

# Go 每日一库之 zap

Original dj GoUpUp 2020-04-24

收录于话题

#Go 每日一库

48个 >

## 简介

在很早之前的文章中，我们介绍过 Go 标准日志库 `log` 和结构化的日志库 `logrus`。在热点函数中记录日志对日志库的执行性能有较高的要求，不能影响正常逻辑的执行时间。`uber` 开源的日志库 `zap`，对性能和内存分配做了极致的优化。

## 快速使用

先安装：

```
$ go get go.uber.org/zap
```

后使用：

```
package main

import (
```

```

"time"

"go.uber.org/zap"
)

func main() {
    logger := zap.NewExample()
    defer logger.Sync()

    url := "http://example.org/api"
    logger.Info("failed to fetch URL",
        zap.String("url", url),
        zap.Int("attempt", 3),
        zap.Duration("backoff", time.Second),
    )

    sugar := logger.Sugar()
    sugar.Infow("failed to fetch URL",
        "url", url,
        "attempt", 3,
        "backoff", time.Second,
    )
    sugar.Infof("Failed to fetch URL: %s", url)
}

```

`zap` 库的使用与其他的日志库非常相似。先创建一个 `logger`，然后调用各个级别的方法记录日志（`Debug/Info/Error/Warn`）。`zap` 提供了几  
个快速创建 `logger` 的方法，`zap.NewExample()`、`zap.NewDevelopment()`、`zap.NewProduction()`，还有高度定制化的创建方法 `zap.Ne`

`w()`。创建前 3 个 `logger` 时，`zap` 会使用一些预定义的设置，它们的使用场景也有所不同。`Example` 适合用在测试代码中，`Development` 在开发环境中使用，`Production` 用在生成环境。

`zap` 底层 API 可以设置缓存，所以一般使用 `defer logger.Sync()` 将缓存同步到文件中。

由于 `fmt.Printf` 之类的方法大量使用 `interface{}` 和反射，会有不少性能损失，并且增加了内存分配的频次。`zap` 为了提高性能、减少内存分配次数，没有使用反射，而且默认的 `Logger` 只支持强类型的、结构化的日志。必须使用 `zap` 提供的方法记录字段。`zap` 为 Go 语言中所有的基本类型和其他常见类型都提供了方法。这些方法的名称也比较好记忆，`zap.Type`（`Type` 为 `bool/int/uint/float64/complex64/time.Time/time.Duration/error` 等）就表示该类型的字段，`zap.Typep` 以 `p` 结尾表示该类型指针的字段，`zap.Types` 以 `s` 结尾表示该类型切片的字段。如：

- `zap.Bool(key string, val bool) Field` : `bool` 字段
- `zap.Boolp(key string, val *bool) Field` : `bool` 指针字段；
- `zap.Bools(key string, val []bool) Field` : `bool` 切片字段。

当然也有一些特殊类型的字段：

- `zap.Any(key string, value interface{}) Field` : 任意类型的字段；
- `zap.Binary(key string, val []byte) Field` : 二进制串的字段。

当然，每个字段都用方法包一层用起来比较繁琐。`zap` 也提供了便捷的方法 `SugarLogger`，可以使用 `printf` 格式符的方式。调用 `logger.Sugar()` 即可创建 `SugaredLogger`。`SugaredLogger` 的使用比 `Logger` 简单，只是性能比 `Logger` 低 50% 左右，可以用在非热点函数中。调用 `SugarLogger` 以 `f` 结尾的方法与 `fmt.Printf` 没什么区别，如例子中的 `Infof`。同时 `SugarLogger` 还支持以 `w` 结尾的方法，这种方式不需要先创建字段对象，直接将字段名和值依次放在参数中即可，如例子中的 `Infow`。

默认情况下，`Example` 输出的日志为 JSON 格式：

```
{ "level": "info", "msg": "failed to fetch URL", "url": "http://example.org/api", "attempt": 3, "backoff": "1s" }
{ "level": "info", "msg": "failed to fetch URL", "url": "http://example.org/api", "attempt": 3, "backoff": "1s" }
{ "level": "info", "msg": "Failed to fetch URL: http://example.org/api" }
```

## 记录层级关系

前面我们记录的日志都是一层结构，没有嵌套的层级。我们可以使用 `zap.Namespace(key string) Field` 构建一个命名空间，后续的 `Field` 都记录在此命名空间中：

```
func main() {
    logger := zap.NewExample()
    defer logger.Sync()

    logger.Info("tracked some metrics",
        zap.Namespace("metrics"),
        zap.Int("counter", 1),
    )

    logger2 := logger.With(
        zap.Namespace("metrics"),
        zap.Int("counter", 1),
    )
    logger2.Info("tracked some metrics")
}
```

