



下载APP



21 | 日志处理（下）：手把手教你从 0 编写一个日志包

2021-07-13 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 16:54 大小 15.49M



你好，我是孔令飞。

上一讲我介绍了如何设计日志包，今天是实战环节，我会手把手教你从 0 编写一个日志包。

在实际开发中，我们可以选择一些优秀的开源日志包，不加修改直接拿来使用。但更多时候，是基于一个或某几个优秀的开源日志包进行二次开发。想要开发或者二次开发一个日志包，就要掌握日志包的实现方式。那么这一讲中，我来带你从 0 到 1，实现一个具备基本功能的日志包，让你从中一窥日志包的实现原理和实现方法。



在开始实战之前，我们先来看下目前业界有哪些优秀的开源日志包。

有哪些优秀的开源日志包？

在 Go 项目开发中，我们可以通过修改一些优秀的开源日志包，来实现项目的日志包。Go 生态中有很多优秀的开源日志包，例如标准库 log 包、glog、logrus、zap、seelog、zerolog、log15、apex/log、go-logging 等。其中，用得比较多的是标准库 log 包、glog、logrus 和 zap。

为了使你了解开源日志包的现状，接下来我会简单介绍下这几个常用的日志包。至于它们的具体使用方法，你可以参考我整理的一篇文章：[🔗 优秀开源日志包使用教程](#)。

标准库 log 包

标准库 log 包的功能非常简单，只提供了 Print、Panic 和 Fatal 三类函数用于日志输出。因为是标准库自带的，所以不需要我们下载安装，使用起来非常方便。

标准库 log 包只有不到 400 行的代码量，如果你想研究如何实现一个日志包，阅读标准库 log 包是一个不错的开始。Go 的标准库大量使用了 log 包，例如 net/http、net/rpc 等。

glog

[🔗 glog](#) 是 Google 推出的日志包，跟标准库 log 包一样，它是一个轻量级的日志包，使用起来简单方便。但 glog 比标准库 log 包提供了更多的功能，它具有如下特性：

支持 4 种日志级别：Info、Warning、Error、Fatal。

支持命令行选项，例如 -alsologtostderr、-log_backtrace_at、-log_dir、-logtostderr、-v 等，每个参数实现某种功能。

支持根据文件大小切割日志文件。

支持日志按级别分类输出。

支持 V level。V level 特性可以使开发者自定义日志级别。

支持 vmodule。vmodule 可以使开发者对不同的文件使用不同的日志级别。

支持 traceLocation。traceLocation 可以打印出指定位置的栈信息。

Kubernetes 项目就使用了基于 glog 封装的 klog，作为其日志库。

logrus

🔗 **logrus** 是目前 GitHub 上 star 数量最多的日志包，它的优点是功能强大、性能高效、高度灵活，还提供了自定义插件的功能。很多优秀的开源项目，例如 Docker、Prometheus 等，都使用了 logrus。除了具有日志的基本功能外，logrus 还具有如下特性：

支持常用的日志级别。logrus 支持 Debug、Info、Warn、Error、Fatal 和 Panic 这些日志级别。

可扩展。logrus 的 Hook 机制允许使用者通过 Hook 的方式，将日志分发到任意地方，例如本地文件、标准输出、Elasticsearch、Logstash、Kafka 等。

支持自定义日志格式。logrus 内置了 JSONFormatter 和 TextFormatter 两种格式。除此之外，logrus 还允许使用者通过实现 Formatter 接口，来自定义日志格式。

结构化日志记录。logrus 的 Field 机制允许使用者自定义日志字段，而不是通过冗长的消息来记录日志。

预设日志字段。logrus 的 Default Fields 机制，可以给一部分或者全部日志统一添加共同的日志字段，例如给某次 HTTP 请求的所有日志添加 X-Request-ID 字段。

Fatal handlers。logrus 允许注册一个或多个 handler，当产生 Fatal 级别的日志时调用。当我们的程序需要优雅关闭时，这个特性会非常有用。

zap

🔗 **zap** 是 uber 开源的日志包，以高性能著称，很多公司的日志包都是基于 zap 改造而来。除了具有日志基本的功能之外，zap 还具有很多强大的特性：

