Logrus

Logrus is a structured logger for Go (golang), completely API compatible with the standard library logger.

Logrus is in maintenance-mode. We will not be introducing new features. It's simply too hard to do in a way that won't break many people's projects, which is the last thing you want from your Logging library (again...).

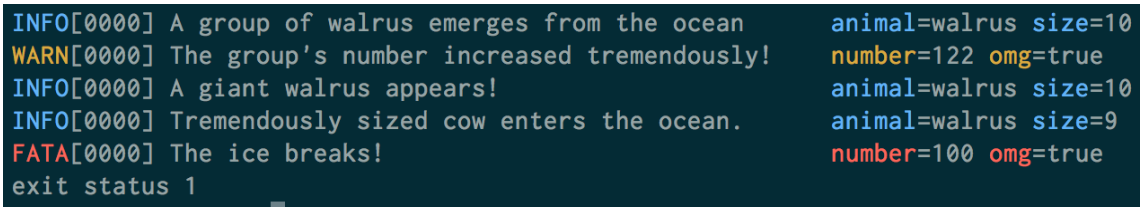
This does not mean Logrus is dead. Logrus will continue to be maintained for security, (backwards compatible) bug fixes, and performance (where we are limited by the interface).

I believe Logrus' biggest contribution is to have played a part in today's widespread use of structured logging in Golang. There doesn't seem to be a reason to do a major, breaking iteration into Logrus V2, since the fantastic Go community has built those independently. Many fantastic alternatives have sprung up. Logrus would look like those, had it been re-designed with what we know about structured logging in Go today. Check out, for example, Zerolog, Zap, and Apex.

Seeing weird case-sensitive problems? It's in the past been possible to import Logrus as both upper- and lower-case. Due to the Go package environment, this caused issues in the community and we needed a standard. Some environments experienced problems with the upper-case variant, so the lower-case was decided. Everything using `logrus` will need to use the lower-case: `github.com/sirupsen/logrus`. Any package that isn't, should be changed.

To fix Glide, see these comments. For an in-depth explanation of the casing issue, see this comment.

Nicely color-coded in development (when a TTY is attached, otherwise just plain text):



With `log.SetFormatter(&log.JSONFormatter{})`, for easy parsing by logstash or Splunk:

```
{"animal":"walrus","level":"info","msg":"A group of walrus emerges from the
ocean","size":10,"time":"2014-03-10 19:57:38.562264131 -0400 EDT"}

{"level":"warning","msg":"The group's number increased tremendously!",
"number":122,"omg":true,"time":"2014-03-10 19:57:38.562471297 -0400 EDT"}

{"animal":"walrus","level":"info","msg":"A giant walrus appears!",
"size":10,"time":"2014-03-10 19:57:38.562500591 -0400 EDT"}

{"animal":"walrus","level":"info","msg":"Tremendously sized cow enters the ocean.",
"size":9,"time":"2014-03-10 19:57:38.562527896 -0400 EDT"}
```

```
{"level":"fatal","msg":"The ice breaks!","number":100,"omg":true,
"time":"2014-03-10 19:57:38.562543128 -0400 EDT"}
```

With the default `log.SetFormatter(&log.TextFormatter{})` when a TTY is not attached, the output is compatible with the logfmt format:

```
time="2015-03-26T01:27:38-04:00" level=debug msg="Started observing beach"
animal=walrus number=8
time="2015-03-26T01:27:38-04:00" level=info msg="A group of walrus emerges from the
ocean" animal=walrus size=10
time="2015-03-26T01:27:38-04:00" level=warning msg="The group's number increased
tremendously!" number=122 omg=true
time="2015-03-26T01:27:38-04:00" level=debug msg="Temperature changes"
temperature=-4
time="2015-03-26T01:27:38-04:00" level=panic msg="It's over 9000!" animal=orca
size=9009
time="2015-03-26T01:27:38-04:00" level=fatal msg="The ice breaks!" err=&{0x2082280c0
map[animal:orca size:9009] 2015-03-26 01:27:38.441574009 -0400 EDT panic It's over
9000!} number=100 omg=true
```

To ensure this behaviour even if a TTY is attached, set your formatter as follows:

```
    log.SetFormatter(&log.TextFormatter{
        DisableColors: true,
        FullTimestamp: true,
    })
```

Logging Method Name

If you wish to add the calling method as a field, instruct the logger via:

```
log.SetReportCaller(true)
```