输出：

```
{"level":"info","msg":"tracked some metrics","metrics":{"counter":1}}
{"level":"info","msg":"tracked some metrics","metrics":{"counter":1}}
```

上面我们演示了两种 `Namespace` 的用法，一种是直接作为字段传入 `Debug/Info` 等方法，一种是调用 `With()` 创建一个新的 `Logger`，新的 `Logger` 记录日志时总是带上预设的字段。`With()` 方法实际上是创建了一个新的 `Logger`：

```
// src/go.uber.org/zap/logger.go
func (log *Logger) With(fields ...Field) *Logger {
    if len(fields) == 0 {
        return log
    }
    l := log.clone()
    l.core = l.core.With(fields)
    return l
}
```

## 定制Logger

调用 `NexExample()/NewDevelopment()/NewProduction()` 这 3 个方法，`zap` 使用默认的配置。我们也可以手动调整，配置结构如下：

```
// src/go.uber.org/zap/config.go
type Config struct {
    Level AtomicLevel `json:"level" yaml:"level"`
    Encoding string `json:"encoding" yaml:"encoding"`
    EncoderConfig zapcore.EncoderConfig `json:"encoderConfig" yaml:"encoderConfig"`
}
```

```

OutputPaths []string `json:"outputPaths" yaml:"outputPaths"`
ErrorOutputPaths []string `json:"errorOutputPaths" yaml:"errorOutputPaths"`
InitialFields map[string]interface{} `json:"initialFields" yaml:"initialFields"`
}

```

- **Level** : 日志级别；
- **Encoding** : 输出的日志格式，默认为 JSON；
- **OutputPaths** : 可以配置多个输出路径，路径可以是文件路径和 **stdout**（标准输出）；
- **ErrorOutputPaths** : 错误输出路径，也可以是多个；
- **InitialFields** : 每条日志中都会输出这些值。

其中 **EncoderConfig** 为编码配置：

```

// src/go.uber.org/zap/zapcore/encoder.go
type EncoderConfig struct {
    MessageKey    string `json:"messageKey" yaml:"messageKey"`
    LevelKey      string `json:"levelKey" yaml:"levelKey"`
    TimeKey       string `json:"timeKey" yaml:"timeKey"`
    NameKey       string `json:"nameKey" yaml:"nameKey"`
    CallerKey     string `json:"callerKey" yaml:"callerKey"`
    StacktraceKey string `json:"stacktraceKey" yaml:"stacktraceKey"`
    LineEnding    string `json:"lineEnding" yaml:"lineEnding"`
    EncodeLevel   LevelEncoder `json:"levelEncoder" yaml:"levelEncoder"`
    EncodeTime    TimeEncoder  `json:"timeEncoder" yaml:"timeEncoder"`
    EncodeDuration DurationEncoder `json:"durationEncoder" yaml:"durationEncoder"`
}

```

```
EncodeCaller CallerEncoder `json:"callerEncoder" yaml:"callerEncoder"`
EncodeName NameEncoder `json:"nameEncoder" yaml:"nameEncoder"`
}
```

- **MessageKey** : 日志中信息的键名, 默认为 **msg** ;
- **LevelKey** : 日志中级别的键名, 默认为 **level** ;
- **EncodeLevel** : 日志中级别的格式, 默认为小写, 如 **debug/info** 。

调用 **zap.Config** 的 **Build()** 方法即可使用该配置对象创建一个 **Logger** :

```
func main() {
    rawJSON := []byte(`{
        "level": "debug",
        "encoding": "json",
        "outputPaths": ["stdout", "server.log"],
        "errorOutputPaths": ["stderr"],
        "initialFields": {"name": "dj"},
        "encoderConfig": {
            "messageKey": "message",
            "levelKey": "level",
            "levelEncoder": "lowercase"
        }
    }`)

    var cfg zap.Config
    if err := json.Unmarshal(rawJSON, &cfg); err != nil {
```

```

    panic(err)
}
logger, err := cfg.Build()
if err != nil {
    panic(err)
}
defer logger.Sync()

logger.Info("server start work successfully!")
}

```

上面创建一个输出到标准输出 `stdout` 和文件 `server.log` 的 `Logger`。观察输出：

```

{"level":"info","message":"server start work successfully!","name":"dj"}

