

24 | 日志处理：日志规范与最佳实践

2022-12-03 郑建勋 来自北京



《Go进阶·分布式爬虫实战》

[课程介绍 >](#)



讲述：郑建勋

时长 10:29 大小 9.58M



你好，我是郑建勋。

这节课，我们需要构建项目的日志组件，方便我们收集打印的日志信息。运行中的程序就像一个黑盒，好在日志为我们记录了系统在不同时刻的运行状态。日志的好处主要有下面四点。

1. 打印调试：日志可以记录变量或者某一段逻辑，记录程序运行的流程。虽然用日志来调试通常被人认为是技术手段落后，但它确实能够解决某些难题。例如，一个场景线下无法复现，我们又不希望对线上系统产生破坏性的影响，这时打印调试就派上用场了。
2. 问题定位：有时候，系统或者业务出现问题，我们需要快速排查原因，这时我们就要用到日志的功能了。例如，Go 程序突然 `panic`，被 `recover` 捕获之后，打印出当前的详细堆栈信息，就需要通过日志来定位。
3. 用户行为分析：日志的大量数据可以作为大数据分析的基础，例如可以分析用户的行为偏好等。

4. 监控：日志数据通过流处理生成的连续指标数据，可以存储起来并对接监控告警平台，这有助于我们快速发现系统的异常。监控的指标可能包括：核心接口调用量是否突然下降或上升，核心的业务指标（GMV 是否同比和环比稳定，是否出现了不合理的订单，是否出现了零元或者天价账单）等。

标准库 log

而 Go 语言标准库就为我们提供了一个简单的 log 日志库，我们先从它的使用方法讲起。

标准库 log 提供了 3 个打印日志的接口，分别为 `log.Println`、`log.Fatalln` 和 `log.Panicln`。另外 log 还可以通过 `log.SetPrefix` 设置打印日志时的前缀，通过 `log.SetFlags` 设置打印的时间和文件的格式。

我们举一个简单的例子，下面这段代码使用了 Go 标准库。log.SetFlags 中的参数 `log.Ldate` 代表日志会打印日期，`log.Lmicroseconds` 代表日志会打印微秒，`log.Llongfile` 的意思是日志会输出长文件名的形式。

复制代码

```
1 func init() {
2     log.SetPrefix("TRACE: ")
3     log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile)
4 }
5
6 func main() {
7     log.Println("message")
8     log.Fatalln("fatal message")
9     log.Panicln("panic message")
10 }
```

输出如下所示：

复制代码

```
1 TRACE: 2022/10/12 22:22:36.540776 a/b/main.go:19: message
2 TRACE: 2022/10/12 22:22:36.541046 a/b/main.go:20: fatal message
```

要注意的是，`log.Fatalln` 会调用 `os.Exit(1)` 强制退出程序，所以就算没有打印出第三条日志，`log.Panicln` 也会使程序 panic，但是我们可以通过 `recover` 完成捕获。

可能你已经发现了，标准 `log` 库的不足之处在于，无法对日志进行分级，在生产环境中日志可能会有 `DEBUG`、`INFO` 等级别。并且我们也并不希望在打印日志时触发 `panic`，或者让程序直接退出。



`log` 提供了一些扩展能力，它让我们可以自定义不同级别的日志，例如借助 `log.New` 完成日志分级的功能，`log.New` 可以指定新 `log` 的输出位置、日志的前缀和格式。`Error.Println` 最终会将数据输出到文件中。

复制代码

```
1 var (  
2     Error    *log.Logger  
3     Warning  *log.Logger  
4 )  
5  
6 func init() {  
7     file, err := os.OpenFile("errors.txt",  
8         os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)  
9     if err != nil {  
10        log.Fatalf("Failed to open error log file:", err)  
11    }  
12  
13    Warning = log.New(os.Stdout,  
14        "WARNING: ",  
15        log.Ldate|log.Ltime|log.Lshortfile)  
16  
17    Error = log.New(file,  
18        "ERROR: ",  
19        log.Ldate|log.Ltime|log.Lshortfile)  
20 }  
21  
22 func main() {  
23     Warning.Println("There is something you need to know about")  
24     Error.Println("Something has failed")  
25 }
```