This adds the caller as 'method' like so:

```
{"animal":"penguin","level":"fatal","method":"github.com/sirupsen/arcticcreatures.migra
 penguin swims by",
"time":"2014-03-10 19:57:38.562543129 -0400 EDT"}
```

```
time="2015-03-26T01:27:38-04:00" level=fatal
method=github.com/sirupsen/arcticcreatures.migrate msg="a penguin swims by"
animal=penguin
```

Note that this does add measurable overhead - the cost will depend on the version of Go, but is between 20 and 40% in recent tests with 1.6 and 1.7. You can validate this in your environment via benchmarks:

```
go test -bench=.*CallerTracing
```

Case-sensitivity

The organization's name was changed to lower-case--and this will not be changed back. If you are getting import conflicts due to case sensitivity, please use the lower-case import: `github.com/sirupsen/logrus`.

Example

The simplest way to use Logrus is simply the package-level exported logger:

```
package main

import (
  log "github.com/sirupsen/logrus"
)

func main() {
  log.WithFields(log.Fields{
    "animal": "walrus",
  }).Info("A walrus appears")
}
```

Note that it's completely api-compatible with the stdlib logger, so you can replace your `log` imports everywhere with `log "github.com/sirupsen/logrus"` and you'll now have the flexibility of Logrus. You can customize it all you want:

```
package main

import (
  "os"
  log "github.com/sirupsen/logrus"
)

func init() {
  // Log as JSON instead of the default ASCII formatter.
  log.SetFormatter(&log.JSONFormatter{})

  // Output to stdout instead of the default stderr
  // Can be any io.Writer, see below for File example
  log.SetOutput(os.Stdout)

  // Only log the warning severity or above.
  log.SetLevel(log.WarnLevel)
}

func main() {
  log.WithFields(log.Fields{
    "animal": "walrus",
    "size":   10,
  }).Info("A group of walrus emerges from the ocean")

  log.WithFields(log.Fields{
    "omg":    true,
```

```
    "number": 122,
  }).Warn("The group's number increased tremendously!")

  log.WithFields(log.Fields{
    "omg":    true,
    "number": 100,
  }).Fatal("The ice breaks!")

  // A common pattern is to re-use fields between logging statements by re-using
  // the logrus.Entry returned from WithFields()
  contextLogger := log.WithFields(log.Fields{
    "common": "this is a common field",
    "other": "I also should be logged always",
  })

  contextLogger.Info("I'll be logged with common and other field")
  contextLogger.Info("Me too")
}
```

For more advanced usage such as logging to multiple locations from the same application, you can also create an instance of the `logrus` Logger:

```
package main

import (
  "os"
  "github.com/sirupsen/logrus"
)

// Create a new instance of the logger. You can have any number of instances.
var log = logrus.New()

func main() {
  // The API for setting attributes is a little different than the package level
  // exported logger. See Godoc.
  log.Out = os.Stdout

  // You could set this to any `io.Writer` such as a file
  // file, err := os.OpenFile("logrus.log", os.O_CREATE|os.O_WRONLY|os.O_APPEND,
0666)
  // if err == nil {
  //  log.Out = file
  // } else {
  //  log.Info("Failed to log to file, using default stderr")
  // }

  log.WithFields(logrus.Fields{
    "animal": "walrus",
    "size":   10,
  }).Info("A group of walrus emerges from the ocean")
}
```

Fields

Logrus encourages careful, structured logging through logging fields instead of long, unparseable error messages. For example, instead of: `log.Fatalf("Failed to send event %s to topic %s with key %d")`, you should log the much more discoverable:

```
log.WithFields(log.Fields{
  "event": event,
  "topic": topic,
  "key": key,
}).Fatal("Failed to send event")
```

We've found this API forces you to think about logging in a way that produces much more useful logging messages. We've been in countless situations where just a single added field to a log statement that was already there would've saved us hours. The `WithFields` call is optional.

In general, with Logrus using any of the `printf`-family functions should be seen as a hint you should add a field, however, you can still use the `printf`-family functions with Logrus.

Default Fields

Often it's helpful to have fields *always* attached to log statements in an application or parts of one. For example, you may want to always log the `request_id` and `user_ip` in the context of a request. Instead of writing `log.WithFields(log.Fields{"request_id": request_id, "user_ip": user_ip})` on every line, you can create a `logrus.Entry` to pass around instead:

```
requestLogger := log.WithFields(log.Fields{"request_id": request_id, "user_ip":
user_ip})
requestLogger.Info("something happened on that request") # will log request_id and
user_ip
requestLogger.Warn("something not great happened")
```

Hooks

You can add hooks for logging levels. For example to send errors to an exception tracking service on `Error`, `Fatal` and `Panic`, info to StatsD or log to multiple places simultaneously, e.g. syslog.

