value`. This program creates a new logger using a `TextHandler` and makes the same call to the `Info` method:

```
logger := slog.New(slog.NewTextHandler(os.Stdout, nil))
logger.Info("hello, world", "user", os.Getenv("USER"))
```

Now the output looks like this:

```
time=2023-08-04T16:56:03.786-04:00 level=INFO msg="hello, world" user=jba
```

Everything has been turned into a key-value pair, with strings quoted as needed to preserve structure.

For JSON output, install the built-in `JSONHandler` instead:

```
logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
logger.Info("hello, world", "user", os.Getenv("USER"))
```

Now our output is a sequence of JSON objects, one per logging call:

```
{"time":"2023-08-04T16:58:02.939245411-04:00","level":"INFO","msg":"hello,
world","user":"jba"}
```

You are not limited to the built-in handlers. Anyone can write a handler by implementing the `slog.Handler` interface. A handler can generate output in a particular format, or it can wrap another handler to add functionality. One of the examples in the `slog` documentation shows how to write a wrapping handler that changes the minimum level at which log messages will be displayed.

The alternating key-value syntax for attributes that we've been using so far is convenient, but for frequently executed log statements it may be more efficient to use the `Attr` type and call the `LogAttrs` method. These work together to minimize memory allocations. There are functions for building `Attrs` out of strings, numbers, and other common types. This call to `LogAttrs` produces the same output as above, but does it faster:

```
slog.LogAttrs(context.Background(), slog.LevelInfo, "hello, world",
    slog.String("user", os.Getenv("USER")))
```

There is a lot more to `slog`:

As the call to `LogAttrs` shows, you can pass a `context.Context` to some log functions so a handler can extract context information like trace IDs. (Canceling the context does not prevent the log entry from being written.)

You can call `Logger.With` to add attributes to a logger that will appear in all of its output, effectively factoring out the common parts of several log statements. This is not only convenient, but it can also help performance, as discussed below.

Attributes can be combined into groups. This can add more structure to your log output and can help to disambiguate keys that would otherwise be identical.

You can control how a value appears in the logs by providing its type with a `LogValue` method. That can be used to log the fields of a struct as a group or redact sensitive data, among other things.

The best place to learn about all of `slog` is the package documentation.

Performance

We wanted `slog` to be fast. For large-scale performance gains, we designed the `Handler` interface to provide optimization opportunities. The `Enabled` method is called at the beginning of every log event, giving the handler a chance to drop unwanted log events quickly. The `WithAttrs` and `WithGroup` methods let the handler format attributes added by `Logger.With` once, rather than at each logging call. This pre-formatting can provide a significant speedup when large attributes, like an `http.Request`, are added to a `Logger` and then used in many logging calls.

To inform our performance optimization work, we investigated typical patterns of logging in existing open-source projects. We found that over 95% of calls to logging methods pass five or fewer attributes. We also categorized the types of attributes, finding that a handful of common types accounted for the majority. We then wrote benchmarks that captured the common cases, and used them as a guide to see where the time went. The greatest gains came from paying careful attention to memory allocation.

The design process

The `slog` package is one of the largest additions to the standard library since Go 1 was released in 2012. We wanted to take our time designing it, and we knew that community feedback would be essential.

By April 2022, we had gathered enough data to demonstrate the importance of structured logging to the Go community. The Go team decided to explore adding it to the standard library.

We began by looking at how the existing structured logging packages were designed. We also took advantage of the large collection of open-source Go code stored on the Go module proxy to learn how these packages were actually used. Our first design was informed by this research as well as Go's spirit of simplicity. We wanted an API that is light on the page and easy to understand, without sacrificing performance.

It was never a goal to replace existing third-party logging packages. They are all good at what they do, and replacing existing code that works well is rarely a good use of a developer's time. We divided the API into a frontend, `Logger`, that calls a backend interface, `Handler`. That way, existing logging packages can talk to a common backend, so the packages that use them can interoperate without having to be rewritten. Handlers are written or in progress for many common logging packages, including Zap, logr and hclog.

We shared our initial design within the Go team and other developers who had extensive logging experience. We made alterations based on their feedback, and by August of 2022 we felt we had a workable design. On August 29, we made our experimental implementation public and began a GitHub discussion to hear what the community had to say. The response was enthusiastic and largely positive. Thanks to insightful comments from the designers and users of other structured logging packages, we made several changes and added a few features, like groups and the `LogValuer` interface. We changed the mapping from log levels to integers twice.

After two months and about 300 comments, we felt we were ready for an actual proposal and accompanying design doc. The proposal issue garnered over 800 comments and resulted in many improvements to the API and the implementation. Here are two examples of API changes, both concerning `context.Context`:

Originally the API supported adding loggers to a context. Many felt that this was a convenient way to plumb a logger easily through levels of code that didn't care about it. But others felt it was smuggling in an implicit dependency, making the code harder to understand. Ultimately, we removed the feature as being too controversial.

We also wrestled with the related question of passing a context to logging methods, trying a number of designs.

We initially resisted the standard pattern of passing the context as the first argument because we didn't want every logging call to require a context, but ultimately created two sets of logging methods, one with a context and one without.

One change we did not make concerned the alternating key-and-value syntax for expressing attributes:

```
slog.Info("message", "k1", v1, "k2", v2)
```

Many felt strongly that this was a bad idea. They found it hard to read and easy to get wrong by omitting a key or value. They preferred explicit attributes for expressing structure:

```
slog.Info("message", slog.Int("k1", v1), slog.String("k2", v2))
```

But we felt that the lighter syntax was important to keeping Go easy and fun to use, especially for new Go programmers. We also knew that several Go logging packages, like `logr`, `go-kit/log` and `zap` (with its `SugaredLogger`) successfully used alternating keys and values. We added a vet check to catch common mistakes, but did not change the design.

On March 15, 2023, the proposal was accepted, but there were still some minor unresolved issues. Over the next few weeks, ten additional changes were proposed and resolved. By early July, the `log/slog` package implementation was complete, along with the `testing/slogtest` package for verifying handlers and the vet check for correct usage of alternating keys and values.

And on August 8, Go 1.21 was released, and `slog` with it. We hope you find it useful, and as fun to use as it was to build.

And a big thanks to everyone who participated in the discussion and the proposal process. Your contributions improved `slog` immensely.

Resources

The documentation for the `log/slog` package explains how to use it and provides several examples.

The wiki page has additional resources provided by the Go community, including a variety of handlers.

If you want to write a handler, consult the handler writing guide.