支持常用的日志级别，例如：Debug、Info、Warn、Error、DPanic、Panic、Fatal。

性能非常高。zap 具有非常高的性能，适合对性能要求比较高的场景。

支持针对特定的日志级别，输出调用堆栈。

像 logrus 一样，zap 也支持结构化的目录日志、预设日志字段，也因为支持 Hook 而具有可扩展性。

开源日志包选择

上面我介绍了很多日志包，每种日志包使用的场景不同，你可以根据自己的需求，结合日志包的特性进行选择：

标准库 log 包：标准库 log 包不支持日志级别、日志分割、日志格式等功能，所以在大型项目中很少直接使用，通常用于一些短小的程序，比如用于生成 JWT Token 的 main.go 文件中。标准库日志包也很适合一些简短的代码，用于快速调试和验证。

glog：glog 实现了日志包的基本功能，非常适合一些对日志功能要求不多的小型项目。

logrus：logrus 功能强大，不仅实现了日志包的基本功能，还有很多高级特性，适合一些大型项目，尤其是需要结构化日志记录的项目。

zap：zap 提供了很强大的日志功能，性能高，内存分配次数少，适合对日志性能要求很高的项目。另外，zap 包中的子包 zapcore，提供了很多底层的日志接口，适合用来做二次封装。

举个我自己选择日志包来进行二次开发的例子：我在做容器云平台开发时，发现 Kubernetes 源码中大量使用了 glog，这时就需要日志包能够兼容 glog。于是，我基于 zap 和 zapcore 封装了 github.com/marmotedu/iam/pkg/log 日志包，这个日志包可以很好地兼容 glog。

在实际项目开发中，你可以根据项目需要，从上面几个日志包中进行选择，直接使用，但更多时候，你还需要基于这些包来进行定制开发。为了使你更深入地掌握日志包的设计和开发，接下来，我会从 0 到 1 带你开发一个日志包。

从 0 编写一个日志包

接下来，我会向你展示如何快速编写一个具备基本功能的日志包，让你通过这个简短的日志包实现掌握日志包的核心设计思路。该日志包主要实现以下几个功能：

支持自定义配置。

支持文件名和行号。

支持日志级别 Debug、Info、Warn、Error、Panic、Fatal。

支持输出到本地文件和标准输出。

支持 JSON 和 TEXT 格式的日志输出，支持自定义日志格式。

支持选项模式。

日志包名称为 `cuslog`，示例项目完整代码存放在 [🔗 cuslog](#)。


具体实现分为以下四个步骤：

1. 定义：定义日志级别和日志选项。
2. 创建：创建 `Logger` 及各级别日志打印方法。
3. 写入：将日志输出到支持的输出中。
4. 自定义：自定义日志输出格式。

定义日志级别和日志选项

一个基本的日志包，首先需要定义好日志级别和日志选项。本示例将定义代码保存在 [🔗 options.go](#) 文件中。

可以通过如下方式定义日志级别：

 复制代码

```
1 type Level uint8
2
3 const (
4     DebugLevel Level = iota
5     InfoLevel
6     WarnLevel
7     ErrorLevel
8     PanicLevel
9     FatalLevel
10 )
11
12 var LevelNameMapping = map[Level]string{
13     DebugLevel: "DEBUG",
14     InfoLevel:  "INFO",
15     WarnLevel:  "WARN",
16     ErrorLevel: "ERROR",
17     PanicLevel: "PANIC",
18     FatalLevel: "FATAL",
19 }
```

在日志输出时，要通过对比开关级别和输出级别的大小，来决定是否输出，所以日志级别 Level 要定义成方便比较的数值类型。几乎所有的日志包都是用常量计数器 iota 来定义日志级别。

另外，因为要在日志输出中，输出可读的日志级别（例如输出 INFO 而不是 1），所以需要 Level 到 Level Name 的映射 LevelNameMapping，LevelNameMapping 会在格式化时用到。

接下来看定义日志选项。日志需要是可配置的，方便开发者根据不同的环境设置不同的日志行为，比较常见的配置选项为：

日志级别。

输出位置，例如标准输出或者文件。

输出格式，例如 JSON 或者 Text。

是否开启文件名和行号。

本示例的日志选项定义如下：

```
1 type options struct {
2     output      io.Writer
3     level       Level
4     stdLevel    Level
5     formatter   Formatter
6     disableCaller bool
7 }
```

[复制代码](#)

