A minimal logging API for Go

GO reference

logr offers an(other) opinion on how Go programs and libraries can do logging without becoming coupled to a particular logging implementation. This is not an implementation of logging - it is an API. In fact it is two APIs with two different sets of users.

The `Logger` type is intended for application and library authors. It provides a relatively small API which can be used everywhere you want to emit logs. It defers the actual act of writing logs (to files, to stdout, or whatever) to the `LogSink` interface.

The `LogSink` interface is intended for logging library implementers. It is a pure interface which can be implemented by logging frameworks to provide the actual logging functionality.

This decoupling allows application and library developers to write code in terms of `logr.Logger` (which has very low dependency fan-out) while the implementation of logging is managed "up stack" (e.g. in or near `main()`.) Application developers can then switch out implementations as necessary.

Many people assert that libraries should not be logging, and as such efforts like this are pointless. Those people are welcome to convince the authors of the tens-of-thousands of libraries that *DO* write logs that they are all wrong. In the meantime, logr takes a more practical approach.

Typical usage

Somewhere, early in an application's life, it will make a decision about which logging library (implementation) it actually wants to use. Something like:

```
func main() {
    // ... other setup code ...

    // Create the "root" logger.  We have chosen the "logimpl" implementation,
    // which takes some initial parameters and returns a logr.Logger.
    logger := logimpl.New(param1, param2)

    // ... other setup code ...
```

Most apps will call into other libraries, create structures to govern the flow, etc. The `logr.Logger` object can be passed to these other libraries, stored in structs, or even used as a package-global variable, if needed. For example:

```
app := createTheAppObject(logger)
app.Run()
```

Outside of this early setup, no other packages need to know about the choice of implementation. They write logs in terms of the `logr.Logger` that they received:

```
type appObject struct {
    // ... other fields ...
    logger logr.Logger
    // ... other fields ...
}

func (app *appObject) Run() {
    app.logger.Info("starting up", "timestamp", time.Now())

    // ... app code ...
```

Background

If the Go standard library had defined an interface for logging, this project probably would not be needed. Alas, here we are.

Inspiration

Before you consider this package, please read this blog post by the inimitable Dave Cheney. We really appreciate what he has to say, and it largely aligns with our own experiences.

Differences from Dave's ideas

The main differences are:

1. Dave basically proposes doing away with the notion of a logging API in favor of `fmt.Printf()`. We disagree, especially when you consider things like output locations, timestamps, file and line decorations, and structured logging. This package restricts the logging API to just 2 types of logs: info and error.

Info logs are things you want to tell the user which are not errors. Error logs are, well, errors. If your code receives an `error` from a subordinate function call and is logging that `error` *and not returning it*, use error logs.

2. Verbosity-levels on info logs. This gives developers a chance to indicate arbitrary grades of importance for info logs, without assigning names with semantic meaning such as "warning", "trace", and "debug." Superficially this may feel very similar, but the primary difference is the lack of semantics. Because verbosity is a numerical value, it's safe to assume that an app running with higher verbosity means more (and less important) logs will be generated.

Implementations (non-exhaustive)

There are implementations for the following logging libraries:

- **a function** (can bridge to non-structured libraries): funcr
- **github.com/google/glog**: glogr
- **k8s.io/klog** (for Kubernetes): klogr
- **go.uber.org/zap**: zapr
- **log** (the Go standard library logger): stdr
- **github.com/sirupsen/logrus**: logrusr
- **github.com/wojas/genericr**: genericr (makes it easy to implement your own backend)
- **logfmt** (Heroku style logging): logfmtr
- **github.com/rs/zerolog**: zerologr

FAQ

Conceptual

Why structured logging?

- **Structured logs are more easily queryable**: Since you've got key-value pairs, it's much easier to query your structured logs for particular values by filtering on the contents of a particular key -- think searching request logs for error codes, Kubernetes reconcilers for the name and namespace of the reconciled object, etc.
- **Structured logging makes it easier to have cross-referenceable logs**: Similarly to searchability, if you maintain conventions around your keys, it becomes easy to gather all log lines related to a particular concept.
- **Structured logs allow better dimensions of filtering**: if you have structure to your logs, you've got more precise control over how much information is logged -- you might choose in a particular configuration to log certain keys but not others, only log lines where a certain key matches a certain value, etc., instead of just having v-levels and names to key off of.
- **Structured logs better represent structured data**: sometimes, the data that you want to log is inherently structured (think tuple-link objects.) Structured logs allow you to preserve that structure when outputting.

