



下载APP

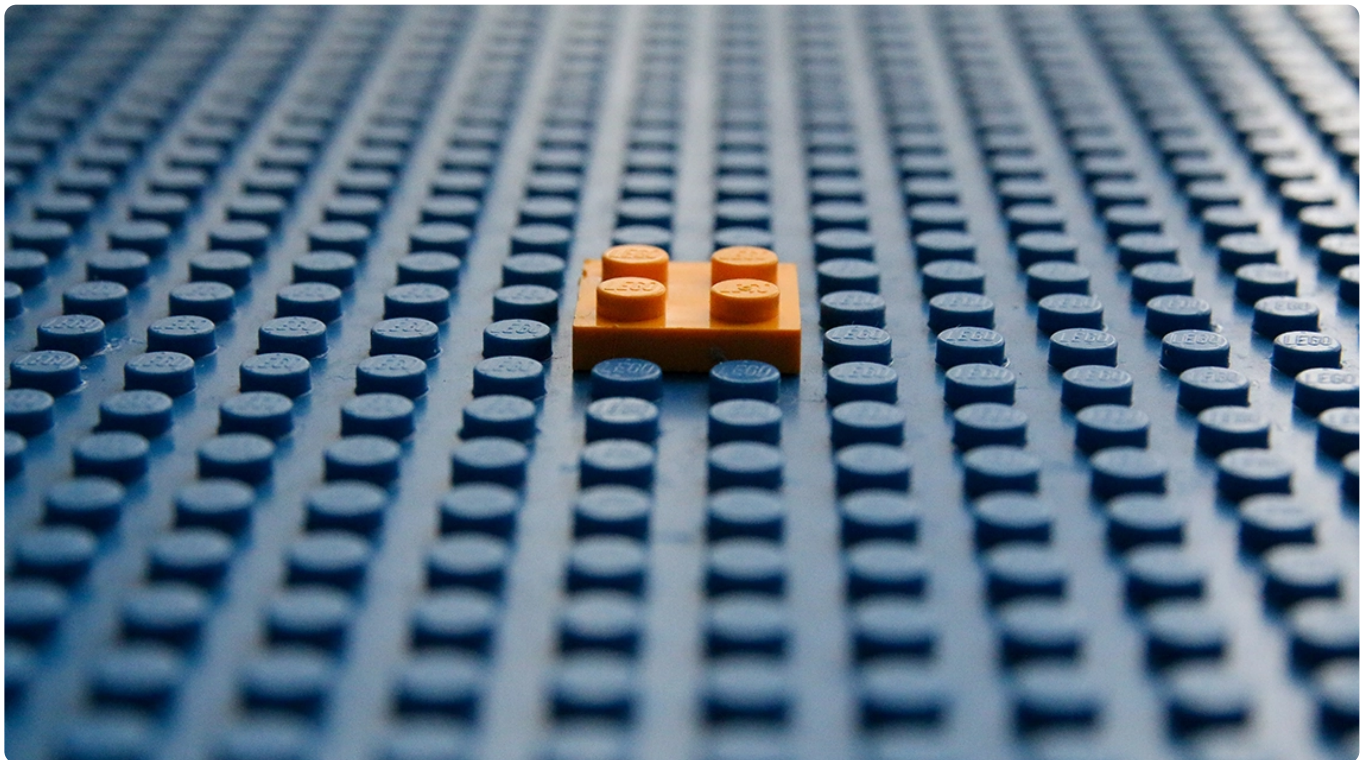


17 | 日志：如何设计多输出的日志服务？

2021-10-25 叶剑峰

《手把手带你写一个Web框架》

课程介绍 >



讲述：叶剑峰

时长 17:36 大小 16.12M



你好，我是轩脉刃。

上面两节课，我们将环境变量和配置服务作为一个服务注册到容器中了。这样，在业务中就能很方便获取环境变量和配置了。不知道你有没有逐渐体会到这种“一切皆服务”思想的好处。

就像堆积积木，只要想好了一个服务的接口，我们逐步实现服务之后，这一个服务就是一块积木，之后可以用相同的思路实现各种服务的积木块，用它们来拼出我们需要的业务逻辑。这节课，我们就来实现另一个框架最核心的积木：日志服务。



实现一个框架的服务，我们习惯要创建三个文件：接口协议文件
framework/contract/log.go、服务提供者 framework/provider/log/provider.go、接

□实例 `framework/provider/log/service.go`。

日志接口协议

说到日志服务，最先冒出来的一定是三个问题：什么样的日志需要输出？日志输出哪些内容？日志输出到哪里？一个个来分析。

日志级别

什么样的日志需要输出，这是个关于日志级别的问题。我们想要把日志分为几个级别？每个级别代表什么？这个日志级别其实在不同系统中有不同划分，比如 Linux 的 [syslog](#) 中将系统划分为以下八种日志级别：

Numerical Code	Severity
0	Emergency: system is unusable
1	Alert: action must be taken immediately
2	Critical: critical conditions
3	Error: error conditions
4	Warning: warning conditions
5	Notice: normal but significant condition
6	Informational: informational messages
7	Debug: debug-level messages