为了灵活地设置日志的选项，你可以通过选项模式，来对日志选项进行设置：

```
1 type Option func(*options)
2
3 func initOptions(opts ...Option) (o *options) {
4     o = &options{}
5     for _, opt := range opts {
6         opt(o)
7     }
8 }
```

[复制代码](#)

```
8     if o.output == nil {
9         o.output = os.Stderr
10    }
11
12    if o.formatter == nil {
13        o.formatter = &TextFormatter{}
14    }
15
16    return
17 }
18
19 func WithLevel(level Level) Option {
20     return func(o *options) {
21         o.level = level
22     }
23 }
24 ...
25 func SetOptions(opts ...Option) {
26     std.SetOptions(opts...)
27 }
28
29 func (l *logger) SetOptions(opts ...Option) {
30     l.mu.Lock()
31     defer l.mu.Unlock()
32
33     for _, opt := range opts {
34         opt(l.opt)
35     }
36 }
37
```

具有选项模式的日志包，可通过以下方式，来动态地修改日志的选项：

[📄 复制代码](#)

```
1  cuslog.SetOptions(cuslog.WithLevel(cuslog.DebugLevel))
```

你可以根据需要，对每一个日志选项创建设置函数 `WithXXXX`。这个示例日志包支持如下选项设置函数：

`WithOutput (output io.Writer)`：设置输出位置。

`WithLevel (level Level)`：设置输出级别。

`WithFormatter (formatter Formatter)`：设置输出格式。

`WithDisableCaller (caller bool)`：设置是否打印文件名和行号。

创建 Logger 及各级别日志打印方法

为了打印日志，我们需要根据日志配置，创建一个 Logger，然后通过调用 Logger 的日志打印方法，完成各级别日志的输出。本示例将创建代码保存在 [logger.go](#) 文件中。

可以通过如下方式创建 Logger：

[复制代码](#)

```
1 var std = New()
2
3 type logger struct {
4     opt      *options
5     mu       sync.Mutex
6     entryPool *sync.Pool
7 }
8
9 func New(opts ...Option) *logger {
10     logger := &logger{opt: initOptions(opts...)}
11     logger.entryPool = &sync.Pool{New: func() interface{} { return entry(logge
12     return logger
13 }
```

上述代码中，定义了一个 Logger，并实现了创建 Logger 的 New 函数。日志包都会有一个默认的全局 Logger，本示例通过 `var std = New()` 创建了一个全局的默认 Logger。`cuslog.Debug`、`cuslog.Info` 和 `cuslog.Warnf` 等函数，则是通过调用 std Logger 所提供的方法来打印日志的。

定义了一个 Logger 之后，还需要给该 Logger 添加最核心的日志打印方法，要提供所有支持级别的日志打印方法。

如果日志级别是 Xyz，则通常需要提供两类方法，分别是非格式化方法 `Xyz(args ...interface{})` 和格式化方法 `Xyzf(format string, args ...interface{})`，例如：

[复制代码](#)

```
1 func (l *logger) Debug(args ...interface{}) {
2     l.entry().write(DebugLevel, FmtEmptySeparate, args...)
3 }
4 func (l *logger) Debugf(format string, args ...interface{}) {
5     l.entry().write(DebugLevel, format, args...)
```



```
6 }
```

本示例实现了如下方法：Debug、Debugf、Info、Infof、Warn、Warnf、Error、Errorf、Panic、Panicf、Fatal、Fatalf。更详细的实现，你可以参考 cuslog/logger.go。