Logrus comes with built-in hooks. Add those, or your custom hook, in `init`:

```
import (
  log "github.com/sirupsen/logrus"
  "gopkg.in/gemnasium/logrus-airbrake-hook.v2" // the package is named "airbrake"
  logrus_syslog "github.com/sirupsen/logrus/hooks/syslog"
  "log/syslog"
)

func init() {

  // Use the Airbrake hook to report errors that have Error severity or above to
  // an exception tracker. You can create custom hooks, see the Hooks section.
  log.AddHook(airbrake.NewHook(123, "xyz", "production"))
```

```
  hook, err := logrus_syslog.NewSyslogHook("udp", "localhost:514", syslog.LOG_INFO,
"")
  if err != nil {
    log.Error("Unable to connect to local syslog daemon")
  } else {
    log.AddHook(hook)
  }
}
```

Note: Syslog hook also support connecting to local syslog (Ex. "/dev/log" or "/var/run/syslog" or "/var/run/log"). For the detail, please check the syslog hook README.

A list of currently known service hooks can be found in this wiki page

Level logging

Logrus has seven logging levels: Trace, Debug, Info, Warning, Error, Fatal and Panic.

```
log.Trace("Something very low level.")
log.Debug("Useful debugging information.")
log.Info("Something noteworthy happened!")
log.Warn("You should probably take a look at this.")
log.Error("Something failed but I'm not quitting.")
// Calls os.Exit(1) after logging
log.Fatal("Bye.")
// Calls panic() after logging
log.Panic("I'm bailing.")
```

You can set the logging level on a `Logger`, then it will only log entries with that severity or anything above it:

```
// Will log anything that is info or above (warn, error, fatal, panic). Default.
log.SetLevel(log.InfoLevel)
```

It may be useful to set `log.Level = logrus.DebugLevel` in a debug or verbose environment if your application has that.

Entries

Besides the fields added with `WithField` or `WithFields` some fields are automatically added to all logging events:

1. `time`. The timestamp when the entry was created.
2. `msg`. The logging message passed to `{Info,Warn,Error,Fatal,Panic}` after the `AddFields` call. E.g. `Failed to send event.`
3. `level`. The logging level. E.g. `info`.

Environments

Logrus has no notion of environment.

If you wish for hooks and formatters to only be used in specific environments, you should handle that yourself. For example, if your application has a global variable `Environment`, which is a string representation of the

environment you could do:

```
import (
  log "github.com/sirupsen/logrus"
)

init() {
  // do something here to set environment depending on an environment variable
  // or command-line flag
  if Environment == "production" {
    log.SetFormatter(&log.JSONFormatter{})
  } else {
    // The TextFormatter is default, you don't actually have to do this.
    log.SetFormatter(&log.TextFormatter{})
  }
}
```

This configuration is how `logrus` was intended to be used, but JSON in production is mostly only useful if you do log aggregation with tools like Splunk or Logstash.

Formatters

The built-in logging formatters are:

- `logrus.TextFormatter`. Logs the event in colors if stdout is a tty, otherwise without colors.
  - *Note:* to force colored output when there is no TTY, set the `ForceColors` field to `true`. To force no colored output even if there is a TTY set the `DisableColors` field to `true`. For Windows, see github.com/mattn/go-colorable.
  - When colors are enabled, levels are truncated to 4 characters by default. To disable truncation set the `DisableLevelTruncation` field to `true`.
  - When outputting to a TTY, it's often helpful to visually scan down a column where all the levels are the same width. Setting the `PadLevelText` field to `true` enables this behavior, by adding padding to the level text.
  - All options are listed in the generated docs.
- `logrus.JSONFormatter`. Logs fields as JSON.
  - All options are listed in the generated docs.

Third party logging formatters:

- `FluentdFormatter`. Formats entries that can be parsed by Kubernetes and Google Container Engine.
- `GELF`. Formats entries so they comply to Graylog's GELF 1.1 specification.
- `logstash`. Logs fields as Logstash Events.
- `prefixed`. Displays log entry source along with alternative layout.
- `zalgo`. Invoking the Power of Zalgo.
- `nested-logrus-formatter`. Converts logrus fields to a nested structure.
- `powerful-logrus-formatter`. get fileName, log's line number and the latest function's name when print log; Sava log to files.
- `caption-json-formatter`. logrus's message json formatter with human-readable caption added.