```

使用 `NewDevelopment()` 创建的 `Logger` 使用的是如下的配置：

```

// src/go.uber.org/zap/config.go
func NewDevelopmentConfig() Config {
    return Config{
        Level:          NewAtomicLevelAt(DebugLevel),
        Development:    true,
        Encoding:       "console",
        EncoderConfig:  NewDevelopmentEncoderConfig(),
        OutputPaths:    []string{"stderr"},
        ErrorOutputPaths: []string{"stderr"},
    }
}

```



```

}
}

func NewDevelopmentEncoderConfig() zapcore.EncoderConfig {
    return zapcore.EncoderConfig{
        // Keys can be anything except the empty string.
        TimeKey:      "T",
        LevelKey:     "L",
        NameKey:      "N",
        CallerKey:    "C",
        MessageKey:   "M",
        StacktraceKey: "S",
        LineEnding:   zapcore.DefaultLineEnding,
        EncodeLevel:   zapcore.CapitalLevelEncoder,
        EncodeTime:    zapcore.ISO8601TimeEncoder,
        EncodeDuration: zapcore.StringDurationEncoder,
        EncodeCaller:  zapcore.ShortCallerEncoder,
    }
}

```

`NewProduction()` 的配置可自行查看。

## 选项

`NewExample()/NewDevelopment()/NewProduction()` 这 3 个函数可以传入若干类型为 `zap.Option` 的选项，从而定制 `Logger` 的行为。又一次见到了选项模式！！

`zap` 提供了丰富的选项供我们选择。

## 输出文件名和行号

调用 `zap.AddCaller()` 返回的选项设置输出文件名和行号。但是有一个前提，必须设置配置对象 `Config` 中的 `CallerKey` 字段。也因此 `NewExample()` 不能输出这个信息（它的 `Config` 没有设置 `CallerKey`）。

```
func main() {
    logger, _ := zap.NewProduction(zap.AddCaller())
    defer logger.Sync()

    logger.Info("hello world")
}
```

输出：

```
{"level":"info","ts":1587740198.9508286,"caller":"caller/main.go:9","msg":"hello world"}
```

`Info()` 方法在 `main.go` 的第 9 行被调用。`AddCaller()` 与 `zap.WithCaller(true)` 等价。

有时我们稍微封装了一下记录日志的方法，但是我们希望输出的文件名和行号是调用封装函数的位置。这时可以使用 `zap.AddCallerSkip(skip int)` 向上跳 1 层：

```
func Output(msg string, fields ...zap.Field) {
    zap.L().Info(msg, fields...)
}
```

```
func main() {  
    logger, _ := zap.NewProduction(zap.AddCaller(), zap.AddCallerSkip(1))  
    defer logger.Sync()  
  
    zap.ReplaceGlobals(logger)  
  
    Output("hello world")  
}
```

输出：

```
{"level":"info","ts":1587740501.5592482,"caller":"skip/main.go:15","msg":"hello world"}
```

输出在 `main` 函数中调用 `Output()` 的位置。如果不指定 `zap.AddCallerSkip(1)`，将输出 `"caller":"skip/main.go:6"`，这是在 `Output()` 函数中调用 `zap.Info()` 的位置。因为这个 `Output()` 函数可能在很多地方被调用，所以这个位置参考意义并不大。试试看！

---

## 输出调用堆栈

有时候在某个函数处理中遇到了异常情况，因为这个函数可能在很多地方被调用。如果我们能输出此次调用的堆栈，那么分析起来就会很方便。我们可以使用 `zap.AddStackTrace(lvl zapcore.LevelEnabler)` 达成这个目的。该函数指定 `lvl` 和之上的级别都需要输出调用堆栈：

```
func f1() {  
    f2("hello world")  
}
```

```
func f2(msg string, fields ...zap.Field) {
    zap.L().Warn(msg, fields...)
}

func main() {
    logger, _ := zap.NewProduction(zap.AddStacktrace(zapcore.WarnLevel))
    defer logger.Sync()

    zap.ReplaceGlobals(logger)

    f1()
}
```

将 `zapcore.WarnLevel` 传入 `AddStacktrace()`，之后 `Warn()/Error()` 等级别的日志会输出堆栈，`Debug()/Info()` 这些级别不会。运行结果：

```
{ "level": "warn", "ts": 1587740883.4965692, "caller": "stacktrace/main.go:13", "msg": "hello world", "stacktrace": "main.f2\n\td:/code/golang/src
```