虽然提供了有限的扩展能力，标准库 `log` 仍然存在一些不足之处（例如对参数的处理借助了 `fmt.Sprintf` 函数，中间包含了大量反射，性能比较低下；而且 `log` 不会输出类似 `JSON` 的结构化数据，要想实现文件切割也比较困难），所以我们一般是在本地开发环境中使用它。


那么一个工业级的日志组件应该具备什么特性呢，我列出了一些需要考虑的点：

- 不同的环境有不同的输出行为，比如测试和开发环境要输出到控制台，而生产环境则要输出到文件中；
- 日志分等级；
- 类似 JSON 的结构化输出，这会使日志更容易阅读，也有助于后续查找和存储日志；
- 支持日志文件切割（可以按照日期、时间间隔或者文件大小对日志进行切割）；
- 可以定义输出的格式，例如打印日志的函数、所在的文件、行号、记录时间等；
- 速度快。

Zap

在 Go 语言中，满足这些条件而且比较知名的日志库有 **Zap**、**Logrus** 和 **Zerolog**。

其中，**Zap** 是 **Uber** 开源的日志组件，当前在社区非常受欢迎。在满足了日志库应该具备的基本功能的基础上，**Zap** 在内存分配量与速度方面相比其他日志组件都有较大的优势。所以在我们后面的项目中将通过封装 **Zap** 来实现日志的打印能力。

 **Zap** 提供了两种类型的日志，分别是 **Logger** 与 **SugaredLogger**。其中，**Logger** 是默认的，每个要写入的字段都得指定对应类型的方法，这种方式可以不使用反射，因此效率更高。为了避免在异常情况下丢失日志（尤其是在崩溃时），**logger.Sync()** 会在进程退出之前落盘所有位于缓冲区中的日志条目。

 复制代码

```
1 package main
2 import "go.uber.org/zap"
3 func main() {
4     logger, _ := zap.NewProduction()
5     defer logger.Sync()
6     url := "www.google.com"
7     logger.Info("failed to fetch URL",
8         zap.String("url", url),
9         zap.Int("attempt", 3),
10        zap.Duration("backoff", time.Second))
11 }
```

而 **SugaredLogger** 的性能稍微低于 **Logger**，但是它提供了一种更灵活的打印方式：

```

1 func main() {
2     logger, _ := zap.NewProduction()
3     defer logger.Sync()
4     sugar := logger.Sugar()
5     url := "www.google.com"
6     sugar.Infow("failed to fetch URL",
7         "url", url,
8         "attempt", 3,
9         "backoff", time.Second,
10    )
11 }

```



Zap 在默认情况下会输出 JSON 格式的日志，上面这个例子中的输出为：

```

1 {"level":"info","ts":1665669124.251897,"caller":"a/b.go:20","msg":"failed to fe

```

Zap 预置了 3 种格式的 Logger，分别为 `zap.NewExample()`、`zap.NewDevelopment()` 和 `zap.NewProduction()`。这 3 个函数可以传入若干类型为 `zap.Option` 的选项，从而扩展 Logger 的行为。选项包括 `zap.WithCaller` 打印文件、行号、`zap.AddStacktrace` 打印堆栈信息。

除此之外，我们还可以定制自己的 Logger，提供比预置的 Logger 更灵活的能力。举一个例子，`zap.NewProduction()` 实际调用了 `NewProductionConfig().Build()`，而 `NewProductionConfig()` 生成的 `zap.Config` 可以被定制化。

```

1 func NewProduction(options ...Option) (*Logger, error) {
2     return NewProductionConfig().Build(options...)
3 }