Table 2. Syslog Message Severities

而 Java 的 [log4j](#) 将日志分为以下七种日志级别：

级别	描述
OFF	最高级别，用于关闭日志记录。
FATAL	导致应用程序提前终止的严重错误。一般这些信息将立即呈现在状态控制台上。
ERROR	其他运行时错误或意外情况。一般这些信息将立即呈现在状态控制台上。
WARN	使用已过时的API，API的滥用，潜在错误，其他不良的或意外的运行时的状况（但不一定是错误的）。一般这些信息将立即呈现在状态控制台上。
INFO	令人感兴趣的运行时事件（启动/关闭）。一般这些信息将立即呈现在状态控制台上。因而要保守使用，并保持到最低限度。
DEBUG	流经系统的详细信息。一般这些信息只记录到日志文件中。
TRACE	最详细的信息。一般这些信息只记录到日志文件中。自版本1.2.12。



其实仔细看，它们的日志级别差别都不大。比如都同意用 Error 级别代表运行时的错误情况，而 Warn 级别代表运行时可以弥补的错误、Info 级别代表运行时信息、debug 代表调试的时候需要打印的信息。

不同点就在是否有 trace 级别以及 Error 级别往上的级别定义。syslog 没有 trace 级别，而在 Error 级别往上分别定义了 Emergency 级别、Alert 级别、Critical 级别。而 log4j 在 ERROR 上定义了两个级别 FATAL 和 OFF，也同时保留了 Trace 级别。

在我看来，syslog 和 log4j 的日志区分主要是由于场景不同。syslog 比较偏向于**操作系统的使用场景**，它的分级语义更多是告诉我们系统的情况，比如 Alert 这个级别表示“系统有问题，需要立即采取行动”；而 log4j 的日志级别定义是**从一个应用出发的**，它的影响范围理论上会小一些，所以它很难定义比如像“需要立即采取行动”这样的级别。

所以，这里我们主要参考 log4j 的日志级别方法，并做了一些小调整，归并为下列七种日志级别：

- panic，表示会导致整个程序出现崩溃的日志信息
- fatal，表示会导致当前这个请求出现提前终止的错误信息
- error，表示出现错误，但是不一定影响后续请求逻辑的错误信息
- warn，表示出现错误，但是一定不影响后续请求逻辑的报警信息

info，表示正常的日志信息输出


debug，表示在调试状态下打印出来的日志信息

trace，表示最详细的信息，一般信息量比较大，可能包含调用堆栈等信息

在 error 级别之上，我们把导致程序崩溃和导致请求结束的错误拆分出来，分为 panic 和 fatal 两个类型来定义级别。而其他的 error、warn、info、debug 都和其他的日志系统一致。另外也增加一个 trace 级别，当需要打印调用堆栈等这些比较详细的信息的时候，可以使用这种日志级别。

日志级别是按照从下往上的严重顺序排列的。也就是说，如果我们设置了日志输出级别为 info，那么 info 级别的日志及 info 级别往上，日志级别更高的 warn、error、fatal、panic 的日志，也需要被打印出来。

按照这个思路，我们在 framework/contract/log.go 中定义的接口协议如下：

 复制代码

```
1 package contract
2 import (
3     "context"
4     "io"
5     "time"
6 )
7
8 // 协议关键字
9 const LogKey = "hade:log"
10
11 type LogLevel uint32
12
13 const (
14     // UnknownLevel 表示未知的日志级别
15     UnknownLevel LogLevel = iota
16     // PanicLevel level, panic 表示会导致整个程序出现崩溃的日志信息
17     PanicLevel
18     // FatalLevel level. fatal 表示会导致当前这个请求出现提前终止的错误信息
19     FatalLevel
20     // ErrorLevel level. error 表示出现错误，但是不一定影响后续请求逻辑的错误信息
21     ErrorLevel
22     // WarnLevel level. warn 表示出现错误，但是一定不影响后续请求逻辑的报警信息
23     WarnLevel
24     // InfoLevel level. info 表示正常的日志信息输出
25     InfoLevel
26     // DebugLevel level. debug 表示在调试状态下打印出来的日志信息
27     DebugLevel
```

```
28 // TraceLevel level. trace 表示最详细的信息，一般信息量比较大，可能包含调用堆栈等信息
29 TraceLevel
30 )
31 ...
32
33 // Log define interface for log
34 type Log interface {
35     // Panic 表示会导致整个程序出现崩溃的日志信息
36     Panic(ctx context.Context, msg string, fields map[string]interface{})
37     // Fatal 表示会导致当前这个请求出现提前终止的错误信息
38     Fatal(ctx context.Context, msg string, fields map[string]interface{})
39     // Error 表示出现错误，但是不一定影响后续请求逻辑的错误信息
40     Error(ctx context.Context, msg string, fields map[string]interface{})
41     // Warn 表示出现错误，但是一定不影响后续请求逻辑的报警信息
42     Warn(ctx context.Context, msg string, fields map[string]interface{})
43     // Info 表示正常的日志信息输出
44     Info(ctx context.Context, msg string, fields map[string]interface{})
45     // Debug 表示在调试状态下打印出来的日志信息
46     Debug(ctx context.Context, msg string, fields map[string]interface{})
47     // Trace 表示最详细的信息，一般信息量比较大，可能包含调用堆栈等信息
48     Trace(ctx context.Context, msg string, fields map[string]interface{})
49     // SetLevel 设置日志级别
50     SetLevel(level LogLevel)
51     ...
52 }
```