这里要注意，Panic、Panicf 要调用 panic() 函数，Fatal、Fatalf 函数要调用 os.Exit(1) 函数。

将日志输出到支持的输出中

调用日志打印函数之后，还需要将这些日志输出到支持的输出中，所以需要实现 write 函数，它的写入逻辑保存在 [entry.go](#) 文件中。实现方式如下：

[复制代码](#)

```
1 type Entry struct {
2     logger *logger
3     Buffer *bytes.Buffer
4     Map    map[string]interface{}
5     Level  Level
6     Time   time.Time
7     File   string
8     Line   int
9     Func   string
10    Format string
11    Args   []interface{}
12 }
13
14 func (e *Entry) write(level Level, format string, args ...interface{}) {
15     if e.logger.opt.level > level {
16         return
17     }
18     e.Time = time.Now()
19     e.Level = level
20     e.Format = format
21     e.Args = args
22     if !e.logger.opt.disableCaller {
23         if pc, file, line, ok := runtime.Caller(2); !ok {
24             e.File = "???"
25             e.Func = "???"
26         } else {
27             e.File, e.Line, e.Func = file, line, runtime.FuncForPC(pc).Name()
28             e.Func = e.Func[strings.LastIndex(e.Func, "/")+1:]
29         }
30     }
31 }
```

```
30     }
31     e.format()
32     e.writer()
33     e.release()
34 }
35
36 func (e *Entry) format() {
37     _ = e.logger.opt.formatter.Format(e)
38 }
39
40 func (e *Entry) writer() {
41     e.logger.mu.Lock()
42     _, _ = e.logger.opt.output.Write(e.Buffer.Bytes())
43     e.logger.mu.Unlock()
44 }
45
46 func (e *Entry) release() {
47     e.Args, e.Line, e.File, e.Format, e.Func = nil, 0, "", "", ""
48     e.Buffer.Reset()
49     e.logger.entryPool.Put(e)
50 }
```

上述代码，首先定义了一个 Entry 结构体类型，该类型用来保存所有的日志信息，即日志配置和日志内容。写入逻辑都是围绕 Entry 类型的实例来完成的。

用 Entry 的 write 方法来完成日志的写入，在 write 方法中，会首先判断日志的输出级别和开关级别，如果输出级别小于开关级别，则直接返回，不做任何记录。

在 write 中，还会判断是否需要记录文件名和行号，如果需要则调用 runtime.Caller() 来获取文件名和行号，调用 runtime.Caller() 时，要注意传入正确的栈深度。

write 函数中调用 e.format 来格式化日志，调用 e.writer 来写入日志，在创建 Logger 传入的日志配置中，指定了输出位置 output io.Writer，output 类型为 io.Writer，示例如下：

[复制代码](#)

```
1 type Writer interface {
2     Write(p []byte) (n int, err error)
3 }
```

io.Writer 实现了 Write 方法可供写入，所以只需要调用 `e.logger.opt.output.Write(e.Buffer.Bytes())` 即可将日志写入到指定的位置中。最后，会调用 `release()` 方法来清空缓存和对象池。至此，我们就完成了日志的记录和写入。

自定义日志输出格式

cuslog 包支持自定义输出格式，并且内置了 JSON 和 Text 格式的 Formatter。Formatter 接口定义为：

[复制代码](#)

```
1 type Formatter interface {  
2     Format(entry *Entry) error  
3 }
```

cuslog 内置的 Formatter 有两个：[JSON](#)和[TEXT](#)。

测试日志包

cuslog 日志包开发完成之后，可以编写测试代码，调用 cuslog 包来测试 cuslog 包，代码如下：

[复制代码](#)


```
1 package main  
2  
3 import (  
4     "log"  
5     "os"  
6  
7     "github.com/marmotedu/gopractise-demo/log/cuslog"  
8 )  
9  
10 func main() {  
11     cuslog.Info("std log")  
12     cuslog.SetOptions(cuslog.WithLevel(cuslog.DebugLevel))  
13     cuslog.Debug("change std log to debug level")  
14     cuslog.SetOptions(cuslog.WithFormatter(&cuslog.JsonFormatter{IgnoreBasicFi  
15     cuslog.Debug("log in json format")  
16     cuslog.Info("another log in json format")  
17  
18     // 输出到文件  
19     fd, err := os.OpenFile("test.log", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 06
```

```

20     if err != nil {
21         log.Fatalln("create file test.log failed")
22     }
23     defer fd.Close()
24
25     l := cuslog.New(cuslog.WithLevel(cuslog.InfoLevel),
26         cuslog.WithOutput(fd),
27         cuslog.WithFormatter(&cuslog.JsonFormatter{IgnoreBasicFields: false}),
28     )
29     l.Info("custom log with json formatter")
30 }

```

将上述代码保存在 main.go 文件中，运行：

 复制代码

```

1 $ go run example.go
2 2020-12-04T10:32:12+08:00 INFO example.go:11 std log
3 2020-12-04T10:32:12+08:00 DEBUG example.go:13 change std log to debug level
4 {"file":"/home/colin/workspace/golang/src/github.com/marmotedu/gopractise-demo
5 {"level":"INFO","time":"2020-12-04T10:32:12+08:00","file":"/home/colin/workspa