Why V-levels?

V-levels give operators an easy way to control the chattiness of log operations. V-levels provide a way for a given package to distinguish the relative importance or verbosity of a given log message. Then, if a particular logger or package is logging too many messages, the user of the package can simply change the v-levels for that library.

Why not named levels, like Info/Warning/Error?

Read Dave Cheney's post. Then read Differences from Dave's ideas.

Why not allow format strings, too?

Format strings negate many of the benefits of structured logs:

- They're not easily searchable without resorting to fuzzy searching, regular expressions, etc.
- They don't store structured data well, since contents are flattened into a string.
- They're not cross-referenceable.
- They don't compress easily, since the message is not constant.

(Unless you turn positional parameters into key-value pairs with numerical keys, at which point you've gotten key-value logging with meaningless keys.)

Practical

Why key-value pairs, and not a map?

Key-value pairs are *much* easier to optimize, especially around allocations. Zap (a structured logger that inspired logr's interface) has performance measurements that show this quite nicely.

While the interface ends up being a little less obvious, you get potentially better performance, plus avoid making users type `map[string]string{}` every time they want to log.

What if my V-levels differ between libraries?

That's fine. Control your V-levels on a per-logger basis, and use the `WithName` method to pass different loggers to different libraries.

Generally, you should take care to ensure that you have relatively consistent V-levels within a given logger, however, as this makes deciding on what verbosity of logs to request easier.

But I really want to use a format string!

That's not actually a question. Assuming your question is "how do I convert my mental model of logging with format strings to logging with constant messages":

1. Figure out what the error actually is, as you'd write in a TL;DR style, and use that as a message.
2. For every place you'd write a format specifier, look to the word before it, and add that as a key value pair.

For instance, consider the following examples (all taken from spots in the Kubernetes codebase):

- `klog.V(4).Infof("Client is returning errors: code %v, error %v", responseCode, err)` becomes `logger.Error(err, "client returned an error", "code", responseCode)`
- `klog.V(4).Infof("Got a Retry-After %ds response for attempt %d to %v", seconds, retries, url)` becomes `logger.V(4).Info("got a retry-after response when requesting url", "attempt", retries, "after seconds", seconds, "url", url)`

If you *really* must use a format string, use it in a key's value, and call `fmt.Sprintf` yourself. For instance: `log.Printf("unable to reflect over type %T")` becomes `logger.Info("unable to reflect over type", "type", fmt.Sprintf("%T"))`. In general though, the cases where this is necessary should be few and far between.

How do I choose my V-levels?

This is basically the only hard constraint: increase V-levels to denote more verbose or more debug-y logs.

Otherwise, you can start out with `0` as "you always want to see this", `1` as "common logging that you might *possibly* want to turn off", and `10` as "I would like to performance-test your log collection stack."

Then gradually choose levels in between as you need them, working your way down from 10 (for debug and trace style logs) and up from 1 (for chattier info-type logs.)

How do I choose my keys?

Keys are fairly flexible, and can hold more or less any string value. For best compatibility with implementations and consistency with existing code in other projects, there are a few conventions you should consider.

- Make your keys human-readable.
- Constant keys are generally a good idea.
- Be consistent across your codebase.
- Keys should naturally match parts of the message string.
- Use lower case for simple keys and lowerCamelCase for more complex ones. Kubernetes is one example of a project that has adopted that convention.

While key names are mostly unrestricted (and spaces are acceptable), it's generally a good idea to stick to printable ascii characters, or at least match the general character set of your log lines.

Why should keys be constant values?

The point of structured logging is to make later log processing easier. Your keys are, effectively, the schema of each log message. If you use different keys across instances of the same log line, you will make your structured logs much harder to use. `Sprintf()` is for values, not for keys!

Why is this not a pure interface?

The Logger type is implemented as a struct in order to allow the Go compiler to optimize things like high-V `Info` logs that are not triggered. Not all of these implementations are implemented yet, but this structure was suggested as a way to ensure they *can* be implemented. All of the real work is behind the `LogSink` interface.