You can define your formatter by implementing the `Formatter` interface, requiring a `Format` method. `Format` takes an `*Entry`. `entry.Data` is a `Fields` type (`map[string]interface{}`) with all your

fields as well as the default ones (see Entries section above):

```
type MyJSONFormatter struct {
}

log.SetFormatter(new(MyJSONFormatter))

func (f *MyJSONFormatter) Format(entry *Entry) ([]byte, error) {
  // Note this doesn't include Time, Level and Message which are available on
  // the Entry. Consult `godoc` on information about those fields or read the
  // source of the official loggers.
  serialized, err := json.Marshal(entry.Data)
    if err != nil {
      return nil, fmt.Errorf("Failed to marshal fields to JSON, %w", err)
    }
  return append(serialized, '\n'), nil
}
```

Logger as an `io.Writer`

Logrus can be transformed into an `io.Writer`. That writer is the end of an `io.Pipe` and it is your responsibility to close it.

```
w := logger.Writer()
defer w.Close()

srv := http.Server{
    // create a stdlib log.Logger that writes to
    // logrus.Logger.
    ErrorLog: log.New(w, "", 0),
}
```

Each line written to that writer will be printed the usual way, using formatters and hooks. The level for those entries is `info`.

This means that we can override the standard library logger easily:

```
logger := logrus.New()
logger.Formatter = &logrus.JSONFormatter{}

// Use logrus for standard log output
// Note that `log` here references stdlib's log
// Not logrus imported under the name `log`.
log.SetOutput(logger.Writer())
```

Rotation

Log rotation is not provided with Logrus. Log rotation should be done by an external program (like `logrotate(8)`) that can compress and delete old log entries. It should not be a feature of the application-level logger.

Tools

Tool	Description
Logrus Mate	Logrus mate is a tool for Logrus to manage loggers, you can initial logger's level, hook and formatter by config file, the logger will be generated with different configs in different environments.
Logrus Viper Helper	An Helper around Logrus to wrap with spf13/Viper to load configuration with fangs! And to simplify Logrus configuration use some behavior of Logrus Mate. sample

Testing

Logrus has a built in facility for asserting the presence of log messages. This is implemented through the `test` hook and provides:

- decorators for existing logger (`test.NewLocal` and `test.NewGlobal`) which basically just adds the `test` hook
- a test logger (`test.NewNullLogger`) that just records log messages (and does not output any):

```
import(
  "github.com/sirupsen/logrus"
  "github.com/sirupsen/logrus/hooks/test"
  "github.com/stretchr/testify/assert"
  "testing"
)

func TestSomething(t*testing.T){
  logger, hook := test.NewNullLogger()
  logger.Error("Helloerror")

  assert.Equal(t, 1, len(hook.Entries))
  assert.Equal(t, logrus.ErrorLevel, hook.LastEntry().Level)
  assert.Equal(t, "Helloerror", hook.LastEntry().Message)

  hook.Reset()
  assert.Nil(t, hook.LastEntry())
}
```

Fatal handlers

Logrus can register one or more functions that will be called when any `fatal` level message is logged. The registered handlers will be executed before logrus performs an `os.Exit(1)` . This behavior may be helpful if callers need to gracefully shutdown. Unlike a `panic("Something went wrong...")` call which can be intercepted with a deferred `recover` a call to `os.Exit(1)` can not be intercepted.

```
...
handler := func() {
  // gracefully shutdown something...
}
```

```
logrus.RegisterExitHandler(handler)
...
```

Thread safety

By default, Logger is protected by a mutex for concurrent writes. The mutex is held when calling hooks and writing logs. If you are sure such locking is not needed, you can call logger.SetNoLock() to disable the locking.

Situation when locking is not needed includes:

- You have no hooks registered, or hooks calling is already thread-safe.
- Writing to logger.Out is already thread-safe, for example:
    1. logger.Out is protected by locks.
    2. logger.Out is an os.File handler opened with `O_APPEND` flag, and every write is smaller than 4k. (This allows multi-thread/multi-process writing)

       (Refer to http://www.notthewizard.com/2014/06/17/are-files-appends-really-atomic/)