在接口中，我们针对七种日志级别设置了七个不同的方法，并且提供 SetLevel 方法，来设置当前这个日志服务需要输出的日志级别。

日志格式

定义好了日志级别，下面该定义日志格式了。日志格式包括输出哪些内容、如何输出？

首先明确下需要输出的日志信息，不外乎有下面四个部分：


日志级别，输出当前日志的级别信息。

日志时间，输出当前日志的打印时间。

日志简要信息，输出当前日志的简要描述信息，一句话说明日志错误。

日志上下文字段，输出当前日志的附带信息。这些字段代表日志打印的上下文。


比如这就是一个完整的日志信息：

 复制代码

```
1 [Info] 2021-09-22T00:04:21+08:00 "demo test error" map[api:demo/d
```

上面那条日志，日志级别为 Info，时间为 2021-09-21 年 15:40:03，时区为 +08:00。简要信息 demo test error 表示这个日志希望打印的信息，剩下的 map 表示的 key、value 为补充的日志上下文字段。

它对应的调用函数如下：

 复制代码

```
1 logger.Info(c, "demo test error", map[string]interface{}{  
2     "api": "demo/demo",  
3     "user": "jianfengye",  
4 })
```

这里我额外说一下日志上下文字段。它是一个 map 值，来源可能有两个：一个是用户在打印日志的时候传递的 map，比如上面代码中的 api 和 user；而另外一部分数据是可能来自 context，因为在具体业务开发中，我们很有可能把一些通用信息，比如 trace_id 等放在 context 里，这一部分信息也会希望取出放在日志的上下文字段中。

所以这里有一个从 context 中获取日志上下文字段的方法。在 framework/contract/log.go 中定义其为 CtxFielder。

 复制代码

```
1 // CtxFielder 定义了从context中获取信息的方法  
2 type CtxFielder func(ctx context.Context) map[string]interface{}
```

明确了打印哪些信息，更要明确这些信息按照什么输出格式输出。这个输出格式也是一个通用方法 Fomatter，它的传入参数就是刚才的四个日志信息。

 复制代码

```
1 // Formatter 定义了将日志信息组织成字符串的通用方法  
2 type Formatter func(level LogLevel, t time.Time, msg string, fields map[string
```

同时在 log 服务协议中增加了 SetFormatter 和 SetCtxFielder 的方法。

 复制代码


```
1 // Log 定义了日志服务协议
2 type Log interface {
3     ...
4     // SetCtxFielder 从context中获取上下文字段field
5     SetCtxFielder(handler CtxFielder)
6     // SetFormatter 设置输出格式
7     SetFormatter(formatter Formatter)
8     ...
9 }
```

日志输出

已经解决了前两个问题，明确了日志的级别和输出格式，那日志可以输出在哪些地方？

其实我们在定义接口的时候，并不知道它会输出到哪里。但是只需要知道一定会输出到某个输出管道就可以了，之后在每个应用中使用的时候，我们再根据每个应用的配置，来确认具体的输出管道实现。

目前这个输出管道我们使用 io.Writer 来进行设置：

 复制代码

```
1 // Log 定义了日志服务协议
2 type Log interface {
3     ...
4     // SetOutput 设置输出管道
5     SetOutput(out io.Writer)
6 }
```

日志服务提供者

日志接口协议文件就完成了，下面来实现日志的服务提供者，在 framework/provider/log/provider.go 中。

在编写服务提供者的时候，我们需要先明确最终会提供哪些服务。对于日志服务，按照我们平时的使用情况，可以分为四类：

控制台输出

本地单个日志文件输出

本地单个日志文件，自动进行切割输出

自定义输出

这四种输出我们都各自定义一个服务，分别放在 `framework/provider/log/service/` 目录下的四个文件里：


`console.go` 表示控制台输出，定义初始化实例方法 `NewHadeConsoleLog`；

`single.go` 表述单个日志文件输出，定义初始化实例方法 `NewHadeSingleLog`；

`rotate.go` 表示单个文件输出，但是自动进行切割，定义初始化实例方法 `NewHadeRotateLog`；

`custom.go` 表示自定义输出，定义实例化方法 `NewHadeCustomLog`。

那在服务提供者的 `Register` 注册服务实例方法中，我们设计成根据配置项 `"log.driver"`，来选择不同的实例化方法，默认为 `NewHadeConsoleLog` 方法。

 复制代码

```
1 // Register 注册一个服务实例
2 func (l *HadeLogServiceProvider) Register(c framework.Container) framework.New
3     if l.Driver == "" {
4         tcs, err := c.Make(contract.ConfigKey)
5         if err != nil {
6             // 默认使用console
7             return services.NewHadeConsoleLog
8         }
9         cs := tcs.(contract.Config)
10        l.Driver = strings.ToLower(cs.GetString("log.Driver"))
11    }
12    // 根据driver的配置项确定
13    switch l.Driver {
14    case "single":
15        return services.NewHadeSingleLog
16    case "rotate":
17        return services.NewHadeRotateLog
18    case "console":
19        return services.NewHadeConsoleLog
20    case "custom":
21        return services.NewHadeCustomLog
22    default:
```



```

23     return services.NewHadeConsoleLog
24 }
25 }