把 `stacktrace` 单独拉出来：

```
main.f2
d:/code/golang/src/github.com/darjun/go-daily-lib/zap/option/stacktrace/main.go:13
main.f1
d:/code/golang/src/github.com/darjun/go-daily-lib/zap/option/stacktrace/main.go:9
main.main
d:/code/golang/src/github.com/darjun/go-daily-lib/zap/option/stacktrace/main.go:22
```

```
runtime.main
C:/Go/src/runtime/proc.go:203
```

很清楚地看到调用路径。

## 全局Logger

为了方便使用，`zap` 提供了两个全局的 `Logger`，一个是 `*zap.Logger`，可调用 `zap.L()` 获得；另一个是 `*zap.SugaredLogger`，可调用 `zap.S()` 获得。需要注意的是，全局的 `Logger` 默认并不会记录日志！它是一个无实际效果的 `Logger`。看源码：

```
// go.uber.org/zap/global.go
var (
    _globalMu sync.RWMutex
    _globalL  = NewNop()
    _globalS  = _globalL.Sugar()
)
```

我们可以使用 `ReplaceGlobals(logger *Logger) func()` 将 `logger` 设置为全局的 `Logger`，该函数返回一个无参函数，用于恢复全局 `Logger` 设置：

```
func main() {
    zap.L().Info("global Logger before")
    zap.S().Info("global SugaredLogger before")

    logger := zap.NewExample()
    defer logger.Sync()
```

```
zap.ReplaceGlobals(logger)
zap.L().Info("global Logger after")
zap.S().Info("global SugaredLogger after")
}
```

输出：

```
{"level":"info","msg":"global Logger after"}
{"level":"info","msg":"global SugaredLogger after"}
```

可以看到在调用 `ReplaceGlobals` 之前记录的日志并没有输出。

---

## 预设日志字段

如果每条日志都要记录一些共用的字段，那么使用 `zap.Fields(fs ...Field)` 创建的选项。例如在服务器日志中记录可能都需要记录 `serverId` 和 `serverName`：

```
func main() {
    logger := zap.NewExample(zap.Fields(
        zap.Int("serverId", 90),
        zap.String("serverName", "awesome web"),
    ))

    logger.Info("hello world")
}
```

输出：

```
{"level":"info","msg":"hello world","serverId":90,"serverName":"awesome web"}
```

### 与标准日志库搭配使用

如果项目一开始使用的是标准日志库 `log`，后面想转为 `zap`。这时不必修改每一个文件。我们可以调用 `zap.NewStdLog(l *Logger) *log.Logger` 返回一个标准的 `log.Logger`，内部实际上写入的还是我们之前创建的 `zap.Logger`：

```
func main() {
    logger := zap.NewExample()
    defer logger.Sync()

    std := zap.NewStdLog(logger)
    std.Print("standard logger wrapper")
}
```

输出：

```
{"level":"info","msg":"standard logger wrapper"}
```

很方便不是吗？我们还可以使用 `NewStdLogAt(l *logger, level zapcore.Level) (*log.Logger, error)` 让标准接口以 `level` 级别写入内部的 `*zap.Logger`。

如果我们只是想在一小段代码内使用标准日志库 `log`，其它地方还是使用 `zap.Logger`。可以调用 `RedirectStdLog(l *Logger) func()`。它会返回一个无参函数恢复设置：

```
func main() {
    logger := zap.NewExample()
    defer logger.Sync()

    undo := zap.RedirectStdLog(logger)
    log.Print("redirected standard library")
    undo()

    log.Print("restored standard library")
}
```

看前后输出变化：

```
{"level":"info","msg":"redirected standard library"}
2020/04/24 22:13:58 restored standard library
```

当然 `RedirectStdLog` 也有一个对应的 `RedirectStdLogAt` 以特定的级别调用内部的 `*zap.Logger` 方法。

## 总结

`zap` 用在日志性能和内存分配比较关键的地方。本文仅介绍了 `zap` 库的基本使用，子包 `zapcore` 中有更底层的接口，可以定制丰富多样的 `Logger`。



大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😊

## 参考

1. zap GitHub： <https://github.com/jordan-wright/zap>
2. Go 每日一库 GitHub： <https://github.com/darjun/go-daily-lib>

## 我

我的博客：<https://darjun.github.io>

欢迎关注我的微信公众号【GoUpUp】，共同学习，一起进步~



People who liked this content also liked

Go 每日一库之 reflect

GoUpUp



百万系列57—119DF（冠军风采 & 拥有七牛之力的牛云长爱好者）

有树的神武分享



打屁股会影响孩子智商？家长们可长点心吧...

科普中国