```

在下面这个例子中，我修改了 Zap 中打印时间的格式：

```

1 func main() {
2     loggerConfig := zap.NewProductionConfig()
3     loggerConfig.EncoderConfig.TimeKey = "timestamp"
4     loggerConfig.EncoderConfig.EncodeTime = zapcore.TimeEncoderOfLayout(time.RFC3

```

```
5     logger, err := loggerConfig.Build()
6     if err != nil {
7         log.Fatal(err)
8     }
9
10    sugar := logger.Sugar()
11
12    sugar.Info("Hello from zap logger")
13 }
14 }
```



输出如下，根据合理灵活的日志格式配置，我们就可以满足项目的不同需要了。

 复制代码

```
1 {"level":"info","timestamp":"2022-10-13T23:12:17+08:00","caller":"a/main.go:169"}
```

日志切割

在 Zap 中，我们也可以通过底层的 Zap.New 函数的扩展能力完成更加定制化的操作。例如，指定日志输出的 Writer 行为。

不同的 Writer 可能有不同的写入行为，像是输出到文件还是控制台，是否需要根据时间和文件的大小对日志进行切割等。Zap 将日志切割的能力开放了出来，只要日志切割组件实现了 zapcore.WriteSyncer 接口，就可以集成到 Zap 中。比较常用的日志切割组件为 [lumberjack.v2](#)。下面这个例子将 lumberjack.v2 组件集成到了 Zap 中：

 复制代码

```
1 w := &lumberjack.Logger{
2     Filename:   "/var/log/myapp/foo.log",
3     MaxSize:    500, // 日志的最大大小，以M为单位
4     MaxBackups: 3,   // 保留的旧日志文件的最大数量
5     MaxAge:     28,  // 保留旧日志文件的最大天数
6 }
7 core := zapcore.NewCore(
8     zapcore.NewJSONEncoder(zap.NewProductionEncoderConfig()),
9     w,
10    zap.InfoLevel,
11 )
12 logger := zap.New(core)
```


日志分级

设置好日志组件的基本属性之后就可以打印日志了。在输出日志时，我们需要根据日志的用途进行分级。目前最佳的实践是将日志级别分为了五类。



- **DEBUG**

DEBUG 级别的日志，顾名思义主要是用来调试的。通过打印当前程序的调用链，我们可以知道程序运行的逻辑，了解关键分支中详细的变量信息、上下游请求参数和耗时等，帮助开发者调试错误，判断逻辑是否符合预期。**DEBUG** 日志一般用在开发和测试初期。不过，由于太多的 **DEBUG** 日志会降低线上程序的性能、同时导致成本上升，因此，在生产环境中一般不会打印 **DEBUG** 日志。

- **INFO**

INFO 级别的日志记录了系统中核心的指标。例如，初始化时配置文件的路径，程序所处集群的环境和版本号，核心指标和状态的变化；再比如，监听到外部节点数量的变化，或者外部数据库地址的变化。**INFO** 信息有助于我们了解程序的整体运行情况，快速排查问题。

- **WARNING**

WARNING 级别的日志用于输出程序中预知的，程序目前仍然能够处理的问题。例如，打车服务中，行程信息会存储在缓存中方便我们快速查找。但是如果在缓存中查不到用户的行程信息，这时我们可以用一些兜底的策略重建行程，继续完成后续的流程。这种不影响正常流程，但又不太符合预期的情况就适合用 **WARNING**。**WARNING** 还可以帮助我们在事后分析异常情况出现的原因。