```

而上一节里面分析的，日志的几个配置：日志级别、输出格式方法、context 内容获取方法、输出方法，都以服务提供者 provider.go 中参数的方式提供。

```

1 // HadeLogServiceProvider 服务提供者
2 type HadeLogServiceProvider struct {
3     ...
4
5     // 日志级别
6     Level contract.LogLevel
7     // 日志输出格式方法
8     Formatter contract.Formatter
9     // 日志context上下文信息获取函数
10    CtxFielder contract.CtxFielder
11    // 日志输出信息
12    Output io.Writer
13 }

```

[复制代码](#)

默认提供两种输出格式，一种是文本输出形式，比如上面举的那个例子，

```

1 [Info]  2021-09-22T00:04:21+08:00      "demo test error"      map[api:demo/d

```

[复制代码](#)

另外一种 JSON 输出形式，如下：

```

1 {"api":"demo/demo","cspan_id":"","level":5,"msg":"demo1","parent_id":"","span_

```


[复制代码](#)

这两种输出除了格式不同，其中的内容应该是相同的。具体使用起来，文本输出更便于我们阅读，而 JSON 输出更便于机器或者程序阅读。

在实现文件夹 framework/provider/log/formatter/ 里，我们增加两个文件 json.go 和 text.go 表示两种格式输出，对应的 TextFormatter 和 JsonFormatter 是对应的文本格式

输出方法，

这里就贴出 text.go 的具体实现，很简单，其他的差别不大，可以参考 [G@it@H@ub](#)。

 复制代码

```
1 // TextFormatter 表示文本格式输出
2 func TextFormatter(level contract.LogLevel, t time.Time, msg string, fields map[string]string) []byte {
3     bf := bytes.NewBuffer([]byte{})
4     Separator := "\t"
5
6     // 先输出日志级别
7     prefix := Prefix(level)
8
9     bf.WriteString(prefix)
10    bf.WriteString(Separator)
11
12    // 输出时间
13    ts := t.Format(time.RFC3339)
14    bf.WriteString(ts)
15    bf.WriteString(Separator)
16
17    // 输出msg
18    bf.WriteString("\n")
19    bf.WriteString(msg)
20    bf.WriteString("\n")
21    bf.WriteString(Separator)
22
23    // 输出map
24    bf.WriteString(fmt.Sprintf("%v", fields))
25    return bf.Bytes(), nil
26 }
```

再回到 framework/provider/log/provider.go，定义服务提供者的 Params 方法。比如获取格式化方法 Formatter，我们就设定成，先判断在初始化的时候，是否定义了服务提供者；如果没有，再判断配置项 log.formatter 是否指定了格式化方法 json/text，设置最终的 Formatter，并且传递实例化的方法。

 复制代码

```
1 // Params 定义要传递给实例化方法的参数
2 func (l *HadeLogServiceProvider) Params(c framework.Container) []interface{} {
3     // 获取configService
4     configService := c.MustMake(contract.ConfigKey).(contract.Config)
5
6     // 设置参数formatter
```

```

7     if l.Formatter == nil {
8         l.Formatter = formatter.TextFormatter
9         if configService.IsExist("log.formatter") {
10             v := configService.GetString("log.formatter")
11             if v == "json" {
12                 l.Formatter = formatter.JsonFormatter
13             } else if v == "text" {
14                 l.Formatter = formatter.TextFormatter
15             }
16         }
17     }
18
19     if l.Level == contract.UnknownLevel {
20         l.Level = contract.InfoLevel
21         if configService.IsExist("log.level") {
22             l.Level = logLevel(configService.GetString("log.level"))
23         }
24     }
25
26     // 定义5个参数
27     return []interface{}{c, l.Level, l.CtxFielder, l.Formatter, l.Output}
28 }

```


至于日志服务提供者的其他几个方法（Register、Boot、IsDefer、Name），就不在这里说明了，可以参考 [GitHub](#) 上的代码。

日志服务的具体实现

最后就到具体的日志服务的实现了。上面我们说，针对四种不同的输出方式，定义了四个不同的服务实例，**这四个不同的服务实例都需要实现前面定义的日志服务协议。如果每个实例都实现一遍，还是非常麻烦的。这里可以使用一个技巧：类型嵌套。**

我们先创建一个通用的服务实例 HadeLog，在 HadeLog 中存放通用的字段，比如上述日志服务提供者传递的五个参数：container、level、ctxFielder、formatter、output。

在 provider/log/services/log.go 中定义这个结构：

 复制代码

```

1 // HadeLog 的通用实例
2 type HadeLog struct {
3     // 五个必要参数
4     level      contract.LogLevel // 日志级别
5     formatter  contract.Formatter // 日志格式化方法
6     ctxFielder contract.CtxFielder // ctx获取上下文字段

```

```
7    output    io.Writer           // 输出
8    c         framework.Container // 容器
9 }
```