```


到这里日志包就开发完成了，完整包见 [log/cuslog](#)。

IAM 项目日志包设计

这一讲的最后，我们再来看下我们的 IAM 项目中，日志包是怎么设计的。

先来看一下 IAM 项目 log 包的存放位置：[pkg/log](#)。放在这个位置，主要有两个原因：第一个，log 包属于 IAM 项目，有定制开发的内容；第二个，log 包功能完备、成熟，外部项目也可以使用。

该 log 包是基于 go.uber.org/zap 包封装而来的，根据需要添加了更丰富的功能。接下来，我们通过 log 包的 [Options](#)，来看下 log 包所实现的功能：

 复制代码

```

1 type Options struct {
2     OutputPaths      []string `json:"output-paths"      mapstructure:"output
3     ErrorOutputPaths []string `json:"error-output-paths" mapstructure:"error-
4     Level            string  `json:"level"            mapstructure:"level"
5     Format           string  `json:"format"          mapstructure:"format"

```

```

6      DisableCaller      bool      `json:"disable-caller"      mapstructure:"disabl
7      DisableStacktrace bool      `json:"disable-stacktrace" mapstructure:"disabl
8      EnableColor        bool      `json:"enable-color"       mapstructure:"enable
9      Development        bool      `json:"development"       mapstructure:"develo
10     Name                string     `json:"name"              mapstructure:"name"`
11 }

```

Options 各配置项含义如下：

development：是否是开发模式。如果是开发模式，会对 DPanicLevel 进行堆栈跟踪。

name：Logger 的名字。

disable-caller：是否开启 caller，如果开启会在日志中显示调用日志所在的文件、函数和行号。

disable-stacktrace：是否在 Panic 及以上级别禁止打印堆栈信息。

enable-color：是否开启颜色输出，true，是；false，否。

level：日志级别，优先级从低到高依次为：Debug, Info, Warn, Error, Dpanic, Panic, Fatal。

format：支持的日志输出格式，目前支持 Console 和 JSON 两种。Console 其实就是 Text 格式。

output-paths：支持输出到多个输出，用逗号分开。支持输出到标准输出（stdout）和文件。

error-output-paths：zap 内部（非业务）错误日志输出路径，多个输出，用逗号分开。


log 包的 Options 结构体支持以下 3 个方法：

Build 方法。Build 方法可以根据 Options 构建一个全局的 Logger。

AddFlags 方法。AddFlags 方法可以将 Options 的各个字段追加到传入的 pflag.FlagSet 变量中。

String 方法。String 方法可以将 Options 的值以 JSON 格式字符串返回。

log 包实现了以下 3 种日志记录方法：


 复制代码

```
1 log.Info("This is a info message", log.Int32("int_key", 10))
2 log.Infof("This is a formatted %s message", "info")
3 log.Infow("Message printed with Infow", "X-Request-ID", "fbf54504-64da-4088-9b
```

Info 使用指定的 key/value 记录日志。Infof 格式化记录日志。Infow 也是使用指定的 key/value 记录日志，跟 Info 的区别是：使用 Info 需要指定值的类型，通过指定值的日志类型，日志库底层不需要进行反射操作，所以使用 Info 记录日志性能最高。

log 包支持非常丰富的类型，具体你可以参考 [types.go](https://golang.org/pkg/types/#pkg-overview)。

上述日志输出为：


 复制代码

```
1 2021-07-06 14:02:07.070 INFO This is a info message {"int_key": 10}
2 2021-07-06 14:02:07.071 INFO This is a formatted info message
3 2021-07-06 14:02:07.071 INFO Message printed with Infow {"X-Request-ID": "fbf5
```

log 包为每种级别的日志都提供了 3 种日志记录方式，举个例子：假设日志格式为 Xyz，则分别提供了 Xyz(msg string, fields ...Field)，Xyzf(format string, v ...interface{})，Xyzw(msg string, keysAndValues ...interface{}) 3 种日志记录方法。

另外，log 包相较于一般的日志包，还提供了众多记录日志的方法。

第一个方法，log 包支持 V Level，可以通过整型数值来灵活指定日志级别，数值越大，优先级越低。例如：

 复制代码

```
1 // V level使用
2 log.V(1).Info("This is a V level message")
3 log.V(1).Infof("This is a %s V level message", "formatted")
4 log.V(1).Infow("This is a V level message with fields", "X-Request-ID", "7a7b9
```