- **ERROR**

ERROR 级别的日志主要针对一些不可预知的问题，例如网络通信或者数据库连接的异常等。

- **FATAL**

FATAL 级别的日志主要针对程序遇到的严重问题，意味着需要立即终止程序。例如遇到了不能容忍的并发冲突时，就应该使用 **FATAL** 级别的日志。

Zap 日志库在上面五个分级的基础上还增加了 **Panic** 和 **DPanic** 级别，如下所示。**Panic** 级别打印后会触发 **panic**。而 **DPanic** 级别比较特殊，它在 **development** 模式下会 **panic**，相当于 **PanicLevel** 级别，而在其他模式下相当于 **ErrorLevel** 级别。

```
2  DebugLevel = zapcore.DebugLevel
3  InfoLevel  = zapcore.InfoLevel
4  WarnLevel  = zapcore.WarnLevel
5  ErrorLevel = zapcore.ErrorLevel
6  DPanicLevel = zapcore.DPanicLevel
7  PanicLevel = zapcore.PanicLevel
8  FatalLevel = zapcore.FatalLevel
9  }
```



日志格式规范

对于打印出来的日志格式，我们希望它尽量符合通用的规范，以便公共的采集通道进行进一步的日志处理。一个合格的日志至少应该有具体的时间、打印的行号、日志级别等重要信息。同时，在大规模的集群中，还可能包含机器的 IP 地址，调用者 IP 和其他业务信息。规范日志示例如下：

 复制代码

```
1 { "level": "info", "timestamp": "2022-10-13T23:29:10+08:00", "caller": "a/main.go:168"
```

下面列出一个实际中在使用的日志规范，可以作为一个参考：

- 每一行日志必须至少包含日志 Key、TimeStamp、打印的行号、日志级别字段；
- 单条日志长度在 1MB 以内，单个服务节点每秒的日志量小于 20MB；
- 建议使用 UTF-8 字符集，避免入库时出现乱码；
- Key 为每个字段的字段名，一行示例中的 Key 是唯一的；
- Key 的命名只能包含数字、字母、下划线，且必须以字母开头；
- Key 的长度不可大于 80 个字符；
- Key 中的字母均为小写。

构建项目日志组件

接下来，让我们利用 Zap 构建项目的日志组件，将代码放入到新的 log 文件夹中。关于日志组件的代码，你可以查看代码库 v0.1.1 分支。

我们把 `zapcore.Core` 作为一个自定义类型 `Plugin`，`zapcore.Core` 定义了日志的编码格式以及输出位置等核心功能。作为一个通用的库，下面我们实现了 `NewStdoutPlugin`、`NewStderrPlugin`、`NewFilePlugin` 这三个函数，分别对应了输出日志到 `stdout`、`stderr` 和文件中。这三个函数最终都调用了 `zapcore.NewCore` 函数。

复制代码

```
1 type Plugin = zapcore.Core
2
3 func NewStdoutPlugin(enabler zapcore.LevelEnabler) Plugin {
4     return NewPlugin(zapcore.Lock(zapcore.AddSync(os.Stdout)), enabler)
5 }
6
7 func NewStderrPlugin(enabler zapcore.LevelEnabler) Plugin {
8     return NewPlugin(zapcore.Lock(zapcore.AddSync(os.Stderr)), enabler)
9 }
10
11 // Lumberjack logger虽然持有File但没有暴露sync方法，所以没办法利用zap的sync特性
12 // 所以额外返回一个closer，需要在进程退出前close以保证写入的内容可以全部刷到磁盘
13 func NewFilePlugin(
14     filePath string, enabler zapcore.LevelEnabler) (Plugin, io.Closer) {
15     var writer = DefaultLumberjackLogger()
16     writer.Filename = filePath
17     return NewPlugin(zapcore.AddSync(writer), enabler), writer
18 }
19
20 func NewPlugin(writer zapcore.WriteSyncer, enabler zapcore.LevelEnabler) Plugin
21     return zapcore.NewCore(DefaultEncoder(), writer, enabler)
22 }
```