接着在通用实例中，使用这几个必要的参数，就能实现日志协议的所有接口了，这里展示了 Info 方法是怎么打印信息的：

[复制代码](#)

```
1 // Info 会打印出普通的日志信息
2 func (log *HadeLog) Info(ctx context.Context, msg string, fields map[string]interface{}) {
3     log.logf(contract.InfoLevel, ctx, msg, fields)
4 }
5
6 // logf 为打印日志的核心函数
7 func (log *HadeLog) logf(level contract.LogLevel, ctx context.Context, msg string, fields map[string]interface{}) {
8     // 先判断日志级别
9     if !log.IsLevelEnable(level) {
10         return nil
11     }
12
13     // 使用ctxFielder 获取context中的信息
14     fs := fields
15     if log.ctxFielder != nil {
16         t := log.ctxFielder(ctx)
17         if t != nil {
18             for k, v := range t {
19                 fs[k] = v
20             }
21         }
22     }
23
24     ...
25
26     // 将日志信息按照formatter序列化为字符串
27     if log.formatter == nil {
28         log.formatter = formatter.TextFormatter
29     }
30     ct, err := log.formatter(level, time.Now(), msg, fs)
31     if err != nil {
32         return err
33     }
34
35     // 如果是panic级别，则使用log进行panic
36     if level == contract.PanicLevel {
37         pkgLog.Panicln(string(ct))
38         return nil
39     }
40 }
```

```

41 // 通过output进行输出
42 log.output.Write(ct)
43 log.output.Write([]byte("\r\n"))
44 return nil
45 }

```

可以看到，Info 打印最终调用 logf 方法，而 logf 方法的实现步骤也很清晰，简单梳理一下：

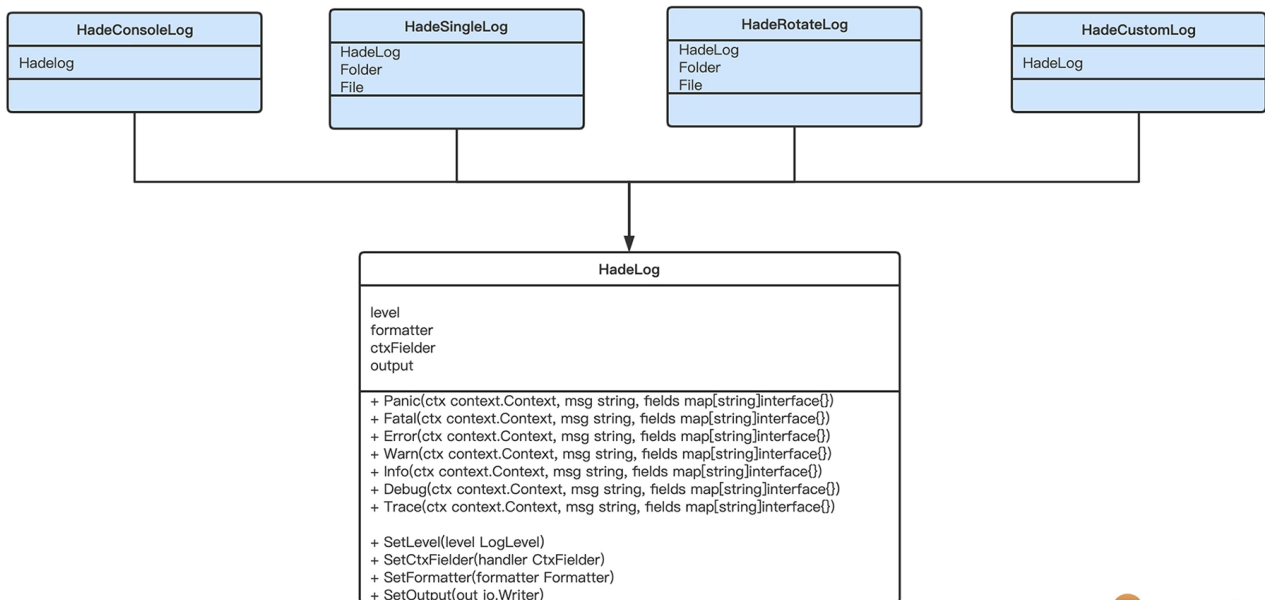
先判断日志级别是否符合要求，如果不符合要求，则直接返回，不进行打印；

再使用 ctxFielder，从 context 中获取信息放在上下文字段中；

接着将日志信息按照 formatter 序列化为字符串；

最后通过 output 进行输出。

HadeLog 其他方法的实现和 Info 大同小异，这里就不展示所有代码了。实现了基础的 HadeLog 实例，接下来，就实现对应的四个不同输出类型的实例 HadeConsoleLog、HadeSingleLog、HadeRotateLog、HadeCustomLog。



这里四个具体实例使用类型嵌套的方式，就能自动拥有 HadeLog 已经实现了的那些方法。

比如在 `framework/provider/log/service/console.log` 中，使用类型嵌套实现 `HadeConsoleLog`：

[复制代码](#)

```
1 // HadeConsoleLog 代表控制台输出
2 type HadeConsoleLog struct {
3     // 类型嵌套HadeLog
4     HadeLog
5 }
```

相当于 `HadeConsoleLog` 就已经实现了日志服务协议了。我们唯一要做的就是实例化 `HadeConsoleLog` 的时候，将基础 `HadeLog` 中的通用字段进行填充。比如 `HadeConsoleLog` 最重要就是将输出类型 `output` 设置为控制台 `os.stdout`：