这里要注意，Log.V 只支持 Info、Infof、Infow 三种日志记录方法。

第二个方法，log 包支持 WithValues 函数，例如：

[复制代码](#)

```
1 // WithValues使用
2 lv := log.WithValues("X-Request-ID", "7a7b9f24-4cae-4b2a-9464-69088b45b904")
3 lv.Infow("Info message printed with [WithValues] logger")
4 lv.Infow("Debug message printed with [WithValues] logger")
```

上述日志输出如下：

[复制代码](#)

```
1 2021-07-06 14:15:28.555 INFO Info message printed with [WithValues] logger {"X
2 2021-07-06 14:15:28.556 INFO Debug message printed with [WithValues] logger {"
```

WithValues 可以返回一个携带指定 key-value 的 Logger，供后面使用。

第三个方法，log 包提供 WithContext 和 FromContext 用来将指定的 Logger 添加到某个 Context 中，以及从某个 Context 中获取 Logger，例如：

[复制代码](#)

```
1 // Context使用
2 ctx := lv.WithContext(context.Background())
3 lc := log.FromContext(ctx)
4 lc.Info("Message printed with [WithContext] logger")
```

WithContext和FromContext非常适合用在以context.Context传递的函数中，例如：

[复制代码](#)


```
1 func main() {
2
3     ...
4
5     // WithValues使用
6     lv := log.WithValues("X-Request-ID", "7a7b9f24-4cae-4b2a-9464-69088b45b904")
7
8     // Context使用
9     lv.Infof("Start to call pirntString")
```

```

10     ctx := lv.WithContext(context.Background())
11     pirntString(ctx, "World")
12 }
13
14 func pirntString(ctx context.Context, str string) {
15     lc := log.FromContext(ctx)
16     lc.Infof("Hello %s", str)
17 }

```

上述代码输出如下：

 复制代码


```

1 2021-07-06 14:38:02.050 INFO Start to call pirntString {"X-Request-ID": "7a7b9
2 2021-07-06 14:38:02.050 INFO Hello World {"X-Request-ID": "7a7b9f24-4cae-4b2a-

```

将 Logger 添加到 Context 中，并通过 Context 在不同函数间传递，可以使 key-value 在不同函数间传递。例如上述代码中，X-Request-ID 在 main 函数、printString 函数中的日志输出中均有记录，从而实现了一种调用链的效果。

第四个方法，可以很方便地从 Context 中提取出指定的 key-value，作为上下文添加到日志输出中，例如 [internal/apiserver/api/v1/user/create.go](#) 文件中的日志调用：

 复制代码

```

1 log.L(c).Info("user create function called.")

```

通过调用 Log.L() 函数，实现如下：

 复制代码

```

1 // L method output with specified context value.
2 func L(ctx context.Context) *zapLogger {
3     return std.L(ctx)
4 }
5
6 func (l *zapLogger) L(ctx context.Context) *zapLogger {
7     lg := l.clone()
8
9     requestID, _ := ctx.Value(KeyRequestID).(string)
10    username, _ := ctx.Value(KeyUsername).(string)
11    lg.zapLogger = lg.zapLogger.With(zap.String(KeyRequestID, requestID), zap.
12

```



```

13     return lg
14 }

```

L() 方法会从传入的 Context 中提取出 requestID 和 username，追加到 Logger 中，并返回 Logger。这时候调用该 Logger 的 Info、Infof、Infof 等方法记录日志，输出的日志中均包含 requestID 和 username 字段，例如：

```

1 2021-07-06 14:46:00.743 INFO      apiserver      secret/create.go:23      create

```

[复制代码](#)

通过将 Context 在函数间传递，很容易就能实现调用链效果，例如：

```

1 // Create add new secret key pairs to the storage.
2 func (s *SecretHandler) Create(c *gin.Context) {
3     log.L(c).Info("create secret function called.")
4
5     ...
6
7     sec, err := s.store.Secrets().List(c, username, metav1.ListOptions{
8         Offset: pointer.ToInt64(0),
9         Limit:  pointer.ToInt64(-1),
10    })
11
12    ...
13
14    if err := s.srv.Secrets().Create(c, &r, metav1.CreateOptions{}); err != nil {
15        core.WriteResponse(c, err, nil)
16    }
17    return
18 }

```

[复制代码](#)