`NewFilePlugin` 中暴露出了两个参数：`filePath` 和 `enabler`。`filePath` 表示输出文件的路径，而 `enabler` 代表当前环境中要打印的日志级别。刚才我们梳理了 Zap 中的七种日志级别，它们是一个整数，按照等级从上到下排列的，等级最低的是 `Debug`，等级最高的为 `Fatal`。在生产环境中，我们并不希望打印 `Debug` 日志，因此我们可以在生产环境中指定 `enabler` 参数为 `InfoLevel` 级别，这样，就只有大于等于 `enabler` 的日志等级才会被打印了。

复制代码

```
1 const (
2     DebugLevel = zapcore.DebugLevel
3     InfoLevel  = zapcore.InfoLevel
4     WarnLevel  = zapcore.WarnLevel
5     ErrorLevel = zapcore.ErrorLevel
6     DPanicLevel = zapcore.DPanicLevel
7     PanicLevel = zapcore.PanicLevel
8     FatalLevel  = zapcore.FatalLevel
9 )
```



下一步是日志切割，在 `NewFilePlugin` 中，我们借助 `lumberjack.v2` 来完成日志的切割。

[复制代码](#)

```
1 // 1.不会自动清理backup
2 // 2.每200MB压缩一次，不按时间切割
3 func DefaultLumberjackLogger() *lumberjack.Logger {
4     return &lumberjack.Logger{
5         MaxSize:    200,
6         LocalTime:  true,
7         Compress:   true,
8     }
9 }
```

最后，我们要暴露一个通用的函数 `NewLogger` 来生成 `logger`。默认选项会打印调用时的文件与行号，并且只有当日志等级在 `DPanic` 等级之上时，才输出函数的堆栈信息。

[复制代码](#)

```
1 func NewLogger(plugin zapcore.Core, options ...zap.Option) *zap.Logger {
2     return zap.New(plugin, append(DefaultOption(), options...)...)
3 }
4
5 func DefaultOption() []zap.Option {
6     var stackTraceLevel zap.LevelEnablerFunc = func(level zapcore.Level) bool {
7         return level >= zapcore.DPanicLevel
8     }
9     return []zap.Option{
10         zap.AddCaller(),
11         zap.AddStacktrace(stackTraceLevel),
12     }
13 }
```

现在让我们在 `main` 函数中集成 `log` 组件，文件名和日志级别现在是写死的，后续我们会统一放入到配置文件中。

[复制代码](#)

```
1 func main() {
2     plugin, c := log.NewFilePlugin("./log.txt", zapcore.InfoLevel)
3     defer c.Close()
4     logger := log.NewLogger(plugin)
```

```
5 logger.Info("log init end")
6 }
```



输出为:

复制代码

```
1 {"level":"INFO","ts":"2022-10-14T23:59:15.701+0800","caller":"crawler/main.go:1"}
```

这样，我们就可以在项目中愉快地打印日志了。

总结

日志可以帮助我们了解程序的运行状态，在调试、问题定位、监控等场景下都有诸多的用处。

但是，Go 标准库虽然提供了简单的日志功能，却不具备日志分级、结构化输出、日志切割、自定义输出格式的功能，性能也相对低下，我们一般在项目中并不能直接使用它。

好在满足我们使用需求的日志组件有很多，最出名的就有 Zap、Logrus 和 Zerolog。其中，Zap 在速度和内存分配上都有明显的优势。这节课，我们就封装了 Zap 来实现了项目的日志组件，并让它具备了可扩展、可切割的能力。后续日志的打印我们都将使用这个项目封装的日志组件。

课后题

今天的思考题是这样的。Go 团队正在开发一个日志库 [slog](#)，你觉得 Go 团队为什么要设置这个日志库，它和其他的日志库有什么区别？

欢迎你在留言区与我交流讨论，我们下节课见。

分享给需要的人，Ta 购买本课程，你将得 20 元

生成海报并分享



天下无鱼

<https://shikey.com/>

[上一篇](#) 23 | 偷梁换柱：为爬虫安上代理的翅膀

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。