[复制代码](#)

```
1 // NewHadeConsoleLog 实例化HadeConsoleLog
2 func NewHadeConsoleLog(params ...interface{}) (interface{}, error) {
3     c := params[0].(framework.Container)
4     level := params[1].(contract.LogLevel)
5     ctxFielder := params[2].(contract.CtxFielder)
6     formatter := params[3].(contract.Formatter)
7
8     log := &HadeConsoleLog{}
9
10    log.SetLevel(level)
11    log.SetCtxFielder(ctxFielder)
12    log.SetFormatter(formatter)
13
14    // 最重要的将内容输出到控制台
15    log.SetOutput(os.Stdout)
16    log.c = c
17    return log, nil
18 }
```

四种输出文件其实都大同小异，这里就挑选一个最复杂的带有日志切割的 `HadeRotateLog` 来讲解。

Golang 中日志切割有个非常好用的 [file-rotate](#)，这个库的使用方法也不复杂，最核心的就是一个初始化操作 `New`：


```
1 func New(p string, options ...Option) (*RotateLogs, error)
```

[复制代码](#)

它有两个参数，第一个参数 `p` 是带目录的日志地址，可以允许有通配符代表日期的日志文件名。这里的通配符符合 Linux 的 `strftime` 的定义，具体哪个通配符代表日期、小时、分钟等可以参考 `strftime` 的 [文档说明](#)。而第二个参数是 `Option` 数组，表示这个切割日志的一些配置，比如多久切割一次日志文件、切割后的日志文件保存多少天等。

使用很简单，直接看我们对 `HadeRotateLog` 的具体实现。大致思路就是**先定义结构，再实现初始化方法，在初始化方法中，我们实例化 `file-rotatelogs` 的初始化操作 `New`。**

首先定义了 `HadeRotateLog` 的结构，其中嵌套了基础实例结构 `HadeLog`，同时有这个结构特定的字段 `folder` 和 `file`：

```
1 // HadeRotateLog 代表会进行切割的日志文件存储
2 type HadeRotateLog struct {
3     HadeLog
4     // 日志文件存储目录
5     folder string
6     // 日志文件名
7     file string
8 }
```

[复制代码](#)

实例化的 `NewHadeRotateLog` 先获取参数，然后从配置文件中获取参数属性 `folder`、`file`、`date_format`、`rotate_count`、`rotate_size`、`max_age`、`rotate_time`，这些属性都和 `file-rotatelogs` 库实例化的 `Option` 参数一一对应。

所以这里也展示一下我们的 `log.yaml` 配置文件可配置的 `rotate`：

```
1 driver: rotate # 切割日志
2 level: trace # 日志级别
3 file: coredemo.log # 保存的日志文件
4 rotate_count: 10 # 最多日志文件个数
5 rotate_size: 120000 # 每个日志大小
6 rotate_time: "1m" # 切割时间
7 max_age: "10d" # 文件保存时间
8 date_format: "%Y-%m-%d-%H-%M" # 文件后缀格式
```

[复制代码](#)

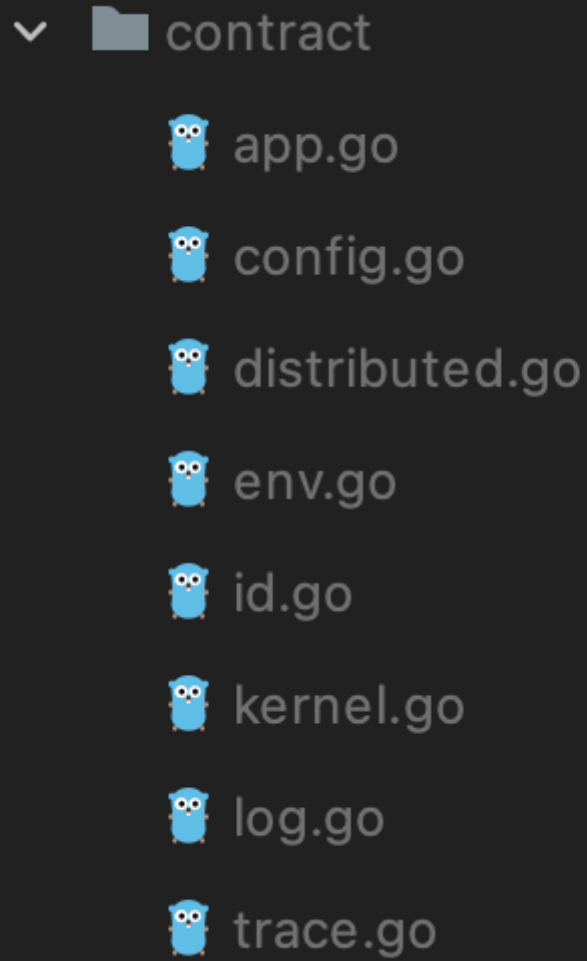
再回到 NewHadeRotateLog，设置了这些配置属性之后，我们实例化 file-rotatlogs，得到了一个符合 io.Writer 的输出，将这个输出使用 SetOutput 设置到嵌套的 HadeLog 中即可。