上述代码输出为：

```

1 2021-07-06 14:46:00.743 INFO      apiserver      secret/create.go:23      create
2 2021-07-06 14:46:00.744 INFO      apiserver      secret/create.go:23      list s
3 2021-07-06 14:46:00.745 INFO      apiserver      secret/create.go:23      insert

```

[复制代码](#)

这里要注意，`log.L` 函数默认会从 `Context` 中取 `requestID` 和 `username` 键，这跟 IAM 项目有耦合度，但这不影响 `log` 包供第三方项目使用。这也是我建议你自己封装日志包的原因。

总结

开发一个日志包，我们很多时候需要基于一些业界优秀的开源日志包进行二次开发。当前很多项目的日志包都是基于 `zap` 日志包来封装的，如果你有封装的需要，我建议你优先选择 `zap` 日志包。

这一讲中，我先给你介绍了标准库 `log` 包、`glog`、`logrus` 和 `zap` 这四种常用的日志包，然后向你展现了开发一个日志包的四个步骤，步骤如下：

1. 定义日志级别和日志选项。
2. 创建 `Logger` 及各级别日志打印方法。
3. 将日志输出到支持的输出中。
4. 自定义日志输出格式。

最后，我介绍了 IAM 项目封装的 `log` 包的设计和使用方式。`log` 包基于 `go.uber.org/zap` 封装，并提供了以下强大特性：

`log` 包支持 `V Level`，可以灵活的通过整型数值来指定日志级别。

`log` 包支持 `WithValues` 函数，`WithValues` 可以返回一个携带指定 `key-value` 对的 `Logger`，供后面使用。

`log` 包提供 `WithContext` 和 `FromContext` 用来将指定的 `Logger` 添加到某个 `Context` 中和从某个 `Context` 中获取 `Logger`。

`log` 包提供了 `Log.L()` 函数，可以很方便的从 `Context` 中提取出指定的 `key-value` 对，作为上下文添加到日志输出中。

课后练习

1. 尝试实现一个新的 `Formatter`，可以使不同日志级别以不同颜色输出（例如：`Error` 级别的日志输出中 `Error` 字符串用红色字体输出，`Info` 字符串用白色字体输出）。

2. 尝试将 `runtime.Caller(2)` 函数调用中的 2 改成 1，看看日志输出是否跟修改前有差异，如果有差异，思考差异产生的原因。

欢迎你在留言区与我交流讨论，我们下一讲见。

分享给需要的人，Ta 订阅后你可得 **24 元现金奖励**

👍 赞 2 💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | 日志处理（上）：如何设计日志包并记录日志？

下一篇 22 | 应用构建三剑客：Pflag、Viper、Cobra 核心功能介绍

更多课程推荐

容器实战高手课

在实战中深入理解容器技术的本质

李程远

eBay 总监级工程师
云平台架构师



涨价倒计时 ⏰

今日订阅 **¥69**，7月20日涨价至 **¥129**

精选留言 (2)

💬 写留言

**nio**

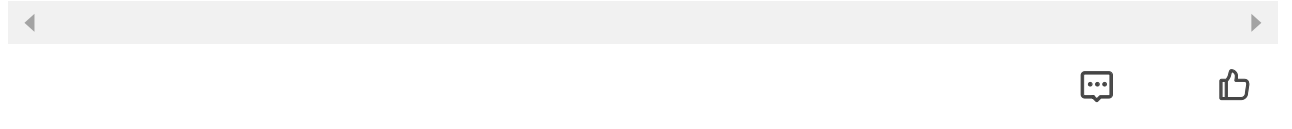
2021-07-13

IAM 项目 log 包的性能比较大概是什么样子呢

展开 ▾

作者回复: 性能跟github.com/pkg/log性能接近一致。github.com/pkg/log这个包很多生产环境在用，所以iam的log包应用在生产环境完全没问题。

你要感兴趣，可以跟其它包对比下，比如：logrus，zap，glog等。也欢迎在留言区分享对比结果。

**helloworld**

2021-07-13

log.Int32("int_key", 10)还有V Level这两处没有get到是干啥用的

作者回复: log.Int32直接指定了字段类型，log不需要再做反射，这种疾苦方式可以提高性能。

V Level可以允许指定任意优先级的日志级别。你可以参考glog的用法来理解V level。

有时候日志包预定义的日志级别可能不够用，这时候可以试试V Level