[复制代码](#)

```
1 // NewHadeRotateLog 实例化HadeRotateLog
2 func NewHadeRotateLog(params ...interface{}) (interface{}, error) {
3     // 参数解析
4     c := params[0].(framework.Container)
5     level := params[1].(contract.LogLevel)
6     ctxFielder := params[2].(contract.CtxFielder)
7     formatter := params[3].(contract.Formatter)
8
9     appService := c.MustMake(contract.AppKey).(contract.App)
10    configService := c.MustMake(contract.ConfigKey).(contract.Config)
11
12    // 从配置文件中获取folder信息，否则使用默认的LogFolder文件夹
13    folder := appService.LogFolder()
14    if configService.IsExist("log.folder") {
15        folder = configService.GetString("log.folder")
16    }
17    // 如果folder不存在，则创建
18    if !util.Exists(folder) {
19        os.MkdirAll(folder, os.ModePerm)
20    }
21
22    // 从配置文件中获取file信息，否则使用默认的hade.log
23    file := "hade.log"
24    if configService.IsExist("log.file") {
25        file = configService.GetString("log.file")
26    }
27
28    // 从配置文件获取date_format信息
29    dateFormat := "%Y%m%d%H"
30    if configService.IsExist("log.date_format") {
31        dateFormat = configService.GetString("log.date_format")
32    }
33
34    linkName := rotatlogs.WithLinkName(filepath.Join(folder, file))
35    options := []rotatlogs.Option{linkName}
36
37    // 从配置文件获取rotate_count信息
38    if configService.IsExist("log.rotate_count") {
39        rotateCount := configService.GetInt("log.rotate_count")
40        options = append(options, rotatlogs.WithRotationCount(uint(rotateCount))
41    }
42}
```

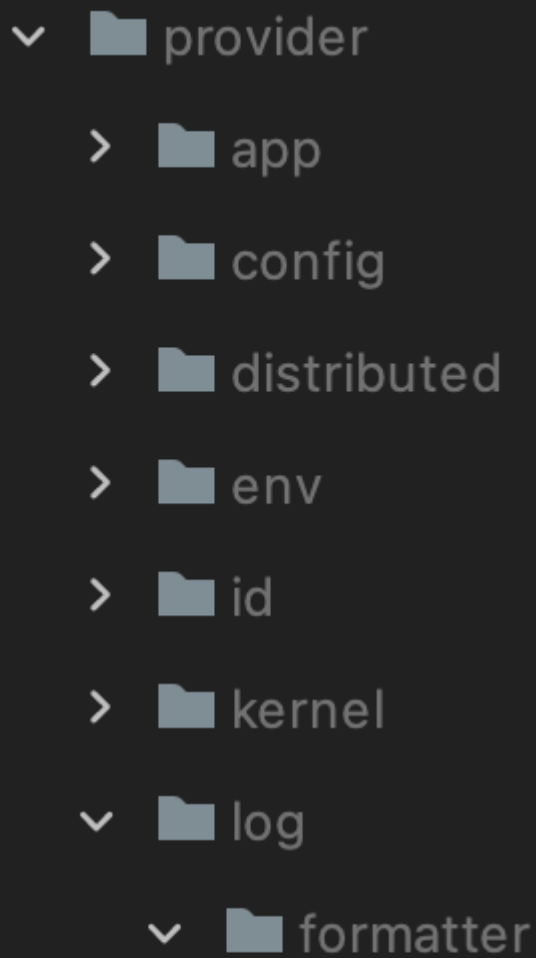
```
43 // 从配置文件获取rotate_size信息
44 if configService.IsExist("log.rotate_size") {
45     rotateSize := configService.GetInt("log.rotate_size")
46     options = append(options, rotatelog.WithRotationSize(int64(rotateSize))
47 }
48
49 // 从配置文件获取max_age信息
50 if configService.IsExist("log.max_age") {
51     if maxAgeParse, err := time.ParseDuration(configService.GetString("log.m
52         options = append(options, rotatelog.WithMaxAge(maxAgeParse))
53     }
54 }
55
56 // 从配置文件获取rotate_time信息
57 if configService.IsExist("log.rotate_time") {
58     if rotateTimeParse, err := time.ParseDuration(configService.GetString("l
59         options = append(options, rotatelog.WithRotationTime(rotateTimeParse
60     }
61 }
62
63 // 设置基础信息
64 log := &HadeRotateLog{}
65 log.SetLevel(level)
66 log.SetCtxFielder(ctxFielder)
67 log.SetFormatter(formatter)
68 log.folder = folder
69 log.file = file
70
71 w, err := rotatelog.New(fmt.Sprintf("%s.%s", filepath.Join(log.folder, log
72 if err != nil {
73     return nil, errors.Wrap(err, "new rotatelog error")
74 }
75 log.SetOutput(w)
76 log.c = c
77 return log, nil
78 }
```

本节课我们只是修改了框架目录中的日志服务相关的文件。文件目录：



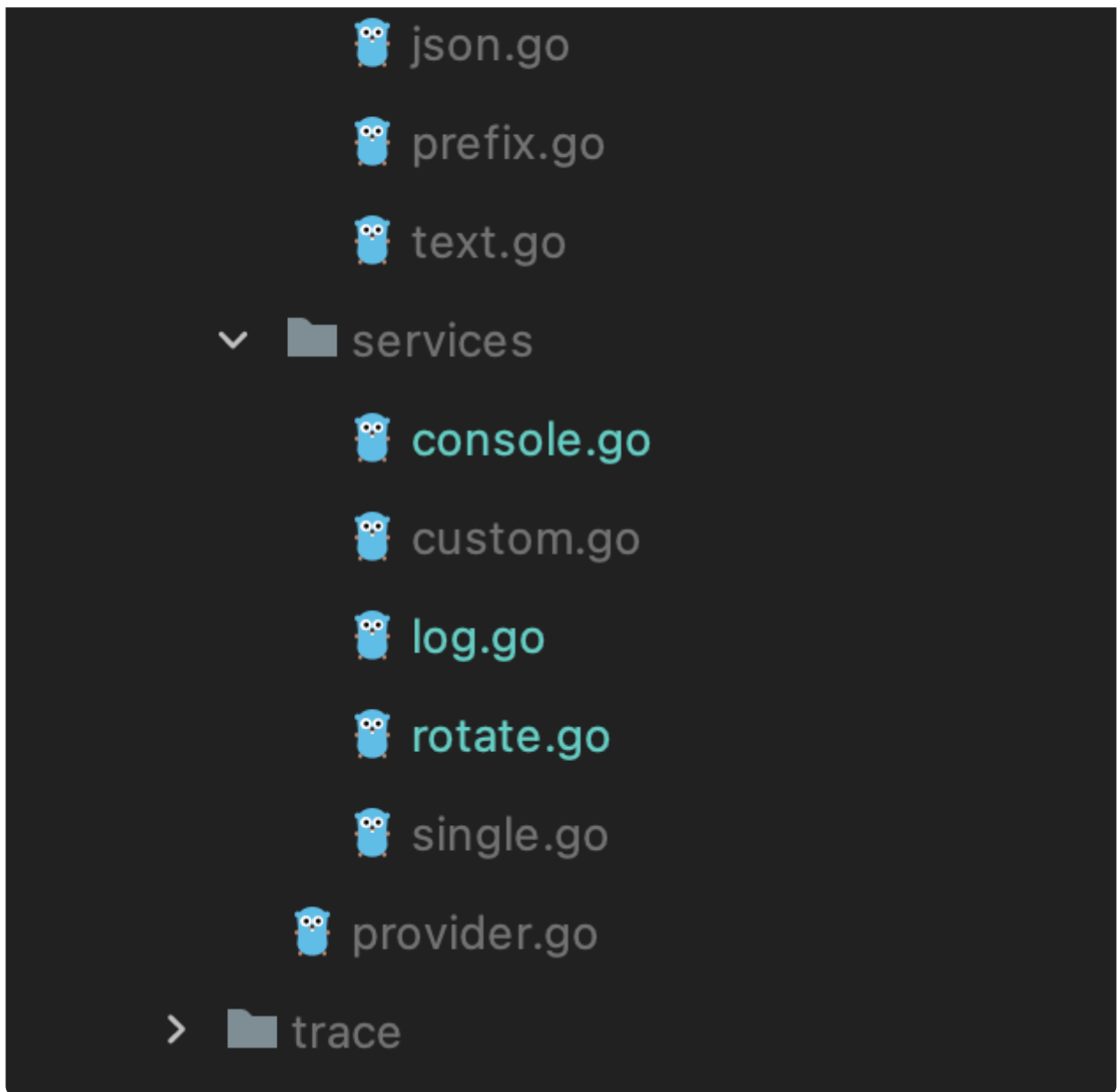
contract

- app.go
- config.go
- distributed.go
- env.go
- id.go
- kernel.go
- log.go
- trace.go



provider

- app
- config
- distributed
- env
- id
- kernel
- log
 - formatter



所有代码都放在 GitHub 上的 [@geekbang/17](https://github.com/geekbang/17) 分支，欢迎比对查看。

小结

我们这节课通过定义了日志级别、日志格式、日志输出，来实现了日志的级别，并且使用类型嵌套方法实现了四种本地日志输出方式。

回顾今天实现的日志服务，你会发现和其他服务的实现思路是差不多的。我们**在一个服务中，实现了多个实现类，但是所有的实现类都实现了同样的服务接口**，最后能让我们根据配置来决定这个服务使用哪个实现类，其中还使用了嵌套方式，能节省大量重复性的代码。

希望你能熟练掌握这种实现方式，因为我们的服务会越来越多，越上层的服务，比如数据库、缓存，它的具体实现就越多种多样，到时候我们都可以用同样的套路来进行。

思考题

在微服务盛行的今天，全链路日志是非常重要的一个需求。全链路日志的需求本质就是在日志中增加 `trace_id`、`span_id` 这样的链路字段。具体实现有三点：

在接收请求的时候，从请求 `request` 中解析全链路字段，存放进入 `context` 中

在打印日志的时候从 `context` 中获取全链路字段序列化进入日志

在发送请求的时候将全链路字段加入到 `request` 中

你可以思考下这个功能应该怎么实现？

欢迎在留言区分享你的思考。感谢你的收听，如果觉得有收获，也欢迎把今天的内容分享给你身边的朋友，邀他一起学习。我们下节课见。

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0  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 16 | 配置和环境：配置服务中的设计思路(下)

1024 活动特惠

VIP 年卡直降 ¥2000

新课上线即解锁，享 365 天畅看全场

超值拿下 ¥999



精选留言 (1)

写留言



qinsi

2021-10-25

接收下游请求和返回响应时打印日志可以在中间件中解决，向上游发送请求时打印日志可能需要框架提供一种方法截获请求并向其中注入trace信息，或者是自行封装一个http请求服务

作者回复：是的，一般是在中间件中解决

1

